

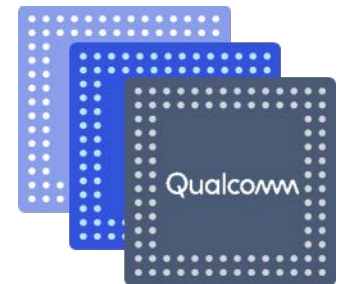
Sensors Execution Environment (SEE) Sensors Deep Dive

80-P9301-35 Rev. B

Confidential and Proprietary – Qualcomm Technologies, Inc.

NO PUBLIC DISCLOSURE PERMITTED: Please report postings of this document on public servers or websites to: DocCtrlAgent@qualcomm.com.

Restricted Distribution: Not to be distributed to anyone who is not an employee of either Qualcomm Technologies, Inc. or its affiliated companies without the express approval of Qualcomm Configuration Management.



Confidential and Proprietary – Qualcomm Technologies, Inc.

Qualcomm
2019-04-14 20:26:28 PDT
zk_sw@wingtech.com

Confidential and Proprietary – Qualcomm Technologies, Inc.

NO PUBLIC DISCLOSURE PERMITTED: Please report postings of this document on public servers or websites to: DocCtrlAgent@qualcomm.com.

Restricted Distribution: Not to be distributed to anyone who is not an employee of either Qualcomm Technologies, Inc. or its affiliated companies without the express approval of Qualcomm Configuration Management.

Not to be used, copied, reproduced, or modified in whole or in part, nor its contents revealed in any manner to others without the express written permission of Qualcomm Technologies, Inc.

All Qualcomm products mentioned herein are products of Qualcomm Technologies, Inc. and/or its subsidiaries.

Qualcomm, Snapdragon, MSM, and QXDM Professional are trademarks of Qualcomm Incorporated, registered in the United States and other countries. Qualcomm All-Ways Aware is a trademark of Qualcomm Incorporated. Other product and brand names may be trademarks or registered trademarks of their respective owners.

This technical data may be subject to U.S. and international export, re-export, or transfer (“export”) laws. Diversion contrary to U.S. and international law is strictly prohibited.

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

© 2017, 2019 Qualcomm Technologies, Inc. and/or its subsidiaries. All rights reserved.

Revision History

Revision	Date	Description
A	October 2017	Initial release
B	March 2019	Numerous updates have been made to this document; it should be read in its entirety

Contents

- [Introduction](#)
- [Sensors Execution Environment](#)
- [SEE Sensor API](#)
- [Protocol Buffers in SEE](#)
- [Sensor API Messages](#)
- [SEE Service Manager and Services](#)
- [SEE Platform Sensors](#)
- [SEE Utilities](#)
- [SEE Sensor Heartbeat Feature](#)
- [SEE Multisensor Design](#)
- [Application Processor Sensors Software Stack](#)
- [Registry in SEE](#)
- [SEE Log Packets and Debug Messages](#)
- [Sensor Driver Development](#)
- [Driver Components](#)
- [Integrating New Sensor Driver in SEE Framework](#)
- [Test Tools](#)
- [Debugging](#)
- [Qualcomm All-Ways Aware Hub Validation Tool](#)
- [Frequently Asked Questions](#)
- [References](#)
- [Questions?](#)



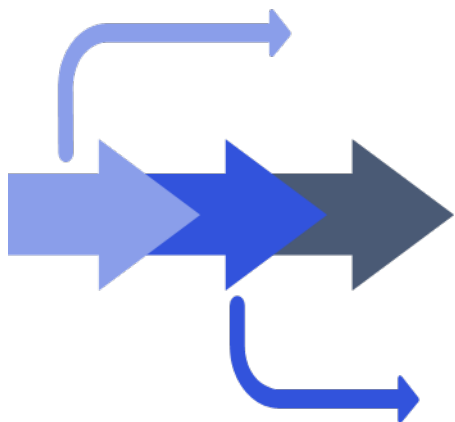
Qualcomm
2019-04-14 20:26:28 PDT
zk_sw@wingtech.com

Introduction

Overview

- This document is intended for engineers who want to understand, develop, integrate, and debug the Sensors Execution Environment (SEE) sensor management software on QTI chipsets.
- Prerequisite – It is strongly recommended that you read the relevant chipset overview document. The following table lists chipset-specific overview documents (as of February 2019):

SEE-supported chipset	Overview document
<ul style="list-style-type: none">▪ SM8150▪ SDX50+SM8150, SDX55M+SM8150P	80-PF777-23
SC8180X	80-PG470-24
SDM845	80-P9301-34
<ul style="list-style-type: none">▪ SM7150, SDM712,SDM710▪ SM6150/SM6150P, SM6125, SDM670▪ QCS605	80-PD126-9
SDW3100	80-PF839-3

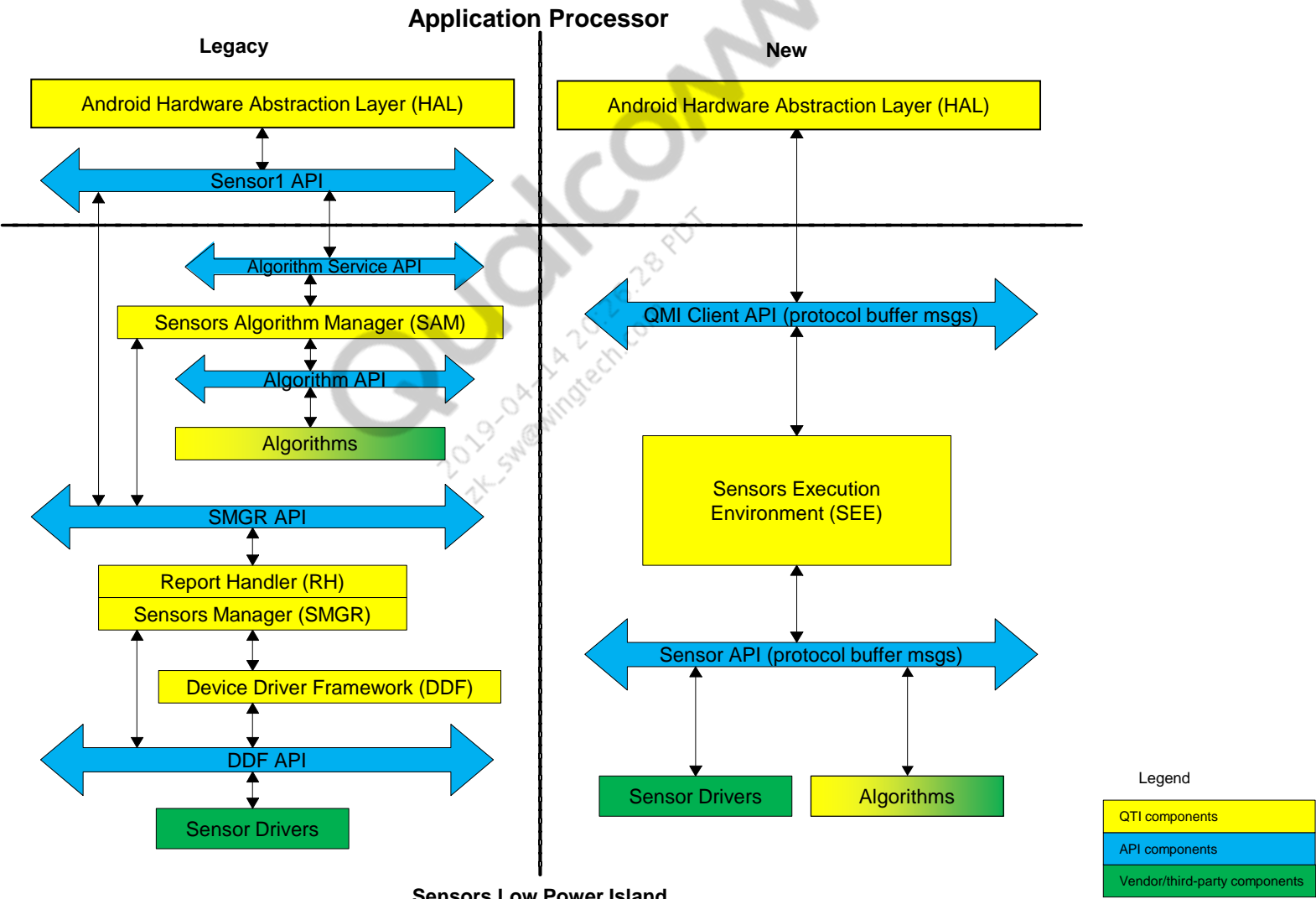


Sensors Execution Environment

SEE Definition and Motivation

- Sensors Execution Environment (SEE) – New generation of sensor management software on Qualcomm All-Ways Aware™ Hub (AAH) introduced in SDM845 in 2017 and implemented in subsequent chipsets
 - AAH is analogous to the Qualcomm® Snapdragon™ Sensors Core (SSC); AAH and SSC are used interchangeably.
- SEE provides the following benefits:
 - Unified event-driven framework for drivers and algorithms
 - Any algorithm that runs on the SEE must be signed; contact Sensors Product Management for more information.
 - Simple and symmetrical APIs (sensors and services)
 - Simple interface to init/enable/active/de-active/sample sensors
 - Driver has improved control on how it handles client requests
 - Easily extendable to add new/custom driver features
 - Better testability directly at driver API
 - Support for asynchronous COM bus transfers

SEE High-Level Design Comparison with Legacy Framework



SEE Sensor API Comparison with Device Driver Framework (DDF)

DDF	SEE sensor API
init	init
probe	
set_attr (multiple)	set_client_request
get_attr (multiple)	
run_test	
enable_sched_data	
get_attr	sns_attribute_service_api:: publish_attribute()
<i>Not supported</i>	notify_event
<i>Not supported</i>	deinit
<i>Not supported</i>	get_sensor_uid

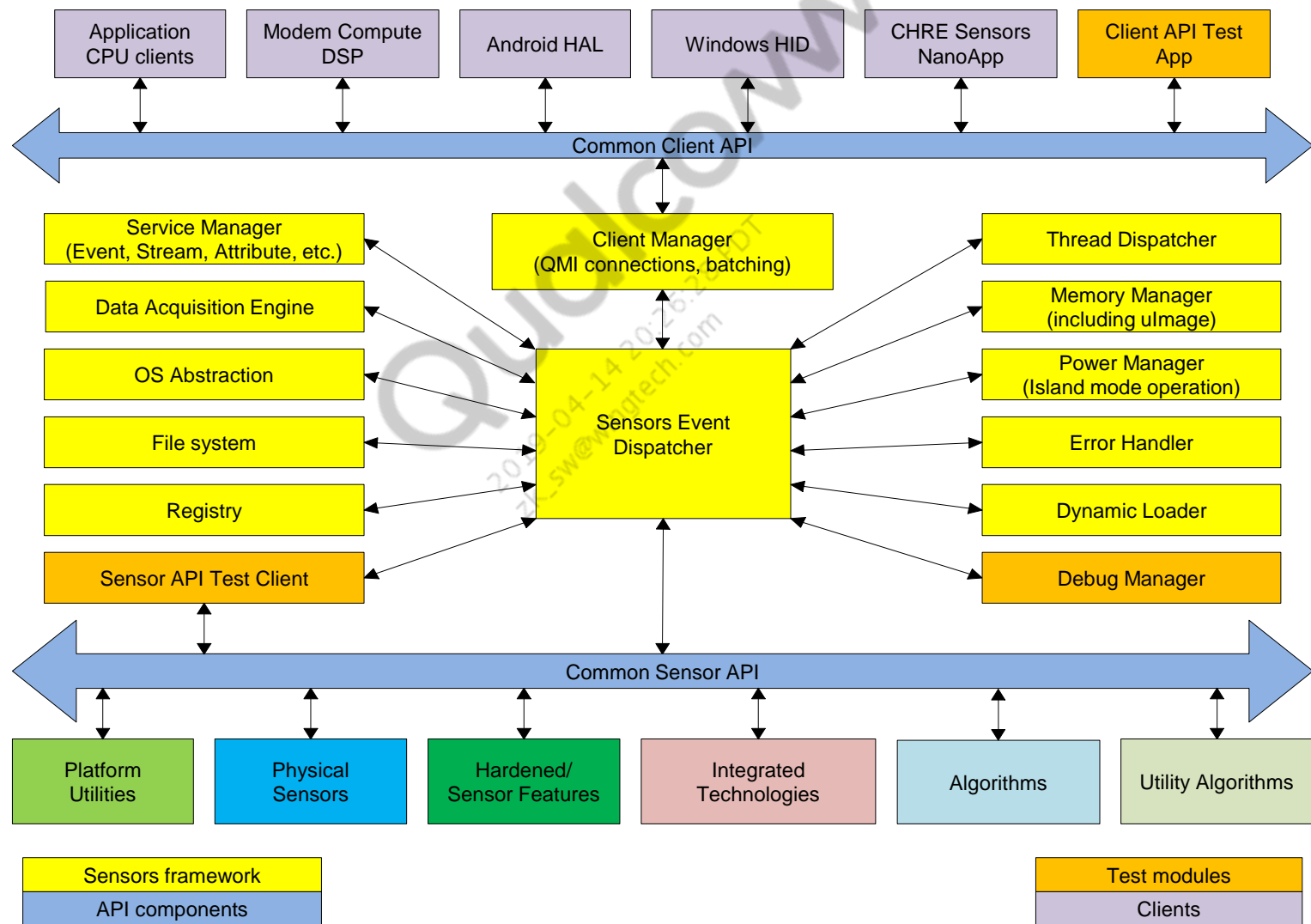
DDF	SEE sensor instance API
reset	init
get_data	notify_event
handle_timer	
handle_irq	
set_attr (multiple)	set_client_config
get_attr (multiple)	
run_test	
enable_sched_data	
trigger_fifo_data	
process_daf_req	
cancel_daf_trans	
<i>Not supported</i>	deinit

Notable Changes with SEE

Item	DDF	SEE
Coordinate system	North East Down (NED)	Android coordinate system
Sensor sample data type	Fixed point (Q16)	Floating point (single precision)
API message definition	N/A	Protocol buffers

Qualcomm
2019-04-14 20:26:28 PDT
zk_sw@wingtech.com

SEE Block Diagram



SEE High-Level Components

- API components
 - Client API – SEE interface with external (to the SSC) clients
 - Sensor API – Common sensor API for all device drivers
- Sensors framework
 - Core SEE framework that provides services to sensors, manages registry, handles client interface, manages power features, etc.
- Qualcomm Technologies, Inc. (QTI)-implemented platform sensors
 - Modules that provide asynchronous platform feature support
- Vendor implemented sensors
 - Device drivers written by sensor/third-party vendors
- Test modules
 - Test clients to test at each API component
- Clients
 - External clients to the SEE

SEE Definitions

- **Sensor**
 - A module that produces and/or consumes asynchronous data
- **Sensor instance**
 - An instantiation of a sensor that runs at a specific configuration
- **Sensor unique identifier (SUID)**
 - A unique 128-bit ID per 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 sent by a client to a sensor
 - See `sns_request.h`
- **Event**
 - Asynchronous output data message generated by a sensor instance
 - See `sns_sensor_event.h`

Communication Between Sensors

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



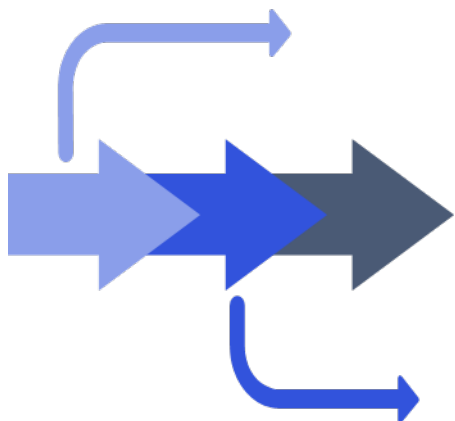
- Request messages are sent to enable, disable, and/or reconfigure a sensor.
 - Request messages are always addressed to a specific SUID.
 - Once the target sensor receives the request message, it sends the request to the sensor instance for proper handling.
- Event messages are sent asynchronously by sensor instances to their registered clients, which may be other sensors or sensor instances.

Sensor and Sensor Instance

- Sensors are producers and/or consumers of asynchronous data.
- Each sensor can be instantiated one or more times as sensor instances.
 - Each instance operates with a specific configuration.
 - Any request to a sensor for data may result in creation of a sensor instance or sharing of an existing instance.
- Sensor instances are created on-demand, as determined by the sensor.
 - Sensors fully manage the lifecycle and configuration of their corresponding instances, and are responsible for sending configuration updates and initial state events to their clients.
 - Vendors are strongly encouraged to serve all client requests with as few instances as possible.
 - A stream of data generated by an instance is sent to all active clients.
- A single sensor instance may be shared and configured by multiple sensors.
 - This mode of operation is typically for a combo driver for hardware sensors, where the sensors represent the supported data types, and the sensor instance is the sole module communicating and configuring the hardware.

Sensor and Sensor Instance (cont.)

Sensor	Sensor instance
<ul style="list-style-type: none">▪ An entity that produces a single type of data, for example, accelerometer, gyroscope, timer, interrupt, rotation vector, and so on▪ SUID – 128-bit number unique to each sensor▪ Publishes attributes (mandatory and custom)▪ Manages its instances	<ul style="list-style-type: none">▪ An active instance of a sensor that publishes output data events▪ Sensors may create an instance per client request or share an instance between multiple client requests▪ Physical sensors typically share a single instance



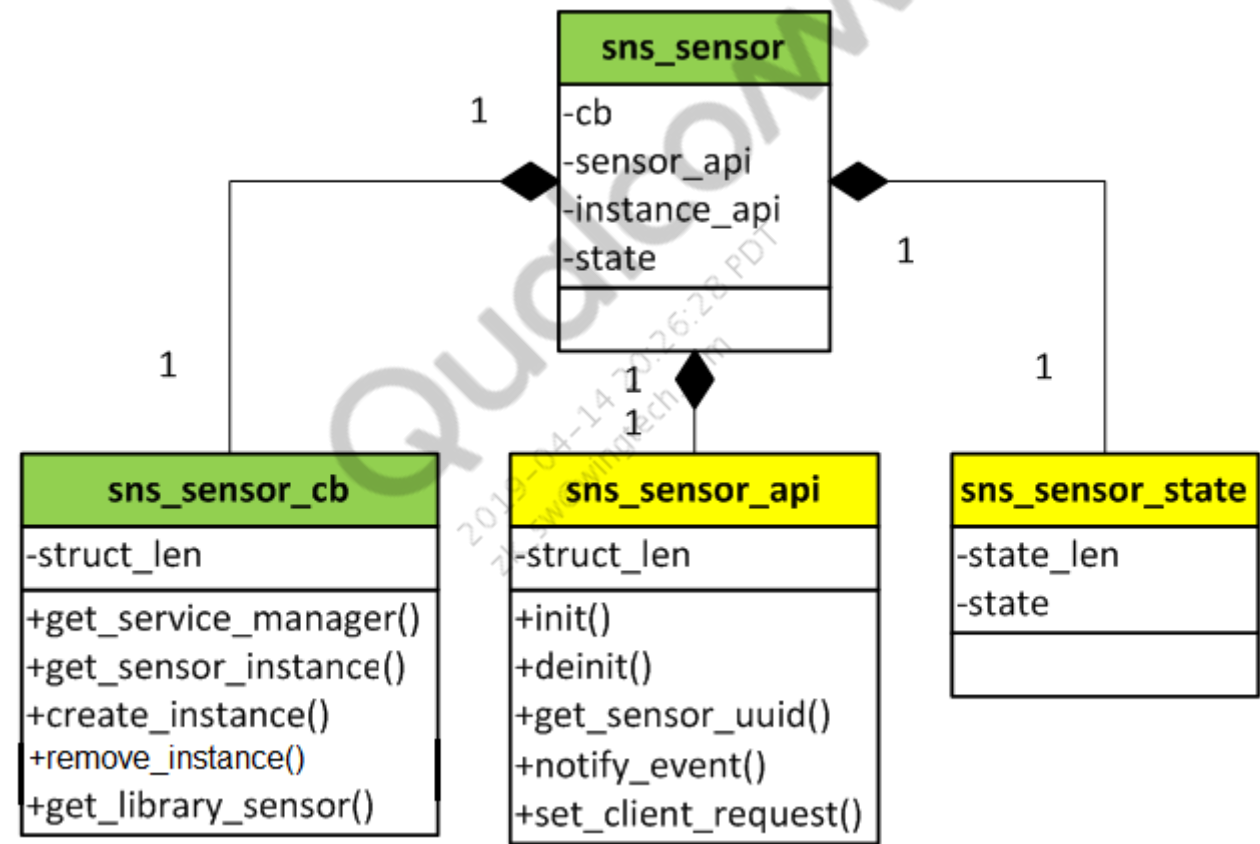
Qualcomm
2019-04-14 20:26:28 PDT
zk_sw@wingtech.com

SEE Sensor API

SEE Sensor API

- Common API
 - Physical sensors (device drivers)
 - Virtual sensors (algorithms)
- API is defined in the following files:
 - `slpi_proc\ssc\inc\sns_sensor.h`
 - `slpi_proc\ssc\inc\sns_sensor_instance.h`
 - `slpi_proc\ssc\inc\sns_register.h`

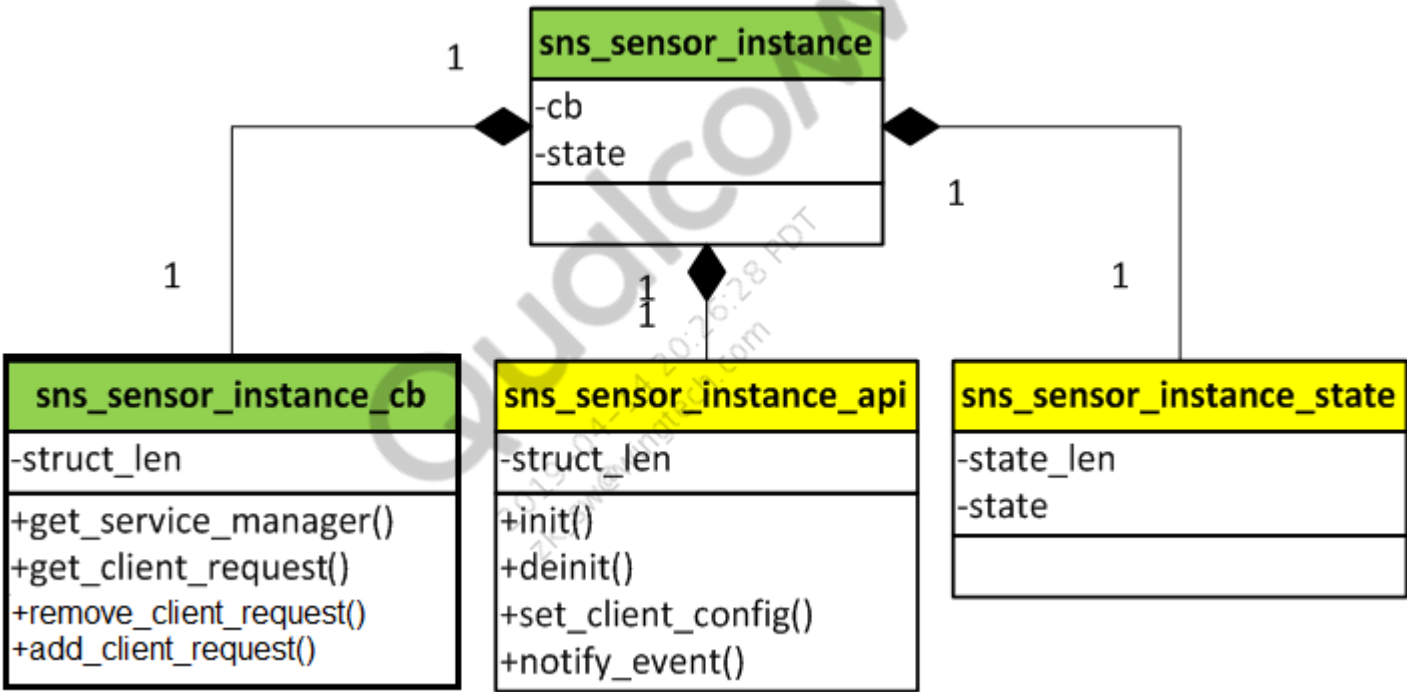
Sensor API



SEE Sensor API – sns_sensor.h

- `sns_sensor` struct defines:
 - `sns_sensor_cb` – Callbacks in the SEE framework available to a sensor
 - Gets the service manager handle (`.get_service_manager()`)
 - Gets the next instance for the sensor (`.get_sensor_instance()`)
 - Creates a new instance (`.create_instance()`)
 - Removes an existing instance (`.remove_instance()`)
 - Gets the next sensor supported in this sensor's library (`.get_library_sensor()`)
 - `sns_sensor_api` – Functions that every sensor must implement and are called only by the SEE framework
 - Initializes a sensor (`.init()`)
 - Called during the sensor registration
 - Destroys a sensor (`.deinit()`)
 - Called when the sensor reports any failure error code
 - Gets the unique sensor identifier (`.get_sensor_uid()`)
 - Notifies a sensor that an event is available from a dependent sensor (`.notify_event()`)
 - Updates a client request for a sensor (`.set_client_request()`)
 - `sns_sensor_state` – Private state maintained by a sensor
 - Memory for the sensor state is allocated by the SEE framework during the sensor registration

Sensor Instance API



SEE Sensor API – sns_sensor_instance.h

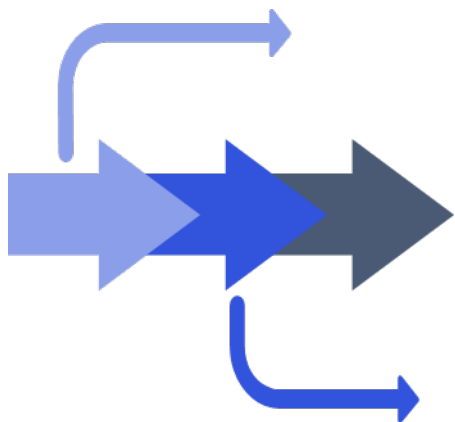
- `sns_sensor_instance` struct defines:
 - `sns_sensor_instance_cb` – Callbacks in the SEE framework available to a sensor instance
 - Gets the service manager handle (`.get_service_manager()`)
 - Gets the next client request associated with an instance (`.get_client_request()`)
 - Removes a client request handled by an instance (`.remove_client_request()`)
 - Adds a client request handled by an instance (`.add_client_request()`)
 - `sns_sensor_instance_state` – Private state maintained by a sensor instance
 - SEE framework allocates the memory for a sensor instance state during the creation of a sensor instance
- `sns_sensor_instance_api` – Functions that every sensor instance must implement and are called by its sensor or the SEE framework
 - Initializes a sensor instance (`.init()`)
 - Called by the SEE framework when an instance is created
 - Destroys a sensor instance (`.deinit()`)
 - Called by the SEE framework when its sensor requests a `remove_instance()`.
 - Sets instance configuration (`.set_client_config()`)
 - Called by the sensor to set the instance configuration
 - Notifies an instance that an event is available from a dependent sensor (`.notify_event()`)
 - Called by the SEE framework

SEE Sensor API – sns_register.h

- Each driver library implements a registration function of type `sns_register_sensors()`.
 - `sns_register_cb` struct defines:
 - Callback to initialize a sensor (`.init_sensor()`)
- This function calls `init_sensor()` on each supported sensor within the library.

Physical Sensor Driver Responsibilities

- Sensor
 - Looks for sensor hardware during initialization and publishes availability only when the hardware is present
 - Publishes all relevant attributes with the correct values
 - Gets dependent SUIDs
 - Gets configuration and calibration information from the registry
 - Manages incoming requests
 - Creates/updates/removes instances when incoming requests update
 - Manages power rails connected to the sensor hardware
 - Power rails must be ON only when there is a client request for the sensor, OFF otherwise
 - Manages COM bus power around COM transfers
 - Releases all resources during deinitialization
- Instance
 - Manages COM bus power around COM transfers
 - Programs the hardware for the requested configuration
 - Publishes a configuration event whenever the hardware configuration changes
 - Publishes data events per the API definition
 - Publishes any error events
 - Releases all resources during deinitialization



Qualcomm
2019-04-14 20:26:28 PDT
zk_sw@windtech.com

Protocol Buffers in SEE

Protocol Buffer and Nanopb

- Google protocol buffers are a data format for serializing data structures into a byte stream agnostic of language and platform.
- Data structure information is defined in the form of a protocol buffer message in a .proto file.
- The .proto file is run through a protocol buffer compiler to generate data structures that can be used programmatically.
- The protocol buffer design also provides special code to encode (serialize at the source)/decode (deserialize at the sink) data to and from byte streams.
- See <https://developers.google.com/protocol-buffers/> for detailed information about Google protocol buffers.
- Nanopb – C implementation of Google protocol buffers
 - Small code size
 - Support 32-bit processor architecture
 - See <https://jpa.kapsi.fi/nanopb/> for detailed information about nanopb

Nanopb in SEE

- SEE uses nanopb protocol buffers for:
 - All request and event messages exchanged between sensors
 - A sensor/instance must encode the payload (if present) for all requests it sends to its dependents.
 - A sensor/instance must decode the payload (if present) for all requests it receives.
 - A sensor/instance must encode the payload (if present) for all events it publishes.
 - A sensor/instance must decode the payload (if present) for all events it receives from its dependents.
 - Some requests/events may not have a message body. In this case, decoding/encoding the payload is not expected. Such messages are typically processed by their message ID.
 - Representing attribute data
 - All attribute values are in nanopb-encoded format.
 - Diag log packet payload
 - All payloads in diag log packets are in nanopb-encoded format.



Qualcomm
2019-04-14 20:26:28 PDT
zk_sw@wingtech.com

Sensor API Messages

Sensor API Messages

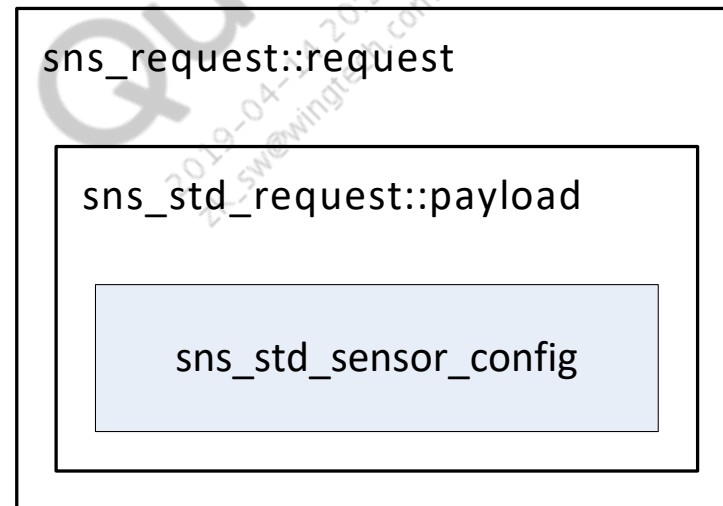
- .proto files contain the following API elements:
 - Protocol buffer message definitions
 - Documentation
- Standard message definitions are in slpi_proc\ssc\sensors\pb\sns_std_*.proto.
 - sns_std.proto contains the following standard definitions:
 - Framework-defined message ID
 - Standard request message
 - Batching spec
 - Attribute request and event
 - Error event message
 - sns_std_sensor.proto contains the following standard definitions:
 - Message IDs for request and event API for standard sensors
 - Streaming request and event messages
 - Sensor sample status types
 - Standard attribute IDs
 - Common attribute types
 - Physical sensor configuration event message

Sensor API Messages (cont.)

- Standard message definitions are in `slpi_proc\ssc\sensors\pb\sns_std_*.proto`.
 - `sns_std_type.proto` contains common API-type definitions, for example:
 - SUID message
 - Attribute event and value message
 - Common error types
 - `sns_std_event_gated_sensor.proto` contains the API for event gated sensors:
 - Config message ID
 - API documentation
- Physical sensor-specific API definitions and documentation are present in sensor-specific .proto files, for example, `sns_accel.proto`, `sns_proximity.proto`, `sns_motion_detect.proto`, etc.
- Platform sensor API definitions and documentation are present at `slpi_proc\ssc\sensors\pb\`, for example, `sns_timer.proto`, `sns_interrupt.proto`, and `sns_async_com_port.proto`.
- Framework-related APIs for SUID, registry, and diag are defined in the following:
 - `sns_suid.proto`
 - `sns_registry.proto`
 - `sns_diag.proto`

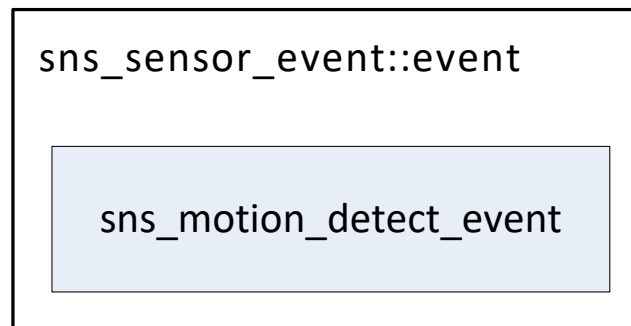
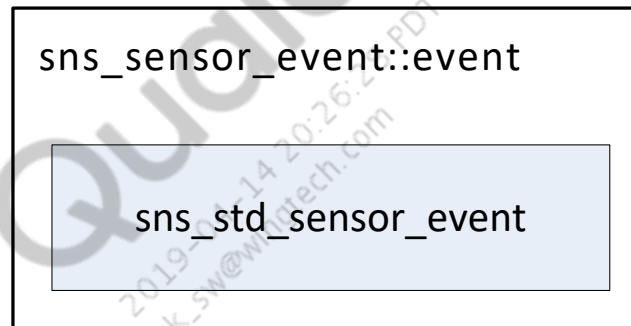
Nanopb-Encoded Request Message

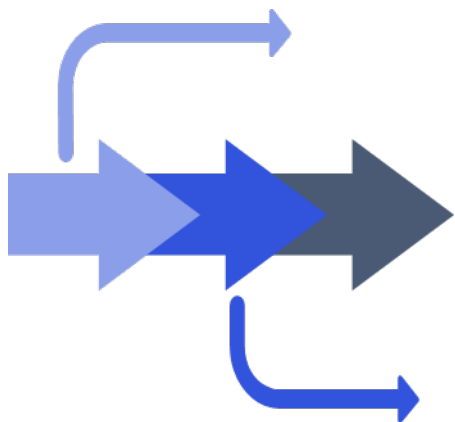
- `sns_request::request` contains the following:
 - Nanopb-encoded `sns_std_request` message
 - `sns_std_request::payload`
 - Typically contains nanopb-encoded `sns_std_sensor_config` message
 - Any other sensor-specific request message
 - May be NULL for empty request payloads



Nanopb-Encoded Event Message

- `sns_sensor_event::event` contains the following:
 - Nanopb-encoded event message; typically, the `sns_std_sensor_event` message but may be different based on the sensor. For example, for `motion_detect`, it is `sns_motion_detect_event`.





SEE Service Manager and Services

SEE Service Manager

- SEE provides synchronous services via the service manager.
- Sensor and sensor instance API have a callback to get a handle to the service manager.
- The service manager API allows getting a handle to the required service.

sns_service_manager
-struct_len
+get_service()

SEE Services

- Services available in the SEE are listed in sns_service.h.
- Key services essential for device drivers:
 - Stream service
 - Provides the ability to create and remove a data stream with a sensor.
 - All data streams are created with a SUID of the source sensors.
 - See sns_data_stream.h for data stream API to send requests and receive events over the data streams.
 - After a stream is removed, it cannot be reused to send requests; a new data stream must be created.
 - See slpi_proc\ssc\inc\services\sns_stream_service.h for API details.

sns_stream_service_api
-struct_len
+create_sensor_stream() +create_sensor_instance_stream() +remove_stream()

SEE Services (cont.)

- Key services essential for device drivers (cont.):
 - Attribute service
 - Allows a sensor to publish its attributes.
 - All standard attribute IDs and expected value type are defined in sns_std_sensor.proto.
 - All attribute values must be in nanopb-encoded format.
 - See slpi_proc\ssc\inc\services\sns_attribute_service.h for API details.
 - Diagnostic service
 - Provides debug message and data log packet services.
 - Defines standard log packet IDs.
 - See slpi_proc\ssc\inc\services\sns_diag_service.h for API details.

sns_attribute_service_api
-struct_len
+publish_attribute()

sns_diag_service_api
-struct_len
+get_max_log_size() +alloc_log() +submit_log() +sensor_printf() +sensor_inst_printf()

SEE Services (cont.)

- Key services essential for device drivers (cont.):
 - Event service
 - Provides the ability to publish output events from source sensor instances

sns_event_service_api
-struct_len
+get_max_event_size() +alloc_event() +publish_event() +publish_error()

Qualcomm

2019-04-14 20:26:28 PDT
zk_sw@wingtech.com

- Power rail service
 - Service available to physical sensors to register power rails and vote them ON/OFF

sns_pwr_rail_service_api
-struct_len
+sns_register_power_rails() +sns_vote_power_rail_update()

SEE Services (cont.)

- Key services essential for device drivers (cont.):
 - Synchronous COM port service
 - Service available to physical sensors to register/deregister COM port and perform synchronous transfers over the COM port

sns_sync_com_port_service_api
-struct_len
+sns_scp_get_version()
+sns_scp_open()
+sns_scp_close()
+sns_scp_update_bus_power()
+sns_scp_register_rw()
+sns_scp_simple_rw()
+sns_scp_get_write_time()
+sns_scp_register_com_port()
+sns_scp_deregister_com_port()



Qualcomm
2019-04-14 20:26:28 PDT
zk_sw@wingtech.com

SEE Platform Sensors

SEE Platform Sensors

- SEE provides some built-in sensors that can be used by other sensors and sensor instances.
- Typical platform feature-specific sensors
 - Registry sensor
 - Timer sensor
 - Interrupt sensor
 - Asynchronous COM port sensor
 - SUID lookup sensor
 - Test sensor

Registry Sensor

- The registry sensor provides an interface to access the registry from persistent memory.
- All sensors that require registry information must create a data stream with the registry sensor and send a request over this data stream.
- When registry data is available, data events over this data stream are delivered to the requesting sensor and must be read by the requesting sensor.
- Registry data can be accessed during boot up and at runtime.
- The registry sensor also provides the ability to subscribe to updates to requested registry data; updates are delivered as data events to the requesting sensor.
- If the sensor does not need to subscribe to changes in the registry data, it must remove the data stream with the registry sensor.
- The API for the registry sensor is documented at `slpi_proc\ssc\framework\registry\pb\sns_registry.proto`.

Timer Sensor

- The timer sensor provides an interface to start periodic or one-shot timers.
- All sensors that require timers must create a data stream with the timer sensor and send a request over this data stream.
- When the requested timer fires, data events over this data stream are delivered to the requesting sensor and must be read by the requesting sensor.
- If the sensor no longer needs the timer, it must remove the data stream with the timer sensor.
- The API for the timer sensor is documented at `slpi_proc\ssc\sensors\pb\sns_timer.proto`.

Interrupt Sensor

- The interrupt sensor provides an interface to register for interrupts.
- All sensors that require interrupts must create a data stream with the interrupt sensor and send a request over this data stream.
- When the interrupt trigger condition on the requested pin is detected, data events over this data stream are delivered to the requesting sensor and must be read by the requesting sensor.
- If the sensor no longer needs the interrupts, it must remove the data stream with the interrupt sensor.
- The API for the interrupt sensor is documented at `slpi_proc\ssc\sensors\pb\sns_interrupt.proto`.

Asynchronous COM Port (ASCP) Sensor

- The ASCP sensor provides an interface to read/write over a COM port asynchronously.
- All sensors that require this feature must create a data stream with the ASCP sensor and send a request over this data stream.
- When the requested data transfer over the port is complete, data events over this data stream are delivered to the requesting sensor and must be read by the requesting sensor.
- If the sensor no longer needs ASCP, it must remove the data stream with the ASCP sensor.
- The API for the ASCP sensor is documented at `slpi_proc\ssc\sensors\pb\sns_async_com_port.proto`.

Note: The ASCP sensor is typically used by physical sensor drivers for reading large FIFO.

SUID Lookup Sensor

- This framework sensor provides an API for all other sensors to get the SUID for their dependent sensors.
- The request to the SUID lookup sensor contains a `data_type` field of the dependent sensor. See sensor-specific proto files at `slpi_proc\ssc\sensors\pb` for the exact string of `data_type` per sensor.
- Any data stream with dependent sensors can be created only after receiving a non-zero SUID event from the SUID lookup sensor.
- The SUID of the SUID lookup sensor itself is available at `sns_get_suid_lookup()` util in `sns_sensor_util.h`.
- See the SUID sensor API at `slpi_proc\ssc\framework\suid_sensor\pb\sns_suid.proto`.
- Refer to the example accelerometer and gyroscope drivers shipped in the package on how to use the SUID lookup sensor.

Test Sensor

- A test sensor is available in source code for developers to customize and run sensor-specific use cases.
 - Standard sensor streaming test – `slpi_proc\ssc\sensors\test`

Qualcomm
2019-04-14 20:26:28 PDT
zk_sw@wingtech.com

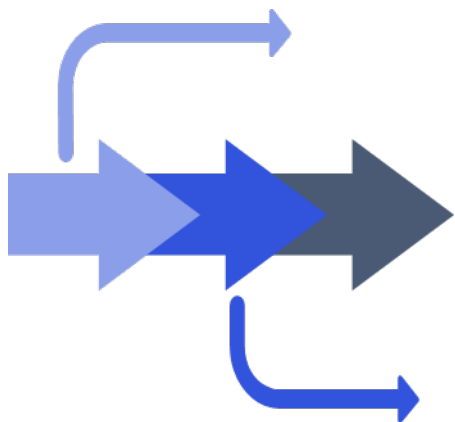


Qualcomm
2019-04-14 20:26:28 PDT
zk_sw@wingtech.com

SEE Utilities

SEE Utilities

- Along with synchronous services, the SEE also provides several helper utilities for sensors and sensor instances.
- See `slpi_proc\inc\utils` for all available utilities.
- Key utilities include:
 - Nanopb encode/decode
 - Provides common encode/decode helper functions for all sensors, for example, encode/decode `sns_request` messages, encode and publish/decode data events, and so on
 - Asynchronous COM port nanopb utilities are available for physical sensor drivers
 - Sensor utils
 - Provides common functionality, for example, finding a sensor instance, getting a SUID of a SUID lookup sensor, and so on
 - Attribute utils
 - Provides helper function that encodes and publishes a sensor attribute
 - Memory, math, printf utils



SEE Sensor Heartbeat Feature

Sensor Heartbeat Feature

1. All physical sensor device drivers must implement logic that determines whether the physical sensor has stopped sampling.
 - Polling sensors – Use a multiple of the polling schedule as the heartbeat timer. Start with five times for nonFIFO and two times for FIFO, based on the polling schedule. In theory, a single timer should suffice both polling and heartbeat functionality.
 - Example: Pressure driver polling at 20 Hz (50 ms sample period/polling timer) determines every 250 ms (five times sample period/polling timer) if the sensor is still producing output.
 - Example: Magnetometer FIFO driver streaming in S4S mode at 100 Hz sample rate (10 ms sample period), watermark (WM) = 20 (200 ms S4S polling timer) determines every 400 ms (two times polling timer) if the sensor is still producing output.
 - Interrupt sensors – Driver must implement a heartbeat timer for this purpose. Heartbeat timer timeout value can be a multiple of the report rate chosen by the driver. Start with two times the chosen report rate.
 - Example: Accelerometer FIFO driver streaming at 13 Hz sample rate (76.9 ms sample period), WM = 100 (7690 ms report rate) determines every 15380 ms (two times report rate) if the sensor is still producing output.

Sensor Heartbeat Feature (cont.)

2. At each heartbeat timeout, if the driver determines that the sensor has stopped sampling (driver could use a combination of current time, previous sample time, previous interrupt time, sample rate, report rate, WM, and so on, to determine a stream stall condition) then the driver must:
 - Perform a reset (hardware reset, software reset, or any other operation that the driver deems fit) operation in an attempt to revive the sensor.
 - Return an `sns_rc` error `SNS_RC_NOT_AVAILABLE` error to indicate that it is attempting to revive the sensor. The SEE framework sends an `SNS_STD_ERROR_NOT_AVAILABLE` error event to all registered clients to this sensor to help clients know that there maybe some sample data gaps due to the reset process.
3. The driver must attempt Step 2 for a finite number of times.
 - a. Start with three revive attempts.
 - b. If sensor streaming is unable to resume after three attempts, the driver must return an `sns_rc` error code `SNS_RC_INVALID_STATE` error code to the SEE framework.
 - c. Upon receiving this error code, the framework deletes the driver instance. The framework also sends an `SNS_STD_ERROR_INVALID_STATE` error event to all registered clients of this sensor indicating that the instance is no longer usable.



Qualcomm
2019-04-14 20:26:28 PDT
zk_sw@windtech.com

SEE Multisensor Design

Multisensor Design

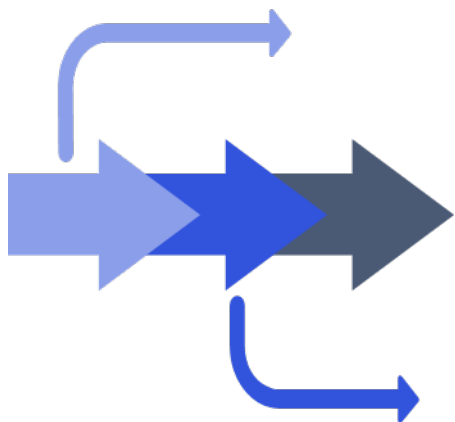
- A single driver library supports multiple instances of the same hardware.
- The expanded scons method `AddSSCSU()` supports multiple registrations of the same driver library. Each registration belongs to a unique hardware instance of the physical sensor.
- The expanded SEE framework provides a callback to indicate which of the sensors/instance belongs to which registration/hardware sensor.

Driver Support

- A single driver is registered N times where N is the number of hardware instances present on the platform. The driver scons file must be updated to use the `AddSSCSU()` method to add N registrations.
- Each driver registration is treated as a separate library within the SEE, for example:
 - LSM6DSM registration for `hw_id=0` is considered one library consisting of accelerometer, gyroscope, sensor temperature, and motion detect sensors.
 - LSM6DSM registration for `hw_id=1` is considered another library consisting of another set of accelerometer, gyroscope, sensor temperature, and motion detect sensors.
- Each set of sensors belonging to a single registration/`hw_id` share a common instance for all streaming, self-test, and custom operations.
- All existing SEE requirements for physical sensor drivers apply to each registration of the driver for multiple sensors.
- Registry configuration contains a platform-specific and driver-specific registry for all N hardware instances of the sensor, for example:
 - `lsm6dsm_0.json`
 - `lsm6dsm_1.json`
 - `sdm845_lsm6dsm_0.json`
 - `sdm845_lsm6dsm_1.json`

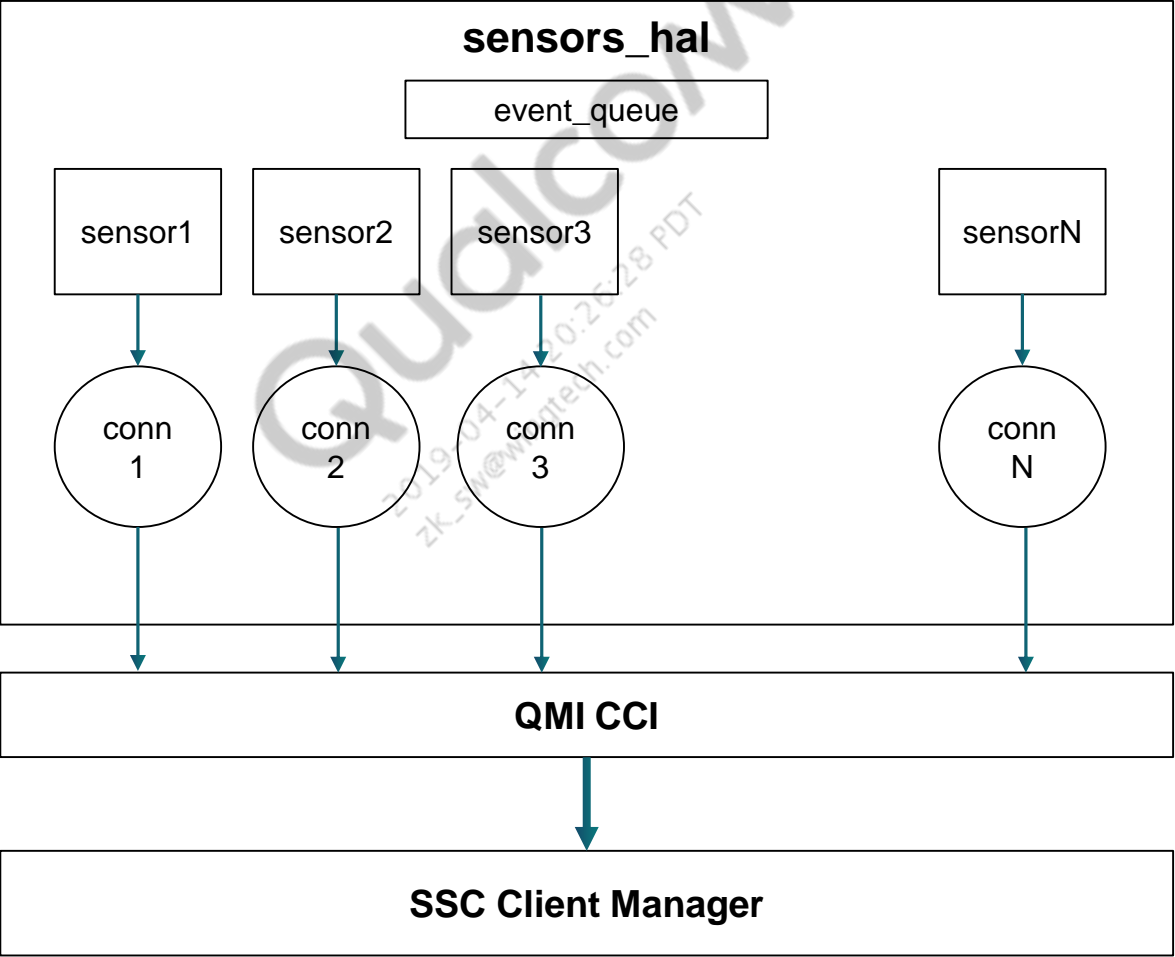
Driver Support (cont.)

- The driver uses the framework callback to obtain its hw_id/registration information to:
 - Publish unique SUIDs for all sensors per hardware instance, for example:
 - LSM6DSM driver publishes unique SUIDs for accel/gyro/sensor_temperature/motion_detect datatypes for hw_id/registration=0 and hw_id/registration=1.
 - Request registry per unique hardware instance, for example:
 - LSM6DSM driver requests for registry "lsm6dsm_0_platform.config" for hw_id=0 and "lsm6dsm_1_platform.config" for hw_id=1.
 - Publish correct attributes for all sensors per hardware instance, in particular, placement of specific attributes, for example:
 - LSM6DSM driver publishes hw_id=0 for all sensors that belong to the first registration and hw_id=1 for the second registration.
- Drivers do not use any mutable global variables; using static lookup tables (for example, ODR match) is acceptable.
- Drivers do not use any local static variables.
- If all sensor and sensor instance API implementations in the driver are reentrant and thread safe, no other driver changes are required.



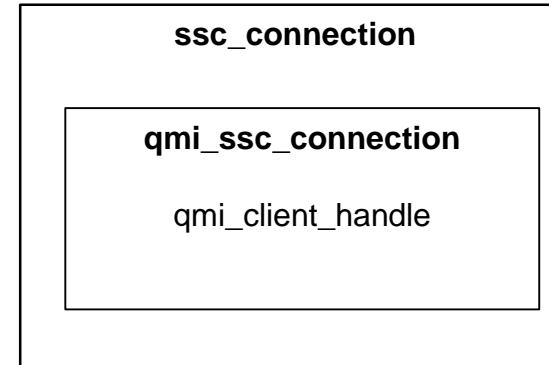
Application Processor Sensors Software Stack

Sensors HAL Overview



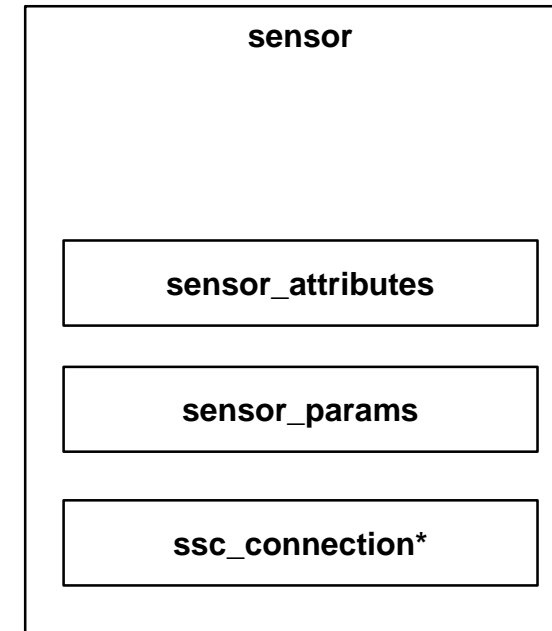
Class ssc_connection

- Manages a connection to the SSC
- Public operations
 - `ssc_connection(event_cb)`
 - Creates a new connection to SSC
 - Registers a callback
 - `send_request(client_req_msg)`
 - Sends a request message to the sensors service
 - `~ssc_connection()`
 - Disconnects from the SSC
 - Deregisters the callback



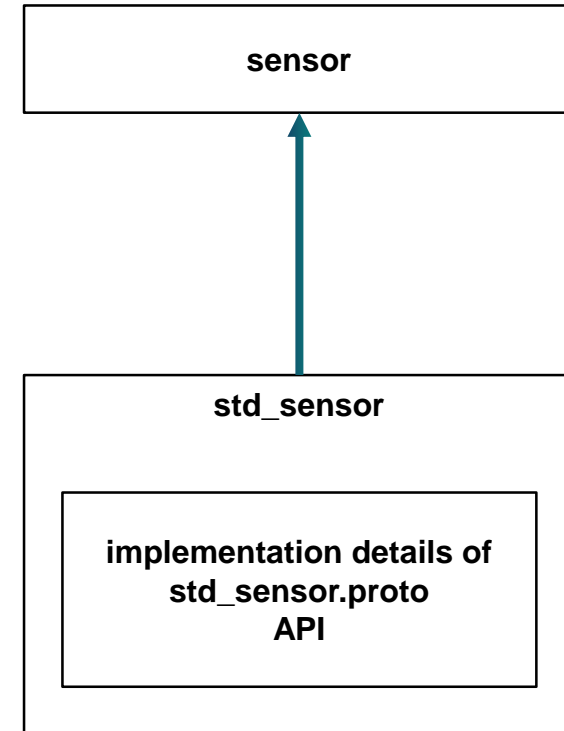
Class sensor

- Manages a single physical/virtual sensor in the SSC
- This class is abstract; specific implementations must derive from it
- Public operations
 - `sensor(datatype, suid, event_cb)`
 - Creates a new sensor object
 - Retrieves and saves attributes for later use
 - Registers a HAL event callback
 - `configure(params)`
 - Sets timing parameters
 - `sample_period` and `latency`
 - `activate()`
 - Creates a connection to SSC
 - Sends a config request
 - `deactivate()`
 - Destroys a connection to the SSC
 - `flush()`
 - Flushes a sensor FIFO



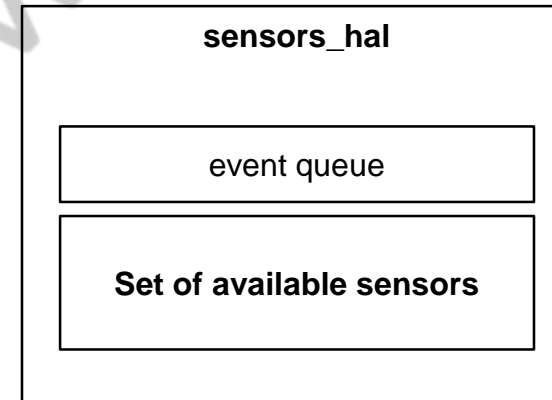
Sensor Derivatives

- Based on the API, the sensor_factory creates concrete objects of these classes.
- These classes provide no extra public operations other than base sensor operations.
- Most sensors are expected to implement the std_sensor API; therefore, they can be enabled without adding a new class.
- Each nonstandard API must have a separate derivative of a class sensor, for example:
 - SENSOR_API_STD
class std_sensor
 - SENSOR_API_GRAVITY
class gravity_sensor
 - SENSOR_API_UNICORN_DETECTOR
class unicorn_detector_sensor



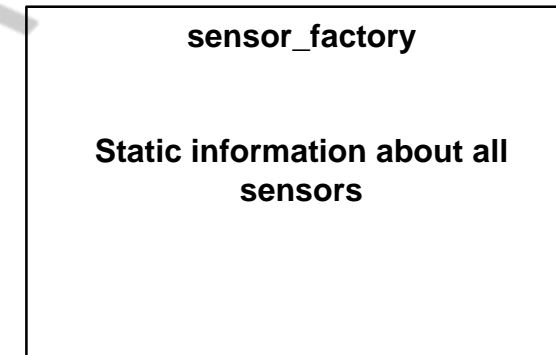
Class sensors_hal

- Public operations
 - sensors_hal()
 - Initializes the HAL
 - Queries available sensors from the SSC
 - Creates sensor objects
 - Prepares a list of available sensors
 - get_sensors_list()
 - Gets a list of available sensors
 - batch(handle)
 - Sets sample_period and latency
 - activate(handle, en)
 - Activates/deactivates a sensor
 - poll(*events, count)
 - Returns up to count number of events
 - Blocks if no events are available
 - flush(handle)
 - Flushes the sensor FIFO



Class `sensor_factory`

- Manages creation of sensor objects and provides static information
- Creates the correct type of sensor given its data type
- Public operations
 - `static create_sensor(datatype, suid, cb)`
 - Creates a new sensor object based on the API represented by the data type
 - `static get_sensor_info(datatype)`
 - Provides static information about a sensor data type
 - `sensor_type`, API, and so on
 - `static get_next_handle()`
 - Returns a unique handle to be used for a new sensor



Operation Sequence

1. Android calls `sensors_open()`.
2. A `sensors_hal` object is created.
 - a. The `sensors_hal` sends lookup requests to the SSC about all known data types.
 - b. The sensor object is created when an SUID is available for a data type.
 - i. Sensor attributes are queried and stored in sensor object.
 - ii. A `sensor_t` object is populated using the attributes.
 - c. A list of available sensors is populated in `sensors_hal`.
3. Android calls `get_sensors_list()`.
 - a. A list of available sensors is returned.
4. Android calls `batch()`.
 - a. The `sensors_hal` finds a sensor corresponding to the handle.
 - b. This sensor is configured with new parameters.
 - i. If sensor is active, a new `config_request` is sent.

Operation Sequence (cont.)

5. Android calls `activate()`.
 - a. The `sensors_hal` finds a sensor corresponding to the handle.
 - b. If this sensor is activated, a new `ssc_connection()` is made and `config_request` is sent.
 - c. If this sensor is deactivated, a `ssc_connection` is closed.
6. Android calls `poll()`.
 - a. A `sensors_hal` returns available events.
7. Android calls `flush()`.
 - a. A `sensors_hal` finds a sensor corresponding to the handle.
 - b. Calls `sensor.flush()`.
 - i. The sensor sends a flush request to the SSC.

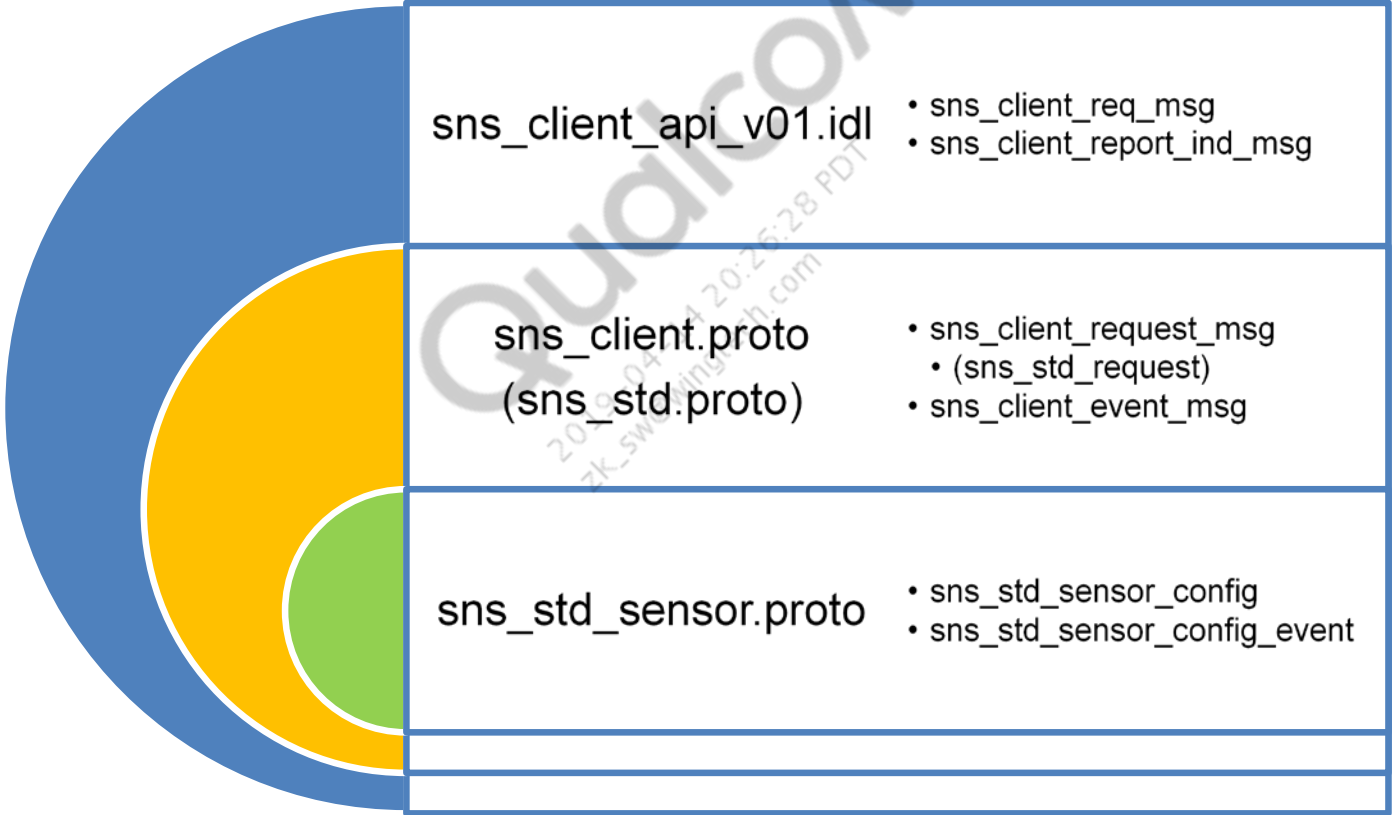
Utilities

- Class `suid_lookup_helper`
 - Creates a connection to SSC with a special SUID for lookup
 - Provides an API for looking up available SUIDs for a data type
 - `sensors_hal` uses this class to discover available sensors
- Class `sensor_attributes`
 - Provides an API for storage and parsing of sensor attribute values
 - Each sensor object has an instance of this class
- Class `concurrent_queue`
 - Provides a thread-safe implementation of a blocking queue
 - `pop()` blocks if no items are available on queue

Sensors Client API

- Sensor clients access the SEE, which resides on the SSC, via the sensors Qualcomm Messaging Interface (QMI) using client APIs.
- For details, see *Sensors Execution Environment Client API Reference* (80-P9301-36).
- Sensor clients must first open a QCCI connection to send requests to the SEE.
 - QCCI/QMI Common Service Interface (QCSI) supports only the transport of IDL-defined messages.
 - `sns_client_api_v01.idl` file is defined for this purpose.
- Sensors QMI client manager resides on the SSC and handles all QMI communication.
 - It is responsible for translating incoming QMI requests into request messages understood by the SEE and translating event messages received from the SEE into outgoing QMI indications.
 - For additional information, see the `sns_client.proto` file.
- The sensor-specific request or event is referred to in `sns_client_request_msg::sns_std_request::payload` and `sns_client_event_msg::sns_client_event::payload`.
 - These 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 in which they communicate. Each sensor type has a corresponding `.proto` file. For example, `sns_accel.proto` describes how to enable an accelerometer stream. In addition, every sensor publishes its list of `.proto` files (see Chapter 6 in 80-P9301-36).

Sensors Client API (cont.)





Qualcomm
2019-04-14 20:26:28 PDT
zk_sw@wingtech.com

Registry in SEE

Registry in SEE

- Registry is persistent storage available to all sensors to store information between reboots.
- It is created/updated from configuration files present on the file system during bootup or created/updated at runtime when requested by sensors.
- Location of registry (json files):
 - Android O: /persist/sensors/registry/config
 - Android P: /vendor/etc/sensors/config
- Location of parsed configuration files:
 - Android O: /persist/sensors/registry/registry
 - Android P: /mnt/vendor/persist/sensors/registry/registry
 - These files are generated from json files on first bootup (or updated subsequently if there are any updates to any of the json files).
 - Calibration biases are saved here as well during self-test execution and by calibration algorithms (QGyroCal and QMagCal).
- Registry contains numerous groups, and each group contains zero or more items, zero or more subgroups.
- Each item consists of item name, version, and data.
- Item data is updated only when the version of a new configuration is greater than the version of the item in the registry.
- Data item and group names may consist only of a-z, 0-9, _ characters.
- Registry and configuration data format – JSON
- Driver registry owner – Driver library
- All drivers access the registry through the registry sensor.

Driver Registry in SEE

- Typically, there are two JSON files per physical sensor; one contains all platform-specific configuration and the other contains driver-specific configuration.
- Two types of JSON files are in this directory.
 - Platform-specific configuration for the driver
 - Filename format – <target>_<sensor_name>_<hardware_id>
 - Example – /vendor/etc/sensors/config/<chipset>_lsm6dso_0.json
 - If multiple platforms have identical configurations per target, drop the <platform> field.
 - The filename format only identifies the correct file for a target, which helps developers update the correct file easily. The filename is not used by the registry sensor nor the driver.
 - Use a file naming convention to make identification easier; for example, different platform information for two different models:
 - Chipset X-based device 1 – chipsetX_lsm6dso_0_device1.json
 - Chipset X-based device 2 – chipsetX_lsm6dso_0_device2.json
 - Driver-specific configuration for the driver
 - Filename format – <sensor_name>_<hardware_id>
 - Example – /vendor/etc/sensors/config/lsm6dso_0.json
- If the configuration file does not exist and the sensor driver supports default values, the sensor library populates default values in the registry.
- The driver can update its registry at runtime by sending write requests to the registry sensor.

Driver Registry in SEE (cont.)

- Platform-specific configuration for the driver
 - Registry groups/items contain sensor hardware and platform configuration.
 - All platform-specific configuration files contain a “config” group on the top of the JSON file.
 - After the config group, all platform configuration for the driver is enclosed within a top-level group called <sensor_name>_<hardware_id>_platform.
 - Registry items marked mandatory must be populated in the sensor’s configuration file. The items marked optional may be absent in the sensor’s configuration file if not applicable to the sensor.
 - The configuration file may contain data_type specific registry subgroups (for example, accelerometer) that contains data type-specific platform configuration such as factory calibration parameters.

Driver Platform Configuration File Example

- /vendor/etc/sensors/config/<target>_lsm6dsm_0.json

```
{
  "config":{
    "hw_platform": ["MTP"],
    "soc_id": ["321"]
  },
  "lsm6dsm_0_platform":{
    "owner": "lsm6dsm",
    ".config":{
      "owner": "lsm6dsm",
      "bus_type":{ "type": "int", "ver": "0",
        "data": "1"
      },
      "bus_instance":{ "type": "int", "ver": "0",
        "data": "2"
      },
      "slave_config":{ "type": "int", "ver": "0",
        "data": "0"
      },
      .
      .
    }
  }
}
```

Driver Registry in SEE

- Driver-specific configuration
 - Registry groups/items contain driver-specific configuration.
 - All driver-specific configuration files contain a “config” group on the top of the JSON file.
 - All configuration is enclosed within a top-level <sensor_name>_<hardware_id> registry group.
 - The configuration file contains data_type specific registry groups (for example, accelerometer) that contain data type-specific configuration.
- Custom registry
 - All sensors can add custom registry groups/items in the driver-specific configuration file for any custom requirements for persistent data for sensor/algo operation.
 - These items must be added in a registry group <sensor_name>_<hardware_id>_custom.

Driver-Specific Configuration File Example

- /vendor/etc/sensors/config/lsm6dsm_0.json

```
{
  "config":
  {
    "hw_platform": ["MTP", "Dragon", "Surf"],
    "soc_id": ["291", "246", "305", "321"]
  },
  "lsm6dsm_0":{
    "owner": "lsm6dsm",
    ".accel":{
      "owner": "lsm6dsm",
      ".config":{
        "owner": "lsm6dsm",
        "is_dri":{ "type": "int", "ver": "0",
          "data": "1"
        },
        "hw_id":{ "type": "int", "ver": "0",
          "data": "0"
        },
        "res_idx":{ "type": "int", "ver": "0",
          "data": "2"
        },
        "sync_stream":{ "type": "int", "ver": "0",
          "data": "0"
        }
      }
    },
    ".gyro":{
      "owner": "lsm6dsm",
      ...
    }
  }
}
```

Driver Platform-Specific Configuration

Registry group/ item name	Registry item type	Mandatory/ Optional	Notes
"bus_type"	int	Mandatory	This item identifies to which communication bus the sensor is connected. Possible values are from sns_bus_type in sns_com_port_types.h. <ul style="list-style-type: none">▪ I²C: 0 (SNS_BUS_I2C)▪ SPI: 1 (SNS_BUS_SPI)▪ UART: 2 (SNS_BUS_UART)▪ I³C: 3 (SNS_BUS_I3C_SDR)
"bus_instance"	int	Mandatory	This item identifies the platform bus instance for the communication bus. This depends on QUP number for every chipset; refer to the chipset overview document for details.
"slave_config"	int	Mandatory	This item identifies the slave on the communication bus. <ul style="list-style-type: none">▪ For I²C/I³C, this is the slave/static address.▪ For SPI, this is the chip select line for the slave. Typically this is zero as most chipsets only support one chip-select.
"min_bus_speed_khz"	int	Mandatory	This item identifies the minimum COM bus clock speed in kHz.
"max_bus_speed_khz"	int	Mandatory	This item identifies the maximum COM bus clock speed in kHz.
"reg_addr_type"	int	Mandatory	This item identifies the register address type support by the sensor. See sns_com_port_types.h. <ul style="list-style-type: none">▪ 8-bit: 0 (SNS_REG_ADDR_8_BIT)▪ 16-bit: 1 (SNS_REG_ADDR_16_BIT)▪ 32-bit: 2 (SNS_REG_ADDR_32_BIT)
"dri_irq_num"	int	Optional (Required for interrupt)	If the sensor uses an interrupt pin, this item identifies the interrupt pin connected to the sensor, which is the MSM GPIO number where the sensor interrupt is connected. Polling sensors may not use this item.

Driver Platform-Specific Configuration (cont.)

Registry group/ item name	Registry item type	Mandatory/ Optional	Notes
"irq_pull_type"	int	Optional (Required for interrupt- based sensors)	<p>If the sensor supports DRI modes, this item identifies the GPIO pull configuration of the dri_irq_num pin (active configuration). Valid values are listed below from sns_interrupt_pull_type in sns_interrupt.proto. Polling sensors may not use this item.</p> <ul style="list-style-type: none"> • No pull – 0 (SNS_INTERRUPT_PULL_TYPE_NO_PULL) • Pull Down – 1 (SNS_INTERRUPT_PULL_TYPE_PULL_DOWN) • Keeper – 2 (SNS_INTERRUPT_PULL_TYPE_KEEPER) • Pull-up – 3 (SNS_INTERRUPT_PULL_TYPE_PULL_UP)
"irq_is_chip_pin"	int		<p>If a sensor uses dri_irq_num, set this to "1" for sensors with DRI interrupt support (this indicates the MSM GPIO is used for interrupt)</p>
"irq_drive_strength"	int		<p>If a sensor uses dri_irq_num, this item identifies the drive strength configuration for the pin. Valid values for sns_interrupt_drive_strength from sns_interrupt.proto:</p> <ul style="list-style-type: none"> • 0 – 2 mA (SNS_INTERRUPT_DRIVE_STRENGTH_2_MILLI_AMP) • 1 – 4 mA (SNS_INTERRUPT_DRIVE_STRENGTH_4_MILLI_AMP) • 2 – 6 mA (SNS_INTERRUPT_DRIVE_STRENGTH_6_MILLI_AMP) • 3 – 8 mA (SNS_INTERRUPT_DRIVE_STRENGTH_8_MILLI_AMP) • 4 – 10 mA (SNS_INTERRUPT_DRIVE_STRENGTH_10_MILLI_AMP) • 5 – 12 mA (SNS_INTERRUPT_DRIVE_STRENGTH_12_MILLI_AMP) • 6 – 14 mA (SNS_INTERRUPT_DRIVE_STRENGTH_14_MILLI_AMP) • 7 – 16 mA (SNS_INTERRUPT_DRIVE_STRENGTH_16_MILLI_AMP)
"irq_trigger_type"	int		<p>If a sensor uses dri_irq_num, this item identifies the irq trigger type. See sns_interrupt.proto.</p> <ul style="list-style-type: none"> • Rising edge – 0 (SNS_INTERRUPT_TRIGGER_TYPE_RISING) • Falling edge – 1 (SNS_INTERRUPT_TRIGGER_TYPE_FALLING) • Rising & Falling edge – 2 (SNS_INTERRUPT_TRIGGER_TYPE_DUAL_EDGE) • Level triggered: High – 3 (SNS_INTERRUPT_TRIGGER_TYPE_HIGH) • Level triggered: Low – 4 (SNS_INTERRUPT_TRIGGER_TYPE_LOW)

Driver Platform-Specific Configuration (cont.)

Registry group/ item name	Registry item type	Mandatory/ Optional	Notes
"num_rail"	int	Mandatory	This item provides the number of power rails connected to the sensor; it includes VDD and VDDIO rails. Example: If a sensor has the same VDD and VDDIO, set it to 1. This case is for most sensors with QTI reference designs except for ALS/proximity.
"rail_on_state"	int	Mandatory	This item identifies the ON state (LPM or NPM) of the power rail. A valid value is an enum value from sns_power_rail_state in sns_pwr_rail_service.h. <ul style="list-style-type: none"> ▪ Low Power mode – 1 (SNS_RAIL_ON_LPM) – Must be used as an ON state only by accelerometer sensor drivers ▪ Normal Power mode – 2 (SNS_RAIL_ON_NPM) – ON state used by all other sensors except accelerometer
"vddio_rail"	string	Optional	If the physical sensor is connected to the VDDIO (1.8 V typically) rail, this item identifies the VDDIO rail. By default, this is set to /pmic/client/sensor_vddio, which maps to a particular power rail on an individual chipset. Refer to the overview document for a particular chipset for more details. If any custom power rails are used different from rails used on QTI reference design on that chipset, contact the QTI Sensors, PMIC, and Hardware teams by filing cases in Salesforce as this needs changes in PMIC software and review from PMIC and Hardware teams.
"vdd_rail"	string	Optional	If the physical sensor is connected to the VDD (3.0 V rail typically) rail, this item identifies the VDD rail. Most sensors use a single rail; in that case, "vdd_rail" does not need to be specified since "vddio_rail" would be populated already. For ALS/proximity, set it to /pmic/client/sensor_vdd, and refer to the overview document for a particular chipset for more details. If any custom power rails are used different from rails used on QTI reference design on that chipset, contact the QTI Sensors, PMIC, and Hardware teams by filing cases in Salesforce as this needs changes in PMIC software and review from PMIC and Hardware teams.

Driver Platform-Specific Configuration (cont.)

Registry group/ item name	Registry item type	Mandatory/ Optional	Notes
"rigid_body_type"	int	Mandatory	<p>This item provides rigid body information with regard to the sensor's placement. A valid value is from sns_std_sensor_rigid_body_type in sns_std_sensor.proto.</p> <ul style="list-style-type: none"> For a sensor mounted on the same rigid body as the display, set it to 0. (SNS_STD_SENSOR_RIGID_BODY_TYPE_DISPLAY) For a sensor mounted on the same rigid body as a keyboard, set it to 1. (SNS_STD_SENSOR_RIGID_BODY_TYPE_KEYBOARD) For sensor mounted on an external device, set it to 2. (SNS_STD_SENSOR_RIGID_BODY_TYPE_EXTERNAL)
".placement"	Registry group	Mandatory	<p>This item is a registry group. It contains 12 registry items of type float that are interpreted as (float[12]): Location and orientation of sensor element in the device frame. See Android definition AINFO_SENSOR_PLACEMENT in sensors/1.0/types.h.</p> <p>These items are:</p> <p>"0"</p> <p>.</p> <p>.</p> <p>"11"</p>
".orient"	Registry group	Optional	<p>This item is a registry group. It contains three items of type string each.</p> <p>"x"</p> <p>"y"</p> <p>"z"</p> <p>This registry group is applicable for inertial sensors.</p>
".fac_cal"	Registry group	Optional	<p>This item is a registry group that is placed within a data type-specific registry group.</p> <p>Example: ".gyro" group contains ".fac_cal" group with gyroscope factory calibration parameters.</p> <p>Items within this group are of type float. See .fac_cal for format of these items per data_type.</p>

Driver Platform-Specific Configuration (cont.)

Registry group/ item name	Registry item type	Mandatory/ Optional	Notes
"min_odr"	int	Optional	This registry item can be used by a physical sensor driver to define a minimum ODR that needs to be supported. For targets supporting DAE, such as SM8150, min_odr needs to be 20. This provides flexibility to define the minimum ODR without changing the driver code.
"max_odr"	int	Optional	This registry item can be used by a physical sensor driver to define a maximum ODR that needs to be supported. This provides flexibility to define the maximum ODR without changing the driver code.

Driver-Specific Configuration

Registry group/ item name	Registry item type	Mandatory/ Optional	Notes
“is_dri”	int	Mandatory	This item identifies whether the sensor stream is interrupt based (data ready, watermark, motion, and so on) or polling. <ul style="list-style-type: none">▪ DRI sensors use value 1.▪ Polling sensors use value 0.
“hw_id”	int	Mandatory	This item is the unique identifier for the sensor hardware. It is typically used to differentiate between multiple sensors of the same hardware.
“res_idx”	int	Mandatory	Physical sensors typically have multiple supported resolutions (and corresponding ranges). Sensors publish an array of supported resolutions. This item identifies the default resolution used by the sensor and is the index into the array of supported resolutions.
“sync_stream”	int	Mandatory	This item identifies whether the sensor supports any synchronous streaming mode such as S4S, and so.

.fac_cal

- The table shows a summary of registry items within .fac_cal per data_type. Each item is of type float.

data_type	Registry items
accel gyro mag	Tri-axial sensors use a 3x3 ".corr_mat" registry group to apply factory calibration. Each item within this group is float and is named: "0_0", "0_1", "0_2", "1_0", "1_1", "1_2", "2_0", "2_1", "2_2" Tri-axial sensors also use a group named ".bias" with three float items named: "x", "y", "z"
pressure ambient_light ambient_temperature sensor_temperature humidity thermopile	"scale" "bias"
proximity hall	"near_threshold" "far_threshold"

factory_cal_parameters

data_type	Registry items
rgb	"r_channel_scale"
	"r_channel_bias"
	"g_channel_scale"
	"g_channel_bias"
	"b_channel_scale"
	"b_channel_bias"
	"color_temp_scale"
	"color_temp_bias"
uv	"a_scale"
	"a_bias"
	"b_scale"
	"b_bias"
	"total_scale"
	"total_bias"

.orient

- The driver adjusts sensor axes orientation to the device axes orientation (Android coordinate format).
- Each registry item represents orientation information per axis, and the registry item name is same as the device axis name.
- Data for each item is of type string and is a combination of axis value and sign.
- Value:

Axis	Value
x-axis	“x”
y-axis	“y”
z-axis	“z”

- Direction:

Sign	Axis direction
“+”	Sensor axis direction is same as device axis direction
“-”	Sensor axis direction is opposite to device axis direction

- Example:
 - If the sensor y-axis is the device x-axis and in opposite direction, the registry item “x” has value “-y”.
 - If the sensor z-axis is the same as device z-axis, the registry item “z” has value “+z”.

Registry Error Handling and Recovery

- Several errors may occur during normal registry processing. Key points for error handling and recovery include the following:
 - At first bootup, the /config directory contains some registry configuration files. It is likely, but not required, that the /registry directory is empty.
 - The registry module uses the `opendir` and `readdir` file system functions to determine the list of files in the /registry directory. If either of these functions fail, the registry module would be unaware of the presence of one or more files.
 - Without being aware of the file for an existing registry group, the registry may accidentally overwrite its contents. In this rare error case, the registry aborts its initialization procedure and forces the sensors protection domain (PD) to restart.
 - For each file found in the /registry directory, `fopen` and `fread` are invoked. If either function fails, it is reattempted several times. If all attempts fail, the data contained in that registry group is inaccessible and marked *unwritable* to avoid accidental overwrites.
 - After a successful `fread`, the file is parsed as a registry group and must be formatted in a valid JSON, for example, paired open/closing brackets. A failed parsing is logged, but the file is not marked *unwritable*. Assuming no file system corruption, the only cause of an invalid formatted file in the /registry directory is a failed `fwrite` on a previous device power cycle. In this case, the registry replaces the invalid file while processing the /config directory.

Registry Error Handling and Recovery (cont.)

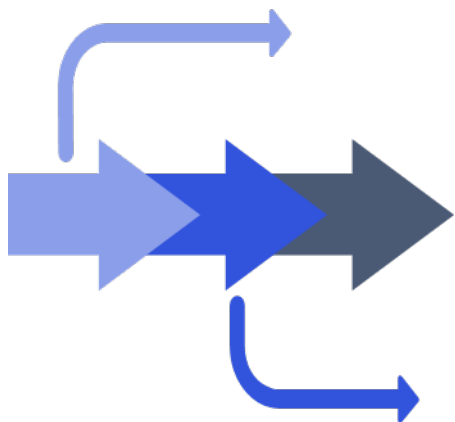
- For each successfully parsed registry group, the contents are stored in DDR and maintained in a linked-list by the registry module (*registry proper*).
- The contents of the /config directory are processed. `opendir` and `readdir` are invoked to determine the list of available configuration files. If either function fails, they similarly cause the initialization process to fail and abort, and the sensors PD restarts.
- For each configuration file, before calling `fopen` and `fread`, the registry checks the file's last modified timestamp and compares it against the timestamp of the file during its last successful parsing. If the configuration file is not parsed successfully, or if the timestamp does not match, the file processing continues. `fopen` and `fread` are reattempted upon failure. If all attempts fail, that configuration file is not parsed, and the file timestamp is not saved.
- For each successfully opened and read configuration file, the config header section of the file is parsed. Each entry is compared against the device hardware configuration. If any of the entries does not contain a match, processing of this file stops, and the registry marks the file as being parsed successfully (that is, it does not need to be retried until its file timestamp changes).

Registry Error Handling and Recovery (cont.)

- If all entries find a match, the rest of the file is parsed. Configuration files typically contain one or more registry groups. The entire file is parsed, and the resulting groups are temporarily stored.
- The contents are merged into the *registry proper* after the entire file is parsed successfully. This process occurs as follows for each group.
 1. Checks whether the existing group exists in the *registry proper*.
 2. If the existing group does not exist in the *registry proper*, merges and saves the new group.
 3. If the existing group does exist, iterates over all group items, and checks whether the existing version is 0 or less than the new version.
 4. If the version is 0 or less than the new version, updates the value of the existing item; new items are simply added to the existing group.

Registry Error Handling and Recovery (cont.)

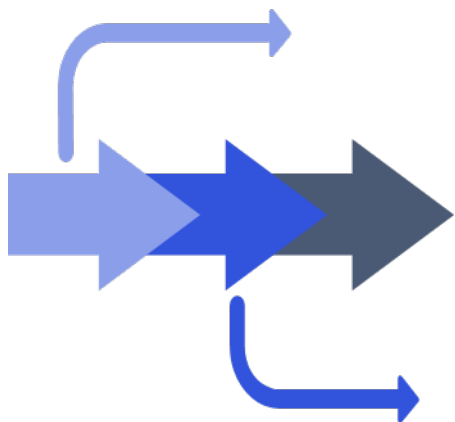
- After the group merges, and if any changes to the group occur, the registry attempts to save the JSON string formed from the registry group to file. This process involves `fopen` and `fread`. `fopen` is reattempted upon failure; `fwrite` is reattempted upon complete or partial failure. If `fopen` and `fwrite` are ultimately unsuccessful, the timestamp of the configuration file is not saved and is reparsed upon the next boot.
- After the configuration file is parsed successfully, and all resulting registry groups are correctly saved to file, the configuration file's last modified timestamp is saved to disk. If the file containing all configuration file timestamps becomes corrupted or is not saved correctly, it is discarded. Upon the next boot, the registry attempts to reparse all configuration files.
- This process continues for all files in the `/config` directory. Once completed, the Registry Sensor publishes the `Available` attribute, which notifies algorithms and drivers that they may now query for their registry groups. Any corrupted or invalid file in the `/registry` directory is detected during this process, and an attempt is made to reparse the necessary configuration files and resave the invalid files.
- If an outside source modifies the contents of the `/registry` directory (for example, a developer using the `adb` shell deletes a file), there is no mechanism to recognize its absence.
- These error detection processes typically take effect upon the subsequent power cycle. File system errors and failures are rare, and initializing the SSC with most functionality present, rather than aggressively failing, aborting, and restarting the initialization process, is preferred.



SEE Log Packets and Debug Messages

Driver Log Packets

- The diag service provides an API to allocate and submit driver log packets.
 - Drivers are required to support the following log packets:
 - SNS_DIAG_SENSOR_STATE_LOG_RAW
 - Nanopb-encoded raw sensor sample data.
 - Uses the sns_diag_sensor_state_raw message in sns_diag.proto.
 - SNS_DIAG_SENSOR_STATE_LOG_INTERRUPT
 - If the sensor supports any interrupt as defined in sns_diag_interrupt (sns_diag.proto), it must publish this log packet.
 - Nanopb-encoded interrupt information.
 - Uses the sns_diag_sensor_state_interrupt message in sns_diag.proto.
- The diag service provides an API to submit debug messages.
 - Separate printf for sensor and sensor instance.
 - Sensor instance printf uses SUID input.
 - QTI recommends using the appropriate SUID when adding a debug message from an instance, because diag may filter debug messages based on the SUID.
 - Low, Medium, High, Error, and Fatal message priority can be used.
 - See *Sensor Execution Environment F3 Message Guidelines for Driver Developers* (80-P9361-6) for guidance on using the appropriate message priority.
- All sensors must publish at least the SNS_STD_SENSOR_ATTRID_VENDOR attribute before the diag service can process requests from the sensor.



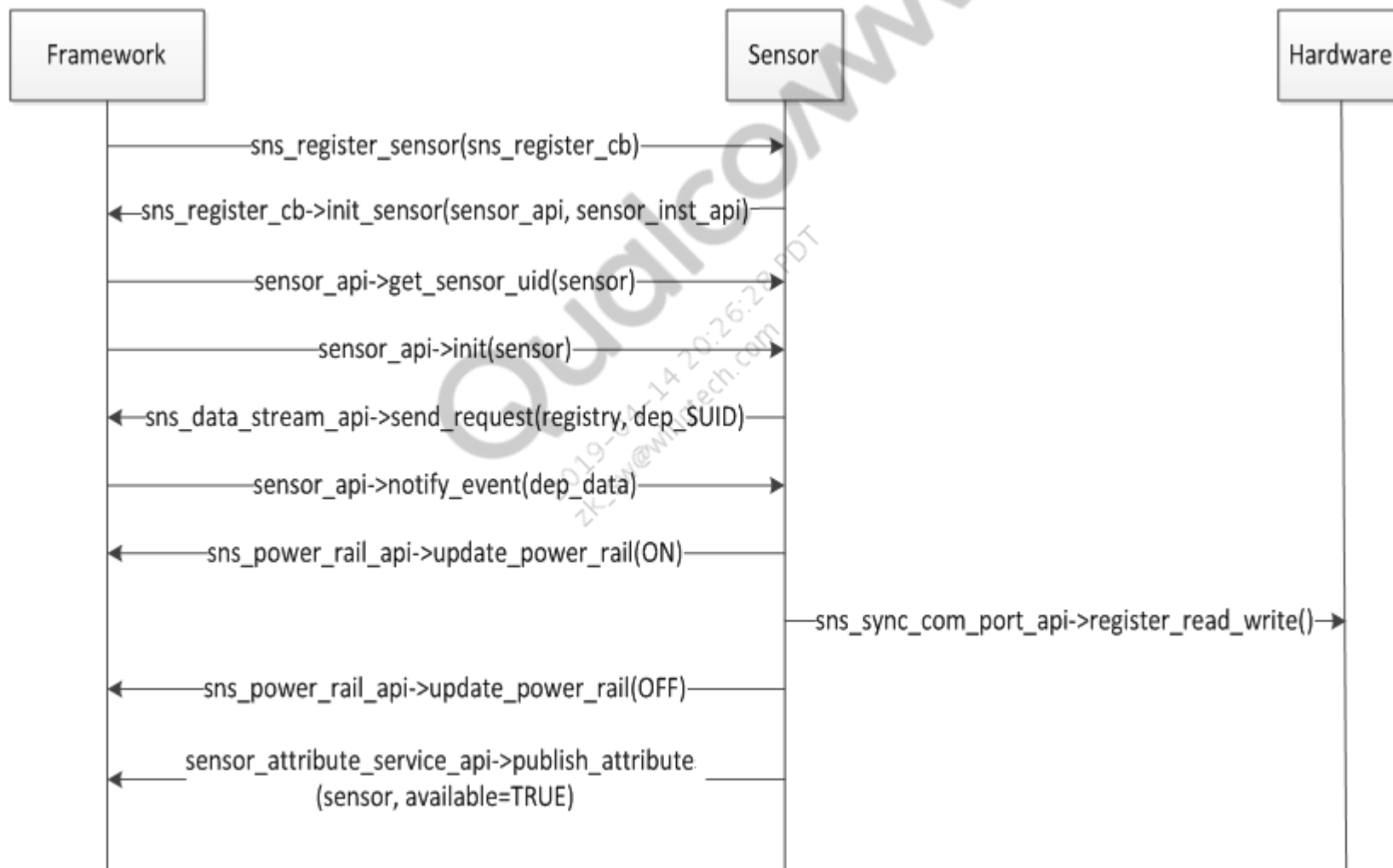
Qualcomm
2019-04-14 20:26:28 PDT
zk_sw@vtech.com

Sensor Driver Development

Sensor Initialization and Activation

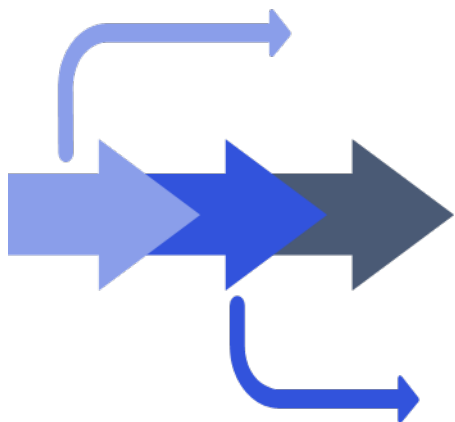
- The SEE framework provides a SUID lookup sensor. Its SUID is returned using the `sns_get_suid_lookup()` function.
- During initialization, the vendor's sensor must make a request to the SUID lookup sensor to get the SUIDs of their dependent platform sensors. These requests are made using well-known string literals, for example, `interrupt`, `timer`, `async_com_port`, and so on.
- As the dependent platform sensors become available, the SUID lookup sensor provides events to the vendor's sensor with the SUID for each. After the vendor's sensor has all required SUIDs, it publishes itself as available and is ready to support sensor requests for its supported sensors.

Sensors Initialization



Sensors Activation





Qualcomm
2019-04-14 20:26:28 PDT
zk_sw@wingtech.com

Driver Components

Driver Components

Sensor API

File	Description
sns_<drv_name>.c	Function to register driver library
sns_<drv_name>_<sensor>_sensor.c	Sensor-specific normal (also known as bigImage) mode functions for the combo driver
sns_<drv_name>_sensor.c sns_<drv_name>_sensor.h	<ul style="list-style-type: none">▪ Common normal mode sensor functions for combo driver▪ Common sensor data types for combo driver
sns_<drv_name>_sensor_island.c	Common Island (also known as uimg) mode sensor functions (code and data) for combo driver
sns_<drv_name>_hal.c (optional) sns_<drv_name>_hal.h	<ul style="list-style-type: none">▪ Hardware-specific functions for normal mode▪ Hardware-specific data types for combo driver
sns_<drv_name>_hal_island.c	Hardware-specific functions (code and data) for Island mode functions

Driver Components (cont.)

Sensor instance API

File	Description
sns_<drv_name>_sensor_instance.c sns_<drv_name>_sensor_instance.h	<ul style="list-style-type: none">▪ Normal mode functions for the sensor instance▪ Sensor instance data types for the driver
sns_<drv_name>_sensor_instance_island.c	Island mode functions (code and data) for the sensor instance

sns_<drv_name>.c

- Contains a single function, `sns_register_<drv_name>`, to register all sensors supported by this driver.
- Each physical and virtual data event supported by the driver is registered as a separate sensor API.
- For example, in the case of a combo driver that can support three sensors (accelerometer, gyroscope, and motion detect), the register function registers all three separate sensor APIs.

```
sns_rc sns_register_<drv_name>(
    sns_register_cb const *register_api)
{
    /* Register the Accelerometer Sensor API */
    register_api->init_sensor(
        sizeof( <drv_name>_state ), &<drv_name>_accel_api, &<drv_name>_instance_api);

    /* Register the Gyroscope Sensor API */
    register_api->init_sensor(
        sizeof( <drv_name>_state ), &<drv_name>_gyro_api, &<drv_name>_instance_api);

    /* Register the Motion Detect Sensor API */
    register_api->init_sensor(
        sizeof( <drv_name>_state ), &<drv_name>_md_api, &<drv_name>_instance_api);
}
```

- Driver registration is a normal mode operation.
- For more information about the `sns_register_cb` definition, see `slpi_proc\ssc\inc\sns_register.h`.

Refactoring for Island Mode

- Refactor all driver functions and code such that all Island mode functionality is pulled into a separate *_island.c file.
- A dedicated environment flag *must* be used per driver to enable Island mode support for the driver, for example:
 - SNS_ISLAND_INCLUDE_LSM6DSM in sns_lsm6dsm.scons
- Use this flag to decide the value of the `add_island_files` field when `AddSSCSU()` method is called in the driver .scons file.
 - Refer to sns_lsm6dsm.scons

```
lsm6dsm_island_enable = False
if 'SNS_ISLAND_INCLUDE_LSM6DSM' in env:
    lsm6dsm_island_enable = True
if ('SSC_TARGET_HEXAGON' in env['CPPDEFINES']) and ('SENSORS_DD_DEV_FLAG' not in env):
    env.AddSSCSU(inspect.getfile(inspect.currentframe()),
                 register_func_name = "sns_register_lsm6dsm",
                 binary_lib = False,
                 add_island_files = lsm6dsm_island_enable)
```

Refactoring Code for Island Mode

Operation mode	API	Notes
Normal	Sensor	Place in sns_<drv_name>_<sensor>_sensor.c and/or sns_<drv_name>_sensor.c. Example: sns_lsm6dsm_accel_sensor.c
	init() deinit() set_client_request() for non-accel driver libraries notify_event()	
	Sensor instance	Place sns_<drv_name>_sensor_instance.c Example: sns_lsm6dsm_sensor_instance.c
	init() deinit() set_client_config() only for non-accel driver libraries	
	HAL	Place in sns_<drv_name>_hal.c
	All functions called from sensor and sensor instance normal mode API	

Refactoring Code for Island Mode (cont.)

Operation mode	API	Notes
Island	Sensor	Place in sns_<drv_name>_sensor_island.c Example: sns_lsm6dsm_sensor_island.c
	sns_sensor_api vtable get_sensor_uid() set_client_request() only for accel driver libraries	
	Sensor instance	Place in sns_<drv_name>_sensor_instance_island.c Example: sns_lsm6dsm_sensor_instance_island.c
	sns_sensor_instance_api vtable notify_event() set_client_config() only for accel driver libraries	
	HAL	Place in sns_<drv_name>_hal_island.c Example: sns_lsm6dm_hal_island.c
	All functions called from sensor and sensor instance Island mode API	

Refactoring Data for Island Mode

- All data accessed by driver Island mode functions must be placed in a *_island.c driver file, for example, any ODR mapping table used by the sensor instance or HAL.

Qualcomm
2019-04-14 20:26:28 PDT
zk_sw@wingtech.com

Driver Versioning

- It is important to track changes in drivers.
 - For QTI third-party driver developers and customers
- QTI recommends the following format:
 - Add a *_ver.h file as part of the driver code that maintains change history comments
 - When – Date of modification
 - Who – Initials/company who made the update
 - Version – Version associated with this update
 - What – Brief information of changes
 - Add a version macro in this file that is used to publish the SNS_STD_SENSOR_ATTRID_VERSION attribute



Integrating New Sensor Drivers in SEE Framework

Adding New Sensors Drivers in SEE

This section lists the steps for integrating a non-PoR sensor driver:

- For non-PoR sensors, OEMs can request the (well-tested) driver from the vendor for their sensor part.
 - OEMs should request a Driver Acceptance checklist, mandated by QTI for adding a driver to PVL, and full CTS results.
 - Vendor-delivered package to OEMs includes:
 - SEE-compliant driver source code
 - Registry (json files)
 - Test results: Driver Acceptance checklist and CTS results with same source code and registry

Integrating SEE-Compliant Device Driver to SLPI Build

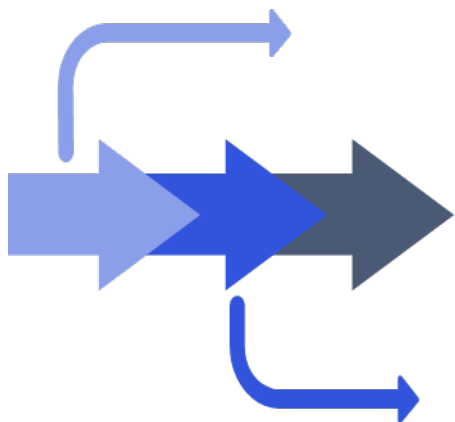
1. Create a new folder based on the part name at `\slpi_proc\ssc\sensors`.
 - `slpi_proc\ssc\sensors\<new_driver>`
 - Example: `\slpi_proc\ssc\sensors\lsm6dso`
2. Within this folder:
 - a. Create a build folder and add the driver `.scons` file.
 - `slpi_proc\ssc\sensors\<new_driver>\build\<new_driver>.scons`
 - Example: `slpi_proc\ssc\sensors\<new_driver>\build\lsm6dso.scons`
 - b. Create an `src` folder and add all the driver source and header files.
 - `slpi_proc\ssc\sensors\<new_driver>\src*`
 - Example: `slpi_proc\ssc\sensors\lsm6dso\src*`
 - c. Compile the SLPI build. On successful compilation, you should see:
 - *.lib files in `slpi_proc\ssc\sensors\<new_driver>\build\ssc_slpi_user\qdsp6\AAAAAAAAA\`
 - Check file “`slpi_proc\ssc\framework\src\sns_static_sensors.c`” and verify that the following are present:

```
sns_rc sns_register_<new_driver>(sns_register_cb const *register_api);  
const sns_register_entry sns_register_sensor_list[] =  
    { sns_register_<new_driver>, 1},
```

- d. Load the device with compiled SLPI.

Integrating Registry for New Sensor Driver in Application Processor

- Though the driver is integrated in SLPI, most of the driver configuration is from registry (JSON files). As covered in [Registry in SEE](#), there are typically two JSON files, one for platform-specific configuration and the other for driver-specific configuration. Follow these steps to update the registry to configure the newly added driver on SLPI.
- 1. Add two JSON files required for the physical sensor.
 - Platform-specific .json file:
 - Android O: vendor/qcom/proprietary/sensors-see/ssc/registry/config/<chipset_name>_<sensor_part_name>_0.json
 - Android P: vendor/qcom/proprietary/sensors-see/registry/config/<chipset_name>/<chipset_name>_<sensor_part_name>_0.json
 - Driver-specific configuration for the physical sensor:
 - Android O: vendor/qcom/proprietary/sensors-see/ssc/registry/config/<sensor_part_name>_0.json
 - Android P: vendor/qcom/proprietary/sensors-see/registry/config/common/<sensor_part_name>_0.json
- 2. Compile the application processor build and load the device with the compiled application processor build.
 - Registry (.json) files are present at:
 - Android O: /persist/sensors/registry/config
 - Android P: /vendor/etc/sensors/config
 - On successful parsing of these files, the appropriate parsed files appear in:
 - Android O: /persist/sensors/registry/registry
 - Android P: /mnt/vendor/persist/sensors/registry/registry

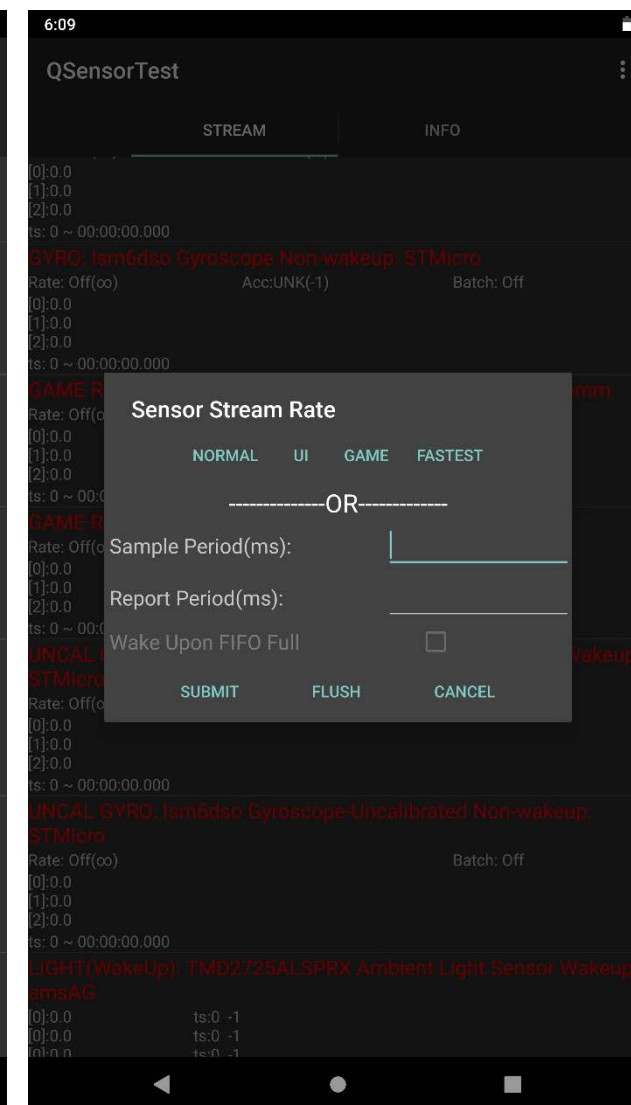


Qualcomm
2019-04-14 20:26:28 PDT
zk_sw@wingtech.com

Test Tools

QSensorTest Application (Testing at HAL)

- QSensorTest app is provided in default QTI releases.
 - Source code available in vendor/qcom/proprietary/sensors-see/QSensorTest/
- HAL support is required for any physical sensor to be tested from the QSensorTest app.
- Ability to stream at Android rates (Normal, UI, Game, Fastest) is available.



Unified Sensor Test Application (Testing at Client API)

- Unified Sensor Test Application (USTA) enables testing every sensor type available from the client API.
- USTA can create and send any request to any sensor, and parse all the events back to the client and display the events in the logcat.
- USTA is provided in default QTI releases:
 - vendor/qcom/proprietary/sensors-see/USTA
- For additional information on USTA, see *Unified Sensor Test Application (USTA) User Guide* (80-P9301-85).
- For any new sensor type to work with USTA, customers do not need to add HAL; the only requirement is that the appropriate *.proto file for new sensor (if any) be present in AP build:
 - AP build location
 - Android O: vendor/qcom/proprietary/sensors-see/ssc/proto
 - Android P: vendor/qcom/proprietary/commonsys-intf/sensors-see/ssc/proto
 - Device: /etc/sensors/proto/*
 - You can simply push required proto file « adb push sns_<new_sensor>.*proto /etc/sensors/proto »

Unified Sensor Test App

USTA-TAB REGISTRY-TAB SENSORS-INFO CAL-TAB

Sensor: accel-STMicro

SUID Low: 9a418d19f8d7ba44

SUID High: 1fbb6afc01727ea6

Scripts : Nothing is selected

ReqMsgs: sns_std_sensor_config - 513

Request:

Suspend Config:

Client Processor

sns_std_client_processor_apss

Wakeup Delivery

sns_client_delivery_wakeup

Batch Spec

Batch Period (u-sec):

Flush Period (u-sec):

Flush Only: ☐

Max Batch: ☐

Unified Sensor Test App

USTA-TAB REGISTRY-TAB SENSORS-INFO CAL-TAB

Send Request

Flush Request

Stop Request

Sensor Event Payload :

Samples Counted:	5879
Time Elapsed:	14
Expected Samples:	5824
Message ID:	1025
Timestamp:	248349013870
Data:	[-0.071823,0.143646,9.772711]
Status:	SNS_STD_SENSOR_SAMPLE_STATUS_ACCURACY_HIGH

Standard Sensor Config:

Sample Rate	416.0
Water Mark:	8
Float Resolution:	0.488
Float Range:	[-16,16]

Toggle JSON View

```
{
  "sns_client_event_msg": {
    "suid": {
      "suid_low": "0x9a418d19f8d7ba44",
      "suid_high": "0x1fbb6afc01727ea6"
    },
    "events": [

```

SEE Test Sensors (Testing at SEE API on SLPI)

- Test sensors enable validating a physical sensor driver or algorithms without involving the application processor (HAL or client API layer).
 - `slpi_proc\ssc\sensors\test\src\`
 - Test sensors enable streaming of a selected sensor (based on the compile time option) on bootup.
- The following test sensors are available in the build:
 - SEE standard stream test sensor – `slpi_proc\ssc\sensors\test\src\sns_test_std_sensor*`
 - Flush test sensor
 - Motion detect and event-gated stream sensor
- To enable a test sensor, enable a desired build flag – Options for various test sensors are available in `slpi_proc\ssc\sensors\test\src\sns_test_sensor.c`.

```
#if defined(SNS_TEST_ACCEL)
#include "sns_test_std_sensor.h"
const sns_test_implementation test_sensor_impl = {
    "accel",
    sizeof("accel"),
    sns_test_std_sensor_create_request,
    sns_test_std_sensor_process_event
};
```

- For example, to enable testing of accelerometer, add `SNS_TEST_ACCEL` to the existing build command.
- If the existing build command is “`build.py -c sdm<XXX> -o all`”, enable the accelerometer test sensor with “`build.py -c sdm<XXX> -f SNS_TEST_ACCEL -o all`”.

Driver Acceptance Test (ssc_drva_test)

- Executes various sensor use cases at the SEE API layer by passing the parameters from an adb command without requiring any compile time options, such as [SEE Test Sensors](#).
- Sensor vendors can use this utility to validate physical sensors with OpenSSC; OEMs can use it for basic test/validation purposes.

- adb (Android debug bridge) shell command syntax:

```
adb shell ssc_drva_test -sensor=<value> -duration=<value> -sample_rate=<value> -  
batch_period=<value> -iterations=<value> -num_samples=<value> -factory_test=<value>
```

Tags	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 – Limited to available sensor types

Driver Acceptance Test (ssc_drva_test) (cont.)

Tags	Type	Value range	Units	Notes
sample_rate	float	Positive floating point numbers Special: "-1" – Max sampling rate "-2" – Min Sampling rate	Hz	Mandatory for streaming sensors, optional for on-change sensors
batch_period	float	Positive floating point numbers	Seconds	This is same as batch period or report period.
num_samples	int	Positive values only	N/A	num_samples indicates the minimum number of samples desired to be collected. If this parameter is specified and the test does not collect enough samples during the test, a FAIL is generated by the test sensor. This 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.
factory_test	int	<ul style="list-style-type: none"> 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) 	N/A	This parameter selects the type of factory test you want to run.

Driver Acceptance Test (ssc_drva_test) (cont.)

- Examples

- Stream single sensor at selected sampling frequency for known duration

```
adb shell ssc_drva_test -sensor=accel -duration=30 -sample_rate=100
```

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

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

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

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

- Output

- On an adb command line, this test only outputs a pass or fail, which indicates only the test execution status (whether or not the test completed)
- Refer to the QXDM Professional™ (QXDM Pro) log for the SSC operation

Client APIs (sns_client_example)

- For details about client APIs and writing custom code for factory requirements, see *Sensors Execution Environment Client API Reference* (80-P9301-36).
 - Describes the functions of the sensor clients with the SEE
 - Includes example client code located in the Linux Android build, which provides examples to most of the referenced messages and procedures
- Source code location
 - vendor/qcom/proprietary/sensors-see/test/sns_client_example/ src/sns_client_example.c
- Usage

```
adb shell sns_client_example
```
- Notes:
 - Examples shown on the [SALT API](#) page and the API examples that follow are for informational purposes only. Changes to the examples may occur at any time without notice to customers.
 - QTI recommends writing code that is compliant to the APIs shown in *Sensors Execution Environment Client API Reference* (80-P9301-36), for example, sns_client_example.

SEE SALT

- Sensor Abstraction Layer for Testing (SALT)
 - vendor/qcom/proprietary/sensors-see/test/see_salt
- SEE SALT includes a set of C++ classes and APIs for use by sensor test applications.
 - It includes a set of Linux command line utilities.
 - Utility programs are generally useful and provide coding examples for the see_salt APIs.
 - APIs are examples only and subject to change.
- SEE SALT uses the libUSTANative library to access SSC sensors.
 - Using libUSTANative provides extensible access to new sensors and features.
 - SSC sensor messages/APIs are defined using protocol buffers files.
 - Many SEE SALT APIs are a simple abstraction of the protocol buffer messages.
- Adb shell command line applications use the SEE SALT APIs to implement their functionality.
 - see_workhorse
 - see_resampler
 - see_selftest

see_workhorse Example

- **see_workhorse** – Runs a particular sensor configured by command line arguments
 - -sensor selects using sns_std_sensor.proto → sns_std_sensor_attr_id → SNS_STD_SENSOR_ATTRID_TYPE
 - -sample_rate sets sns_std_sensor.proto → sns_std_sensor_config → sample_rate
 - -sample_rate=min | -sample_rate=max | -sample_rate=number to select the minimum odr | maximum odr | numerical rate in Hz
 - -batch_period sets → sns_std.proto → sns_std_request → batch_spec → batch_period
 - -calibrated=<0 | 1>
 - 0 – For all sensors except gyroscope and magnetometer
 - 1 – If sensor_type is gyroscope/magnetometer then also activates gyro_cal/mag_cal
 - -wakeup=<0 | 1>
 - 0 – Sets suspend_config wakeup to SEE_CLIENT_DELIVERY_NO_WAKEUP
 - 1 – Sets suspend_config wakeup to SEE_CLIENT_DELIVERY_WAKEUP (default)
- **Source code location**
 - vendor/qcom/proprietary/sensors-see/test/see_workhorse/
- **Usage**
 - adb shell see_workhorse -sensor=<sensor_type> [-sample_rate=<min | max | number>] [-batch_period=<seconds>] [-calibrated=<0 | 1>] [-wakeup=<0 | 1>]
 - -duration=<seconds>
 - where: <sensor_type> :: accel | gyro | ...
- **Example** – adb shell see_workhorse -sensor=accel -sample_rate=100 -duration=10

Note: Code samples are provided as examples only and subject to change.

see_resampler Example

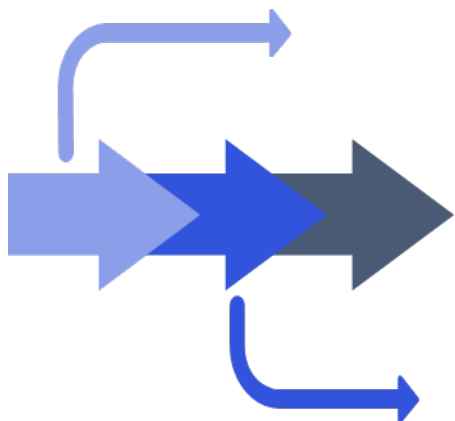
- `see_resampler` – Streams a particular sensor using resampler configured by command line arguments
 - `-sample_rate` sets parameter `sns_resampler.proto` → `sns_resampler_config` → `resampled_rate`
 - `-rate_type` sets `sns_resampler.proto` → `sns_resampler_config` → `rate_type`
 - `SNS_RESAMPLER_RATE_FIXED` | `SNS_RESAMPLER_RATE_MINIMUM`
 - `-filter` set to `true` to enable filtering, else `false`: `sns_resampler.proto` → `sns_resampler_config` → `filter`
- Source code location
 - `vendor/qcom/proprietary/sensors-see/test/see_resampler/`
- Usage
 - `adb shell see_resampler -sensor=<sensor_type> [-name=<name>] [-sample_rate=<hz>] -duration=<seconds>`
 - `[-rate_type=<fixed | minimum>] [-filter=<0 | 1>]`
- Example – `adb shell see_resampler -sensor=accel -sample_rate=10 -rate_type=fixed -duration=10`

Note: Code samples are provided as examples only and subject to change.

see_selftest Example

- `see_selftest` – Runs a particular self-test for a physical sensor driver
 - `-testtype=number` specifies `sns_physical_sensor_test.proto` → `sns_physical_sensor_test_type`
 - 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`
- Source code location
 - `vendor/qcom/proprietary/sensors-see/test/see_selftest`
- Usage
 - `adb shell see_selftest -sensor=<sensor_type> [-name=<name>] -testtype=<number>`
- Example – `adb shell see_selftest -sensor=accel -testtype=2`

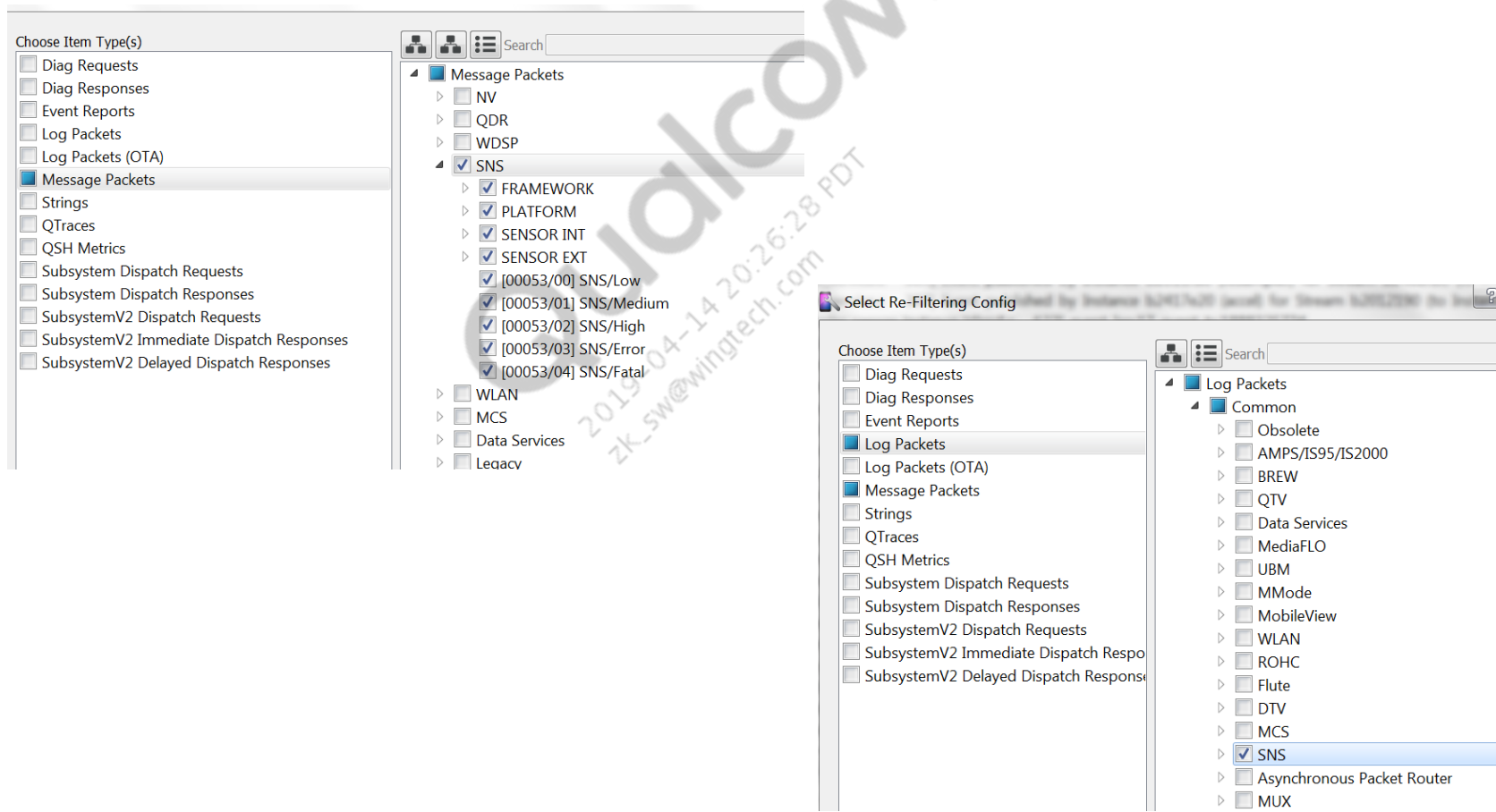
Note: Code samples are provided as examples only and subject to change.



Qualcomm
2019-04-14 20:26:28 PDT
zk_sw@wingtech.com

Debugging

- Sensors message packets and log packets



Note: Use QXDM Pro v04.00.187 or later for SEE debugging to view these message/log packets.

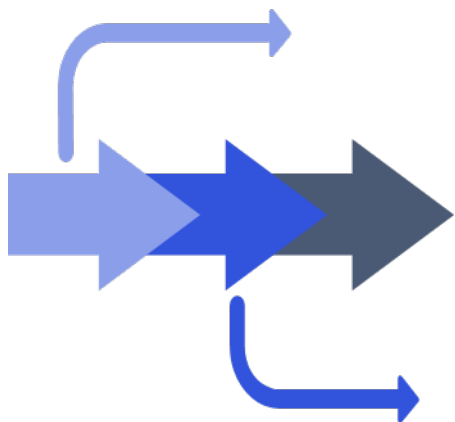
Sensors Device Drivers Logs

- With SEE, sensor device drivers logs appear under “SENSOR EXT” message packets, for example:

```
[0125/00... MSG      21:56:29.599  SENSOR EXT/Low    [ sns_bmp285_sensor.c 1272] <sns_see_if__ set_client_request> for 1
[0125/00... MSG      21:56:29.599  SENSOR EXT/Low    [ sns_bmp285_sensor.c 1385] new request from 1 message id:513 add to the client request list
[0125/00... MSG      21:56:29.599  SENSOR EXT/Low    [ sns_bmp285_sensor.c 1386] instance2 is B243E1C0
[0125/00... MSG      21:56:29.599  SENSOR EXT/Low    [ sns_bmp285_sensor.c 1399] try to configure
[0125/00... MSG      21:56:29.599  SENSOR EXT/Low    [ sns_bmp285_sensor.c 498] sensor type: 1 1
[0125/00... MSG      21:56:29.599  SENSOR EXT/Low    [ sns_bmp285_sensor.c 409] pressure sample rete 25 pressure present 1
[0125/00... MSG      21:56:29.599  SENSOR EXT/Low    [ sns_bmp285_sensor.c 523] sensor type:1, enable sensor flag:0x1 publish sensor flag:0x1
[0125/00... MSG      21:56:29.599  SENSOR EXT/Low    [ sns_bmp285_sensor.c 266] sensor type:1, sample rate 25
[0125/00... MSG      21:56:29.599  SENSOR EXT/Low    [sns_bmp285_sensor_instance.c 729] <sns_see_if__ set_client_config>
[0125/00... MSG      21:56:29.599  SENSOR EXT/Low    [sns_bmp285_sensor_instance.c 470] pressure odr = 25
[0125/00... MSG      21:56:29.599  SENSOR EXT/Low    [sns_bmp285_sensor_instance.c 506] pressure timer_value = 768000
[0125/00... MSG      21:56:29.599  SENSOR EXT/Low    [sns_bmp285_sensor_instance.c 299] bmp285_reconfig_hw state->config_step = 0
[0125/00... MSG      21:56:29.599  SENSOR EXT/Low    [sns_bmp285_sensor_instance.c 303] enable sensor flag:0x1 publish sensor flag:0x1
[0125/00... MSG      21:56:29.600  SENSOR EXT/Low    [sns_bmp285_sensor_instance.c 157] pressure_info.timer_is_active:1 state->pressure_info.sampling_intvl:768000
[0125/00... MSG      21:56:29.600  SENSOR EXT/Low    [sns_bmp285_sensor_instance.c 116] bmp285_start_sensor_pressure_polling_timer
```

- All message and log packets related to individual sensors (driver, platform sensors) are filtered by diag filter “vendor/qcom/proprietary/sensors-see/ssc/registry/config/sns_diag_filter.json”.
 - Add custom sensors to sns_diag_filter.json to view the logs in QXDM, for example:

```
"<custom_sensor>":
{
  "type" : "int",
  "ver" : "0",
  "data" : "1"
},
```



Qualcomm All-Ways Aware Hub Validation Tool

Introduction to Qualcomm All-Ways Aware Hub Validation Tool

- Standalone tool for the analysis of log packets generated by the SEE
- Features supported
 - Parses SEE logs from a DLF log file
 - Converts binary log data into a human-readable text format
 - Aggregates log data of similar type into data sets, such as CSV data files
 - Plots log data and reports data statistics
- Requirements
 - Compatible with x64 Windows 7 or later operating systems
 - .NET framework v4.5
- Optional requirements
 - To generate plots, the following is required:
 - A fully licensed version of MATLAB; see <http://www.MathWorks.com>
 - Full path to the MATLAB.exe program must be present within the "path" environment variable

Download Portal in CreatePoint

HomeDocumentsHardware ComponentsSoftware CodeToolsSupportBuy

All Products | Edit

Tools: Choose a Tools Suite

Tools: Choose a Tools Suite

All Suites

Filter Results ?

All-Ways

Apply

Your Filters

Clear All

System OS

Full Name

☐ Qualcomm All-Ways Aware™ Hub Validation Tool (2)

All Tools

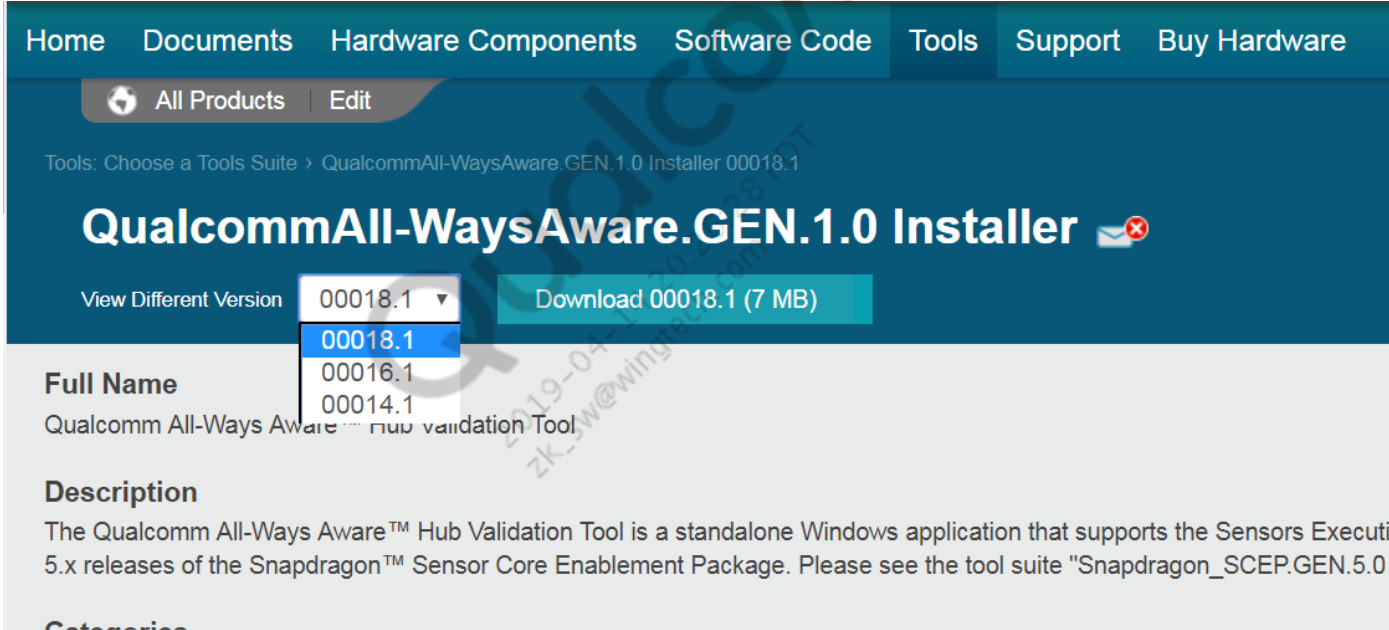
Subscribe

Unsubscribe

<input type="checkbox"/>	Name	Full Name	Version
<input type="checkbox"/>	QualcommAll-WaysAware.GEN.1.0 Installer	Qualcomm All-Ways Aware™ Hub Validation Tool	00014.1

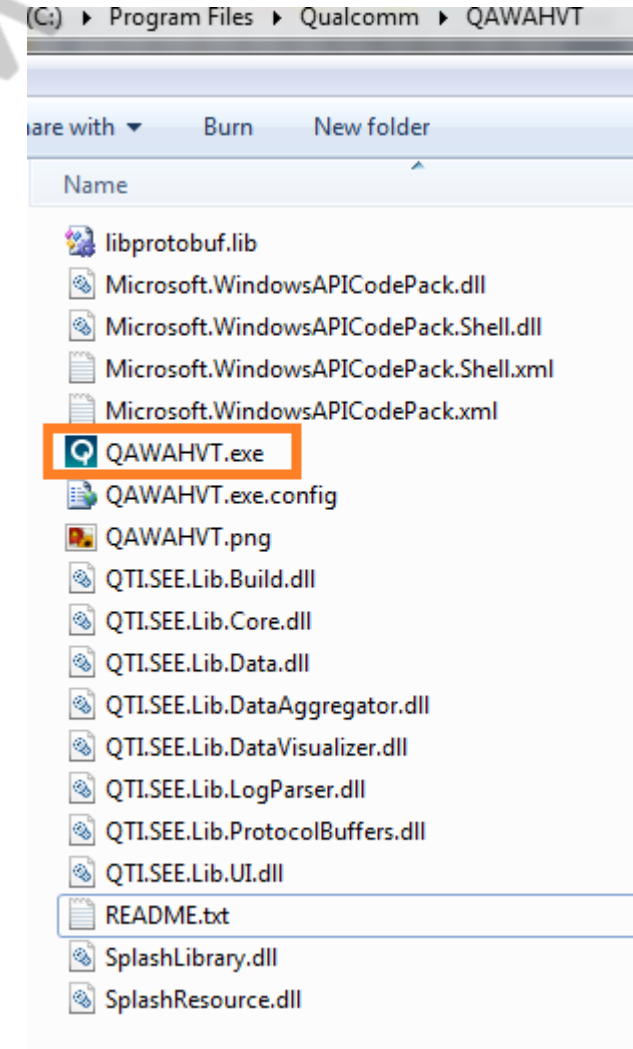
Versions

- Use the latest version available in CreatePoint.

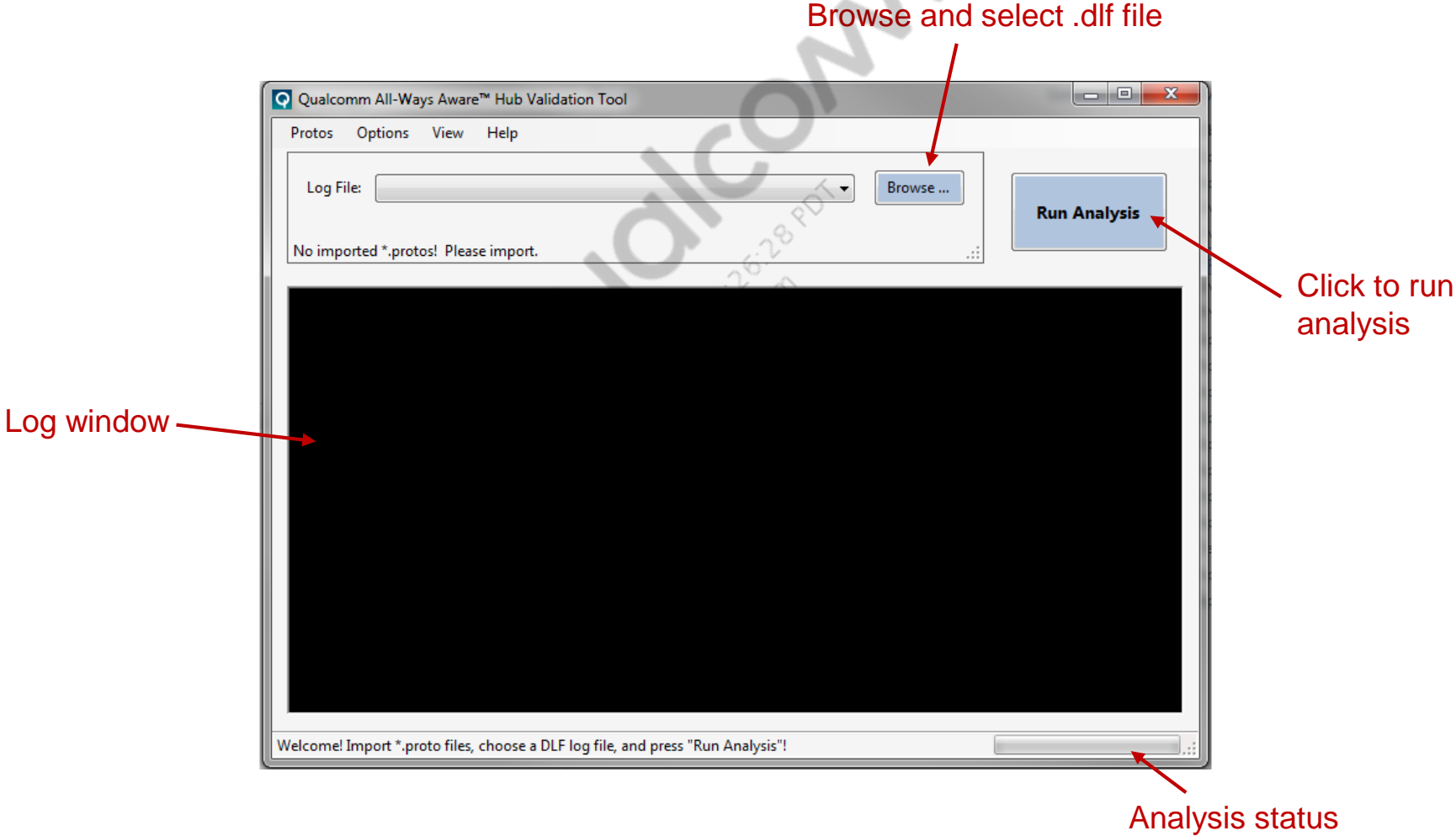


Installation

- A QAWAHVT-setup.exe is included in the download.
- Run the executable to install.
- Install the tool at C:\Program Files\Qualcomm\QAWAHVT.
- Launch QAWAHVT.exe to open the tool.



Window Menu



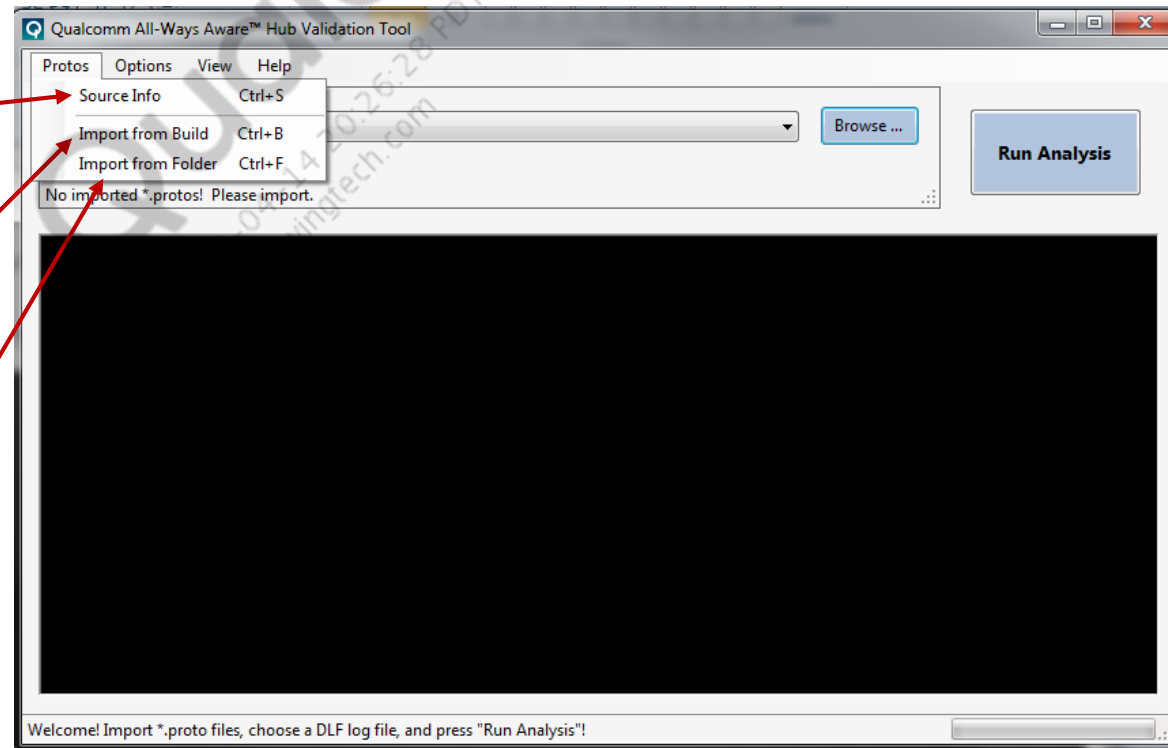
Protos Menu

- The tool requires importing .proto files to analyze logs.
- The .proto files must be the same version as those in SLPI build that was used to generate a .dlf log.

View location of imported .proto files in log window.

Import .proto files from SLPI build. Location must be \slpi_proc\ssc folder.

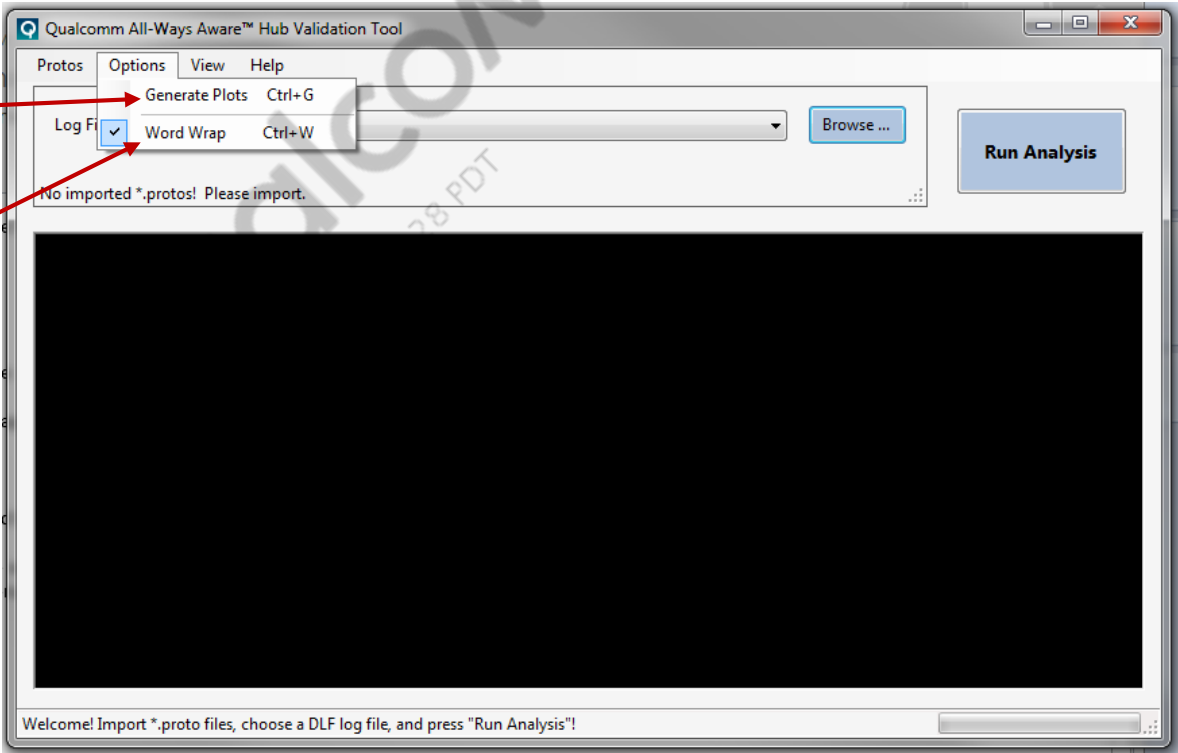
Import from a folder that contains all .proto files from the build.



Options Menu

Generate
MATLAB plots

Word Wrap
output in log
window

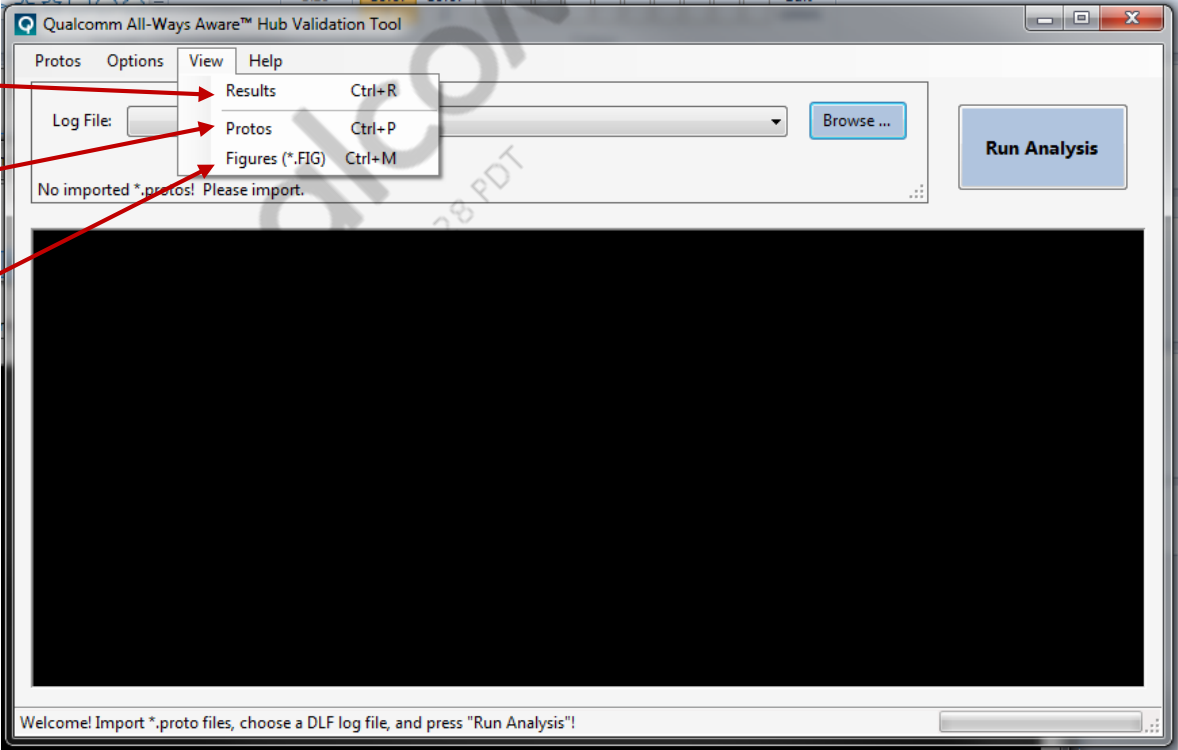


View Menu

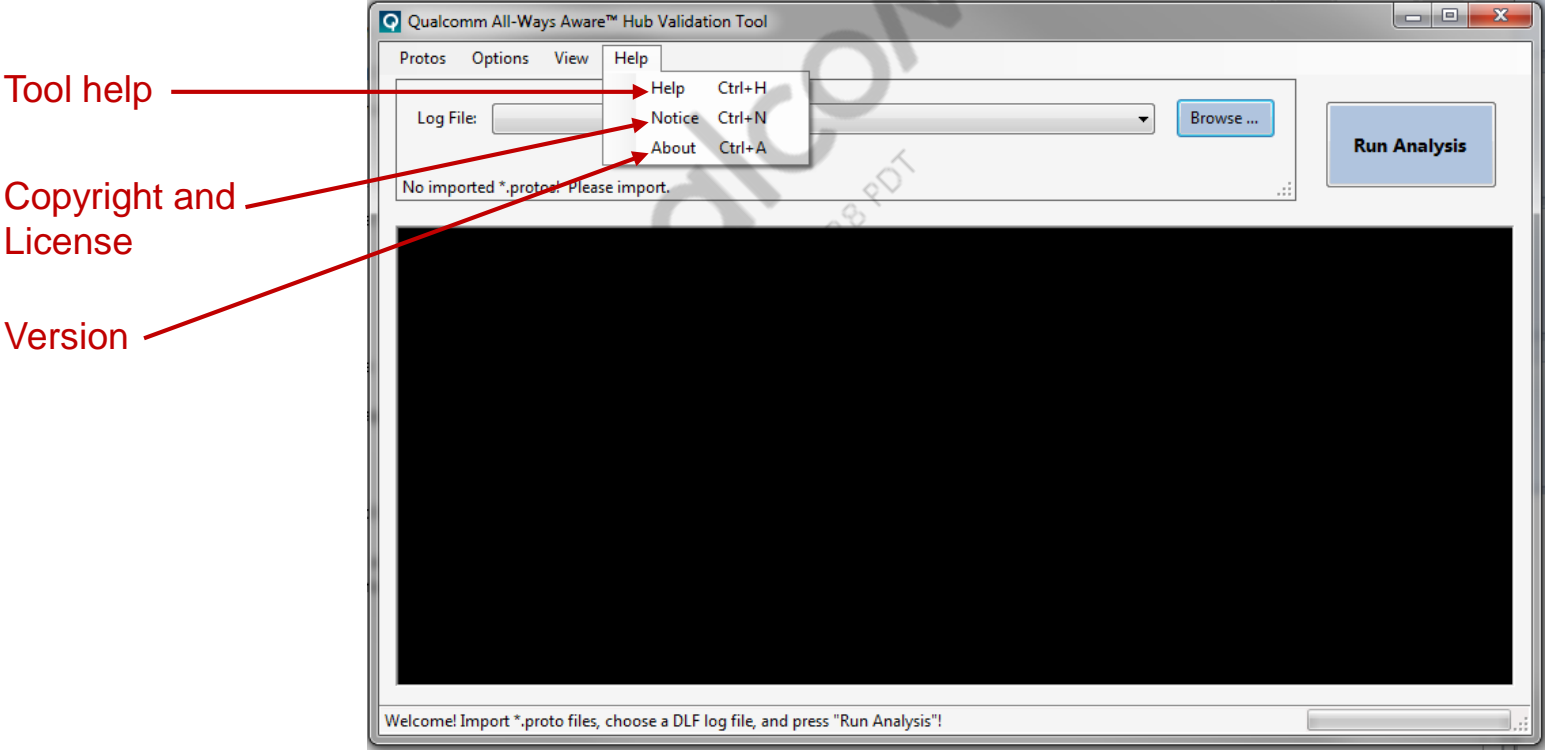
Results location

Proto file location

Generated MATLAB figures

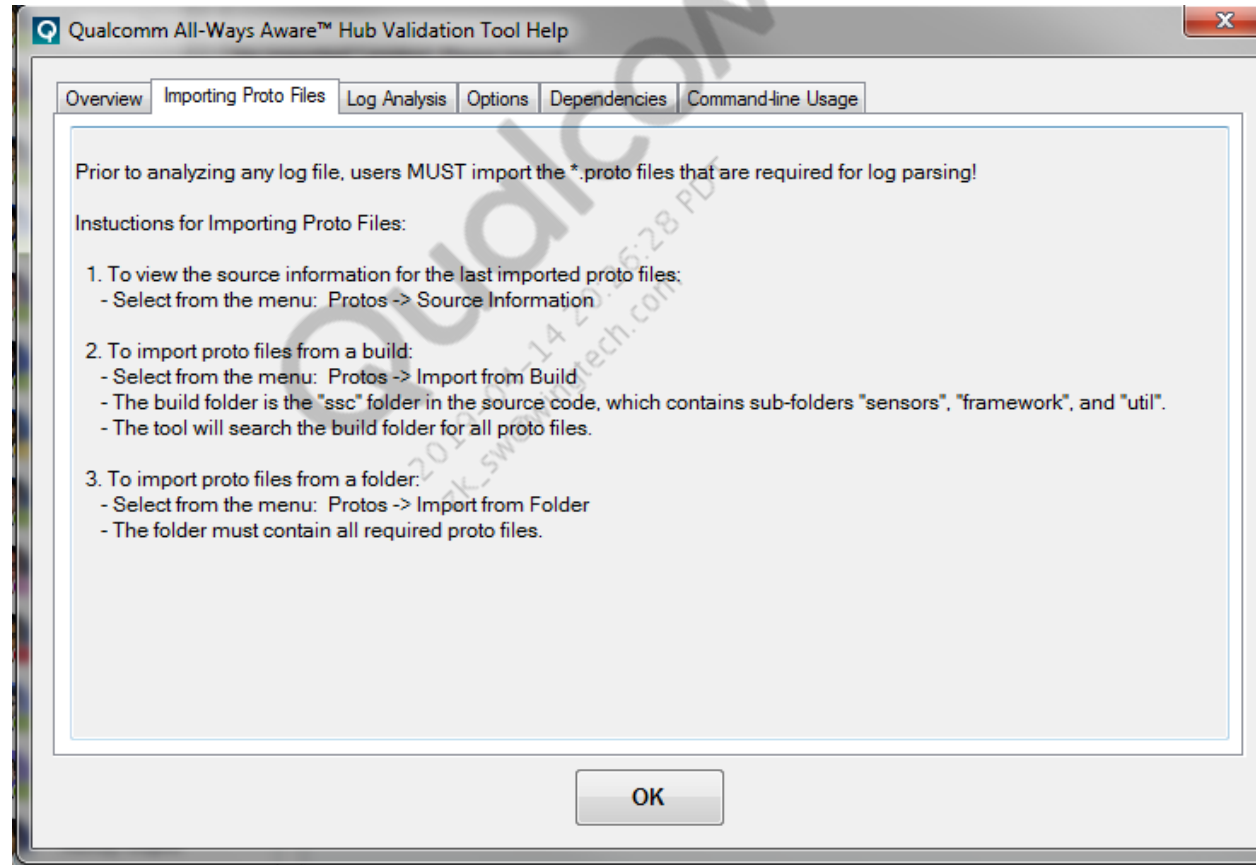


Help Menu



Help Menu (cont.)

- Detailed help information



Log Analysis

1. Import .proto files from an SLPI build that is compiled successfully.
 - This must be done each time there is change in the .proto files in use.
2. Select the .dlf file to analyze.
 - Convert the .isf file to .dlf using QXDM before using the Qualcomm All-Ways Aware Hub Validation Tool.
 - .isf can be converted to .dlf in QXDM menu → Tools → ISF to DLF Converter.
3. Enable plot generation if MATLAB is installed.
4. Select **Run Analysis**.
5. View the results and plots.

Log Analysis Example

- Accelerometer normal rate 10 sec test

Log file to be analyzed

Last import time

Information (white)
Error (red)
Success (green)
Warning (orange)

Tool status message

Qualcomm All-Ways Aware™ Hub Validation Tool

Protos Options View Help

Log File: C:\Temp\game_rv.dlf

Browse ...

Run Analysis

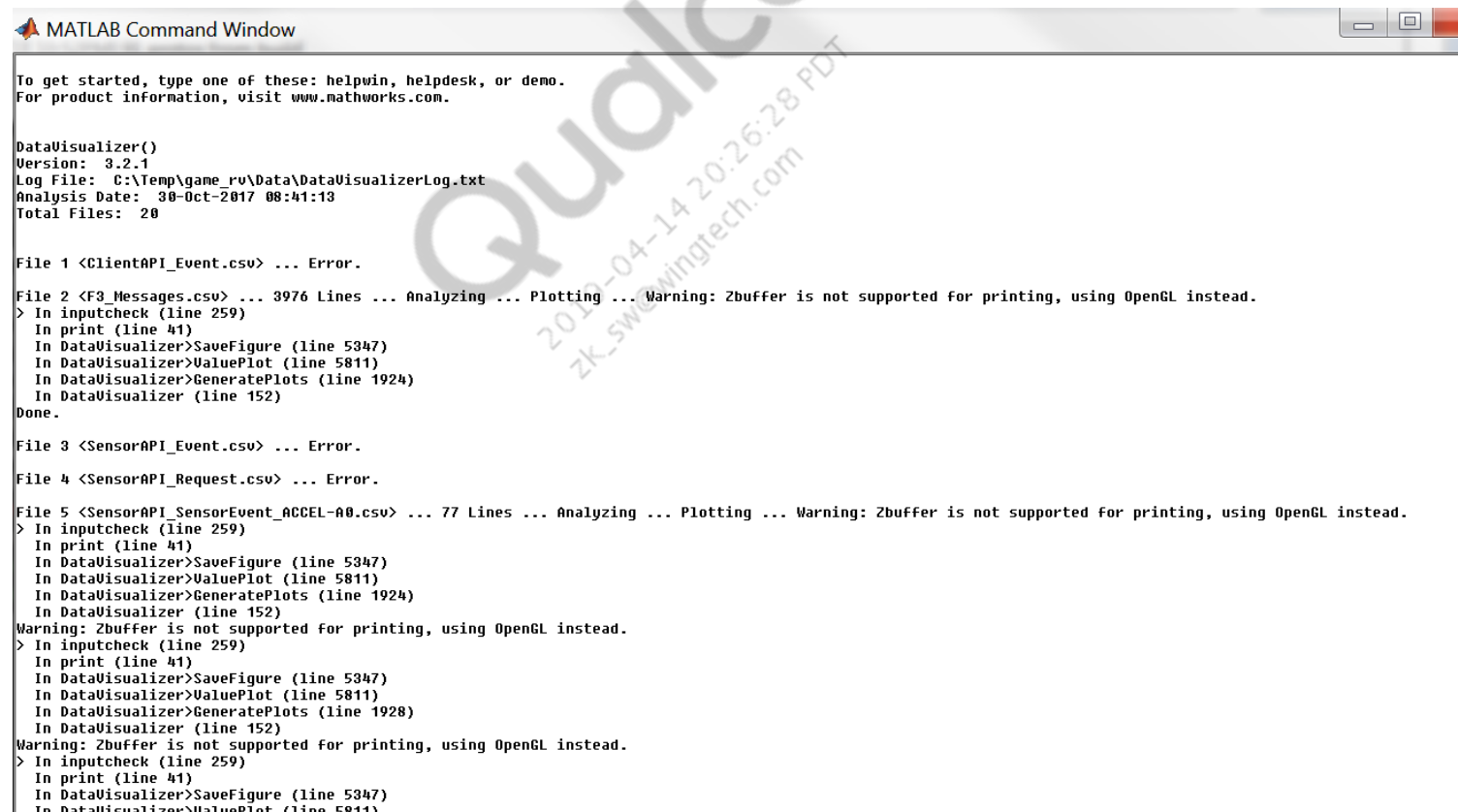
Last import: [10/18 10:52PM] 91 protos from build
"\\snowcone\builds797\PROD\SLPI.HY.1.0-00192-SDM845AZL-1\slpi_proc\ssc"

Log file: "C:\Temp\game_rv.dlf"
Output folder: "C:\Temp\game_rv"
LogParser Starting ...
DLF log file: "C:\Temp\game_rv.dlf"
Protocol buffer directory: "C:\Temp\game_rv\Protos"
Output folder: "C:\Temp\game_rv\Parser"
DataAggregator Starting ...
Output log folder: "C:\Temp\game_rv\Aggregator"
Output data folder: "C:\Temp\game_rv\Data"
Errors encountered while parsing logs! See the log file for more details: C:\Temp\game_rv\Parser\Parser.log.xml
DataAggregator Stopping ...
Data aggregator done!
Log parser done!
Data Visualizer Starting ...
Data Visualizer tool done!
The Qualcomm All-Ways Aware™ Hub Validation Tool completed!

Validating ... Initializing ... Parsing ... Analyzing (Matlab will open) ... Done!

Log Analysis Example (cont.)

- If plotting is enabled, a MATLAB window opens that shows progress
- MATLAB is optional; without MATLAB, tool would still run and analyze, only plotting of the data would be skipped.



The screenshot shows a MATLAB Command Window with the following text:

```
MATLAB Command Window

To get started, type one of these: helpwin, helpdesk, or demo.
For product information, visit www.mathworks.com.

DataVisualizer()
Version: 3.2.1
Log File: C:\Temp\game_ru\Data\DataVisualizerLog.txt
Analysis Date: 30-Oct-2017 08:41:13
Total Files: 20

File 1 <ClientAPI_Event.csv> ... Error.

File 2 <F3_Messages.csv> ... 3976 Lines ... Analyzing ... Plotting ... Warning: Zbuffer is not supported for printing, using OpenGL instead.
> In inputcheck (line 259)
  In print (line 41)
  In DataVisualizer>SaveFigure (line 5347)
  In DataVisualizer>ValuePlot (line 5811)
  In DataVisualizer>GeneratePlots (line 1924)
  In DataVisualizer (line 152)
Done.

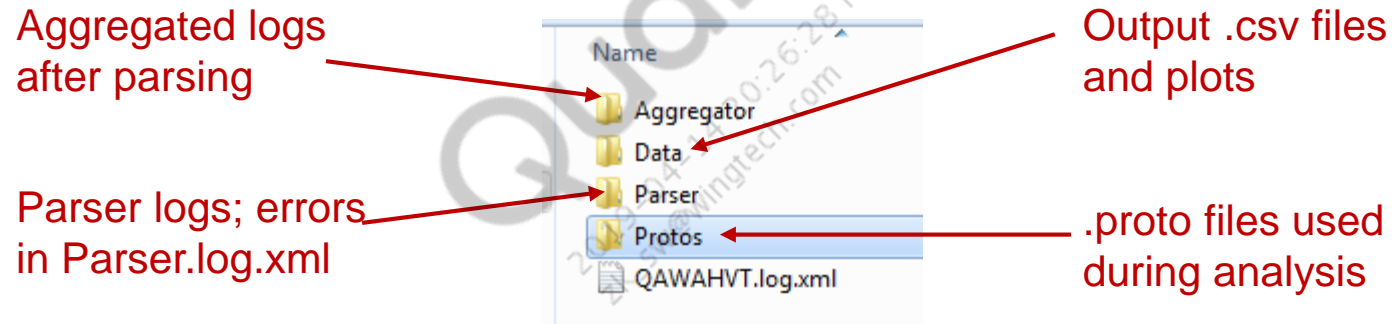
File 3 <SensorAPI_Event.csv> ... Error.

File 4 <SensorAPI_Request.csv> ... Error.

File 5 <SensorAPI_SensorEvent_ACCEL-A0.csv> ... 77 Lines ... Analyzing ... Plotting ... Warning: Zbuffer is not supported for printing, using OpenGL instead.
> In inputcheck (line 259)
  In print (line 41)
  In DataVisualizer>SaveFigure (line 5347)
  In DataVisualizer>ValuePlot (line 5811)
  In DataVisualizer>GeneratePlots (line 1924)
  In DataVisualizer (line 152)
Warning: Zbuffer is not supported for printing, using OpenGL instead.
> In inputcheck (line 259)
  In print (line 41)
  In DataVisualizer>SaveFigure (line 5347)
  In DataVisualizer>ValuePlot (line 5811)
  In DataVisualizer>GeneratePlots (line 1928)
  In DataVisualizer (line 152)
Warning: Zbuffer is not supported for printing, using OpenGL instead.
> In inputcheck (line 259)
  In print (line 41)
  In DataVisualizer>SaveFigure (line 5347)
  In DataVisualizer>ValuePlot (line 5811)
```

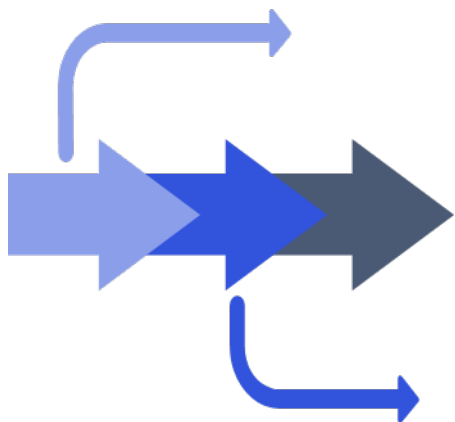

Log Analysis Example (cont.)

- Output files are created in the same folder as the .dlf log.



Error Logs

- If blocking error logs occur during analysis, send the following information to QTI:
 - .dlf file used for analysis
 - .proto files used for analysis
 - QAWAHVT.log.xml
 - Qualcomm All-Ways Aware Hub Validation Tool version used for analysis



Qualcomm
2019-04-14 20:26:28 PDT
zk_sw@wiprtech.com

Frequently Asked Questions

How is Autodetect Enabled for Multiple Sensors with SEE?

To add two accelerometer sensors (for example, LSM6DSM and BMI160) for autodetection:

▪ Android O

1. Add the JSON files to vendor/qcom/proprietary/sensors-see/ssc/registry/config directory.
 - vendor/qcom/proprietary/sensors-see/ssc/registry/config/ <chipset_name>_lsm6dsm_0.json & lsm6dsm_0.json
 - vendor/qcom/proprietary/sensors-see/ssc/registry/config/ <chipset_name>_bmi160_0.json & bmi160_0.json
2. Add the requisite JSON files to config_list.txt.
 - vendor/qcom/proprietary/sensors-see/ssc/registry/config/config_list.txt
 - Add the first accelerometer registry file (<chipset_name>_lsm6dsm_0.json) to config_list.txt.
 - Add second accelerometer registry file (<chipset_name>_bmi160_0.json) to config_list.txt.
3. Add drivers for both sensors to the SLPI build.

▪ Android P

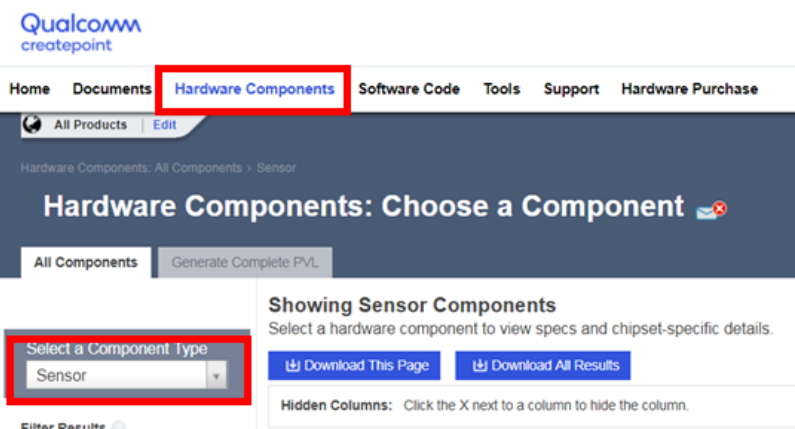
1. Add the JSON files to vendor/qcom/proprietary/sensors-see/registry/config directory.
 - vendor/qcom/proprietary/sensors-see/ssc/registry/config/<chipset_name>/<chipset_name>_lsm6dsm_0.json
 - vendor/qcom/proprietary/sensors-see/ssc/registry/config/common/lsm6dsm_0.json
 - vendor/qcom/proprietary/sensors-see/ssc/registry/config/<chipset_name>/<chipset_name>_bmi160_0.json
 - vendor/qcom/proprietary/sensors-see/ssc/registry/config/common/bmi160_0.json
2. Add drivers for both sensors to the SLPI build.

Which Sensor Parts are Supported by the SEE Framework?

- There are two categories of sensor parts
 - Plan of record (PoR) parts – Shipped and tested by QTI in mainline software releases, present on QTI reference platforms
 - Non-PoR parts – All other parts (not present on QTI reference platforms) – Drivers are available from sensor vendors directly
 - OEMs should always request and use fully-tested drivers from vendors, including:
 - Driver Acceptance checklist
 - CTS results for corresponding Android version

Which Sensor Parts are Supported by the SEE Framework? (cont.)

- For commercially available chipsets, supported parts are documented in CreatePoint → Hardware Components → Sensor. See the following example for accessing CreatePoint.



Example of search for SM8150-support mag (e-compass) sensor parts

Vendor

- ☐ InvenSense, Inc. (590)
- ☐ ROHM Co., Ltd. (570)
- ☐ Robert Bosch LLC (534)
- ☐ STMicroelectronics (China) Investment Co., Ltd (430)
- ☐ Asahi Kasei Microdevices Corporation (269)

[More...](#)

Sensor Types

- ☐ Accelerometer (1522)
- ☐ E-Compass (991)
- ☐ Gyroscope (891)
- ☐ Ambient Light (591)
- ☐ Proximity (542)

[More...](#)

Chipset

Level

- ☒ Gold (4046)

Select a Component Type

Sensor

Filter Results

Apply

Your Filters [Clear All](#)

Level

- ☒ Gold

Chipset

- ☒ SM8150

Sensor Types

- ☒ E-Compass

Vendor

- ☐ Asahi Kasei Microdevices Corporation (3)
- ☐ MEMSIC, Inc. (2)
- ☐ Yamaha Corporation (1)

Sensor Types

- ☒ E-Compass (6)

Chipset

Level

- ☒ Gold (6)

[Download This Page](#) [Download All Results](#)

Hidden Columns: Click the X next to a column to hide the column.

Name	Chipset	SW Product	Vendor	Part #
Yamaha-YAS539 Android P/HIFI/MTP855	SM8150	SM8150.LA.1.0	Yamaha Corporation	YAS539 Android P/HIFI/MTP855
MEMSIC-MMC5603 Android P/HIFI/MTP855	SM8150	SM8150.LA.1.0	MEMSIC, Inc.	MMC5603 Android P/HIFI/MTP855
MEMSIC-MMC3530 Android P/HIFI/MTP855	SM8150	SM8150.LA.1.0	MEMSIC, Inc.	MMC3530 Android P/HIFI/MTP855
AKM-AK09918C Android P/HIFI/MTP855	SM8150	SM8150.LA.1.0	Asahi Kasei Microdevices Corporation	AK09918C Android P/HIFI/MTP855
AKM-AK09917D	SM8150	SM8150.LA.1.0	Asahi Kasei Microdevices Corporation	AK09917D
AKM-AK09916C Android P/HIFI/MTP855	SM8150	SM8150.LA.1.0	Asahi Kasei Microdevices Corporation	AK09916C Android P/HIFI/MTP855

1

Important: Parts for new chipsets not yet public are not listed in CreatePoint. Contact QTI and vendor for new chipset support.

How Can a GPIO be Controlled in a Sensor Driver or Algorithm?

- With SEE, a GPIO can be controlled by sensors (physical sensor driver or algorithms) using the GPIO service.
- Refer to `slpi_proc\ssc\inc\services\sns_gpio_service.h` for details and options.
 - `sns_rc (*read_gpio)(uint32_t gpio, bool is_chip_pin, sns_gpio_state *state)`
 - `sns_rc (*write_gpio)(uint32_t gpio, bool is_chip_pin, sns_gpio_drive_strength drive_strength, sns_gpio_pull_type pull, sns_gpio_state state)`
- `is_chip_pin` is used to identify an SSC vs. MSM GPIO; configure as follows:
 - `is_chip_pin = True` for MSM GPIOs
 - `is_chip_pin = False` for SSC GPIOs
- An example is available in the template driver.

```
void lsm6dsm_write_gpio(sns_sensor_instance *instance, uint32_t gpio
...
sns_service_manager *smgr = instance->cb->get_service_manager(instance);
sns_gpio_service *gpio_svc = (sns_gpio_service*)smgr->get_service(smgr, SNS_GPIO_SERVICE);
...
rc = gpio_svc->api->write_gpio(gpio, is_chip_pin, drive_strength, pull, gpio_state);
```

How Do I generate .c and .h Files from Proto Files?

- SLPI build instructions are provided by QTI compile proto files.
- This also applies to any custom proto files.
 - To add a custom proto file, place the proto file under one of the following two directories:
 - slpi_proc\ssc\sensors\<custom_sensor>\pb
 - slpi_proc\ssc\sensors\pb\
- Creating .c and .h files and placing them in the build for proto files is not required, unlike the legacy QMI IDL compiler.
- During compilation, all proto files are compiled first using the nanopb tool and then followed by SLPI source code compilation.
- Build-log (slpi_proc\build\ms\build-log.txt) contains logs similar to the following:

```
...
slpi_proc\ssc\tools\nanopb\generator-win\protoc.exe --plugin=<>slpi_proc\ssc\tools\nanopb\generator-
win\protoc-gen-nanopb.exe --nanopb_out=D:\slpi_proc\ssc\inc\pb --
proto_path=D:\slpi_proc\ssc\utils\nanopb\pb --proto_path=D:\slpi_proc\ssc\utils\nanopb\pb --
proto_path=D:\slpi_proc\ssc\sensors\pb D:\slpi_proc\ssc\utils\nanopb\pb\nanopb.proto
```


Why Can't I see F3 messages for Vendor Drivers?

- With SEE, “SENSOR_EXT” message packets are introduced. Verify that the latest version of QXDM Pro from CreatePoint is installed, along with using the correct filter with “Message Packets → SNS → SENSOR_EXT” type.

```
[0125/0000] MSG 21:56:30.078 SENSOR_EXT/Low [sns_tmx4903_sensor.c 1986] AMS: ams_set_client_request:
enter: sensor_this 0xB201AC10, exist_rq 0x0, new_rq 0xB2018300, rmv=0; inst 0xB20222A0
[0125/0000] MSG 21:56:30.078 SENSOR_EXT/Low [sns_tmx4903_sensor.c 2079] AMS: ams_set_client_request:
instance B20222A0; new or chg rqst
[0125/0000] MSG 21:56:30.078 SENSOR_EXT/Low [sns_tmx4903_sensor.c 2101] AMS: ams_set_client_request:
adding new request
[0125/0000] MSG 21:56:30.078 SENSOR_EXT/Low [sns_tmx4903_sensor.c 2118] AMS: ams_set_client_request:
call ams_reval_instance_config
```

- Verify that sensor in which you are interested has an entry present in vendor/qcom/proprietary/sensors-see/ssc/registry/config/sns_diag_filter.json.

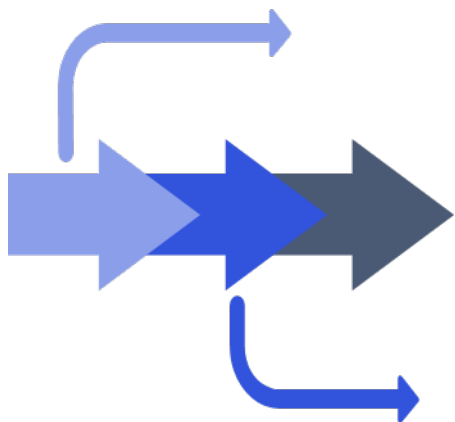
How Can I Route “SENSOR_EXT” Logs to “SNS”?

- Routing of non-QTI vendor logs to “SENSOR_EXT” logs are managed in the diag service as shown below: slpi_proc\ssc\framework\src\sns_diag_service.c. Remove the “if” statement in the following to route all logs to “SNS” instead of “SNS_EXT”:

```
SNS_SECTION(".text.sns") uint32_t sns_diag_get_debug_ssid(char const *vendor)
{
    . . .
    . . .
    //default ssid; assumed to be non-qualcomm
    uint32_t debug_ssid = MSG_SSID_SNS_SENSOR_EXT;

    if(0 == strcmp(vendor, "qualcomm"))
    {
        debug_ssid = MSG_SSID_SNS;
    }

    return debug_ssid;
}
```



Qualcomm
2019-04-14 20:26:28 PDT
zk_sw@wingtech.com

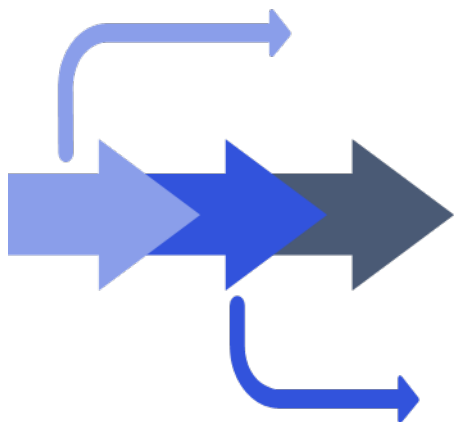
References

References

Documents	
Title	Number
Qualcomm Technologies, Inc.	
<i>SDM845 Sensors Overview</i>	80-P9301-34
<i>SM7150, SM6150/SM6150P, SM6125, SDM712/SDM710/SDM670/QCS605 Sensors Overview</i>	80-PD126-9
<i>SM8150, SDX50+SM8150, and SDX55M+SM8150P Sensors Overview</i>	80-PF777-23
<i>Adding a Custom Sensors Algorithm with Sensors Execution Environment (SEE)</i>	80-P9301-67
<i>Sensors Execution Environment Client API Reference</i>	80-P9301-36
<i>Qualcomm AAH Compatible Driver List for SEE</i>	80-NB925-2
<i>Unified Sensor Test Application (USTA) User Guide</i>	80-P9301-85
Resources	
Google protocol buffers	https://developers.google.com/protocol-buffers/
Nanopb	https://jpa.kapsi.fi/nanopb/

References (cont.)

Acronyms	
Acronym or term	Definition
AAH	All-Ways Aware Hub
ASCP	Asynchronous COM port
DDF	Device driver framework
HAL	Hardware abstraction layer
PD	Protection domain
RH	Report handler
SAM	Sensors algorithm manager
SEE	Sensors Execution Environment
SLPI	Sensors Low Power Island
SMGR	Sensors manager
SSC	Snapdragon Sensors Core
SSR	Subsystem restart
SUID	Sensor unique identifier



Qualcomm
2019-04-14 20:26:28 PDT
zk_sw@wingtech.com

Questions?

<https://createpoint.qti.qualcomm.com>
