Qualcomm Technologies, Inc.

Qualcomm Linux Sensors Guide - Addendum

80-70018-7A AA

April 2, 2025

Contents

1 Sensors overview

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

This addendum provides supplementary information about sensor interfaces, bring up, development, configuration, and debugging. This addendum is available to licensed users with authorized access.

Read this addendum along with the Qualcomm Linux Sensors Guide, which provides basic information about sensor features, interfaces, software, samples, tools, calibration, and verification.

1.1 QSH APIs

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

The Qualcomm sensing hub (QSH) framework runs on low-power application digital signal processor (aDSP) and provides APIs at the application processor. The QSH client APIs are used for the applications and the QSH sensor APIs are used to develop new sensors on a low-power processor. For more information, see QSH client APIs in Qualcomm Linux Sensors Guide.

QSH direct channel APIs

The following QSH direct channel APIs provide low latency sensor data handling for the high rate applications. The QSH direct channel APIs use FastRPC, which bypasses the standard IPC mechanism to achieve faster communication.

sns_direct_channel_open()

Opens a communication channel between the client and the QSH framework. The client must close and re-open the channel during the subsystem restart (SSR) or protection domain restart (PDR) using the `sns_direct_channel_close` and `sns_direct_channel_open` APIs.

```
int sns_direct_channel_open(const char* uri, remote_handle64* h)
```

Parameter	Name	Description
input	uri	**Helps FastRPC route to:** • sensors DSP SLPI sns_direct_channel_URI"&_dom=sdsp" • aDSP sns_direct_channel_URI"&_dom=adsp"
output	_h	Handle provided by the direct channel client manager.

Return	Description
0	The operation was successful.
All other values	The operation failed due to an unknown error.

sns_direct_channel_create()

Creates a direct channel with the direct channel client manager.

```
int sns_direct_channel_create(remote_handle64 _h, const unsigned
char* settings,
                int settingsLen, int *channel_handle)
```

Parameter	Name	Description
input	_h	remote_handle64 from the sns_direct_channel_open API.
input	Settings	Protocol buffer (Pb) encoded sns_direct_channel_req_msg raw bytes.
input	settingsLen	Pb encoded buffer length.
output	Handle	Unique channel handle for configuration.

Return	Description
0	The operation was successful.
All other values	The operation failed due to an unknown error.

sns_direct_channel_config()

Configures the channel with the standard sensor request or updates the existing channel runtime parameters.

```
int sns_direct_channel_config(remote_handle64 _h, int handle, const
unsigned char* config_request, int config_requestLen)
```

Parameter	Name	Description
input	_h	remote_handle64 from the sns_direct_channel_open API.
input	config_request	Pb encoded sns_direct_channel_config_msg.
input	config_requestLen	Pb encoded buffer length.
input	Handle	Channel handle from the sns_direct_channel_create API.

Return	Description
0	The operation was successful.
All other values	The operation failed due to an unknown error.

sns_direct_channel_delete()

Deletes the created sensor handle.

```
int sns_direct_channel_delete(remote_handle64 _h, int handle)
```

Parameter	Name	Description
input	_h	remote_handle64 from the sns_direct_channel_open API.
input	Handle	Channel handle from the sns_direct_channel_create API.

Return	Description
0	The operation was successful.
All other values	The operation failed due to an unknown error.

sns_direct_channel_close()

Closes the channels associated with the client handle.

```
int sns_direct_channel_close(remote_handle64 _h)
```

Parameter	Name	Description
input	_h	remote_handle64 from the sns_direct_channel_open API.

Return	Description
0	The operation was successful.
All other values	The operation failed due to an unknown error.

For more information and example codes, see Develop sensors.

Low-power processor APIs

Provides access to the following:

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

You can use the low-power processor APIs to create algorithms and sensor drivers that comply with the QSH standards.

Register APIs

The following API binds the sensor with the QSH framework. Each sensor library must implement a user-defined registration function. QSH invokes this sensor library register API during the sensor library initialization.

```
sns_register_cb const *register_api
```

Parameter	Name	Description
input	register_api	Framework callback function, which the library calls to register the supported sensors.

Return	Description
sns_rc	• SNS_RC_INVALID_STATE: Hardware and/or software requirements aren't met. • SNS_RC_SUCCESS

struct sns_register_cb		
Data field		**Description**
uint32_t	struct_len;	–
sns_rc	(*init_ sensor)(uint32_t state_len, struct sns_sensor_api const *sensor_api, struct sns_sensor_instance_ api const *instance_ api);	Allocates state_len bytes of memory and initializes the sensor.

The QSH framework implements the init_sensor callback function. Each sensor in the library invokes init_sensor function to provide a sensor and sensor instance API along with the sensor private state size to the QSH framework.

```
sns_rc init_sensor (uint32_t state_len,
            struct sns_sensor_api const *sensor_api,
            struct sns_sensor_instance_api const *instance_api)
```

Parameter	Name	Description
input	state_len	Number of bytes required to store the sensor context.
input	sensor_api	APIs implemented by the sensor.
input	instance_api	APIs implemented by the sensor instance.

Return	Description
sns_rc	• SNS_RC_POLICY: state_len is too large. • SNS_RC_NOT_AVAILABLE: Sensor UID is already in-use. • SNS_RC_FAILED: Sensor initialization failed. • SNS_RC_SUCCESS: Sensor initialization is successful.

Sensor data structure APIs

The following sensor data structures reside on the system and are created during sensor initialization by the QSH framework.

struct sns_sensor		
Data field		**Description**
struct sns_sensor_cb	const *cb;	Callbacks provided by the framework to the sensor.
struct sns_sensor_api	const *sensor_api;	API implementation provided for and by this sensor.
struct sns_sensor_instance_api	const *instance_api;	Associated API for a sensor instance created for and by this sensor.
struct sns_sensor_state	*state;	State space allocated by the framework for the sensor developer.

Sensor framework callbacks

The QSH framework provides the following callbacks for the framework functionality.

struct sns_sensor_cb		
Data field		**Description**
uint32_t	struct_len;	Size of the structure.
struct sns_service_manager*	(*get_service_manager)(sns_sensor const *sensor);	Get a handle to the service manager. Framework utility services are obtained using this handle.
struct sns_sensor_instance*	(*get_sensor_instance)(sns_sensor const *sensor, bool first);	Returns the sensor instance associated with this sensor.
struct sns_sensor_instance*	(*create_instance)(sns_sensor *sensor, uint32_t state_len);	Allocates memory and initializes a new sensor instance to be associated with this sensor.

struct sns_sensor_cb		
Data field		**Description**
`void`	`(*remove_instance)(struct sns_sensor_instance *instance);`	Removes and de-allocates memory of the sensor instance.
`struct sns_sensor*`	`(*get_library_sensor)(sns_sensor const *sensor,bool first);`	Returns the sensor associated with this sensor library.
`uint32_t`	`(*get_registration_index)( sns_sensor const *sensor);`	If multiple copies of this sensor library are registered with QSH, then this returns the index (starting at **0**) of this copy.
`struct sns_sensor_instance*`	`(*create_instance_v2)(sns_sensor *sensor,uint32_t state_len,uint32_t state_len_ni);`	Allocates and initializes a new sensor instance to be associated with this sensor.

```
struct sns_service_manager* (*get_service_manager)(sns_sensor const
*sensor);
```

Parameter	Name	Description
`input`	Sensor	Sensor reference

Return	Description
`struct sns_service_manager*`	Service manager handle

```
struct sns_sensor_instance* (*get_sensor_instance)(sns_sensor const
*sensor, bool first);
```

Parameter	Name	Description
`input`	Sensor	Sensor reference
`input`	First	Returns the first sensor instance

Return	Description
`struct sns_sensor_instance*`	Next sensor instance associated with this sensor

```
struct sns_sensor_instance* (*create_instance)(sns_sensor *sensor,
uint32_t state_len);
```

Parameter	Name	Description
`input`	Sensor	Sensor reference
`input`	state_len	Memory size to be allocated for the sensor instance state

Return	Description
`struct sns_sensor_instance*`	Newly created sensor instance

```
void (*remove_instance)(struct sns_sensor_instance *instance);
```

Parameter	Name	Description
`input`	Instance	Instance received within `set_client_request`

Return	Description
`void`	NULL

```
struct sns_sensor* (*get_library_sensor)(sns_sensor const *sensor,
bool first);
```

Parameter	Name	Description
`input`	Sensor	Sensor reference
`input`	First	Returns the first sensor in the library

Return	Description
`struct sns_sensor*`	Next sensor associated with this sensor library

```
uint32_t (*get_registration_index)(sns_sensor const *sensor);
```

Parameter	Name	Description
`input`	Sensor	Sensor reference

Return	Description
uint32_t	Sensor library registration index

```
struct sns_sensor_instance*(*create_instance_v2)
                          (sns_sensor *sensor,
                              uint32_t state_len,
                                  uint32_t state_len_ni);
```

Parameter	Description
sensor	Sensor reference.
state_len	Allocation size, for the instance island state, if the sensor is supported in an island.
state_len_ni	Allocation size, for the instance non-island state.

Return	Description
struct sns_sensor_instance	Newly created sensor instance

Sensor APIs

Each sensor must implement the following functions, which are exclusively called by the QSH framework to control the core functionality of the sensor.

struct sns_sensor_api		
Data field		**Description**
uint32_t	``struct_len;``	Size of the structure.
sns_rc	``(*init)(sns_sensor *const sensor);``	Initializes a sensor to its hard coded/default state.
sns_rc	(*deinit)(sns_sensor *const sensor);	Releases all hardware and software resources associated with this sensor.
sns_sensor_ uid const*	(*get_sensor_ uid)(sns_sensor const *const sensor);	The unique 128-bit identifier defined by the sensor.
sns_rc	(*notify_event)(sns_ sensor *const sensor);	Incoming notification to the sensor.

struct sns_sensor_api		
Data field		**Description**
`struct sns_sensor_instance*`	`(*set_client_request)(sns_sensor *const sensor, struct sns_request const *exist_request, struct sns_request const *new_request, bool remove);`	Add, remove, or update a client request to this sensor.

```
sns_rc (*init)(sns_sensor *const sensor);
```

Parameter	Name	Description
`input`	Sensor	Sensor reference

Return	Description
`sns_rc`	• `SNS_RC_INVALID_STATE`: Required hardware isn't available. • `SNS_RC_POLICY`: Required services aren't available. • `SNS_RC_SUCCESS`: Successful.

```
sns_rc (*deinit)(sns_sensor *const sensor);
```

Parameter	Name	Description
`input`	Sensor	Sensor reference

Return	Description
struct sns_sensor_instance	• SNS_RC_INVALID_STATE: Error occurred; some resources can't be released. • SNS_RC_SUCCESS: Successful.

```
sns_sensor_uid const* (*get_sensor_uid)(sns_sensor const *const
sensor);
```

Parameter	Name	Description
input	Sensor	Sensor reference

Return	Description
sns_sensor_uid const*	The unique 128-bit identifier for this sensor.

```
sns_rc (*notify_event)(sns_sensor *const sensor);
```

Parameter	Name	Description
input	Sensor	Sensor reference

Return	Description
struct sns_sensor_instance	• SNS_RC_INVALID_STATE: A client error occurred. The framework may destroy the client. • SNS_RC_NOT_AVAILABLE: A transitory error occurred. The framework may remove all outstanding input. • SNS_RC_INVALID_LIBRARY_STATE: A permanent error occurred. The framework may destroy all sensors present in the client library. • SNS_RC_SUCCESS: Successful.

```
struct sns_sensor_instance* (*set_client_request)(sns_sensor *const
sensor,
```

```
struct sns_request const *exist_request,
                struct sns_request const *new_request,
              bool remove);
```

Parameter	Name	Description
input	Sensor	Sensor reference.
input	exist_request	The sensor instance handles the existing request.
input	new_request	New request from the client.
input	Remove	The client doesn't require the sensor data.

Return	Description
struct sns_sensor_instance	The sensor instance handles the request.

Sensor instance APIs

The sensor instance must implement the following functions, which the sensor framework exclusively calls.

struct sns_sensor_instance		
Data field		**Description**
struct sns_sensor_instance_cb	const *cb;	Functions that call back into the framework; provided by the framework.
struct sns_sensor_instance_state	*state;	If the sensor is enabled in the island, then the state of the sensor instance is placed in the island region.
struct sns_sensor_instance_state	*state_ni;	Non-island state of the sensor instance.

The following table lists the function callback provided by the framework and called by the sensor instance.

struct sns_sensor_instance_cb		
Data field		**Description**
uint32_t	struct_len;	Size of the structure.
struct sns_service_manager*	(*get_service_manager)(sns_sensor_instance *instance);	Gets a handle to the service manager. Using this handle, the framework utility services are obtained.
struct sns_request const*	(*get_client_request)(sns_sensor_instance *instance, sns_sensor_uid const *suid, bool first);	Returns the next client request associated with this sensor instance and SUID.
void	(*remove_client_request)(sns_sensor_instance *instance, struct sns_request const *request);	Removes a client request from this sensor instance.
void	(*add_client_request)(sns_sensor_instance *instance, struct sns_request const *request);	Adds a client request to this sensor instance.

```
struct sns_service_manager* (*get_service_manager)(sns_sensor_
instance *instance);
```

Parameter	Name	Description
input	Instance	Sensor instance reference

Return	Description
struct sns_service_manager*	Service manager reference

```
struct sns_request const* (*get_client_request)(
                    sns_sensor_instance *instance,
                        sns_sensor_uid const *suid,
                            bool first);
```

Parameter	Name	Description
input	Instance	Sensor instance reference.
input	SUID	Sensor associated with this instance.
input	First	Returns the first request, if the value of the *first* parameter is set to True, otherwise returns the next request.

Return	Description
struct sns_request const*	NULL or request pointer

```
void (*remove_client_request)(
            sns_sensor_instance *instance,
                struct sns_request const *request);
```

Parameter	Name	Description
input	Instance	Sensor instance reference
input	Request	Client request to be removed

Return	Description
void	NULL

```
void (*add_client_request)(sns_sensor_instance *instance,
    struct sns_request const *request);
```

Parameter	Name	Description
input	Instance	Sensor instance reference
input	Request	Client request to be added

Return	Description
void	NULL

The following state is allocated for the sensor instance to be used by the sensor developer.

struct sns_sensor_instance_state		
Data field		**Description**
uint32_t	state_len;	Allocation size for sns_sensor_instance state
uint64_t	state[1];	Sensor instance state pointer

sns_sensor_instance_api		
Data field		**Description**
uint32_t	struct_len;	Size of the structure.
sns_rc	(*init)(sns_sensor_instance *const instance, sns_sensor_state const *sensor_state);	Initializes a sensor instance to its default state.
sns_rc	(*deinit)(sns_sensor_instance *const instance);	Releases all the hardware and software resources associated with this sensor instances.
sns_rc	(*set_client_config)(sns_sensor_instance *const instance, struct sns_request const *client_request);	Updates a sensor instance configuration to this sensor request.
sns_rc	(*notify_event)(sns_sensor_instance *const instance);	Incoming notification to the sensor instance.

The sensor and the framework exclusively call these functions, which every sensor instance must implement.

```
sns_rc (*init)(sns_sensor_instance *const instance,
                    sns_sensor_state const *sensor_state);
```

Parameter	Name	Description
input	Instance	Sensor instance reference
input	sensor_state	State of the sensor that created this instance

Return	Description
sns_rc	• SNS_RC_NOT_AVAILABLE: Sensor state doesn't allow this operation • SNS_RC_SUCCESS: Successful

```
sns_rc (*deinit)(sns_sensor_instance *const instance);
```

Parameter	Name	Description
input	Instance	Sensor instance reference

Return	Description
sns_rc	• SNS_RC_NOT_AVAILABLE: Error occurred; some resources can't be released. • SNS_RC_SUCCESS: Successful.

```
sns_rc (*set_client_config)(
            sns_sensor_instance *const instance,
               struct sns_request const *client_request);
```

Parameter	Name	Description
input	Instance	Sensor instance reference
input	client_request	Client request associated with the instance

Return	Description
sns_rc	• SNS_RC_INVALID_VALUE: Invalid client request • SNS_RC_SUCCESS: Successful.

```
sns_rc (*notify_event)(
    sns_sensor_instance *const instance);
```

Parameter	Name	Description
input	Instance	Sensor instance reference

Return	Description
sns_rc	• SNS_RC_INVALID_STATE: A client error occurred. The framework might destroy the client. • SNS_RC_NOT_AVAILABLE: A transitory error occurred. The framework might remove all the outstanding input. • SNS_RC_SUCCESS: Successful.

Struct sns_sensor_state		
Data field		**Description**
uint32_t	state_len;	Allocation size for the sensor instance.
uint64_t	state[1];	–

QSH services

The following is the list of service APIs to enable the core functionalities of the framework.

Service manager APIs

The service manager APIs provides access to all the service objects. Sensors must send a query each time for the required service.

sns_service_manager		
Data field		**Description**
uint32_t	struct_len;	Size of the structure
sns_service*	(*get_service)(struct sns_service_manager *manager, sns_service_type service);	Query the framework for a particular service

The following table lists the enumeration of services provided by the framework.

Table : Enumeration of services provided by the framework

enum sns_service_type
SNS_STREAM_SERVICE
SNS_ATTRIBUTE_SERVICE
SNS_EVENT_SERVICE
SNS_POWER_RAIL_SERVICE
SNS_DIAG_SERVICE
SNS_SYNC_COM_PORT_SERVICE
SNS_GPIO_SERVICE
SNS_ISLAND_SERVICE
SNS_REGISTRATION_SERVICE
SNS_POWER_MGR_SERVICE
SNS_FILE_SERVICE

```
sns_service* (*get_service)(
struct sns_service_manager *manager,
sns_service_type service);
```

Parameter	Name	Description
input	Manager	Reference to the service manager.
input	Service	The type of service to query according to the service type listed in Table : Enumeration of services provided by the framework.

Return	Description
sns_service	Service reference, or NULL if not present

Note: To get a handle to the service manager, use the `get_service_manager()` API as a prerequisite. The framework utility services are obtained using this handle.

Stream service APIs

The stream service APIs manage the stream creation, and removal for sensors and sensor instances.

sns_stream_service_api		
Data field		**Description**
uint32_t	struct_len;	Size of the structure.
sns_rc	(*create_sensor_stream)(sns_stream_service *service, struct sns_ sensor *sensor, sns_ sensor_uid sensor_uid, struct sns_data_stream **data_stream);	Initializes a new stream to be used by a sensor.
sns_rc	(*create_sensor_instance_ stream)(sns_stream_ service *service, struct sns_sensor_instance *instance, sns_sensor_ uid sensor_uid, struct sns_data_stream **data_ stream);	Initializes a new stream to be used by a sensor instance.
sns_rc	(*remove_stream)(sns_ stream_service *service, struct sns_data_stream *data_stream);	Removes a stream created by a sensor or sensor instance.

```
sns_rc (*create_sensor_stream)(
    sns_stream_service *service,
    struct sns_sensor *sensor,
    sns_sensor_uid sensor_uid,
    struct sns_data_stream **data_stream);
```

Parameter	Name	Description
input	Service	Stream service reference.
input	Sensor	Destination sensor creating the stream.
input	sensor_uid	SUID of the data source sensor.
output	data_stream	Stream references allocated by the framework.

Return	Description
sns_rc	• SNS_RC_POLICY: The sensor has exceeded the maximum stream count. • SNS_RC_NOT_AVAILABLE: The specified SUID isn't presently available. • SNS_RC_SUCCESS

```
sns_rc (*create_sensor_instance_stream)(
            sns_stream_service *service,
             struct sns_sensor_instance *instance,
                    sns_sensor_uid sensor_uid,
                        struct sns_data_stream **data_stream);
```

Parameter	Name	Description
input	Service	Stream service reference
input	Instance	Destination instance that creates the stream
input	sensor_uid	SUID of the data source sensor
output	data_stream	Stream references allocated by the framework

Return	Description
sns_rc	• SNS_RC_POLICY: The sensor has exceeded the maximum stream count. • SNS_RC_NOT_AVAILABLE: The specified SUID isn't presently available. • SNS_RC_SUCCESS

```
sns_rc (*remove_stream)(
    sns_stream_service *service,
    struct sns_data_stream *data_stream);
```

Parameter	Name	Description
input	Service	Stream service reference
input	data_stream	Data stream to be removed and freed

Return	Description
sns_rc	SNS_RC_SUCCESS

Attribute service APIs

All the sensors publish a series of attributes, where each attribute is a key-value pair. The key is an enum value that uniquely identifies the attribute while the value is a protocol-buffer-encoded message of type sns_std_attr.

```
typedef uint32_t sns_attribute_id;
```

sns_attribute_service		
Data field		**Description**
sns_service	service;	Service information.
struct sns_attribute_service_api	*api;	Public API provided by the framework to be used by the sensor.

The following table lists the APIs made available to the sensors to manage their published attributes.

sns_attribute_service_api		
Data field		**Description**
uint32_t	struct_len;	Size of the structure
sns_rc	(*publish_attribute)(sns_attribute_service *service, struct sns_ sensor *sensor, void const *attribute, uint32_t attribute_ len, sns_attribute_ id attribute_id, bool completed);	Publishes a new or an updated attribute

```
sns_rc (*publish_attribute)(
    sns_attribute_service *service,
    struct sns_sensor *sensor,
```

```
    void const *attribute,
    uint32_t attribute_len,
    sns_attribute_id attribute_id,
    bool completed);
```

Parameter	Name	Description
input/output	Service	Attribute service reference
input	Sensor	The sensor for which attributes are published
input	Attribute	Encoded attribute (sns_std_attr) to publish
input	attribute_len	Length of the encoded attribute buffer
input	attribute_id	ID corresponding to the attribute
input	Completed	Specifies if it's the last attribute in the publish set

Return	Description
sns_rc	• SNS_RC_SUCCESS • SNS_RC_INVALID_TYPE: Attribute ID out of valid range • SNS_RC_POLICY: Attribute is rejected due to size • SNS_RC_INVALID_STATE: Encoding error

Diagnostic service APIs

The diagnostic service APIs manages the diagnostic services for sensors and sensor instances.

sns_diag_service		
Data field		**Description**
sns_service	service;	Service information
Struct sns_diag_service_api	*api;	Public API provided by the framework to be used by the sensor

The following enumeration of diagnostics services are provided by the framework.

- enum sns_diag_sensor_state_log
- SNS_DIAG_SENSOR_STATE_LOG_INTERNAL = 0
- SNS_DIAG_SENSOR_STATE_LOG_RAW = 1
- SNS_DIAG_SENSOR_STATE_LOG_INTERRUPT = 2

sns_diag_service_api		
Data field		**Description**
uint32_t	struct_len;	Size of the structure.
uint32_t	(*get_max_log_size)(sns_diag_service const *service);	Returns the maximum log packet size supported by the framework.
void*	(*alloc_log)(sns_diag_service *service, struct sns_sensor_instance *instance, struct sns_sensor_uid const *sensor_uid, uint32_t log_size, sns_diag_sensor_state_log log_type);	Allocates memory for a log packet. Memory allocated by this function is used to store log packet information and is returned as a part of sns_diag_encode_log_cb for encoding.
sns_rc	(*submit_log)(sns_diag_service *service, struct sns_sensor_instance *instance, struct sns_sensor_uid const *sensor_uid, uint32_t log_size, void *log, sns_diag_sensor_state_log log_type, uint32_t encoded_log_size, sns_diag_encode_log_cb encode_log_cb);	Submit a log packet.

sns_diag_service_api		
Data field		**Description**
`void`	`(*sensor_printf_v2)( sns_diag_service *service, struct sns_sensor *sensor, struct sns_msg_const_ type const *msg_ struct, uint32_t nargs, ...);`	It is a `printf()` style function to print a debug message from a sensor. It has limited printf formatting support (supports only a 32-bit int argument).
`void`	`(*sensor_inst_printf_ v2)( sns_diag_service *service, struct sns_sensor_instance *instance, struct sns_msg_const_type const *msg_struct, uint32_t nargs, ...);`	It is a `printf()` style function to print a debug message from a sensor instance. It has limited printf formatting support (supports only a 32-bit int argument).

```
uint32_t (*get_max_log_size)(sns_diag_service const *service);
```

Parameter	Name	Description
`input`	Service	Diag service reference

Return	Description
`uint32_t`	Maximum size of a log packet supported by the framework (in bytes).

```
void* (*alloc_log)(
    sns_diag_service                *service,
    struct sns_sensor_instance      *instance,
    struct sns_sensor_uid const     *sensor_uid,
    uint32_t                        log_size,
    sns_diag_sensor_state_log       log_type);
```

Parameter	Name	Description
`input`	Service	Diag service reference
`input`	Instance	Instance pointer of the sensor

Parameter	Name	Description
input	sensor_uid	UID of the sensor for which a log packet is being created
input	log_size	Requested size in bytes to store log packet information
input	log_type	Type of log packet

Return	Description
–	Allocated memory; returns NULL if out of memory (OOM) or if logging is disabled for this sensor.

```
sns_rc (*submit_log)(
   sns_diag_service                 *service,
   struct sns_sensor_instance       *instance,
   struct sns_sensor_uid const      *sensor_uid,
   uint32_t                         log_size,
   void                             *log,
   sns_diag_sensor_state_log        log_type,
   uint32_t                         encoded_log_size,
   sns_diag_encode_log_cb           encode_log_cb)
```

Parameter	Name	Description
input	Service	Diag service reference
input	Instance	Instance pointer of the sensor submitting the log
input	sensor_uid	UID of the sensor submitting the log
input	log_size	Size of the log packet information (in bytes)
input	Log	Pointer to the log packet information
input	log_type	Type of log
input	encoded_log_size	Size of the encoded log (in bytes)
input	encode_log_cb	Function used to encode the log

Return	Description
sns_rc	• SNS_RC_NOT_AVAILABLE: The log packet is disabled. • SNS_RC_INVALID_TYPE: The SUID isn't recognized. • SNS_RC_SUCCESS: The log packet was successfully submitted.

```
void (*sensor_printf_v2)(
sns diag service              *service,
struct sns sensor            *sensor,
struct sns_msg_const_type const  *msg_struct,
uint32 t                    nargs, ...);
```

Parameter	Name	Description
input	Service	Diag service reference.
input	Sensor	Sensor pointer of the sns_sensor printing this message.
input	msg_struct	Diag defined static const msg_v2/v4_const_type for the message.
input	nargs	Number of arguments included in "...".

Return	Description
void	None

```
void (*sensor_inst_printf_v2)(
sns diag service                *service,
struct sns sensor instance        *instance,
struct sns_msg_const_type const  *msg_struct,
uint32 t                    nargs, ...);
```

Parameter	Name	Description
input	Service	Diag service reference.
input	Instance	Sensor instance pointer to the sensor instance printing this message.
input	msg_struct	Diag defined static const msg_v2/v4_const_type for the message.
input	nargs	Number of arguments included in "...".

Return	Description
void	None

Event service APIs

The following event service APIs allows the sensor instances to publish events.

sns_event_service		
Data field		**Description**
sns_service	service;	Service information.
struct sns_event_service_api	*api;	Public API provided by the framework to be used by the sensor.

sns_event_service_api		
Data field		**Description**
uint32_t	struct_len;	Size of the structure.
uint32_t	(*get_max_event_size)(sns_event_service const *service);	Gets the maximum event size supported by the framework.
struct sns_sensor_event*	(*alloc_event)(sns_event_service *service, struct sns_sensor_instance *instance, uint32_t event_len);	Allocates an empty event buffer.
sns_rc	(*publish_event)(sns_event_service *service, struct sns_sensor_instance *instance, struct sns_sensor_event *event, struct sns_sensor_uid const *sensor_uid);	Publishes an event, which alloc_event allocated previously to all the registered clients.

sns_event_service_api		
Data field		**Description**
sns_rc	(*publish_error)(sns_event_service *service, struct sns_sensor_instance *instance, sns_rc reason);	Publishes a generic error event that is delivered to all the active clients.

```
uint32_t (*get_max_event_size)(sns_event_service const *service);
```

Parameter	Name	Description
input	Service	Event service reference.

Return	Description
uint32_t	Maximum event size that's supported by the framework.

```
struct sns_sensor_event* (*alloc_event)(
    sns_event_service *service,
    struct sns_sensor_instance *instance,
    uint32_t event_len);
```

Parameter	Name	Description
input	Service	Event service reference.
input	Instance	Sensor instance that forms/publishes the event.
input	event_len	Buffer space to allocate for the event.

Return	Description
structsns_sensor_event*	All allocations are guaranteed to succeed unless event_len exceeds get_max_buffer_size().

```
sns_rc (*publish_event)(
    sns_event_service *service,
    struct sns_sensor_instance *instance,
    struct sns_sensor_event *event,
    struct sns_sensor_uid const *sensor_uid);
```

Parameter	Name	Description
input	Service	Event service reference.
input	Instance	Sensor instance that formed the event.
input	Event	Event to publish to all the registered clients.
input	sensor_uid	SUID of this published data. If set to NULL, the framework determines SUID from an instance parameter.

Return	Description
sns_rc	SNS_RC_SUCCESS: Successful.

```
sns_rc (*publish_error)(
    sns_event_service *service,
    struct sns_sensor_instance *instance,
    sns_rc reason);
```

Parameter	Name	Description
input/ouput	Service	Event service reference.
input	Instance	Sensor instance that formed the event.
input	Reason	• SNS_RC_INVALID_STATE: The sensor instance entered an invalid state. • SNS_RC_NOT_AVAILABLE: A software and/or hardware dependency is lost.

Return	Description
sns_rc	SNS_RC_SUCCESS: Successful.

Power rail service APIs

The following are the power rail management APIs.

sns_power_rail_service		
Data field		**Description**
sns_service	service;	Service information
struct sns_pwr_rail_service_api	*api;	Public API provided by the framework to be used by the sensor

sns_power_rail_service_api		
Data field		**Description**
uint32_t	struct_len;	Size of the structure.
sns_rc	(*sns_register_power_rails)(sns_pwr_rail_service* service,sns_rail_config const* rail_config);	Registers power rails for a physical sensor.
sns_rc	(*sns_vote_power_rail_update)(sns_pwr_rail_service* service, struct sns_sensor const* sensor, sns_rail_config const* rails_config, sns_time* rail_on_timestamp);	Votes for a power rail status change.

The following are the enumeration of power rail services:

- enum sns_power_rail_state
- SNS_RAIL_OFF
- SNS_RAIL_ON_LPM

- SNS_RAIL_ON_NPM

Table: Rail name definition

sns_rail_name		
Data field		**Description**
char	name[RAIL_NAME_STRING_SIZE_MAX];	RAIL_NAME_STRING_SIZE_MAX is 30

Table: Rail configuration definition

sns_rail_name		
Data field		**Description**
sns_power_rail_state	rail_vote;	• Overall vote for all the sensor rails. • At any time, all the rails for a sensor can have a single state from sns_power_rail_state.
uint8_t	num_of_rails	Number of rails in the sns_rail_name array.
sns_rail_name	rails[RAIL_NUMBER_MAX];	Array of rail names. These rails are power rails connected to the sensor hardware.


```
sns_rc (*sns_register_power_rails)(sns_pwr_rail_service* service,
sns_rail_config const* rail_config);
```

Parameter	Name	Description
input	*service	Power rail service reference
input	rail_config	Rail configuration that's being registered

Return	Description
sns_rc	• SNS_RC_INVALID_VALUE: Input rail configuration is invalid • SNS_RC_SUCCESS: Rail registration is successful

```
sns_rc (*sns_vote_power_rail_update)(sns_pwr_rail_service* service,
                                     struct sns_sensor const*
sensor,
                                     sns_rail_config const* rails_
config,
                                     sns_time* rail_on_timestamp);
```

Parameter	Name	Description
input	service	Power rail service reference.
input	sensor	Sensor requesting rail update.
input	rails_config	Rails and their status change vote.
output	rail_on_timestamp	• The timestamp in ticks when the rail is turned ON. • When the rails are turned OFF, the client can choose not to get the rail_on_timestamp information.

Return	Description
sns_rc	• SNS_RC_NOT_SUPPORTED: Requested rail isn't registered • SNS_RC_SUCCESS: Operation is successful

Synchronous COM port service APIs

The following are the Synchronous COM port (SCP) APIs that manage the SCP service.

```
typedef void * sns_sync_com_port_handle;
```

sns_sync_com_port_service		
Data field		**Description**
sns_service	service;	Service information.
struct sns_sync_com_port_service_api	*api;	Public API provided by the framework for the sensor.

Table : COM port version struct

sns_sync_com_port_version		
Data field		**Description**
uint16_t	major;	Service information.
uint16_t	minor;	Public API provided by the framework for the sensor.

sns_sync_com_port_service_api		
Data field		**Description**
uint32_t	struct_len;	Size of the structure.
sns_rc	(*sns_scp_register_com_port)(sns_com_port_config const *com_config,sns_sync_com_port_handle **port_handle);	Registers a bus COM port with the SCP utility.

sns_sync_com_port_service_api		
Data field		**Description**
`sns_rc`	`(*sns_scp_deregister_com_port)(sns_sync_com_port_handle **port_handle);`	• Deregisters a bus COM port with the SCP utility and set the port handle to NULL. • If powered on or opened, this powers OFF and closes the COM port.
`sns_rc`	`(*sns_scp_get_version)(sns_sync_com_port_version *version);`	Gets the version of the SCP API.
`sns_rc`	`(*sns_scp_open)(sns_sync_com_port_handle *port_handle);`	Opens a new COM port with the bus configuration `com_config`.
`sns_rc`	`(*sns_scp_close)(sns_sync_com_port_handle *port_handle);`	Closes and switches off the COM port.
`sns_rc`	`(*sns_scp_update_bus_power)(sns_sync_com_port_handle *port_handle, bool power_bus_on);`	Updates the bus power status to ON or OFF.
`sns_rc`	`(*sns_scp_register_rw)(sns_sync_com_port_handle *port_handle, sns_port_vector *vectors, int32_t num_vectors, save_write_time, uint32_t *xfer_bytes);`	Reads and/or writes multiple registers.

sns_sync_com_port_service_api		
Data field		**Description**
sns_rc	(*sns_scp_register_rw_ex)(sns_sync_com_port_handle *port_handle, sns_com_port_config_ex *com_port_ex, _port_vector *vectors, int32_t num_vectors, bool save_write_time, uint32_t *xfer_bytes);	Reads and/or writes multiple registers.
sns_rc	(*sns_scp_simple_rw)(sns_sync_com_port_handle *port_handle, bool is_write, bool save_write_time, uint8_t *buffer, uint32_t bytes, uint32_t *xfer_bytes);	Reads or writes operation directly to the slave. This operation is valid only on a port that is successfully opened using the sns_scp_open() API.
sns_rc	(*sns_scp_get_write_time)(sns_sync_com_port_handle *port_handle, sns_time *write_time);	Gets the timestamp for the most recent write operation.
sns_rc	(*sns_scp_issue_ccc)(sns_sync_com_port_handle *port_handle, sns_sync_com_port_ccc ccc_cmd, uint8_t *buffer, uint32_t bytes, uint32_t *xfer_bytes);	Issues a direct I3C common command code (CCC) to a slave.

sns_sync_com_port_service_api		
Data field		**Description**
sns_rc	(*sns_scp_rw)(sns_sync_com_port_handle *port_handle, uint8_t* read_buffer, uint32_t read_len_bytes, const uint8_t* write_buffer, uint32_t write_len_bytes, uint8_tbits_per_word);	• Full duplex read /write operation directly to the slave. • The operation is valid only on a port that is successfully opened using the sns_scp_open() API and supports full duplex transfers.

```
sns_rc (*sns_scp_register_com_port)(sns_com_port_config const *com_
config,
   sns_sync_com_port_handle **port_handle);
```

Parameter	Name	Description
input	com_config	COM port configuration for the bus.
output	port_handle	Port handle for the bus.

Return	Description
sns_rc	• SNS_RC_INVALID_VALUE: Input parameters are invalid • SNS_RC_FAILED: COM port registration failed • SNS_RC_SUCCESS: Action succeeded

```
sns_rc (*sns_scp_deregister_com_port)(sns_sync_com_port_handle
**port_handle);
```

Parameter	Name	Description
input	port_handle	Reference to the port handle

Return	Description
sns_rc	• SNS_RC_INVALID_VALUE: If the port handle is invalid • SNS_RC_SUCCESS: Port handle is freed

```
sns_rc (*sns_scp_get_version)(sns_sync_com_port_version *version);
```

Parameter	Name	Description
input	version	Version of this API

Return	Description
sns_rc	• SNS_RC_INVALID_VALUE: Version parameter is NULL • SNS_RC_SUCCESS: Action succeeded

```
sns_rc (*sns_scp_open)(sns_sync_com_port_handle *port_handle);
```

Parameter	Name	Description
input	port_handle	Port handle for the bus

Return	Description
sns_rc	• SNS_RC_INVALID_VALUE: Requested bus and/or port handles are invalid • SNS_RC_FAILED: Unable to open underlying bus • SNS_RC_SUCCESS: Action succeeded or already open

```
sns_rc (*sns_scp_close)(sns_sync_com_port_handle *port_handle);
```

Parameter	Name	Description
input	port_handle	Port handle for the bus

Return	Description
sns_rc	• SNS_RC_INVALID_VALUE: Requested bus and/or port handles are invalid • SNS_RC_FAILED: Unable to open underlying bus • SNS_RC_SUCCESS: Action succeeded or already open

```
sns_rc (*sns_scp_update_bus_power)(sns_sync_com_port_handle *port_
handle, bool power_bus_on);
```

Parameter	Name	Description
input	port_handle	Port handle for the bus
input	power_bus_on	True to power on

Return	Description
sns_rc	• SNS_RC_INVALID_VALUE: The passed in port handle is invalid. • SNS_RC_FAILED: Unable to update the bus power. • SNS_RC_SUCCESS: Action succeeded, or no update needed.

```
sns_rc (*sns_scp_register_rw)(sns_sync_com_port_handle *port_handle,
                              sns_port_vector *vectors,
                              int32_t num_vectors,
                              bool save_write_time,
                              uint32_t *xfer_bytes)
```

Parameter	Name	Description
input	port_handle	Port handle for the bus.
input	vectors	An array of register read/write operations.
input	num_vectors	Number of elements in a vector array.
input	save_write_time	True, if the time of the bus write transaction must be saved. If there are multiple write vectors, then this parameter saves only the time of the last write vector.
output	xfer_bytes	The total number of bytes read and written for all registers in this vectored transfer is NULL.

Return	Description
sns_rc	• SNS_RC_INVALID_VALUE: The passed in port handle is invalid • SNS_RC_FAILED: Unable to transfer over bus • SNS_RC_SUCCESS: Action succeeded

```
sns_rc (*sns_scp_register_rw_ex)(sns_sync_com_port_handle *port_
handle,
                                 sns_com_port_config_ex *com_port_
ex,
                                 sns_port_vector *vectors,
                                 int32_t num_vectors,
                                 bool save_write_time,
                                 uint32_t *xfer_bytes);
```


Parameter	Name	Description
input	port_handle	Port handle for the bus.
input	com_port_ex	Structure with the extended COM port parameters.
input	vectors	An array of register read/write operations.
input	num_vectors	Number of elements in a vector array.
input	save_write_time	True, if the time of the bus write transaction must be saved. If there are multiple write vectors, then this parameter saves only the time of the last write vector.
output	xfer_bytes	The total number of bytes read and written for all registers in this vectored transfer is NULL.

Return	Description
sns_rc	• SNS_RC_INVALID_VALUE: The passed in port handle is invalid • SNS_RC_FAILED: Unable to transfer over the bus • SNS_RC_SUCCESS: Action succeeded

```
sns_rc (*sns_scp_simple_rw)(sns_sync_com_port_handle *port_handle,
                            bool is_write,
                            bool save_write_time,
                            uint8_t *buffer,
                            uint32_t bytes,
                            uint32_t *xfer_bytes);
```

Parameter	Name	Description
input	port_handle	Port handle for the bus
input	is_write	True for the write operation, else False.
input	save_write_time	True, if the time of the bus write transaction must be saved
input/output	buffer	Buffer for read/write operation
input	bytes	Number of bytes to read/write
output	xfer_bytes	Actual total number of bytes read/write

Return	Description
sns_rc	• SNS_RC_INVALID_VALUE: The passed in port handle is invalid • SNS_RC_FAILED: Unable to transfer over bus • SNS_RC_NOT_AVAILABLE: Action not supported • SNS_RC_SUCCESS: Action succeeded

```
sns_rc (*sns_scp_get_write_time)(sns_sync_com_port_handle *port_
handle,
                                 sns_time *write_time);
```

Parameter	Name	Description
input	port_handle	Port handle for the bus
output	write_time	System timestamp (in ticks)

Return	Description
sns_rc	• SNS_RC_INVALID_VALUE: The passed in port handle is invalid • SNS_RC_FAILED: Unable to get write time • SNS_RC_SUCCESS: Action succeeded

```
sns_rc (*sns_scp_issue_ccc)(sns_sync_com_port_handle *port_handle,
                            sns_sync_com_port_ccc ccc_cmd,
                            uint8_t *buffer,
                            uint32_t bytes,
                            uint32_t *xfer_bytes);
```

Parameter	Name	Description
input	port_handle	Port handle for the bus.
input	ccc_cmd	CCC command to be issued. Read or write depends on the command.
input/output	buffer	Buffer for read/write operation. It may be NULL if the command has no data.
input	bytes	The size of the buffer may be 0.
output	xfer_bytes	Total bytes actually read/write.

Return	Description
sns_rc	• SNS_RC_NOT_SUPPORTED: CCC isn't supported on this bus • SNS_RC_INVALID_VALUE: Invalid handle, unsupported CCC, or wrong buffer size for CCC • SNS_RC_FAILED: Transfer failed • SNS_RC_SUCCESS: Action succeeded

```
sns_rc (*sns_scp_rw)(sns_sync_com_port_handle *port_handle,
                     uint8_t*                  read_buffer,  //NULL
allowed for write operation
                     uint32_t                  read_len_bytes,
                     const uint8_t*            write_buffer,
                     uint32_t                  write_len_bytes,
                     uint8_t                   bits_per_word);
```

Parameter	Name	Description
input	port_handle	Port handle for the bus
input	read_buffer	Pointer to the read buffer
input	read_len_bytes	Length of the read buffer
input	write_buffer	Pointer to the write buffer
input	write_len_bytes	Length of the write buffer
input	bits_per_word	SPI basic transaction unit size, usually 8-bits or 32-bits

Return	Description
sns_rc	• SNS_RC_INVALID_VALUE: The passed in port handle is invalid. • SNS_RC_FAILED: Unable to transfer over bus. • SNS_RC_NOT_AVAILABLE: Action not supported. • SNS_RC_SUCCESS: Action succeeded.

GPIO service APIs

The GPIO service APIs provides a synchronous API to read/write from/to output GPIO pins.

sns_gpio_service		
Data field		**Description**
sns_service	service;	Service information.
struct sns_gpio_service_api	*api;	The public API provided by the service to be used by the sensor.

sns_gpio_service_api		
Data field		**Description**
uint32_t	struct_len;	Size of the structure.
sns_rc	(*read_gpio)(uint32_t gpio, bool is_chip_pin, sns_gpio_state *state);	Reads the current state of a general-purpose input pin.
sns_rc	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);	Writes a value to a general-purpose output pin.

```
sns_rc (*read_gpio)(uint32_t gpio,
                    bool is_chip_pin,
                    sns_gpio_state *state);
```

Parameter	Name	Description
input	gpio	GPIO pin to read.
input	is_chip_pin	True, if the GPIO is chip level TLMM pin.
output	state	Output GPIO state.

Return	Description
sns_rc	• SNS_RC_SUCCESS: If GPIO read is successful • SNS_RC_NOT_SUPPORTED: If unsupported GPIO input • SNS_RC_FAILED: For other errors

```
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);
```

Parameter	Name	Description
input	gpio	GPIO pin to write
input	is_chip_pin	is_chip_pin is true if the GPIO is chip level TLMM pin, else false
input	drive_strength	Drive strength configuration
input	pull	Pull type configuration
input	state	Output state to set for the GPIO

Return	Description
sns_rc	• SNS_RC_SUCCESS: If the GPIO write is successful • SNS_RC_NOT_SUPPORTED: If unsupported GPIO input • SNS_RC_FAILED: For other errors

Island service APIs

The following island service APIs provides a synchronous API to request an exit from the Island mode.

sns_island_service		
Data field		**Description**
sns_service	service;	Service information.
struct sns_island_service_api	*api;	A public API provided by the service for the sensor.

sns_island_service_api		
Data field		**Description**
uint32_t	struct_len;	Size of the structure.

sns_island_service_api		
Data field		**Description**
sns_rc	(*sensor_island_exit)(sns_island_service *service, struct sns_sensor const* sensor);	Requests for exit from the Island mode available to a sensor.
sns_rc	(*sensor_instance_island_exit)(sns_island_service *service, struct sns_sensor_instance const* instance);	Requests exit from the Island mode available to a sensor instance.
sns_rc	(*island_log_trace)(sns_island_service *service, uint64_t user_defined_id);	Generates a log packet of the sns_island_trace_log type and commits it.

```
sns_rc (*sensor_island_exit)(sns_island_service *service,
                             struct sns_sensor const* sensor);
```

Parameter	Name	Description
input	service	Island service reference.
input	sensor	Sensor reference that is requesting island exit. NULL, if an instance is requesting an island exit.

Return	Description
sns_rc	• SNS_RC_SUCCESS: If the Island mode exit was successful • SNS_RC_FAILED: Otherwise

```
sns_rc (*sensor_instance_island_exit)(sns_island_service *service,
                                      struct sns_sensor_instance
const* instance);
```

Parameter	Name	Description
input	service	Island service reference.
input	instance	• Sensor instance reference that's requesting island exit. • NULL if a sensor is requesting island exit.

Return	Description
sns_rc	• SNS_RC_SUCCESS: If Island mode exit was successful • SNS_RC_FAILED: Otherwise

```
sns_rc (*island_log_trace)(sns_island_service *service,
                           uint64_t user_defined_id);
```

Parameter	Name	Description
input	service	Island service reference.
input	user_defined_id	Values to be added to the cookie field in the log packet.

Return	Description
sns_rc	• SNS_RC_SUCCESS: Log packet operation was successful • SNS_RC_NOT_SUPPORTED: Log packets not supported

Registration service APIs

The registration service API allows the sensors to register new sensors within the same library. The following are the registration service APIs:

sns_registration_service		
Data field		**Description**
sns_service	service;	Service information.
struct sns_ registration_ service_api	*api;	Public API provided by the framework for the sensor.

sns_registration_service_api		
Data field		**Description**
uint32_t	struct_len;	Size of the structure.
sns_rc	sns_rc (*sns_sensor_ create)(const sns_ sensor *sensor, uint32_t state_len, struct sns_sensor_api const *sensor_api, struct sns_sensor_ instance_api const *instance_api);	Allocates and initializes a sensor within the same library as the sensor making the request.

```
sns_rc (*sns_sensor_create)(const sns_sensor *sensor,
                            uint32_t state_len,
                            struct sns_sensor_api const *sensor_
api,
                            struct sns_sensor_instance_api const
*instance_api);
```

Parameter	Name	Description
input	sensor	Sensor reference that requests the creation of a new sensor. These values should never be NULL.
input	state_len	Size to be allocated for the sensor state.
input	sensor_api	Sensor API implemented by the sensor developer.
input	instance_api	Sensor instance API by the sensor developer.

Return	Description
`sns_rc`	• `SNS_RC_POLICY`: `state_len` too large • `SNS_RC_NOT_AVAILABLE`: Sensor UID is already in-use. • `SNS_RC_FAILED`: Sensor initialization has failed. • `SNS_RC_SUCCESS`: Sensor initialization is successful.

sns_power_mgr_service_api	
Data field	**Description**

Power manager service APIs

The following are the power manager service APIs that provide services for sensor power management.

sns_power_mgr_service		
Data field		**Description**
sns_service	service;	Service information.
struct sns_power_mgr_service_api	*api;	Public API provided by the framework for the sensor.

```
typedef void *sns_power_mgr_handle;
```

sns_power_mgr_service_api		
Data field		**Description**
uint32_t	struct_len;	Size of the structure.
sns_rc	(*sns_register_client)(char *client_name, sns_power_mgr_handle **handle);	Registers sensor client with the sensor power manager service. **Note:** This API is not supported in the Island mode.

sns_power_mgr_service_api		
Data field		**Description**
sns_rc	(*sns_deregister_client)(sns_power_mgr_handle **handle);	Deregister sensor client with the sensor power manager service. **Note:** This API is not supported in the Island mode.
sns_rc	(*sns_update_mcps_vote)(sns_power_mgr_handle *handle, int16_t mcps);	Votes for the million cycles per second (MCPS) requirement. **Note:** This API is not supported in the Island mode.

```
sns_rc (*sns_register_client)( char *client_name, sns_power_mgr_
handle **handle );
```

Parameter	Name	Description
input	*client_name	Pointer to the client name. The maximum size is 32 characters.
output	**handle	Handle returned by the power manager service.

Return	Description
sns_rc	• SNS_RC_INVALID_VALUE: Input parameters are invalid • SNS_RC_FAILED: Client registration failed • SNS_RC_SUCCESS: Action succeeded

```
sns_rc (*sns_deregister_client)( sns_power_mgr_handle **handle );
```

Parameter	Name	Description
input/output	**handle	Reference to the handle

Return	Description
sns_rc	SNS_RC_INVALID_VALUE: If the handle is invalid SNS_RC_SUCCESS: The handle is freed

```
sns_rc (*sns_update_mcps_vote)( sns_power_mgr_handle *handle, int16_t
mcps);
```

Parameter	Name	Description
input/output	**handle	Handle returned by the power manager service
input	mcps	Requested MCPS

Return	Description
sns_rc	• SNS_RC_INVALID_VALUE: Input parameters are invalid • SNS_RC_SUCCESS: Action succeeded

sns_file_service	
Data field	**Description**

File service APIs

The following are the file service APIs that provide services for sensor file service management.

sns_file_service		
Data field		**Description**
sns_service	service;	Service information.
struct sns_file_service_api	*api;	Public API provided by the framework to be used by the sensor.

sns_file_service_api		
Data field		**Description**
uint32_t	struct_len;	Size of the structure.
FILE*	(*fopen)(const char *restrict filename, const char *restrict mode);	Opens the file whose name is the string pointed to the filename and associates a stream with it.
size_t	(*fwrite)(void const *ptr, size_t size, size_t count, FILE *stream);	Attempts to write the buffer, pointed to by the pointer and sized by the specified size, up to N elements specified by count, to the file handle.
size_t	(*fread)(void *ptr, size_t size, size_t count, FILE *stream);	Attempts to read size N elements specified by count from the buffer pointer, each size byte long, from the file handle.

sns_file_service_api		
Data field		**Description**
int	(*fclose)(FILE *stream);	Closes the file pointed by a stream.
int	(*fsize)(const char *path, off_t *size);	Gets the size of the file.

```
FILE *(*fopen)(const char *restrict filename, const char *restrict
mode);
```

Parameter	Name	Description
input	filename	Absolute path of the file to open.
input	mode	Open mode to use.

Return	Description
FILE*	Associated file stream.

```
size_t (*fwrite)(void const *ptr, size_t size, size_t count, FILE
*stream);
```

Parameter	Name	Description
input	ptr	Pointer to the buffer.
input	size	Size of the data type.
input	count	Number of elements to be written.
input	stream	File *stream* to write.

Return	Description
size_t	Number of elements written successfully.

```
size_t (*fread)(void *ptr, size_t size, size_t count, FILE *stream);
```

Parameter	Name	Description
input	ptr	Pointer to the buffer.
input	size	Size of the data type.

Parameter	Name	Description
input	count	Number of elements to be read.
input	stream	File *stream* to read.

Return	Description
size_t	Number of elements read successfully.

```
int (*fclose)(FILE *stream);
```

Parameter	Name	Description
input	stream	File stream.

Return	Description
int	If successful, it returns **0**. Otherwise, it returns EOF.

```
int (*fsize)(const char *path, off_t *size);
```

Parameter	Name	Description
input	path	Absolute path of the file.
input	size	Destination for the file size.

Return	Description
int	If successful, it returns **0**. Otherwise, it returns **–1** on error and sets errno appropriately.

For more information about the usage of all the QSH sensor APIs, see Develop sensors.

1.2 Software

Describes the low-power processor directory structure and how to flash an aDSP image.

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

For the *Application processor directory structure*, see Software in Qualcomm Linux Sensors Guide.

Low-power processor directory structure

The QSH software stack consists of the framework, API, sensor driver, and algorithm components. The following resources explain the QSH software tree and the location of each software component.

- **qsh_algorithms**

  The `adsp_proc/qsh_algorithms` directory has the QSH algorithm components—all fusion algorithms.

```
./qsh_algorithms/
├── build - Build component files
├── oem1 - Sample 2 Template Algorithm
```

- **qsh_api**

  The `adsp_proc/qsh_api` directory has the QSH API component.

```
./qsh_api/
├── build - Build component files
├── nanopb - NanoPB (Protocol Buffer)
└── pb - Proto files directory
```

- **qsh_platform**

  The `adsp_proc/qsh_platform` directory has the QSH component — core framework, utilities, platform sensors, and chipset-specific files.

```
./qsh_platform/
├── api - Framework & debug proto files
├── build - Build component files
├── chipset - Chipset-specific configuration files
├── framework - QSH framework code
├── inc - QSH common header files
├── sensors - QSH sensors (framework sensors)
├── tools - Tools
└── utils - QSH utilities
```

- **ssc_drivers**

  The `adsp_proc/ssc_drivers` directory has all the sensor drivers.

```
./ssc_drivers/
├── Mag Sensor Driver
├── SAR Sensor Driver
├── build - Build component files
├── Accel Sensor Driver
├── Pressure Sensor Driver
├── Humidity Sensor Driver
├── Als/Prox Sensor Driver
```

2 Bring up sensors

This section lists the steps to bring up the QSH framework with default configuration. It also provides steps to verify if the QSH is initialized, and to debug and fix any failures in the process.

2.1 Build and flash an aDSP image

The following procedures explain building and flashing the aDSP image on the target device.

Build

To build the software, see the Qualcomm Linux Build Guide specific to the Development Kit.

Flash

Use the following options to flash aDSP.

- Option 1: Flash all the build images—to flash the images, see Qualcomm Linux Build Guide specific to the Development Kit.
- Option 2: Sideload aDSP firmware and libraries—to sideload aDSP firmware on the device, without flashing all the images, run the following commands. Enter the password when prompted by `scp` command.

  1. To remount with the `rw` permission on the device, run the following command.

     ```
     mount -o rw,remount /
     ```

  2. To sideload aDSP firmware, run the following command on the host computer.

     ```
     scp <FIRMWARE_ROOT>/<distribution>/ADSP.HT.x.y.zz/adsp_proc/
     build/ms/bin/<chipset>.adsp.prod/splitbins/signed/* root@
     <device_ip_address>:/lib/firmware/qcom/<chipset>/
     ```

  3. To sideload aDSP libraries, run the following command on the host computer.

     ```
     scp <FIRMWARE_ROOT>/<distribution>/ADSP.HT.x.y.zz/adsp_proc/
     build/ms/dynamic_modules/<chipset>.adsp.prod/* root@<device_
     ip_address>:/usr/lib/dsp/adsp/
     ```

Verify

Verify that QSH is initialized and working. For instructions, see *Set up sensors* under the Platform in Qualcomm Linux Sensors Guide.

Troubleshoot

Fix QSH/sensor initialization failure. For instructions, see Troubleshoot sensors.

3 Develop sensors

The following sections describe detailed steps to integrate a custom sensor driver and develop a new algorithm.

3.1 QSH direct channel API workflow

Describes the proto messages, rate attributes, and channel types for the QSH direct channel API.

The following figure shows the direct channel workflow for an APPS client:

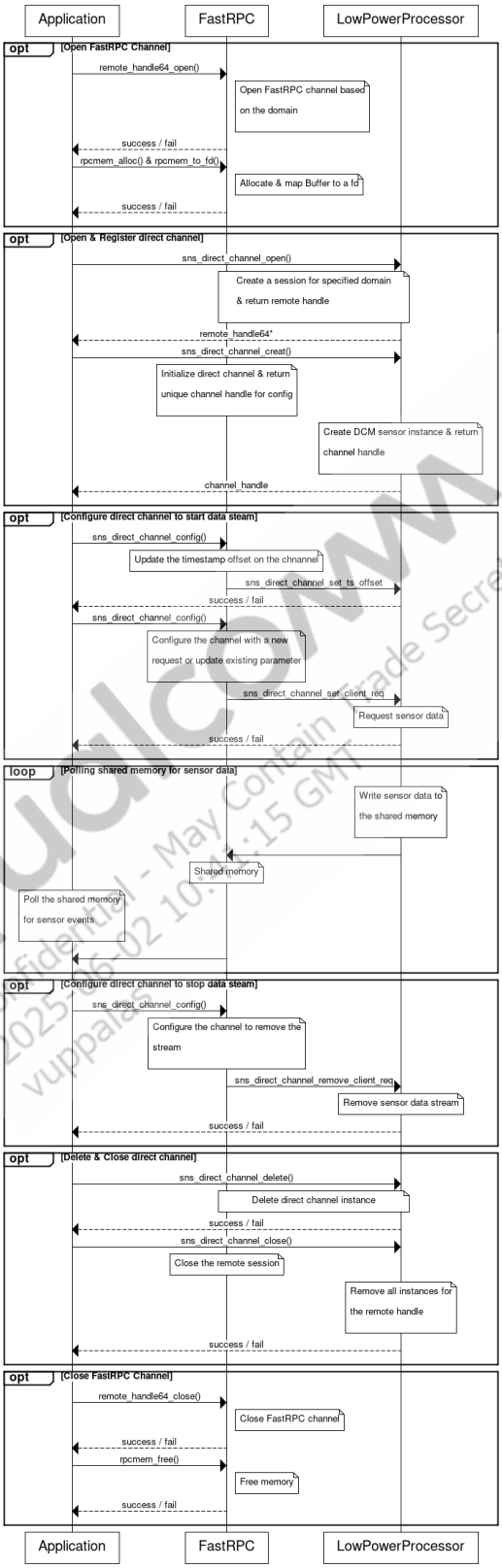


Figure1 Figure : Direct channel workflow for APPS client

Proto messages

Proto messages used to send client requests over the FastRPC stub direct channel APIs are described in the `vendor/qcom/proprietary/sensors-ship/qsh/sns_direct_channel.proto` file.

```
// @file sns_direct_channel.proto
//
// @brief Defines the Sensing Hub Direct Channel Interface
//
// @details ## To activate a Sensor Stream on a new Direct Channel:
// 1. Client must first create a channel using sns_direct_channel_
create_msg.
// 2. Client may then optionally send a sns_direct_channel_set_ts_
offset as part
//    of sns_direct_channel_config_msg to set timestamp offset for
this channel.
//    The offset will be added to timestamps of all Sensor samples
delivered on
//    this channel.
// 3. Client must then send sns_direct_channel_set_client_req as part
of
//    sns_direct_channel_config_msg to start streaming the required
Sensor on
//    this channel.
// 4. An ongoing Sensor stream can be reconfigured by resending the
//    sns_direct_channel_set_client_req with updated settings for the
Sensor.
// 5. If Channel type is DIRECT_CHANNEL_TYPE_STRUCTURED_MUX_CHANNEL,
Client may
//    stream additional Sensors on the same channel by sending
additional
//    sns_direct_channel_set_client_req messages.
// 6. Client may remove an existing Sensor stream on a channel by
sending
//    sns_direct_channel_remove_client_req as part of sns_direct_
channel_config_msg
//    for the Sensor.

syntax = "proto2";
import "nanopb.proto";
import "sns_std.proto";
import "sns_std_type.proto";
```

```
// @brief Direct channel defined Message IDs
enum sns_direct_channel_msgid
{
  option (nanopb_enumopt).long_names = false;

  // @brief Generic Channel Latency Event.
  // Message: sns_generic_channel_latency_msg
  SNS_DIRECT_CHANNEL_MSGID_SNS_GENERIC_CHANNEL_LATENCY_MSG = 20;
}

// @brief Direct Channel Type
enum direct_channel_type {
   option (nanopb_enumopt).long_names = false;

  // @brief Sensor data on this channel is of type sensors_event_t as
defined by the
  // Android Open Source Project.
  // Sensor data from multiple streams can be multiplexed in one
channel.
  DIRECT_CHANNEL_TYPE_STRUCTURED_MUX_CHANNEL = 0;

  // @brief Sensor data on this channel is in the format defined in
the
  // Sensor's proto API. Channel only supports a single stream of
Sensor data.
  DIRECT_CHANNEL_TYPE_GENERIC_CHANNEL = 1;
}

// @brief Sensor Stream Configuration
message sns_direct_channel_stream_id {
  // @brief SUID of the requested Sensor
  required sns_std_suid suid  = 1;

  // @brief if set to true,  auto calibrated data is generated
  // if set to false, factory calibrated data is generated
  optional bool calibrated    = 2 [default = true];

  // @brief if set to true,  resampled data is generated at a rate
which is at most twice the requested rate
  // if set to false, data is generated at the rate defined by
  // SNS_STD_SENSOR_ATTRID_RATES, SNS_STD_SENSOR_ATTRID_ADDITIONAL_
LOW_LATENCY_RATES
  // attributes for the Sensor.
  optional bool resampled     = 3 [ default = true];
```

```
}

// @brief Direct Channel Creation Request Message
message sns_direct_channel_create_msg {

  // @brief Shared memory buffer configuration
  message shared_buffer_config {

    // @brief File descriptor to the shared memory buffer
    required fixed32 fd   = 1;

    // @brief Size of the buffer
    required fixed32 size = 2;
  }
  // @brief shared buffer config
  required shared_buffer_config     buffer_config  = 1;

  // @brief Type of channel
  required direct_channel_type      channel_type   = 2 ;

  // @brief Processor on which the client resides
  optional sns_std_client_processor client_proc    = 3 [ default =
SNS_STD_CLIENT_PROCESSOR_APSS];
}

// @brief Client Request Message
message sns_direct_channel_set_client_req {
  // @brief Msg Id of the Request message as defined in the Sensor's
API
  required fixed32                                 msg_id     = 1;

  // @brief Sensor Stream Configuration
  required sns_direct_channel_stream_id            stream_id  = 2;

  // @brief Request message as specified in sns_std.proto and the
Sensor's API
  required sns_std_request                         request    = 3;

  // @brief Attributes mandatory for DIRECT_CHANNEL_TYPE_STRUCTURED_
MUX_CHANNEL
  message structured_mux_channel_stream_attributes {
    // @brief Sensor ID for requested Sensor as defined by the
Android Open Source
    // Project. This value will be populated in the "sensor_event_t::
```

```
sensor" field
    // for the events delivered from this Sensor.
    required fixed32 sensor_handle = 1;

    // @brief Sensor Type for requested Sensor as defined by the
Android Open Source
    // Project. This value will be populated in the "sensor_event_t::
type" field
    // for the events delivered from this Sensor.
    required fixed32   sensor_type = 2;
  }

  // @brief MUX Stream Attributes
  optional structured_mux_channel_stream_attributes attributes = 4;
}

// @brief Direct Channel Remove Client Request message
message sns_direct_channel_remove_client_req {
  required sns_direct_channel_stream_id stream_id = 1;
}

// @brief Set Timestamp Offset Message
//
// @details Clients requiring timestamps in the Client Processor Time
Domain may use this
// message to convey the offset with Sensing Hub Time Domain.
// This offset will be added to the Sensing Hub timestamps of all
Sensor samples
// delivered on the channel.
message sns_direct_channel_set_ts_offset {
  // @brief Timestamp offset in clock ticks
  required fixed64 ts_offset = 1;
}

// @brief Direct Channel Configuration Message
message sns_direct_channel_config_msg {

  // @brief Channel Config Payload
  oneof channel_config_msg_payload
  {
    sns_direct_channel_set_client_req     set_client_req      = 100;

    sns_direct_channel_remove_client_req  remove_client_req   = 101;
```

```
    sns_direct_channel_set_ts_offset set_ts_offset           = 102;
  }
}

// @brief This message is used to convey latency performance metrics
for the
// DIRECT_CHANNEL_TYPE_GENERIC_CHANNEL
//
// @details Latency is measured as the difference between the time
when the sample is
// written to channel buffer and the sample measurement time.
message sns_generic_channel_latency_msg{

   // @brief Number of samples associated with this measurement
   required fixed64 sample_count = 1;

   // @brief Measurement timestamp of sample with maximum latency
   // Units: nano-seconds
   required fixed64 max_latency_ts = 2;

   // @brief Maximum latency measured across all samples in sample_
count
   // Units: micro-seconds
   required fixed32 max_latency = 3;

   // @brief Average latency calculated across all samples in sample_
count
   // Units: micro-seconds
   required fixed32 avg_latency = 4;
}
```


Direct channel client manager

The direct channel client manager running on the sensors DSP does the following:

- Handles all the incoming direct channel requests using the FastRPC channel.
- Creates the channel instance.
- Configures the channel to request sensor data.
- Deletes the channel instance after the use case is completed.

Sample code

The vendor/qcom/proprietary/sensors-ship/examples/sns_direct_channel_client_example/sns_directchannel_example_cpp test application provides information on how to call direct channel APIs over FastRPC stub.

Rate attributes

Direct channel clients must consider the following attributes before sending requests.

For more information, see adsp_proc/ssc_api/pb/sns_std_sensor.proto file.

Driver low latency rate

This attribute is published by the individual sensor driver.

```
// OPTIONAL
// [float]
// List of additional sample rates for low latency clients in Hz.
// These are additional rates for low latency clients extended from
list of rates published in attribute SNS_STD_SENSOR_ATTRID_RATES.
// This is supported for internal clients only. External clients
shall not use this API.
SNS_STD_SENSOR_ATTRID_ADDITIONAL_LOW_LATENCY_RATES = 25;
```

Acceptable rate

The sample rate must be less than or equal to MIN.

```
SNS_STD_SENSOR_ATTRID_ADDITIONAL_LOW_LATENCY_RATES
```

Channel types

The direct channel supports two channel types as described by the direct_channel_type enum in the vendor/qcom/proprietary/sensors-ship/qsh/sns_direct_channel.proto file.

Table : Direct channel type

	DIRECT_CHANNEL_TYPE_ STRUCTURED_MUX_ CHANNEL	DIRECT_CHANNEL_TYPE_ GENERIC_CHANNEL
Data format	Samples are compatible to the sensors_event_t format defined by the Android specification	Samples are compatible to the sns_std_sensor_event as defined in the sns_std_ sensor.proto
Protobuf encoding	Samples are not PB encoded	Samples are PB encoded

3.2 QSH sample algorithm and integration

Develop custom algorithms for QSH by using the OEM1 sample algorithm.

Write a `.proto` file for custom algorithm

The `.proto` file defines the messages used by the custom algorithm, using the protocol buffer specification language. The proto filename for an algorithm must be the same as the attribute `SNS_STD_SENSOR_ATTRID_API` value set in the respective `sns_oem1_sensor.c` file.

The following code snippet from the `adsp_proc/qsh_algorithms/oem1/src/sns_oem1_ sensor.c` file shows the setting attribute.

```
{
    sns_std_attr_value_data values[] = {SNS_ATTR, SNS_ATTR};
    char const proto1[] = "sns_oem1.proto";
    char const proto2[] = "sns_std_sensor.proto";
    values[0].str.funcs.encode = pb_encode_string_cb;
    values[0].str.arg = &((pb_buffer_arg)
        { .buf = proto1, .buf_len = sizeof(proto1) });
    values[1].str.funcs.encode = pb_encode_string_cb;
    values[1].str.arg = &((pb_buffer_arg)
        { .buf = proto2, .buf_len = sizeof(proto2) });
    sns_publish_attribute(this, SNS_STD_SENSOR_ATTRID_API,
        values, ARR_SIZE(values), false);
  }
```

The `.proto` filename for this algorithm must be `sns_oem1.proto`.

`.proto` file enum and supported messages

Each supported message has a message ID associated with it. All the supported message IDs are listed in an enum in the `.proto` file. Custom algorithms can import the `.proto` file to take advantage of the standard messages defined in the `adsp_proc/qsh_api/pb/sns_std_ sensor.proto` file.

```
import "sns_std_sensor.proto";
```

For more information, see `adsp_proc/qsh_api/pb/sns_oem1.proto` file.

Enum and message format

- The enum name is in `<filename>_msgid` format (without `.proto` extension in the filename).
- The supported message within the enum must be in `<enum_name>_<message_name> = <message_ID>` format.

For example, `SNS_OEM1_MSGID_SNS_OEM1_DATA` is the message ID used for the output data event generated by the OEM1 sensor.

```
enum sns_oem1_msgid
{
option (nanopb_enumopt).long_names = false;
SNS_OEM1_MSGID_SNS_OEM1_DATA = 1024;
}
```

For a message ID added in the enum, there may be a corresponding `protobuf` message that has fields based on the purpose of the message ID.

For example, `SNS_OEM1_MSGID_SNS_OEM1_DATA` message ID uses the `sns_oem1_data` data message for the OEM1 sensor event data that has values along the X-axis, Y-axis, and Z-axis.

```
// Data Message
// Output data event generated by the oem1 sensor.
message sns_oem1_data
{
  // oem1 Vector along axis x,y,z in m/s2
  repeated float oem1 = 1 [(nanopb).max_count = 3];
  // Accuracy of the data
  required sns_std_sensor_sample_status accuracy = 2;
```

Add messages

Additional custom messages within the scope of the algorithm can also be written, named, and used in accordance to the structure mentioned in Enum and message format, as required.

```
import "sns_std_sensor.proto";
enum sns_oem1_msgid
{
  option (nanopb_enumopt).long_names = false;

  // Uses message: sns_oem1_config
```

```
  // Purpose:
  //   1. A stream request from a client to the oem1 sensor.
  //   2. Add more parameters as compared to the sns_std_sensor_
config
  //   3. A config/ack event from a sensor to the client.
  SNS_OEM1_MSGID_SNS_OEM1_CONFIG = 520;

  SNS_OEM1_MSGID_SNS_OEM1_DATA = 1024;
}

// Configuration Message
// Input Configurable data to the oem1 sensor
message sns_oem1_config
{
  // Sample rate in Hz
  required float sample_rate = 1;
  optional float ip_param1 = 2;
  optional int32 ip_param2 = 3;
  optional fixed32 ip_param3 = 4;
```

OEM1 supports `SNS_STD_SENSOR_MSGID_SNS_STD_SENSOR_CONFIG (513)` and `SNS_STD_SENSOR_MSGID_SNS_STD_ON_CHANGE_CONFIG (514)` message IDs that are defined in the `sns_std_sensor.proto` file. The `SNS_OEM1_MSGID_SNS_OEM1_CONFIG (520)` message ID is added to the enum to support a configuration message with additional parameters.

Sample algorithm template: OEM1 sample algorithm

OEM1 is a sample algorithm on top of the QSH framework that Qualcomm provides to check code. It streams the data from *accel* sensor.

Algorithm source code location

Similar to other sensors, the OEM1 algorithm is available in the `adsp_proc/qsh_algorithm/oem1` directory with QSH. OEM1 is provided as a sample algorithm in the default release code. The following table lists important algorithm source code files.

Table : Key files of OEM1 algorithm

Files	Description
`sns_oem1.c`	Function to register the algorithm library.
`sns_oem1_sensor.c`	Normal mode APIs for algorithm.
`sns_oem1_sensor.h`	Data types for the algorithm.
`sns_oem1_sensor_island.c`	Island mode functions for the algorithm.
`sns_oem1_sensor_instance.c`	Normal mode functions for the sensor instance.

Files	Description
sns_oem1_sensor_instance.h	Sensor instance data types for the algorithm.
sns_oem1_sensor_instance_island.c	Island mode functions for the algorithm instance.

sns_oem1.c

The sns_oem1_register function is used to register the sensor. After the build, you can see all the sensors that are registered in the sns_static_sensors.c file. For all the sensors that are up and running, you can see the corresponding registry functions in the qsh_static_algorithms.c file.

```
sns_rc sns_oem1_register(sns_register_cb const *register_api)
{
 register_api->init_sensor(sizeof(sns_oem1_sensor_state),
                      &sns_oem1_api,
                           &sns_oem1_sensor_instance_api);
  return SNS_RC_SUCCESS;
}
```

The sns_register_cb structure has callback functions in the framework, for example, init_sensor is a callback function. The init_sensor function allocates and initializes a sensor.

For more information, see the QSH sensor API sns_register.h.

sns_oem1_sensor.c

The OEM1 sensors API is defined in the following file:

```
sns_sensor_api sns_oem1_api =
{
    .struct_len = sizeof(sns_sensor_api),
    .init = &sns_oem1_init,
    .deinit = &sns_oem1_deinit,
    .get_sensor_uid = &sns_oem1_get_sensor_uid,
    .set_client_request = &sns_oem1_set_client_request,
    .notify_event = &sns_oem1_notify_event,
};
```

The following is the OEM1 function implementation. For more information, see the QSH sensor API sns_sensor.h file.

- The sns_oem1_init function initializes and initiates the lookup data and publishes the sensor attributes. The sensor attributes are published from the sns_oem1_init function through the publish_attributes function.
  - The standard attributes of the sensor are seen in the sns_std_sensor.pb.h file that's

autogenerated from the sns_std_sensor.proto file using nanopb.

The following are a few examples of the attributes:

- SNS_STD_SENSOR_ATTRID_NAME
- SNS_STD_SENSOR_ATTRID_TYPE
- SNS_STD_SENSOR_ATTRID_VENDOR

– The SNS_SUID_LOOKUP_INIT and sns_suid_lookup_add functions are also performed from the sns_oem_init function. For more information, see the sns_suid_util.h file.

- The SNS_SUID_LOOKUP_INIT function initializes the data created by SNS_SUID_LOOKUP_DATA. For example, the following function initializes the lookup data:

```
SNS_SUID_LOOKUP_INIT(state > suid_lookup_data, NULL);
```

- The sns_suid_lookup_add function initiates the SUID lookup for the default sensor of a data type. For example, the following function looks up all the dependency sensors, such as *accel* of the OEM1:

```
sns_suid_lookup_add(this, &state > suid_lookup_data, "accel
");
```

- The sns_oem1_get_sensor_uid function gets the unique sensor identifier for OEM1. For example:

```
static sns_sensor_uid const*sns_oem1_get_sensor_uid(sns_sensor
const *this){
  UNUSED_VAR(this);
  return &oem1_suid;
}
```

- The sns_oem1_set_client_request function adds, removes, or updates the client request to this sensor. Find the instance (if any) that's presently servicing this request and update in curr_inst (current instance). The sns_sensor_util_find_instance is a utility function to find the instance that's presently servicing this request.

– If the message_id in new_req (new request) is SNS_STD_SENSOR_MSGID_SNS_STD_SENSOR_CONFIG, which is a streaming request, or SNS_STD_SENSOR_MSGID_SNS_STD_ON_CHANGE_CONFIG, which is an on-change request, then do the following:

1. Decode the new request using pb_decode (not required for on-change).
   - A new request might contain sample rates and custom parameters.
   - For an on-change, a new request doesn't contain any data, therefore, decoding isn't required.

2. Create an instance of the sensor. Allocate and initialize a new sensor instance to be associated with this sensor. For example:

```
this > cb > create_instance(this,(sizeof(sns_oem1_inst_
state)));
```

3. Assign the sensor instance to fulfill the client request using the `add_client_request` function. For example:

```
add_client_request(rv_inst, new_req)
```

4. Update the sensor instance configuration to this sensor request using `set_client_config`. For example:

```
set_client_config(rv_inst, &decoded_req) set_client_
config(rv_inst, &decoded_req)
```

5. If the current instance isn't NULL and `get_client_request` is NULL, then `remove_instance(curr_inst);`. For example:

```
if(NULL != curr_inst && NULL == curr_inst->cb->get_client_
request(
        curr_inst, this->sensor_api->get_sensor_uid(this),
true))
    {
      this->cb->remove_instance(curr_inst);
    }
```

- The `sns_oem1_notify_event` function notifies the client that data is received. If `suid_lookup` is completed, that is, all the dependent sensors are available, then publish the sensor through the `publish_available` function and call `sns_suid_lookup_deinit` to de-initialize the SUID lookup. For example:

```
 if(sns_suid_lookup_complete(&state->suid_lookup_data))
  {
#ifdef OEM1_SUPPORT_REGISTRY
    if(state->first_pass == true){
      state->first_pass = false;
      sns_oem1_registry_req(this);
    }
#endif
    publish_available(this);
    sns_suid_lookup_deinit(this, &state->suid_lookup_data);
  }

sns_suid_lookup_deinit - Deinitialize the data initialized by
```

```
SNS_SUID_LOOKUP_INIT

static void
publish_available(sns_sensor *const this)
{
  sns_std_attr_value_data value = sns_std_attr_value_data_init_
default;
  value.has_boolean = true;
  value.boolean = true;
  sns_publish_attribute(
      this, SNS_STD_SENSOR_ATTRID_AVAILABLE, &value, 1, true);
}
```

`sns_oem1_sensor_island.c`

The OEM1 SUID is defined in the `sns_oem1_sensor_island.c` file. For example:

```
const sns_sensor_uid oem1_suid = OEM1_SUID;
```

`sns_oem1_sensor_instance.c`

- `sns_oem1_inst_init`: Initializes a sensor instance to its default state:
  1. Get the sensor SUIDs and services and fill the `sns_oem1_inst_state` structure appropriately.
  2. Create `accel_stream`, if necessary. Create connection with the *accel* sensor stream if the direct *accel* sensor stream macro is enabled.

     Note: The sensor creates a stream to all its dependent sensors.

  3. Copy the platform-specific configuration `sns_oem1_sensor_state` to `sns_oem1_inst_state`.
- `sns_oem1_inst_deinit`: Removes all the streams that were created either in `sns_oem1_inst_init` or `sns_oem1_inst_set_client_config`. Release all the hardware and software resources associated with this sensor instance.
- `sns_oem1_inst_set_client_config`: Updates a sensor instance configuration to `sensorRequest`.

  If `message_id` is `SNS_STD_SENSOR_MSGID_SNS_STD_ON_CHANGE_CONFIG` or `SNS_STD_SENSOR_MSGID_SNS_STD_SENSOR_CONFIG`, then do the following:

  1. Initialize the state by updating `client_config` and `sample_rate` from the client request.
  2. If there is a direct *accel* sensor request (no resampler), then `send_request` to the

accel sensor.

```
sns_std_sensor_config accel_config = {.sample_rate = state->
config.sample_rate};
encoded_len = pb_encode_request(buffer,
    sizeof(buffer),
    &accel_config,
    sns_std_sensor_config_fields,
    NULL);
```

For more information about resampler vs. direct requests, see Resampler vs. direct requests.

a. Create a connection with the *accel* sensor. Already created in `sns_oem1_inst_init` function.

b. Update the stream reference allocated by the framework in `accel_stream`.

c. Send a request to another service/sensor using the `send_request` function.

   For example:

   ```
   state > accel_stream > api > send_request(state > accel_
   stream, &request);
   ```

d. The framework copies the request sent on the `accel_stream` data stream.

3. If the resampler is activated, then OEM1 receives accelerometer data using the resampler.

```
sns_resampler_config resampler_config =   sns_resampler_
config_init_default;
sns_memscpy(&resampler_config.sensor_uid,
     sizeof(resampler_config.sensor_uid),
     &state->accel_suid,
     sizeof(state->accel_suid));
     resampler_config.resampled_rate = state->config.sample_
rate;
     resampler_config.rate_type = SNS_RESAMPLER_RATE_MINIMUM;
     resampler_config.filter = false;
     encoded_len = pb_encode_request(buffer, sizeof(buffer),
      &resampler_config, sns_resampler_config_fields, NULL);
```

a. Create a connection with the resampler sensor.

b. Update the stream reference allocated by the framework in `resampler_accel_stream`.

c. Send a request to another service/sensor using `send_request`.

For example:

```
state > resampler_accel_stream > api > send_request(state
> resampler_accel_stream, &request);
```

d. The framework copies the request sent on the `resampler_accel_stream` data stream.

`sns_oem1_sensor_instance_island.c`

The OEM1 sensor instance API is defined in the following file:

```
sns_sensor_instance_api sns_oem1_sensor_instance_api =
{
    .struct_len = sizeof(sns_sensor_instance_api),
    .init = &sns_oem1_inst_init,
    .deinit = &sns_oem1_inst_deinit,
    .set_client_config = &sns_oem1_inst_set_client_config,
    .notify_event = &sns_oem1_inst_notify_event
};
```

For more information about `sns_sensor_instance_api` functions, see the QSH sensor API `sns_sensor_instance.h`.

- The `sns_oem1_inst_init`, `sns_oem1_inst_deinit`, and `sns_oem1_inst_set_client_config` functions are explained in the `sns_oem1_sensor_instance.c` file.
- The `sns_oem1_inst_notify_event` function notifies that the client has received some data and can process incoming events to the OEM1 sensor instance.

  1. Decode and print the `accel` data.
  2. If the resampler isn't enabled, get the event from `accel_stream`.

     For example:

```
accel_event_in = state > accel_stream > api > peek_
input(state > accel_stream);
If (accel_event_in = state > accel_stream > api > peek_
input(state > accel_stream);
```

     a. To get the *accel* sensor data, decode the event. For example, the OEM1 algorithm logic is as follows:

```
//This is dummy logic for OEM1 demonstration purposes
//OEMs can replace with their algo logic
if(0 < temp[2])
{
  current_state = OEM1_FACING_UP;
```

```
  accel_payload[0]=100;
  accel_payload[1]=temp[1];
  accel_payload[2]=temp[2];
}
else
{
  current_state = OEM1_FACING_DOWN;
  accel_payload[0]=state->down_value;
  accel_payload[1]=temp[1];
  accel_payload[2]=temp[2];
}
```

Here,

- `temp[0-2]` contains input to the algorithm from the dependent sensor that is the x, y, and z axes of the accelerometer data in the default implementation
- `accel_payload[0-2]` contains the output of the algorithm generated by the OEM1

b. Encode and publish an outgoing `sns_std_sensor_event` event through the `pb_send_sensor_stream_event` function.

3. If the resampler is enabled, get the data from `resampler_stream`.

For example:

```
state > resampler_stream > api > peek_input(state >
resampler_stream)
If (SNS_STD_SENSOR_MSGID_SNS_STD_SENSOR_EVENT == resampler_
event_in >message_id)
```

a. To get the *accel* sensor data, decode the event. For example, the OEM1 algorithm logic is as follows:

```
//This is dummy logic for OEM1 demonstration purposes
//OEMs can replace with their algo logic
if(0 < data[2])
{
  current_state = OEM1_FACING_UP;
  oem1_payload[0]=100;
  oem1_payload[1]=data[1];
  oem1_payload[2]=data[2];
}
else
{
  current_state = OEM1_FACING_DOWN;
```

```
  oem1_payload[0]= state->down_value;
  oem1_payload[1]=data[1];
  oem1_payload[2]=data[2];
}
```

Here,

– `data[0-2]` contains input to the algorithm from the dependent sensor

– `oem1_payload[0-2]` contains the output of the algorithm generated by the OEM1

b. Encode and publish an outgoing `sns_std_sensor_event` event through the `pb_send_sensor_stream_event` function.

Functionality of sample algorithm OEM1

OEM1 is designed to highlight various functionalities that the QSH algorithm developer needs. For example:

- Supporting output as streaming or on-change
- Algorithm requiring sensor input at a fixed rate
- Registry support (reading certain configuration parameters)

Table : Functionality flags

Flag (in sns_oem1_sensor.h)	Description
OEM1_SUPPORT_DIRECT_SENSOR_REQUEST	• With this flag enabled: – The OEM1 algorithm requests the data from the *accel* sensor directly. – The rate at which the algorithm receives input sensor samples (*accel*) depends on the ODR of the *accel* driver. For example, if there is a concurrent client for the *accel* sensor at a higher rate, then the algorithm receives *accel* data at a higher rate). • With this flag disabled: – OEM1 requests the accelerometer data through a Qualcomm-provided resampler module. – The resampler module always confirms that the rate is the same as requested and it's oblivious to any other concurrent requests.
OEM1_SUPPORT_EVENT_TYPE	• With this flag enabled: – The algorithm is set to have an on-change/event output. – The OEM1 algorithm reports the data only if the output changes (and not every output). • With this flag disabled: – OEM1 output is set to the streaming type. – Every time the algorithm receives input from *accel*, it calculates and generates an event.

Flag (in sns_oem1_sensor.h)	Description
OEM1_SUPPORT_REGISTRY	This flag is enabled if the requirement is for the algorithm to read certain configuration parameters from the registry. Besides enabling the OEM1_SUPPORT_REGISTRY on the aDSP side, the sns_oem1.json file must also be placed on the AP side at <workspace>/build-qcom-wayland/workspace/sources/sensors-ship-qti/sensors-ship/registry/config/<chipset_name>.

OEM1 also depends on the following sensors:

- Accelerometer
- Registry
- Resampler (if enabled)

To add any other dependency, such as streaming gyroscope (or) magnetometer along with the accelerometer, change the appropriate sns_oem1_sensor_state data structure.

For example, SNS_SUID_LOOKUP_DATA(3) suid_lookup_data: Here, the number *3* represents the number of dependencies.

For more information about these customizations and to understand the associated changes required in the algorithm, you must study the code around the flags mentioned in Table : Functionality Flags. You must have a thorough understanding of the OEM1 template code and must map it to the requirements of the algorithm.

Resampler vs. direct requests

Clients/sensors that require a fixed rate of dependent sensors typically register with the resampler. By default, OEM1 is shipped with resampler enabled. For example, OEM1_SUPPORT_DIRECT_SENSOR_REQUEST is disabled.

The SNS_RESAMPLER_MSGID_SNS_RESAMPLER_CONFIG is handled for resampler requests vs.SNS_STD_SENSOR_MSGID_SNS_STD_SENSOR_CONFIG for direct request.

The following code snippet shows the send_request API with the resampler configuration:

```
sns_request request = (sns_request){
  .message_id = SNS_RESAMPLER_MSGID_SNS_RESAMPLER_CONFIG,
  .request_len = encoded_len, .request = buffer };
  state->resampler_stream->api->send_request(state->resampler_stream,
&request);
```

Note: The resampler configuration message is present in the sns_resampler_config structure (sns_resampler.proto).

The following code snippet shows the send_request API with the direct sensor configuration:

```
sns_request request = (sns_request){
  .message_id = SNS_STD_SENSOR_MSGID_SNS_STD_SENSOR_CONFIG,
  .request_len = encoded_len, .request = buffer };
  state->accel_stream->api->send_request(state>accel_stream, &
request);
```

On-change vs. streaming

For an on-change, the request message from the client is empty. Therefore, it's not necessary to decode the request. By default, OEM1 is shipped with the output set to on-change. For example, OEM1_SUPPORT_EVENT_TYPE enabled.

SNS_STD_SENSOR_MSGID_SNS_STD_ON_CHANGE_CONFIG is handled for an on change vs. SNS_STD_SENSOR_MSGID_SNS_STD_SENSOR_CONFIG for streaming.

3.3 Configure sensors

Describes configuring and modifying QSH.

The following resources provide information about the QSH configuration and instructions to modify these configurations based on specific requirements.

Registry configuration in QSH

The QSH registry provides flexibility to configure the sensors, which includes the following aspects of driver-specific and platform-specific configurations.

- Bus protocol configuration
- Interrupt management
- Power rail control
- Sensor placement
- Sensor calibration

The registry adaptability of QSH ensures a custom approach to the sensor configuration and optimizes performance and efficiency. The following resources describes the tools necessary for successful navigation and utilization of the QSH registry.

- Location of registry (input json files): `/etc/sensors/config`
- Location of the parsed output configuration files: `<registry_path>`

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

Important registry files

Important registry files includes the following:

json.lst

This file contains the list of JSON files that the registry sensor must parse from the input configuration directory.

- Location: `<workspace>/build-qcom-wayland/workspace/sources/sensors-ship-qti/sensors-ship/registry/config/<target_name>/json.lst`
- Location on target: `/etc/sensors/config/json.lst`

During QSH framework initialization, the registry sensor parses the `json.lst` file to get a list of files, then parses each file from this list. All the JSON files in the input configuration directory that aren't part of the `json.lst` file, are ignored and aren't parsed by the registry sensor. For example:

```
<chipset_name>_lps22hh_0.json
```

```
<chipset_name>_lsm6dst_0.json
<chipset_name>_lsm6dst_1.json
<chipset_name>_mtp_ak991x_0.json
<chipset_name>_mtp_lsm6dsv_0.json
<chipset_name>_mtp_lsm6dsv_1.json
<chipset_name>_power_0.json
<chipset_name>_qrd_ak991x_0.json
<chipset_name>_qrd_ak991x_1.json
```

sns_reg_config

The registry reads this file and sets all the important parameters, such as input, output, and `hw_platform` during boot up. This file is present in the `<workspace>/build-qcom-wayland/workspace/sources/sensors-ship-qti/sensors-ship/registry/` directory.

```
version=1
file=hw_platform=/sys/firmware/devicetree/base/model
file=platform_subtype_id=/sys/devices/soc0/platform_subtype_id
file=platform_version=/sys/devices/soc0/platform_version
file=soc_id=/sys/devices/soc0/soc_id
file=input=json.lst
file=revision=/sys/devices/soc0/revision
file=output=/var/cache/sensors/registry/registry
```

parsed_file_list.csv

After parsing the registry, `parsed_file_list.csv` file is generated, which contains the list of parsed files. This file is present in the `<registry_path>` directory. The following sample shows the contents of this file:

```
<chipset_name>_ak991x_0.json.ak0991x_0_platform.config
<chipset_name>_ak991x_0.json.ak0991x_0_platform.mag.fac_cal.corr_mat
<chipset_name>_ak991x_0.json.ak0991x_0_platform.mag.fac_cal.bias
<chipset_name>_ak991x_0.json.ak0991x_0_platform.mag.fac_cal_2
<chipset_name>_ak991x_0.json.ak0991x_0_platform.mag
<chipset_name>_ak991x_0.json.ak0991x_0_platform.placement
<chipset_name>_ak991x_0.json.ak0991x_0_platform
<chipset_name>_ak991x_0.json.ak0991x_0.mag.config
<chipset_name>_ak991x_0.json.ak0991x_0.mag.config_2
<chipset_name>_rbx_icm4x6xx_0.json.icm4x6xx_0
<chipset_name>_rbx_icm4x6xx_0.json.icm4x6xx_0.accel
```

```
<chipset_name>_rbx_icm4x6xx_0.json.icm4x6xx_0.accel.config
<chipset_name>_rbx_icm4x6xx_0.json.icm4x6xx_0.freefall
<chipset_name>_rbx_icm4x6xx_0.json.icm4x6xx_0.freefall.config
<chipset_name>_rbx_navmezz_icp101xx_0.json.icp101xx_0_platform.
pressure.fac_cal.bias
<chipset_name>_rbx_navmezz_icp101xx_0.json.icp101xx_0_platform.
pressure.fac_cal.scale
<chipset_name>_rbx_navmezz_icp101xx_0.json.icp101xx_0_platform.temp
```

Driver registry in QSH

The following resources provide an in-depth understanding of how the QSH maintains all platform-specific and driver-specific configurations in the physical sensor JSON files.

- The JSON files are placed in a specific directory.
  - Filename format: `<chipset>_<sensor_name>_<hardware_id>`
  - Example: `/etc/sensors/config/<chipset>_lsm6dst_0.json`

    If multiple platforms have identical configurations for each chipset, then the `<platform>` field is dropped. The filename format only identifies the correct file for a target, which helps you to update the correct file easily. The filename isn't used by the registry sensor or the driver.
  - For easier identification, a file naming convention is used. For example, following are the filenames for different platform information of two different models:
    - Chipset X-based device 1: `chipsetX_lsm6dst_0_device1.json`
    - Chipset X-based device 2: `chipsetX_lsm6dst_0_device2.json`
- If the configuration file doesn't exist and the sensor driver supports the default values, the sensor library populates the default values in the registry.
- The driver can update its registry at runtime by sending the write requests to the registry sensor.

Driver-specific configuration

Registry groups/items also contain driver-specific configurations. All driver-specific configurations start with a *config* group. All configurations are enclosed within a top-level `<sensor_name>_<hardware_id>` registry group. The configuration file includes `data_type` specific registry groups (for example, accelerometer) that contain data-type-specific configurations.

Driver-specific configuration file example

```
{
  "config":
  {
    "hw_platform": ["MTP", "Dragon", "Surf"], "soc_id": [“475”]
  },
"lsm6dst_0":{
  "owner": "lsm6dst",
  ".accel":{
    "owner": "lsm6dst",
    ".config":{
      "owner": "lsm6dst",
      "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": "lsm6dst",
  ...
  }
}
```


The following table lists the driver-specific configuration of the sensor driver available in the QSH registry:

Table : Driver registry fields

Registry group/item name	Registry item type	Mandatory/optional	Description
`is_dri`	`int`	Mandatory	Identifies whether the sensor stream is interrupt-based (data ready, watermark, motion) or polling. • DRI sensors use value 1 • Polling sensors use value 0

hw_id	int	Mandatory	Unique identifier for the sensor hardware. It is used to differentiate between multiple sensors of the same hardware.
res_idx	int	Mandatory	Physical sensors have multiple supported resolutions and corresponding ranges. Sensors publish an array of supported resolutions. This item identifies the default resolution used by the sensor from the array of supported resolutions published.
sync_stream	int	Mandatory	Identifies whether the sensor supports any synchronous streaming mode, such as *S4S*.

Custom registry

Sensors can add custom registry groups/items in the configuration file for any custom requirements for persistent data for sensor/algorithm operation. Add these items in a registry group named, <sensor_name>_<hardware_id>_custom to allow a high degree of customization and flexibility in sensor operations.

Platform-specific configuration for the driver

The registry groups/items encapsulate sensor hardware and platform configuration. Each platform-specific configuration begins with a *config* group at the top, followed by all the platform configurations for the driver enclosed in a top-level group named, <sensor_name>_<hardware_id>_platform. Populate mandatory registry items in the sensor configuration file, and omit optional items if they're not applicable to the sensor. The configuration file must also contain data_type specific registry subgroups (for example, accelerometer) that hold data-type-specific platform configurations, such as factory calibration parameters.

Platform-specific configuration file example

```
{
  "config":{
    "hw_platform": ["MTP"],
    "soc_id": [“475"]
  },
  "lsm6dst_0_platform":{
    "owner": "lsm6dst",
    ".config":{
      "owner": "lsm6dst",
```

```
    "bus_type":{ "type": "int", "ver": "0",
      "data": "3"
    },
    "bus_instance":{ "type": "int", "ver": "0",
      "data": "1"
    },
    "slave_config":{ "type": "int", "ver": "0",
      "data": "106"
    },
    ".accel":{
    "owner": "lsm6dso",
    ".fac_cal":{
      "owner": "lsm6dso",
      ".corr_mat":{
        "owner": "lsm6dso",
        "0_0":{ "type": "flt", "ver": "0",
          "data": "1.0"
        },
        "0_1":{ "type": "flt", "ver": "0",
          "data": "0.0"
        },
        "0_2":{ "type": "flt", "ver": "0",
          "data": "0.0"
        },
        "1_0":{ "type": "flt", "ver": "0",
          "data": "0.0"
        },
        "1_1":{ "type": "flt", "ver": "0",
          "data": "1.0"
        },
        "1_2":{ "type": "flt", "ver": "0",
          "data": "0.0"
        },
        "2_0":{ "type": "flt", "ver": "0",
          "data": "0.0"
        },
        "2_1":{ "type": "flt", "ver": "0",
          "data": "0.0"
        },
        "2_2":{ "type": "flt", "ver": "0",
          "data": "1.0"
        }
      },
  }
```

```
}
```

The following table lists the platform-specific configuration of the sensor driver available in the QSH registry:

Table : Platform registry fields for drivers

Registry group/ item name	Registry item type	Mandatory/ optional	Description
bus_type	int	Mandatory	Identifies the communication bus to which the sensor is connected. Possible values are from sns_bus_type defined in the sns_com_port_types.h file. • 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	• Identifies the platform bus instance for the communication bus. • This item depends on the QUP number for every chipset. For more information, see Platform > Sensors and serial bus configuration section of Qualcomm Linux Sensors Guide.
slave_config	int	Mandatory	Identifies the secondary on the communication bus. • For I2C/I3C, this item is the secondary/static address. • For SPI, this item is the chip-select line for the slave/secondary. Typically, this item is 0 as most chipsets only support one chip-select.
min_bus_speed_khz	int	Mandatory	Identifies the minimum COM bus clock speed in kHz.
max_bus_speed_khz	int	Mandatory	Identifies the maximum COM bus clock speed in kHz.

Registry group/ item name	Registry item type	Mandatory/ optional	Description
reg_addr_type	int	Mandatory	Identifies the register address type supported by the sensor. See sns_com_port_types.h file. • 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 do not use this item.
irq_pull_type	int	Optional (Required for interrupt-based sensors)	If the sensor supports DRI modes, this item identifies the GPIO pull configuration of the dri_irq_num pin (active configuration). The following are the valid values from sns_interrupt_pull_type defined in the sns_interrupt.proto file. Polling sensors may not use this item. • 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	–	If a sensor uses dri_irq_num, set this item to 1 for sensors with the DRI interrupt support (it indicates that the MSM GPIO is used for the interrupt).

Registry group/ item name	Registry item type	Mandatory/ optional	Description
irq_drive_ strength	int	–	If a sensor uses dri_irq_num, this item identifies the drive strength configuration for the pin. Valid values for sns_interrupt_drive_ strength are defined in the sns_ interrupt.proto file: • 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)

Registry group/ item name	Registry item type	Mandatory/ optional	Description
irq_trigger_ type	int	–	If a sensor uses dri_irq_num, this item identifies the interrupt request trigger type. See sns_interrupt. proto file. • Rising edge – 0 (SNS_ INTERRUPT_TRIGGER_ TYPE_RISING) • Falling edge – 1 (SNS_INTERRUPT_ TRIGGER_TYPE_FALLING) • Rising and 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
num_rail	int	Mandatory	Provides the number of power rails connected to the sensor; it includes VDD and VDDIO rails. For example: If a sensor has the same VDD and VDDIO, set it to 1. This case is for most sensors with Qualcomm reference designs except for ALS and proximity.
rail_on_state	int	Mandatory	Identifies the ON state (LPM or NPM) of the power rail. A valid value is an enum value from sns_power_ rail_state defined in the sns_ pwr_rail_service.h file. • Low-power mode: 1 (SNS_ RAIL_ON_LPM): Must be used as an ON state only by the accelerometer sensor drivers. • Normal power mode: 2 (SNS_ RAIL_ON_NPM): ON state used by all the other sensors except the accelerometer.

Registry group/ item name	Registry item type	Mandatory/ optional	Description
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 item is set to /pmic/client/sensor_vddio, which maps to a particular power rail on an individual chipset. If any custom power rails differ from the rails used on the Qualcomm reference design on that chipset, then contact the Qualcomm sensors, PMIC, and hardware teams by filing Salesforce cases at https://support.qualcomm.com. These differences necessitate changes in PMIC software and a review from the PMIC and hardware teams.
rigid_body_type	int	Mandatory	Provides rigid body information regarding the sensor placement. A valid value from sns_std_sensor_rigid_body_type is defined in the sns_std_sensor.proto file. • 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 a sensor mounted on an external device, set it to 2. (SNS_STD_SENSOR_RIGID_BODY_TYPE_EXTERNAL)
.orient	Registry group	Optional	This item is a registry group. It contains 3 items of type string each. *x y z* This registry group is applicable for inertial sensors.

Registry group/ item name	Registry item type	Mandatory/ optional	Description
.fac_cal	Registry group	Optional	This item is a registry group that is placed within a data-type-specific registry group. For example: *.gyro* group contains the *.fac_cal* group with the gyroscope factory calibration parameters. Items within this group are of type float. See .fac_cal for the format of these items of each data_type.
min_odr	int	Optional	This registry item is used by a physical sensor driver to define a minimum ODR that must be supported.
max_odr	int	Optional	This registry item is used by a physical sensor driver to define a maximum ODR that must be supported. This item provides flexibility to define the maximum ODR without changing the driver code.

Island configuration

QSH supports island configuration for hardware-based and software-based sensors. Ensure that a dedicated environment flag is used for each sensor to enable the Island mode support. For example, SNS_ISLAND_INCLUDE_LSM6DST defined in the target specific por.py file.

Use this flag to decide the value of add_island_files field when the AddSSCSU() method is called in the driver .scons file. For more information, see sns_lsm6dst.scons.

```
lsm6dst_island_enable = False
if 'SNS_ISLAND_INCLUDE_LSM6DST' in env:
lsm6dst_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_lsm6dst",
binary_lib = False,
add_island_files = lsm6dst_island_enable)
```

Island memory map reconfiguration

QSH supports multi-island features, such as audio island and sensor island on the low-power processor. Based on the use case, one of the following island specifications is chosen at runtime.

- Audio-only island
- Sensor-only island
- Combined island

Any unused memory from one section can be re-adjusted to another. The island memory re-adjusted size must be in multiples of 64 KB aligned chunks. The `adsp_proc/build/ms/build-log.txt` build log file at its end contains the details of used and available memory.

The configuration file where island memory size for audio and sensor is specified, is `adsp_proc/config/<chipset_name>.adsp/cust_config.xml`.

The island memory pool includes the following:

- `SSC_TCM_PHYSPOOL` and `QURTOS_SSC_ISLAND_POOL` are used for sensor island.
- `AUDIO_TCM_PHYSPOOL` is used for audio island.
- `QURTOS_ISLAND_POOL` is the core services pool.

The following is the default island memory configuration from the `cust_config. xml` file.

```
<physical_pool name="SSC_TCM_PHYSPOOL" island="true">
<region base="0x02C00000" size="0x80000" name="QURTOS_ISLAND_REGION_
TCM" cache_policy="l1_wb_l2_uncacheable"/>
</physical_pool>
<physical_pool name="AUDIO_TCM_PHYSPOOL" island="true">
<region base="0x02C80000" size="0x70000" name="QURTOS_ISLAND_REGION_
TCM" cache_policy="l1_wb_l2_uncacheable"/>
</physical_pool>
<physical_pool name="TCM_PHYSPOOL" island="true">
<region base="0x02CF0000" size="0x90000" name="QURTOS_ISLAND_REGION_
TCM" cache_policy="l1_wb_l2_uncacheable"/>
</physical_pool>

    <physical_pool name="QURTOS_SSC_ISLAND_POOL">
<region allocate="island" size="0x30000" name="QURTOS_ISLAND_REGION_
DDR" cache_policy="l1_wb_l2_cacheable"/>
</physical_pool>

    <physical_pool name="QURTOS_ISLAND_POOL">
<region allocate="island" size="0x50000" name="QURTOS_ISLAND_REGION_
DDR" cache_policy="l1_wb_l2_cacheable"/>
</physical_pool>
```

Multisensor (dual) sensor configuration

QSH supports dual sensors according to the reference platform design configuration. Dual accelerometer/gyroscope sensors are enabled by default for the reference design configuration.

Flags in `adsp_proc/qsh_platform/chipset/<chipset_name>/por.py` file control the dual sensors support.

Build a flag	Usage
`LSM6DST_ENABLE_DUAL_SENSOR`	Enables dual sensor support for the LSM6DST sensor

If your device isn't using dual sensors, then delete the following lines in `adsp_proc/qsh_platform/chipset/<chipset_name>/por.py`.

```
env.AddUsesFlags(['LSM6DST_ENABLE_DUAL_SENSOR'])
```

Configure serial buses in QSH

Details on interfacing a sensor using a different serial bus will be provided in a future update.

3.4 Develop and integrate sensor drivers

Describes developing and integrating custom sensors driver according to the QSH design.

Develop sensor drivers

The QSH compliant sensor drivers must be written for the physical sensors that aren't a part of the default configuration. For more information to develop sensor drivers according to the QSH design, see Related documents.

Multisensor design

Note: The multisensor design section is applicable if you have more than one sensor of the same type.

A single driver library supports multiple instances of the same hardware. The 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 QSH framework provides a callback to indicate which sensor/instance belongs to which registration/hardware sensor.

Driver support

- A single driver is registered multiple times, with the number of registrations equal to the number of hardware instances (N) present on the platform. Update the driver SCons file to use the `AddSSCSU()` method for adding these N registrations.
- Each driver registration is considered as an individual library within the QSH. For example:
  - The LSM6DST registration for `hw_id=0` forms one library that consists of an accelerometer, gyroscope, sensor temperature, and motion detection sensors.
  - The LSM6DST registration for `hw_id=1` forms another library that consists of a different set of accelerometers, gyroscopes, sensor temperature, and motion detection sensors.
- Each set of sensors that belongs to a single hw_id/registration shares a common instance for all streaming, self-test, and custom operations.
- All existing QSH requirements for physical sensor drivers are applicable to each driver registration for multiple sensors.
- The registry configuration includes a platform-specific and a driver-specific registry for all N hardware instances of the sensor. For example, the files can be named as `<chipset_name>_lsm6dst_0.json` and `<chipset_name>_lsm6dst_1.json`.

Guidelines for multisensor design

The following are important multisensor guidelines for a hardware-based sensor driver:

- **Use the framework callback**: The driver uses the framework callback to obtain its `hw_id/registration` information.
  - **Publish unique SUIDs**: The driver must publish unique SUIDs for all the sensors of each hardware instance. For example, the LSM6DST driver publishes unique SUIDs for the `accel/gyro/sensor_temperature/motion_detect`, data types for `hw_id/registration=0` and `hw_id/registration=1`.
  - **Request registry for each unique hardware instance**: The driver must request a registry for each unique hardware instance. For example, the LSM6DST driver requests

for registry `lsm6dst_0_platform.config` for `hw_id=0`, and `lsm6dst_1_platform.config` for `hw_id=1`.

  - **Publish the correct attributes**: The driver must publish correct attributes for all the sensors of each hardware instance, particularly the placement of the specific attributes. For example, the LSM6DST driver publishes `hw_id=0` for all sensors that belong to the first registration and `hw_id=1` for the second registration.

- **Ensure re-entrancy and thread safety**: If all the sensor and sensor instance API implementations in the driver are re-entrant and thread-safe, no other driver changes are required.

  For more information about enabling the dual sensor, see multi_sensor_dual_sensor_config.

Integrate sensor drivers

The sensor driver integration process includes modifying the low-power processor and the application processor.

- Request the driver: Request for the well-tested driver from the sensor vendor for their specific sensor part/hardware.
- Obtain the driver acceptance checklist: Request for a driver acceptance checklist from the sensor vendor. This checklist is mandated by Qualcomm.

A vendor-delivered package includes the following:

- QSH-compliant driver source code
- Registry (JSON files)
- Test results: Driver acceptance checklist with the same source code and registry

Driver integration

The following are the driver integration steps that involve work at both the aDSP and the application processor subsystem.

Low-power processor integration

To integrate the sensor drivers with the low-power processor, do the following:

1. Create a directory based on the part name in `adsp_proc/ssc_drivers/`.

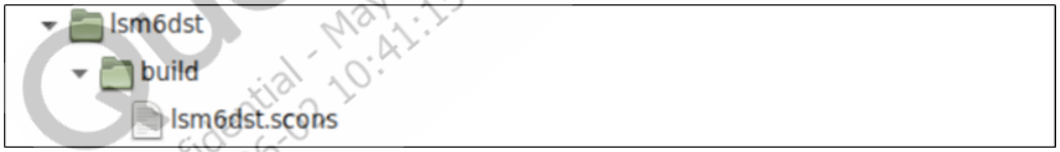
   - The path must be similar to `adsp_proc/ssc_drivers/<new_driver>`. For example, `adsp_proc/ssc_drivers/lsm6dst`.

2. Do the following within this new directory:

   a. Ensure that a build directory is present with the `.scons` file included.

      - The path must be similar to `adsp_proc/ssc_drivers/<new_driver>/build/<new_driver>.scons`.

        For example, `adsp_proc/ssc_drivers/lsm6dst/build/lsm6dst.scons`.



   b. Ensure that the *src* and *inc* directories are present with all the driver source and header files included.

      - The path must be similar to`adsp_proc/ssc_drivers/<new_driver>/src/*`.
      - The path must be similar to`adsp_proc/ssc_drivers/<new_driver>/inc/*`.

        For example, `adsp_proc/ssc_drivers/lsm6dst/src/* & inc/*`.

c. Enable the sensor driver for compilation. For the following steps, see `adsp_proc/qsh_platform/chipset/<chipset_name>/por.py`:

   - To replace a new sensor driver with the existing driver of the same sensor type, remove the driver SCons filename (without the `.scons` extension) of an existing driver from the `include_sensor_vendor_libs.extend` list and add a new driver SCons filename (without the `.scons` extension) into the `include_sensor_vendor_libs.extend` list.

     For example, to remove the existing LSM6DST accelerometer sensor driver, do the following:

```
include_sensor_vendor_libs.extend(['sns_ak0991x',
#MAGNETOMETER
                    -'lsm6dst',        #ACCEL/GYRO/MD/TEMP
                          +'<new_driver>'     #New Driver
                             'sns_tmd3702',      #ALS/PROX
                          'sns_lps22hx',      #PRESSURE
                          ])
```

   - To add a new driver SCons filename (without the `.scons` extension) into the `include_sensor_vendor_libs.extend` list, do the following:

```
include_sensor_vendor_libs.extend(['lsm6dst',
#ACCEL/GYRO/MD/TEMP
                                   'sns_ak0991x',
#MAGNETOMETER
                                    'sns_tmd3702',
#ALS/PROX
                                    'sns_lps22hx',
#PRESSURE
                                       'sns_icm4x6xx'
                                +'<new_driver>'     #New Driver
                                   ])
```

d. To enable the Island mode for a driver that supports the Island mode, add the corresponding flag to `adsp_proc/qsh_platform/chipset/<chipset_name>/por.py`.

```
env.AddUsesFlags(['SNS_ISLAND_INCLUDE_ <new_driver>'])
```

e. Compile the aDSP build. For more information, see Bring up sensors.

f. On successful compilation, the `*.lib` files in the build directory of the new driver are displayed as shown below for the `lm64dst` driver `adsp_proc/ssc_drivers/lsm6dst/build/sensor_img/qdsp6/<chipset_name>.adsp.prod`.

Check `adsp_proc/qsh_platform/framework/build/sensor_img/qdsp6/xxx.adsp.prod/sns_static_drivers.c` file and verify that the following text is 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, true},
```

g. Load the device with the compiled aDSP image. For more information, see Bring up.

Application processor integration

Although the driver is integrated into aDSP, most of the driver configuration is from the registry (JSON files). Each driver has its own JSON file that contains the driver-specific configuration.

To integrate the registry for a new sensor driver, do the following:

1. Update the driver registry files.

   a. Check if the configuration section of each driver configuration JSON file contains values for item `hw_platform`, `soc_id`, and `platfomrm_subtype_id` (if present). These values must match the following command output. If values from the command output aren't present in the respective items, then add the following command output values:

   - For soc_id:

   ```
   cat /sys/devices/soc0/soc_id
   ```

   - For hw_platform:

   ```
   mount -t debugfs debugfs /sys/kernel/debug
   cat /sys/kernel/debug/qcom_socinfo/hardware_platform
   ```

   Note: For the hardware platform output value 32, use `IOT` as the item value.

   - For platform_subtype_id:

   ```
   mount -t debugfs debugfs /sys/kernel/debug
   cat /sys/kernel/debug/qcom_socinfo/hardware_platform_
   subtype
   ```

b. If both the GPIOs and the bus type used for the sensor are the same as the reference design listed in the Platform section of Qualcomm Linux Sensors Guide, then modify the driver configuration JSON file items `bus_type` and `bus_instance` to match the configuration according to the reference design.

   If the GPIOs and bus type used for the sensors aren't the same as the reference design, then modify the driver configuration JSON file items `bus_type` and `bus_instance` according to the custom design.

   For more information about the sensor registry, see Configure sensors.

c. Modify the driver configuration JSON file item `dri_irq_num` according to the supported interrupts in the reference or custom design.

   For more information, see the table specific to the reference design in the Platform section of Qualcomm Linux Sensors Guide.

2. To add a new or modified driver configuration JSON file in the image of the application, do the following:

   a. Add the driver-specific configuration (JSON) file that's required for the physical sensor as described above, to the `<workspace>/build-qcom-wayland/workspace/sources/sensors-ship-qti/sensors-ship/registry/config/<target_name>` HLOS source code path.

   b. Compile the application processor build and load the device with the compiled application processor build. For more information, see Qualcomm Linux Yocto Guide.

      - Registry (.json) files are present in `/etc/sensors/config`.
      - On successful parsing of these files, see the parsed files in `/var/cache/sensors/registry/registry`.

Note: For temporary configuration, push configuration JSON files to the device `/etc/sensors/config/`.

3. Verify and test on platform.

   a. After rebooting, wait for the device to finish bootup. Verify that the driver registry file is present/generated in `/var/cache/sensors/registry/registry/`.

   b. To check if a driver is available/listed, run the `ssc_sensor_info` test utility.

      For more information, see *Tools* in Qualcomm Linux Build Guide.

Note: To develop a hardware-based sensor driver, see Related documents.

3.5 Develop and integrate custom sensor algorithm

Describes developing and integrating a QSH-compliant software sensor algorithm.

Develop custom sensor algorithm

Algorithm source code location and files

Similar to other sensors, the algorithms for QSH-compliant sensors are available in `adsp_proc/qsh_algorithm/<algo_name>/` directory.

Create the files with similar names as shown in the following table. Each file contains different functionality and has a unique purpose.

Table : Important files for algorithm

File	Description
`sns_<algo_name>.c`	Function to register the algorithm library
`sns_<algo_name>_sensor.c`	Normal mode APIs for the algorithm
`sns_<algo_name>_sensor.h`	Data types for the algorithm
`sns_<algo_name>_sensor_island.c`	Island mode functions for the algorithm
`sns_<algo_name>_sensor_instance.c`	Normal mode functions for the sensor instance
`sns_<algo_name>_sensor_instance.h`	Sensor instance data types for the algorithm
`sns_<algo_name>_sensor_instance_island.c`	Island mode functions for the algorithm instance

- The `sns_<algo_name>.c` file must contain a registration function to register as an algorithm library with QSH.

```
sns_rc sns_<algo_name>_register(sns_register_cb const *register_
api)
{
  register_api->init_sensor(sizeof(sns_<algo_name>_sensor_
state),
                            &sns_<algo_name>_api,
                            &sns_<algo_name>_sensor_instance_
api);
  return SNS_RC_SUCCESS;
}
```

- The `sns_<algo_name>_sensor.c` file is used to initialize the sensor, look up for the dependent sensor, and implement other sensor APIs.

```
sns_rc sns_<algo_name>_init(sns_sensor *const this)
{
  SNS_SUID_LOOKUP_INIT(state->suid_lookup_data, NULL);
  sns_suid_lookup_add(this, &state->suid_lookup_data, "accel");
  sns_suid_lookup_add(this, &state->suid_lookup_data, "registry
");
```

- The `sns_<algo_name>_sensor_island.c` and **sns_<algo_name>_sensor_instance_island.c** files must contain functions and code related to the Island mode for a sensor and sensor instance respectively.
- The `sns_<algo_name>_sensor_instance.c` file is important to initialize the sensor instances and implement other sensor instance APIs.

Write a .proto file for custom algorithms

The `.proto` file defines the messages used by the custom algorithm, using the protocol buffer specification language. The proto filename for an algorithm must be the same as the value of the `SNS_STD_SENSOR_ATTRID_API` attribute, set in the respective `sns_<algo_name>_sensor.c` file.

Place the sensor algorithm proto file in the `adsp_proc/qsh_algorithms/oem1/build` directory.

Enable the sensors in the Island mode

QSH supports island configuration for algorithms. Use a dedicated environment flag for each sensor to enable the Island mode support.

For example, `SNS_ISLAND_INCLUDE_<algo_name>` in `sns_<algo_name>.scons`.

Use this flag to decide the value of the `add_island_files` field when a `AddSSCSU()` method is called in the driver `.scons` file.

For more information about island configuration, see Island configuration.

Integrate custom sensor algorithm

To integrate a custom sensor algorithm do the following:

1. Ensure that a build directory is present and includes the SCons file. The path must be similar to `adsp_proc/qsh_algorithms/<algo_name>/build/<algo_name>.scons`.

   The SCons file name must match with the algorithm name in `<algo_name>.scons`.

2. In `adsp_proc/ssc/qsh_algorithms/build/qsh_algorithms.scons` file, add `<algo_name>` entry under the `include_qsh_algorithms_libs` section.

3. If the custom sensor algorithm requires island memory, then add the following entry in the `adsp_proc/qsh_platform/chipset/<chipset_name>/por.py` file.

   ```
   env.AddUsesFlags(['SNS_ISLAND_INCLUDE_<algo_name>'])
   ```

1. Compile the aDSP build. For more information, see Bring up sensors.
2. On successful compilation, you must see: `*.lib` files in the build directory of the new algorithm, `adsp_proc/qsh_algorithms/<algo_name>/build/sensor_img/qdsp6/<chipset_name>.adsp.prod`.
3. Load the device with the compiled aDSP image. For more information, see Bring up sensors.

4 Troubleshoot sensors

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

4.1 Sensor low-power processor compilation failure

Do the following if there is a sensor low-power compilation failure:

1. Follow the compilation instructions provided in the Bring up.
2. Download the tools with the version specified in the software release.
3. Validate that the environment variables (`HEXAGON_ROOT`, `LLVM_ROOT`, and `SECTOOLS_DIR`) are set correctly.

4.2 Non-listing of sensor at QSH client API

Do the following if a sensor isn't listed in the QSH client reference API using the `ssc_sensor_info` tool:

1. To ensure that the sensor is integrated properly, check that the register function is added to the following files:
   - For the sensor algorithm: `adsp_proc/qsh_platform/framework/build/sensor_img/qdsp6/<chipset_name>.adsp.prod/qsh_static_algorithms.c`.
   - For the sensor driver: `adsp_proc/qsh_platform/framework/build/sensor_img/qdsp6/<chipset_name>.adsp.prod/sns_static_drivers.c`.
2. Check that the sensor is probed successfully:
   - To verify that a vendor driver is properly initialized, enable the debug logs in the driver and check them in the sensor QXDM logs.
   - To enable sensor QXDM logs, do the following to select the QDSP6 and SNS log mask in QXDM Professional™:

a. Press **F3**, right-click and select *Configure > Message Packets > Know Messages (By subsystem) > SNS and QDSP6*.

b. Press **F3**, right-click and select *Configure > Log Packets > Know Log Items > Common > SNS (select all)*

For example, in the following magnetometer driver logs, when the bus is configured, the driver reads the *WHO-AM-I* register. If the sensor is connected to the device, it publishes the hardware attributes.

```
[sns_ak0991x_sensor.c 1552] process_timer_events: msg=1025
[sns_ak0991x_sensor.c 1562] process_timer_events: hw_id=0,
state=1
[sns_ak0991x_sensor.c 1570] I3C mode enabled
[sns_ak0991x_sensor.c 1589] Read WHO-AM-I 3144
[sns_ak0991x_sensor.c 1660] Find Sensor. state->hw_is_present=1
state->device_select=8 [sns_ak0991x_sensor.c 1675] start power
timer #0: hw_id=0 pend_state=3
[sns_ak0991x_sensor.c 550] start_power_rail_timer:
timeout=19200000 state=3
[sns_ak0991x_sensor.c 575] power timer is started: hw_id=0,
pend_state=3
[sns_ak0991x_sensor.c 1290] AK0991x HW Present. Publishing
available
[sns_ak0991x_sensor.c 1682] AK0991X HW present. device_select: 8
```


4.3 Sensor is listed but unable to receive sensor data

Do the following if a sensor is listed in the QSH client reference API, but unable to receive the sensor data:

1. Check that the sensor JSON file is present in the `/etc/sensors/config/` location and the corresponding parsed data is present in the `<registry_path>` location.
2. Verify that the JSON platform configurations are set correctly.
3. For a driver interrupt, check that `is_dri` and `dri_irq_num` configuration parameters are set correctly in a driver-specific JSON file.

   For more information about configuration, see registry_in_qsh.
4. Run the following command to stream the specified sensor:

```
see_workhorse [-sensor=][-sample_rate=] [-batch_period=] [-
calibrated=<0 | 1>] [-wakeup=<0 | 1>]
```

For example:

```
see_workhorse -sensor=accel -sample_rate=max -duration=30 -
display_events=1
```

For more information about tools, see Tools section of Qualcomm Linux Sensors Guide.

Note: More use cases will be added in the next revision. Visit Qualcomm Tech support (Salesforce) website directly at https://support.qualcomm.com/.

5 References

The following is the list of resources to develop your own drivers. Sensors-related acronyms and expansions for better understanding.

5.1 Related documents

List of resources to develop your own drivers.

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

Title	Number
Qualcomm Technologies, Inc.	
Sensors Execution Environment Pressure Sensor Driver Requirement Specification	80-P9361-2
Sensors Execution Environment Driver Data Sheet for Motion Detect	80-P9361-4
Sensors Execution Environment Driver Data Sheet for Accelerometer	80-P9361-5
Sensors Execution Environment Accelerometer Sensor Driver Requirement Specification	80-P9361-7
Sensors Execution Environment Gyroscope Sensor Driver Requirement Specification	80-P9361-8
Sensors Execution Environment Magnetometer Sensor Driver Requirement Specification	80-P9361-9
Sensors Execution Environment Driver Data Sheet for Magnetometer	80-P9361-10
Sensors Execution Environment Driver Data Sheet for Gyroscope	80-P9361-11
Sensors Execution Environment Ambient Light Sensor Driver Requirement Specification	80-P9361-12
Sensors Execution Environment Proximity Driver Requirement Specification	80-P9361-13
Sensors Execution Environment Driver Data Sheet for Ambient Light Sensor	80-P9361-14
Sensors Execution Environment Driver Data Sheet for Proximity Sensor	80-P9361-15
Sensors Execution Environment Driver Data Sheet for Pressure Sensor	80-P9361-17

Title	Number
Qualcomm® OpenSSC Driver Guidelines and Bringup Guide	80-P9361-23
Adding a Custom Sensors Algorithm with Sensors Execution Environment (SEE)	80-P9301-67
Qualcomm Linux Build Guide	80-70018-254
Qualcomm Linux Yocto Guide	80-70018-27
RB3 Gen 2 Development Kit	80-70018-251

5.2 Acronyms and terms

Sensors-related acronyms and expansions for better understanding are listed here.

Acronym or term	Definition
ALS	Ambient light sensor
ASCP	Asynchronous COM port
DRI	Data ready interrupt
EIS	Electronic image stabilization
GRV	Game rotation vector
HLOS	High level operating system
IBI	In-band interrupt
IMU	Inertial measurement unit
JSON	JavaScript Object Notation
MCPS	Million cycles per second
MTU	Maximum transmission unit
ODR	Output data rate
PD	Protection domain
PDR	Protection domain restart
QSH	Qualcomm sensing hub
QuP	Qualcomm universal peripherals
QuRT	Qualcomm real time operating system
RTOS	Real time operating system
SCP	Synchronous COM port
SCons	Software construction
SMP2P	Shared memory point to point
SoC	System-on-chip
SSC	Qualcomm® Snapdragon™ sensors core
SSR	Subsystem restart
SUID	Sensor unique identifier
UART	Universal asynchronous receiver/transmitter

LEGAL INFORMATION

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

1) Legal Notice.

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

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

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

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

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

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

This Material is subject to change without further notification.

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

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

2) Trademark and Product Attribution Statements.

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

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