# Amazon Sidewalk Sid API Developer Guide

Protocol Stack 1.0, Document Revision A.1

March 6, 2024

# Contents

# Chapter 1

# Scope

The scope of this document is to explain the usage of the Amazon Sidewalk Application Programming Interface (API) for an application developer. This document does not attempt to discuss the details of Amazon Sidewalk features and functionality. Detailed explanation of Amazon Sidewalk features and functionality is documented by feature specific application notes, specifications, developer guides and the Amazon Sidewalk white paper.

# Chapter 2

# Overview of Functionality

Amazon Sidewalk provides an Application Programming Interface API (the "Sid API") that allows Amazon Sidewalk device creators to onboard their devices to Amazon Sidewalk, and to manage an encrypted data pipe over BLE and sub-GHz radio interfaces.

A developer's application interacts with the Amazon Sidewalk stack through the Sid API. The Sid API supports the following operations for Amazon Sidewalk links:

1. Initialize and de-initialize links.
2. Start and stop links.
3. Send and receive messages.
4. Set and get configuration through IOCTLs.

## 2.1 Supported Links

Amazon Sidewalk supports up to three link types that can be used by devices to transport messages between the application running on the Endpoint device and the application services interacting with AWS IoT Core for Amazon Sidewalk. These links are enumerated in the following code, and described below.

```
/**
 * Describes the link types supported by the Sidewalk stack.
 *
 * Note: Previously SID_LINK_TYPE_BLE used here now maps to SID_LINK_TYPE_1.
 * The change is done to abstract link names from physical link types since
 * more combinations are expected to be added in the future.
 */
enum SID_LINK_TYPE {
    /** Bluetooth Low Energy link */
    SID_LINK_TYPE_1 = 1 << 0,
    /** 900 MHz link for FSK */
    SID_LINK_TYPE_2 = 1 << 1,
    /** 900 MHz link for LoRa */
    SID_LINK_TYPE_3 = 1 << 2,
    /** Any Link Type */
    SID_LINK_TYPE_ANY = INT_MAX,
};

enum SID_LINK_TYPE_idx {
    /** Bluetooth Low Energy link */
    SID_LINK_TYPE_1_IDX = 0,
    /** 900 MHz link for FSK */
    SID_LINK_TYPE_2_IDX = 1,

    /** 900 MHz link for LoRa */
    SID_LINK_TYPE_3_IDX = 2,
    /** sid max supported links */
    SID_LINK_TYPE_MAX_IDX,
};
```

### 2.1.1 BLE link

Amazon Sidewalk over BLE is called `SID_LINK_TYPE_1`. Amazon Sidewalk over BLE uses the Bluetooth Specification v4.2+ standard to transport messages through Amazon Sidewalk enabled Amazon/Ring Gateways.

The Amazon Sidewalk stack configures the BLE advertisement payload according to the use case to allow an Amazon Sidewalk BLE Gateway to detect the Endpoint and establish a connection. The beacons are periodically advertised by the Endpoint. The beacon advertisements switch between fast (160ms) and slow (1s) periodicity.

After detecting a beacon from the Endpoint the Gateway establishes connection only when the Amazon Sidewalk cloud instructs the Gateway to connect. (Note that the Amazon Sidewalk Cloud provides network management services for Amazon Sidewalk. The operation and control of these services is not visible to the developer.) The Gateway uses the standard BLE connection establishment procedure.

If, and only if, one of the following three conditions occurs, a BLE Gateway connects to the Endpoint:

1. The Endpoint requests time synchronization by updating the advertisement payload.
   (a) The advertisement payload in the beacon is correctly received by the Gateway
   (b) The advertisement payload contains information that informs the Gateway that the Endpoint is requesting time synchronization.
2. The Endpoint requests a connection by updating the advertisement payload.
   (a) If the Endpoint has data to send to the AWS IoT Core for Amazon Sidewalk, or Amazon Sidewalk cloud services, it updates the request to the Gateway that it has data to send.
   (b) The Gateway propagates this request to Amazon Sidewalk cloud services.
   (c) The Amazon Sidewalk cloud service requests the Gateway to establish an encrypted connection with the Endpoint.
3. Amazon Sidewalk cloud services or the application on AWS IoT Core for Amazon Sidewalk has data to send to the Endpoint.
   (a) The cloud service instructs a Gateway in range of the Endpoint to establish connection with the Endpoint to send downlink data. (An Endpoint is in range of a Gateway if the Gateway can listen to the beacons advertised by the Endpoint and periodically forward to the Amazon Sidewalk Cloud.)

For more details see the Amazon Sidewalk Specification.

### 2.1.2 FSK Link

Amazon Sidewalk over FSK is called `SID_LINK_TYPE_2`. Amazon Sidewalk over FSK uses a 2-GFSK modulation scheme in sub-GHz spectrum.

If the Gateway is using FSK, it periodically transmits a beacon. The beacon is discovered by FSK capable Endpoints to detect the presence of Amazon Sidewalk sub-GHz capable Gateways. The beacon also allows Endpoints that are using FSK to communicate with the Gateway to synchronize and schedule uplink and downlink messages.

For more details see the Amazon Sidewalk Specification.

### 2.1.3 LoRa Link

Amazon Sidewalk over LoRa is called `SID_LINK_TYPE_3`. Amazon Sidewalk over LoRa uses CSS in sub-GHz spectrum. Amazon Sidewalk over LoRa is an asynchronous radio protocol and so does not use beacons.

For more details see the Amazon Sidewalk Specification.

## 2.2 Supported Link modes

Amazon Sidewalk supports two link modes `SID_LINK_MODE_CLOUD`, and `SID_LINK_MODE_MOBILE`.

1. `SID_LINK_MODE_CLOUD` is used for `SID_LINK_TYPE_1`, `SID_LINK_TYPE_2`, and `SID_LINK_TYPE_3`.
2. `SID_LINK_MODE_MOBILE` is only used for link type `SID_LINK_TYPE_1`, (BLE).

If a communication channel is established between an Endpoint and the Amazon Sidewalk cloud/AWS IoT Core for Amazon Sidewalk through a Gateway, the link mode for each link is set to `SID_LINK_MODE_CLOUD`.

If a communication channel is established between an Endpoint and the Sidewalk Mobile SDK, the link mode for the `SID_LINK_TYPE_1` link is set to `SID_LINK_MODE_MOBILE`.

A session establishment procedure is used to establish the encrypted link between an Endpoint and the Amazon Sidewalk Mobile SDK.

For more details see the Amazon Sidewalk Specification.

```
/**
  * Describes the link modes supported on each link type.
  *
  * Link mode determines the destination of the messages
  * that sid api user can send on a link.
  */
enum sid_link_mode {
    /** Messages can be sent to cloud only, when a link type
     *  notifies support of this mode */
    SID_LINK_MODE_CLOUD = 1 << 0,
    /** Messages can be sent to mobile only, when a link type
     *  notifies support of this mode */
    SID_LINK_MODE_MOBILE = 1 << 1,
    /** Invalid mode */
    SID_LINK_MODE_INVALID = INT_MAX,
};
```

## 2.3 Registration and Deregistration

An Endpoint that is provisioned with Amazon Sidewalk certificates can use Registration to authenticate itself as an Amazon Sidewalk Endpoint. Registration involves transfer of messages between the Endpoint and the Amazon Sidewalk cloud services. During Registration, messages are transferred through Gateways or through the Amazon Sidewalk Mobile SDK. Registration establishes the authenticity of the Endpoint to the Amazon Sidewalk cloud and also establishes the authenticity of the Amazon Sidewalk cloud to the

Endpoint. Successful registration is required before the Endpoint can access services offered by Amazon Sidewalk.

The Endpoint requires security credentials to encrypt communications with Amazon Sidewalk Cloud services, and with AWS IoT Core for Amazon Sidewalk. During the registration process the security credentials are stored in the Endpoint's non-volatile storage.

Messages exchanged between the Endpoint and the AWS IoT application are encrypted with two keys: the Application server key and the Network server key.

The Application server key is derived during registration with AWS IoT Core for Amazon Sidewalk. The Network server key is derived during registration between the Endpoint and the Amazon Sidewalk cloud service.

User payloads are secured using two levels of encryption:

1. The Application server key is used with the AES-CTR encryption scheme, to encrypt the User's payload.
2. The Network server key is used with the AES-GCM and AES-CTR encryption scheme, as a second layer of encryption on the payload.

Internal messages between the Amazon Sidewalk stack and cloud services are only encrypted with the Network server key. For more details on security refer to the Amazon Sidewalk white paper.

Endpoint registration is only supported on `SID_LINK_TYPE_2` (FSK) and `SID_LINK_TYPE_1` (BLE). Endpoints cannot register with Amazon Sidewalk using `SID_LINK_TYPE_3` (LoRa). Please note that only consent-enabled Amazon Sidewalk Gateways can be used to register Endpoints. There is no explicit API call to the Amazon Sidewalk stack to start the process of registration over `SID_LINK_TYPE_2` or `SID_LINK_TYPE_1`.

The Amazon Sidewalk stack automatically detects the registration status and triggers the registration procedure with the Amazon Sidewalk cloud services if the following conditions are met:

1. The Amazon Sidewalk stack is initialized.
2. The link is started.
3. The Endpoint is not already registered.

The APIs to initialize and start the links are described below.

The Amazon Sidewalk stack exposes an API to de-register through the factory reset API, see section 2.3.4.

Registration uses one of three procedures described below.

### 2.3.1 Registration using Amazon Sidewalk Mobile SDK

The Amazon Sidewalk Mobile SDK allows a developer to integrate endpoint registration and de-registration controls into the developer's own mobile application. (The Amazon Sidewalk Mobile SDK also supports a connection to an Endpoint to directly transmit and receive packets.)

The Endpoint uses `SID_LINK_TYPE_1` (BLE) to register, and the Amazon Sidewalk Mobile SDK uses the phone's BLE to communicate with the Endpoint.

During registration the Endpoint communicates with a mobile app that uses the Amazon Sidewalk Mobile SDK. The Amazon Sidewalk Mobile SDK handles the interaction between the Amazon Sidewalk cloud and the edge device, exchanging keys and registering the edge device, so that the edge device can then connect to the Amazon Sidewalk via Amazon Sidewalk gateways.

1. The Endpoint's beacons communicate its pre-registered state to the Gateway.
2. The Amazon Sidewalk Mobile SDK connects to the Endpoint.
3. The Amazon Sidewalk Mobile SDK triggers the registration sequence.

4. The registration sequence completes.

For more details see the Amazon Sidewalk Mobile SDK Developer Guide.

### 2.3.2 Registration using FFS or FFN over BLE

Registration using FFS/FFN (Frustration Free Setup/Frustration Free Networking) over BLE requires `SID_LINK_TYPE_1` (BLE).

The Endpoint uses `SID_LINK_TYPE_1` to register. During registration the Endpoint communicates with an Amazon Sidewalk enabled BLE Gateway. Registration proceeds as follows:

1. The Endpoint's beacons communicate its pre-registered state to the Gateway.
2. The Gateway is instructed by Amazon Sidewalk cloud services to connect to the Endpoint.
3. The Gateway triggers the registration sequence.
4. The registration sequence completes.

### 2.3.3 Registration using FFS or FFN over FSK

Registration using FFS/FFN over FSK requires `SID_LINK_TYPE_2` (FSK)

The Endpoint uses `SID_LINK_TYPE_2` to register. During registration the Endpoint communicates an Amazon Sidewalk enabled FSK Gateway. Registration proceeds as follows:

1. The Endpoint scans for Amazon Sidewalk FSK beacons.
2. The Endpoint synchronizes to a beacon from a Gateway that has Amazon Sidewalk consent enabled.
3. The Endpoint triggers the registration sequence.
4. The registration sequence completes.

Please note that in case of FSK, it is the Endpoint that starts the registration sequence while in the case of BLE, the registration is started by the Amazon Sidewalk cloud services through an Amazon Sidewalk BLE Gateway

### 2.3.4 De-Registration

The Endpoint can be de-registered from Amazon Sidewalk. The de-registration process involves deletion of security keys and configuration parameters from the Endpoint's non-volatile storage. De-registration of an Endpoint uses one of three procedures described below.

1. **Endpoint application initiated**: the Sid API exposes the factory reset API. Calling this API de-registers the Endpoint from Amazon Sidewalk. The `on factory reset` callback notifies the developer's application of the status of de-registration by the Amazon Sidewalk stack. Note that calling this API changes the device's registered state to ready for registration. If the Amazon Sidewalk stack is not stopped or de-initialized, the Endpoint can start the process of registration immediately through FFS/FFN through an Amazon Sidewalk Gateway or through a mobile app that has Amazon Sidewalk Mobile SDK support.
2. **AWS IoT application service initiated**: The AWS IoT application can de-register the device by issuing a factory reset command to the Amazon Sidewalk stack running on the Endpoint. The `on factory reset` callback notifies the developer's Endpoint application that the Endpoint is de-registered from Amazon Sidewalk.
3. **Initiated via Sidewalk Mobile SDK**: The Amazon Sidewalk Mobile SDK exposes an API for the core application to trigger Amazon Sidewalk Endpoint de-registration.

(a) When the Amazon Sidewalk Mobile SDK triggers a request to de-register the Endpoint, The Amazon Sidewalk cloud sends a Factory Reset command which shall be routed via an Amazon Sidewalk Gateway OR the Amazon Sidewalk Mobile SDK. The routing choice for a factory reset command is based on the range of the Endpoint to the mobile.

(b) When the Endpoint is in range of the Amazon Sidewalk Mobile SDK, the de-registration process is simplified because the Amazon Sidewalk Mobile SDK has already established a direct communication path to the Endpoint to de-register. Using the Amazon Sidewalk Mobile SDK to inject a factory reset command is also useful if the developer's application triggers the deregistration procedure when the Endpoint is not in range of an Amazon Sidewalk gateway.

```
/**
 * Describes the Registration status of the Sidewalk stack.
 */
enum sid_registration_status {
    /** Used to indicate  Sidewalk stack is registered with
     *  Sidewalk cloud services */
    SID_STATUS_REGISTERED = 0,
    /** Used to indicate  Sidewalk stack is not registered
     *  with Sidewalk cloud services */
    SID_STATUS_NOT_REGISTERED = 1,
};
```

## 2.4 Time Synchronization

The Amazon Sidewalk stack uses GPS time as a common frame of reference between Endpoints and the Amazon Sidewalk cloud service. The GPS time is used for encryption, and for derivation of the keys used to encrypt messages exchanged between Endpoints and Amazon Sidewalk services. A device has to be in the registered state to request time from the Amazon Sidewalk cloud service. After device registration is complete, or after the device resets in the registered state, the Amazon Sidewalk stack requests GPS time from the Amazon Sidewalk service. The time sync request process is described below:

1. **Time sync on `SID_LINK_TYPE_1` (BLE)**
   (a) The Endpoint updates the advertisement payload on the beacon requesting time.
   (b) Any BLE Gateway in range of the Endpoint forwards the request to the cloud service.
   (c) The cloud service authenticates the Endpoint using its beacon payload.
   (d) If authentication succeeds, the cloud service instructs the BLE Gateway through which the beacon was received to connect to the Endpoint and provide the time.
   (e) The BLE Gateway connects to the Endpoint.
   (f) The Amazon Sidewalk cloud service provides the time through the BLE Gateway. (The Endpoint does not need to send a time sync request message.)
2. **Time sync on `SID_LINK_TYPE_2` (FSK)**
   (a) The Endpoint synchronizes to the beacon.
   (b) The Endpoint sends a time sync request message to the Amazon Sidewalk service through the FSK enabled Gateway.
   (c) The Amazon Sidewalk cloud authenticates the request.
   (d) If authentication succeeds, the cloud service sends a time sync response message to the Endpoint. This message contains the GPS time.

   (e) The EP authenticates the time sync response message.

   (f) If authentication succeeds, the Endpoint accepts the GPS time from the message.

3. **Time sync on `SID_LINK_TYPE_3` (LoRa)**

   (a) The time sync request message is sent by the Endpoint to the Amazon Sidewalk service through the LoRa enabled Amazon Sidewalk Gateway. The time sync request is sent soon after reset. The Amazon Sidewalk cloud authenticates the request and provides time to the Endpoint in the time sync response message. The Endpoint authenticates the time sync response message before accepting the time from the message.

To compensate for clock drift, the Endpoint periodically requests updated GPS time from the Amazon Sidewalk service. The periodicity is dependent on the device tolerance to clock drift and is the same for all of the device's links. For more details on time sync request periodicity of the Endpoint see the Amazon Sidewalk Specification.

When the Amazon Sidewalk stack is in time synced state, the developer's application can get the GPS time by querying the stack. The Amazon Sidewalk stack does not provide UTC or local time.

```
/**
 * Describes the Time synchronization status of the Sidewalk
 * stack with the cloud services.
 * Sidewalk security relies on the synchronization of time
 * between the cloud services and the end device.
 *
 * @note Sidewalk stack can have time synchronization with
 * cloud service only when the device is registered
 *
 */
enum sid_time_sync_status {
    /** Used to indicate Sidewalk stack is registered with
     *  Sidewalk cloud services */
    SID_STATUS_TIME_SYNCED = 0,
    /** Used to indicate Sidewalk stack is not time synchronized
     *  with cloud services */
    SID_STATUS_NO_TIME = 1,
};


/**
 * Option to get current time.
 *
 * @attention  GPS time is the time in milliseconds since 6/1/1980.
 * GPS time is not adjusted by leap seconds.
 * UTC time is the number of milliseconds since 1/1/1970.
 * Local time is the adjusted time taking timezone into consideration.
 */
enum sid_time_format {
    /** Option to get current gps time */
    SID_GET_GPS_TIME = 0,
};
```

## 2.5 Stack States

The Amazon Sidewalk stack state of an Endpoint can be queried through a Sid API. The Amazon Sidewalk stack state is ready only when the Endpoint is registered, time is synced and at least one of the links is connected. The developer's application can only send and receive messages when the Amazon Sidewalk stack is in ready state. For the Amazon Sidewalk stack states see table 2.1.

| State | Type | Value |
|---|---|---|
| Sidewalk registration state | Boolean | Endpoint's registration status<br>`0 = registered`<br>`1 = not registered` |
| Sidewalk time sync state | Boolean | Endpoint's time sync status<br>`0 = Endpoint has time sync`<br>`1 = Endpoint does not have time sync` |
| Sidewalk link status | Boolean | Amazon Sidewalk link connection status (per link type)<br>`0 = Endpoint is connected`<br>`1 = Endpoint is not connected` |

Table 2.1: Amazon Sidewalk stack states.

```
/**
 * Describes the state of the Sidewalk stack.
 */
enum sid_state {
    /** Used when the Sidewalk stack is ready to
     * send and receive messages */
    SID_STATE_READY = 0,
    /** Used when the Sidewalk stack is unable to send or receive
     *  messages, such as when the device is not registered
     *  or link gets disconnected or time is not synced */
    SID_STATE_NOT_READY = 1,
    /** Used when the Sidewalk stack encountered an error.
     *  Use sid_get_error() for a diagnostic
     *  error code */
    SID_STATE_ERROR = 2,
    /** Used when the Sidewalk stack is ready to send and receive
     *  messages only with secure channel establishment completed
     *  but device is not registered and time is not synced */
    SID_STATE_SECURE_CHANNEL_READY = 3,
};
```

## 2.6 Stack Status

The Amazon Sidewalk stack status contains: registration state, time synchronization state, link connection states for each link, and link modes supported for each link.

For registration and time synchronization states see section 2.5.

The `link_status` mask holds the connection status of all the links supported by Amazon Sidewalk. The following code defines the `link_status` mask values. Note that concurrent FSK and LoRa is not supported.

```
// All links are in disconnected state
link_status_mask = 0;
```

```
// Only BLE link is connected
link_status_mask = 1; // ( link_status_mask & SID_LINK_TYPE_1 == 1)

 // Only FSK link is connected
link_status_mask = 2; // ( link_status_mask & SID_LINK_TYPE_2 == 1)

 // Only LoRa link is connected
link_status_mask = 4; // ( link_status_mask & SID_LINK_TYPE_2 == 1)

 // BLE and FSK link is connected
link_status_mask = 3; // ( link_status_mask & SID_LINK_TYPE_1 == 1
                      // and link_status_mask & SID_LINK_TYPE_2 == 1)

  // BLE and LoRa link is connected
link_status_mask = 5; // ( link_status_mask & SID_LINK_TYPE_1 == 1
                      // and link_status_mask & SID_LINK_TYPE_3 == 1)
```

The supported link modes is an array giving the connection mode for each link. Entries in the supported link modes array are accessed using the link type index.

```
/**
 * Describes the link connection status with the gateway device
 *
 */
struct sid_status_detail {
    /**
     * Used to indicate which link is up, if the bit corresponding to a link
     * is set it is up otherwise it is down.
     * For example to check if SID_LINK_TYPE_1 is up,
     *!!(link_status_mask & SID_LINK_TYPE_1) needs to be true
     *
     * supported link modes indicate the modes supported by each link.
     * A link type may support more than one mode simultaneously
     *
     * @see #sid_link_type_idx
     *
     */
    enum sid_registration_status registration_status;
    enum sid_time_sync_status time_sync_status;
    uint32_t link_status_mask;
    uint32_t supported_link_modes[SID_LINK_TYPE_MAX_IDX];
};
```

## 2.7 Link States

The connection establishment and maintenance of the established connection is different for each of the link types supported by Amazon Sidewalk, and is described below.

### 2.7.1 BLE Link States

When the Endpoint has messages to send to AWS IoT Core for Amazon Sidewalk, the Endpoint updates its beacon's advertisement payload to request a connection. If there is an Amazon Sidewalk BLE Gateway in range, the gateway establishes a connection. For details of connection establishment see section 2.1.1

The Endpoint's Amazon Sidewalk stack maintains the connection with the Amazon Sidewalk BLE Gateway only when there is uplink or downlink traffic. If the Endpoint does not transmit uplink traffic to the gateway, or receive downlink traffic from the Gateway for a period of 30 seconds, the Endpoint's stack terminates the connection with the Amazon Sidewalk BLE Gateway and the developer's application is notified. To re-connect, the developer's application again requests the Amazon Sidewalk stack to updates its beacon's advertisement payload to request a connection.

The developer's application can prevent the Amazon Sidewalk stack from terminating the connection by sending at least one uplink message to AWS IoT Core for Amazon Sidewalk every 30 seconds, or if at least one downlink message is received by the Endpoint every 30 seconds the connection is maintained by the Amazon Sidewalk stack.

When the Endpoint on a BLE link does not have time synchronization, the Amazon Sidewalk stack updates the beacon's payload requesting time sync. This is handled automatically by the Amazon Sidewalk stack. When the Endpoint does not have time synchronization, the developer's application is notified of the time acquisition status and the developer's application cannot send or receive messages until GPS time is acquired.

### 2.7.2 FSK Link States

The connection establishment procedure for an Amazon Sidewalk FSK Gateway starts with the Endpoint synchronizing with an FSK Gateway's beacon that has consent enabled. When the beacon is synchronized, the Amazon Sidewalk stack starts the time acquistion process if the End point does not have GPS time. When the Endpoint has acquired GPS time, the developer is notified of the successful time acquisition status. After the Endpoint successfully acquires time, the Amazon Sidewalk stack starts the join procedure with Amazon Sidewalk cloud. See the Amazon Sidewalk Specification for more details on Join procedure.

If the Endpoint has not successfully acquired time or completed the Join procedure, the developer's application cannot send or receive messages. When the Endpoint has completed the Join procedure successfully, the developer's application is notified of the successful link connection status and the developer's application can send and receive messages. When the Endpoint misses 3 consecutive beacons from the already synchronized Amazon Sidewalk FSK gateway, the developer's applcation is notified of link disconnected status. The Amazon Sidewalk stack starts the gateway discovery procedure to discover Amazon Sidewalk FSK gateways after disconnection.

Note that time synchronization process is started by the Amazon Sidewalk stack only when the Endpoint does not have GPS time. The synchronization with a Amazon Sidewalk FSK gateway's beacon will not always trigger time acquisition process. However, every time the Endpoint transistions from gateway discovering state to a successful gateway discovered state, Join procedure is triggered.

The configuration of the device profile settings configuration determines the following:

1. The parameters that are sent in the join request.
2. The downlink schedules of the Endpoint.

See the Amazon Sidewalk Sub-GHz Device Profiles Application Note for more details.

### 2.7.3 LoRa Link States

The LoRa link does not establish a connection with an Amazon Sidewalk LoRa Gateway (the link between the Endpoint and the Gateway is asynchronous and connectionless). After the link is initialized and started, the Amazon Sidewalk stack starts the time acquistion process if the End point does not have GPS time. When the Endpoint has acquired GPS time, the developer is notified of the successful time acquisition status. After the Endpoint successfully acquires time, the Amazon Sidewalk stack starts the join procedure with Amazon Sidewalk cloud. See the Amazon Sidewalk Specification for more details on Join procedure.

If the Endpoint has not successfully acquired time or completed the Join procedure, the developer's application cannot send or receive messages. When the Endpoint has completed the Join procedure successfully, the developer's application is notified of the successful link connection status and the developer's application can send and receive messages. The configuration of the device profile settings configuration determines the following:

1. The parameters that are sent in the join request.
2. The downlink schedules of the Endpoint.
3. In some cases, the requirements on the Endpoint to send periodic uplinks.

See the Amazon Sidewalk Sub-GHz Device Profiles Application Note for more details.

### 2.7.4 Route Selection For Downlinks

The last uplink message affects how Amazon Sidewalk processes downlink messages. Cloud services send any pending downlink packets using the link on which the Endpoint last sent an uplink packet.

For BLE, if there is no active link, downlink packets for an Endpoint are sent to the BLE Gateway that last received a beacon from that Endpoint.

For FSK, downlink packets for an Endpoint are sent to the FSK Gateway that last received an uplink message from that Endpoint.

For LoRa, downlink packets for an Endpoint are sent to the LoRa Gateway that last received an uplink message from that Endpoint.

If BLE beacons and uplinks from FSK or LoRa reach the cloud simultaneously, BLE is the preferred link for the downlink transmission.

## 2.8 Amazon Sidewalk Messages

The Amazon Sidewalk stack provides APIs to send messages to AWS IoT and notifies the developer's application of messages received from AWS IoT Core for Amazon Sidewalk.

Each Amazon Sidewalk message is associated with metadata that uniquely identifies the message. The metadata in a message also notifies the Amazon Sidewalk stack of the attributes that are to be applied to each message.

Amazon Sidewalk messages are agnostic to the link on which the messages are being sent. The only varying factor is the MTU of the link. The message types and attributes of messages are the same irrespective of the link the message is being sent on.

For the MTU of the links supported by Amazon Sidewalk see table 2.2.

| **Link Type** | **MTU (Bytes)** |
|---|---|
| BLE(link type 1) | 255 |
| FSK(link type 2) | 200 |
| LoRa(link type 3) | 19 |

Table 2.2: MTU of Amazon Sidewalk links.

The message consists of `msg` and `msg_desc` data structures. The `msg` data structure has size and a pointer to the payload of the message. The `msg_desc` data structure has message attributes which are described in detail below:

```
union sid_msg_desc_attributes {
    /** Attributes that are applied only when message is
     *  transmitted, see #sid_put_msg */
    struct sid_msg_desc_tx_attributes tx_attr;
    /** Attributes reported per message when the message is
```

```
     *  reported to the developer's application, see #on_msg_received */
    struct sid_msg_desc_rx_attributes rx_attr;
};
struct sid_msg_desc {
    /** The link type on which this message is to be sent
      * or was   received from */
    uint32_t link_type;
    /** The message type */
    enum sid_msg_type type;
    /** The link mode on which message is sent or received */
    enum sid_link_mode link_mode;
    /** The id associated with a message, generated by the
     *  Sidewalk stack
     *  The maximum value the id can take is 0x3FFF after
     *  which the id resets% to
     */
    uint16_t id;
    /** Attributes applied to the message */
    union sid_msg_desc_attributes msg_desc_attr;
};
/**
 * Describes a message payload.
 */
struct sid_msg {
    void *data;
    size_t size;
};
```

### 2.8.1 Amazon Sidewalk Message Types

Sid API supports four types of messages. The same message types are available to the application on AWS IoT Core for Amazon Sidewalk. When messages are exchanged between the Endpoint and AWS IoT Core for Amazon Sidewalk, the message type is also notified.

The four message types are as follows:

1. `GET` to retrieve some data from destination. This message type elicits a mandatory response from the receiver.
2. `SET` requests to write some data on destined devices. These requests can be used to update feature configuration or trigger an operation on device.
3. `RESP` type messages are sent in response to `GET`, `SET` or `NOTIFY`. `RESP` messages are an empty acknowledgement, and shall not contain any payload data. The `RESP` messages are sent in response only when there is an explicit request from the message sender
4. `NOTIFY` messages are sent by a device or application. `NOTIFY` messages do not correspond to any preceding requests.

When the API to send a message is called, the Amazon Sidewalk stack generates a unique ID. This unique ID is the `message ID`. The developer's application is expected to store this `message ID` to compare with the corresponding `message ID` returned in a send status event or in an acknowledgement from the AWS IoT Core for Amazon Sidewalk. The stack generates a unique `message ID` for all message types except `RESP`. For message type `RESP`, the stack expects the `message ID` to be provided by the developer's application.

**There is no restriction on the use of a message type** in a different manner than what is documented above except for message type `RESP`. For example the message type `GET` can be used to trigger an operation on a device or `NOTIFY` can be used by the developer's application to get some data from destination.

```
/**
 * The Sidewalk stack message types.
 * The messages from cloud services to the End device are designated as
 * "Downlink Messages"
 * The messages from End device to Cloud services are designated as
 * "Uplink Messages"
 */
enum sid_msg_type {
    /** #SID_MSG_TYPE_GET is used by the sender to retrieve information from
     * the receiver, the sender
     * expects a mandatory response from the receiver. On reception of
     * SID_MSG_TYPE_GET, the receiver is expected to send a message with type
     * #SID_MSG_TYPE_RESPONSE with the same message id it received from the
     * message type #SID_MSG_TYPE_GET.
     * This is to ensure the sender can map message type #SID_MSG_TYPE_GET
     * with the received #SID_MSG_TYPE_RESPONSE.
     * Both uplink and downlink messages use this message type.
     * @see sid_put_msg().
     * @see on_msg_received in #sid_event_callbacks.
     */
    SID_MSG_TYPE_GET = 0,
    /** #SID_MSG_TYPE_SET indicates that the sender is expecting the receiver
     *  to take an action on receiving the message and the sender does not
     *  expect a response.
     *  #SID_MSG_TYPE_SET type is used typically by Cloud services to trigger
     *  an action to be preformed by the End device.
     *  Typical users for this message type are downlink messages.
     */
    SID_MSG_TYPE_SET = 1,
    /** #SID_MSG_TYPE_NOTIFY is used to notify cloud services of any periodic
     *  events or events triggered/originated from the device. Cloud services
     *  do not typically use #SID_MSG_TYPE_NOTIFY as the nature of messages
     *  from cloud services to the devices are explicit commands instead of
     *  notifications.
     *  Typical users for this message type are uplink messages.
     */
    SID_MSG_TYPE_NOTIFY = 2,
    /** #SID_MSG_TYPE_RESPONSE is sent as a response to the message of type
     *  #SID_MSG_TYPE_GET.
     *  The sender of #SID_MSG_TYPE_RESPONSE is required to the copy the
     *  message id received in the message of type #SID_MSG_TYPE_GET.
     *  Both uplink and downlink messages use this message type.
     *  @see sid_put_msg().
     *  @see on_msg_received in #sid_event_callbacks.
     */
    SID_MSG_TYPE_RESPONSE = 3,
};
```

### 2.8.2 Amazon Sidewalk Message Sequence Numbers

Every message sent by the Endpoint or received by the Endpoint has a unique sequence number associated with it. When Amazon Sidewalk stack successfully accepts message for transmission, an unique sequence number is assigned to the messsage. The sequence number can be used to track messages exchanged between the Endpoint and AWS IoT core for Amazon Sidewalk. Amazon Sidewalk stack does not guarantee the order of messages sent and received by the Endpoint.

### 2.8.3 Amazon Sidewalk Message Attributes

When sending a message, the message attributes can be used to modify each message's behavior. When receiving a message, the Amazon Sidewalk stack uses the received message attributes to notify the developer's application of the attributes AWS IoT Core for Amazon Sidewalk has configured for the message.

#### 2.8.3.1 Amazon Sidewalk Message Transmit Attributes

The Amazon Sidewalk stack notifies the status of the message sent over the air by configured link and also the status of message received by the AWS IoT Core for Amazon Sidewalk based on the transmit attributes configured on every message. The status of the message whether received by AWS IoT Core for Amazon Sidewalk is reported to the developer's application only when `Transport Ack` is requested by the developer's application.

For each message, the Amazon Sidewalk stack performs retries based on the `Number of Retries` configuration.

The developer's application can configure the following attributes of the message that is to be sent by the Amazon Sidewalk stack:

1. `Request Acknowledgment from AWS IoT`: When this configuration is set to true, the Amazon Sidewalk stack requests AWS Io Core for Amazon Sidewalk to send an Acknowledgment when AWS IoT Core for Amazon Sidewalk receives the message. Note that the acknowledgment is sent by AWS IoT Core for Amazon Sidewalk, and not by the developer's AWS IoT application. This is configured with `Transport Ack` set to `TRUE`. Acknowledgements are sent using messages with message type `RESP` and zero size payload.

2. `Number of Retries` This requires `Transport Ack` set to `TRUE`. When set to `TRUE`, AWS IoT Core for Amazon Sidewalk sends an Acknowledgment back to the Amazon Sidewalk stack. If the Acknowledgment is not received, the Amazon Sidewalk stack retries the message the number of times configured in `Number of Retries`. If there is no Acknowledgement received after `Number of Retries` transmissions, then the Amazon Sidewalk stack sends a failure to send status report to the developer's application.

3. `Time to Live`: The time in milliseconds that the Amazon Sidewalk stack holds the message in its queue before reporting to the developer's application the failure to receive the acknowledgment from AWS IoT.

Please note that the retry periodicity is derived from `Number of Retries` and `Time to Live` configuration.
`Retry periodicity = Time to Live / Number of Retries`

```
/** Attributes applied to the message descriptor on tx */
struct sid_msg_desc_tx_attributes {
    /** Whether this message requests an ack from the AWS IOT service */
    bool request_ack;
    /** Number of retries the Sidewalk stack needs to preform in case the
     *  ack is not received. Setting not applicable if request_ack is set
     *  to false
     */
    uint8_t num_retries;
    /** Total time the Sidewalk stack holds the message in its queue in case
     *  the ack is not received. Setting not applicable if request_ack is set
     *  to false
     */
    uint16_t ttl_in_seconds;
}
```

#### 2.8.3.2 Amazon Sidewalk Message Receive Attributes

Each message received by the Amazon Sidewalk stack is notified to the developer's application using the `on msg received` callback registered by the developer's application. The following are the received message attributes reported on every Amazon Sidewalk message:

1. `Acknowledgement`: The received message is an acknowledgement from the AWS IoT Core for Amazon Sidewalk that it has received the message sent by the Endpoint.
2. `Duplicate`: The received message is a duplicate. The Amazon Sidewalk stack keeps track of the last 10 messages received from AWS IoT Core for Amazon Sidewalk and either reports or filters duplicates. The choice to report or filter can be configured, see section 2.9. A duplicate is a message that has the same `message ID` and payload size as a message that has already been reported to the developer's application.
3. `Acknowledgement Request`: The AWS IoT application can request acknowledgment from the device when sending messages to the device. The Amazon Sidewalk stack reports the status that AWS IoT Core for Amazon Sidewalk requested in an acknowledgement for a message. The Amazon Sidewalk stack responds to AWS IoT Core for Amazon Sidewalk immediately with an Acknowledgment and then reports the message to the developer's application with the acknowledgment request set to `TRUE`. When the developer's application receives the message with acknowledgment set to `TRUE`, the Amazon Sidewalk stack had already queued the Acknowledgment to AWS IoT Core for Amazon Sidewalk.
4. `rssi`: the received signal strength at which the message is received by the link's radio.
5. `snr`: the signal to noise ratio reported by the radio on this message.

```
/** Attributes with which the message is received */
struct sid_msg_desc_rx_attributes {
    /** Whether the message received is an acknowledgement. Acknowledgements
     *  have the same message id as that of message sent but with zero payload
     *  size. See #sid_msg_desc_tx_attributes
     */
    bool is_msg_ack;
    /** Whether the message received is a duplicate. If a message arrives at
     *  the Sidewalk stack with message id and payload size equal to an
     *  already reported message, this message is marked as a duplicate
     *  See #SID_OPTION_SET_MSG_POLICY_FILTER_DUPLICATES
     */
    bool is_msg_duplicate;
    /** Whether the message received has requested an acknowledgement to be
     *  sent, Acknowledgements have the same message id as that of the
     *  received message. The Sidewalk stack immediately queues an
     *  acknowledgement to the sender before propagating this message to the
     *  developer's application*/
    bool ack_requested;
    /** rssi of the received message */
    int8_t rssi;
    /** snr of the received message */
    int8_t snr;
}
```

see section 2.3.4.

## 2.9 Stack IOCTLs

The Sid API allows the developer's application to modify certain aspects of the Amazon Sidewalk stack's behavior through IOCTLs. The following IOCTLs are available for the developer's application to configure:

1. `Set BLE Battery level`: The battery level is advertised in the beacon payload. The developer's application can use this IOCTL to set the current battery level as a percentage. The default value is 75%. The developer's application should update the Amazon Sidewalk stack battery level, so that an accurate value is advertised in the BLE beacon. This IOCTL does not play any role in FSK and LoRa links.
2. `Set Device Profile`: This IOCTL is valid only for FSK and LoRa and does not apply for BLE. The default profile setting for FSK is Profile 1 and for LoRa it is Profile B. For more details on FSK and LoRa device profiles, see Amazon Sidewalk Sub-GHz Device Profiles Application Note.
3. `Get Device Profile`: Get the device's current operating profile. See Amazon Sidewalk Sub-GHz Device Profiles Application Note.
4. `Set filter duplicates`: The Amazon Sidewalk stack can filter or report duplicate messages. This setting allows the developer's application to modify the stack behavior to filter or to report the duplicate messages. When set to `1`, the Amazon Sidewalk stack filters the duplicates and does not report duplicate messages to the developer's application. The default setting is `0`, to filter duplicates. When set to `1`, the Amazon Sidewalk stack reports the duplicates using the `on msg received` callback with the `is_msg_duplicate` set to `1` in the metadata of the message.
5. `Get filter duplicates`: Get the current setting of filter duplicates.
6. `Set link connection policy`: Set the link connection policy. See section 2.9.1.
7. `Get link connection policy`: Get the current setting of link connection policy. See section 2.9.1.
8. `Set auto link connection policy parameters`: Set auto link connection policy parameters. See section 2.9.1.2.
9. `Get auto link connection policy parameters`: Get the current setting of auto link connection policy parameters. See section 2.9.1.2.
10. `Set multi-link connection policy`: Set multi-link connection policy. See section 2.9.1.3.
11. `Get multi-link connection policy`: Get the current setting of multi-link connection policy. See section 2.9.1.3.
12. `Get statistics`: Get statistics report from Amazon Sidewalk stack. See section 2.9.2.
13. `Clear statistics`: Clear statistics collected by Amazon Sidewalk stack. See section 2.9.2.

### 2.9.1 Amazon Sidewalk stack Connection Policy

The connection policy adopted by the Amazon Sidewalk stack to establish and maintain connection with Amazon Sidewalk Gateways can be configured by the developer.

```
/**
 *  Describes the connection policy setting
 */
enum sid_link_connection_policy {
    /** Default setting. Sidewalk stack does not apply any
     *  connection policy. */
    SID_LINK_CONNECTION_POLICY_NONE = 0,
    /** Auto connection policy, Sidewalk stack applies auto connection policy
     *  based on the parameters
     *  configured through auto connection policy parameters. */
    SID_LINK_CONNECTION_POLICY_AUTO_CONNECT,
    /** Multi-link connection policy, Sidewalk stack applies multi-link
     *  connection policy based on the policy
     *  configured through multi-link connection policy. */
    SID_LINK_CONNECTION_POLICY_MULTI_LINK_MANAGER,
    /** Delimiter to enum sid_link_connection_policy */
    SID_LINK_CONNECTION_POLICY_LAST
};

// IOCTL to set link connection policy

// Set link connection policy to SID_LINK_CONNECTION_POLICY_AUTO_CONNECT
enum sid_link_connection_policy set_policy =
                                SID_LINK_CONNECTION_POLICY_AUTO_CONNECT;
sid_error_t ret = sid_option (handle, SID_OPTION_SET_LINK_CONNECTION_POLICY,
                              &set_policy, sizeof(set_policy));

// IOCTL to get link connection policy

// Get configured link connection policy
enum sid_link_connection_policy get_policy;
sid_error_t ret = sid_option(handle, SID_OPTION_GET_LINK_CONNECTION_POLICY,
                             &get_policy, sizeof(get_policy));
```

#### 2.9.1.1 SID_LINK_CONNECTION_POLICY_NONE

The Amazon Sidewalk stack does not play a role in the connection establishment and its maintenance with the Amazon Sidewalk Network. The developer is expected to handle the connection establishment and its maintenance in this configuration. The default connection policy the Amazon Sidewalk stack is configured with is SID_LINK_CONNECTION_POLICY_NONE

#### 2.9.1.2 SID_LINK_CONNECTION_POLICY_AUTO_CONNECT

The developer provides the parameters of the connection establishment and maintenance under this configuration of connection policy. For details on auto connection policy and its parameters, Please see Amazon Sidewalk Multi-link Application Note

```
/**
 *  Describes link auto connect parameters
 */
struct sid_link_auto_connect_params {
    /** Sid link type */
    enum sid_link_type link_type;
    /** Auto connect policy if set enable is equal to 0 */
    bool enable;
    /** Priority of the link type when more than one link is specified
     *  using #sid_put_msg
     *  0 is highest priority, 2 is lowest priority */
    uint8_t priority;
    /** Maximum period upto which the stack attempts to form a connection */
    uint16_t connection_attempt_timeout_seconds;
};

// IOCTL to set auto connect parameters
// Set auto connection params for SID_LINK_TYPE_1 with priority 1 and
// connection_attempt_timeout_seconds to 30
struct sid_link_auto_connect_params params_link_1 = {
    .link_type = SID_LINK_TYPE_1,
    .enable = true,
    .priority = 0,
    .connection_attempt_timeout_seconds = 30
};

sid_error_t ret = sid_option(handle,
                             SID_OPTION_SET_LINK_POLICY_AUTO_CONNECT_PARAMS,
                             &params_link_1, sizeof(params_link_1));

// IOCTL to get auto connect parameters

// Get auto connection parameters configured for SID_LINK_TYPE_1
struct sid_link_auto_connect_params get_params_link_1 = {
    .link_type = SID_LINK_TYPE_1,
};
ret = sid_option(handle, SID_OPTION_GET_LINK_POLICY_AUTO_CONNECT_PARAMS,
                 &get_params_link_1, sizeof(get_params_link_1));
```

#### 2.9.1.3 SID_LINK_CONNECTION_POLICY_MULTI_LINK_MANAGER

The multi-link connection policy determines the message uplink attributes and link connection attributes and does not expect the developer to configure these parameters while sending the message. For details on multi-link connection policy, Please see Amazon Sidewalk Multi-link Application Note.

```
/** multi-link policy modes */
enum sid_link_multi_link_policy {
    /** Default policy, Default Setting, All links are enabled */
    SID_LINK_MULTI_LINK_POLICY_DEFAULT = 0,
    /** Policy optimized for better power consumption */
    SID_LINK_MULTI_LINK_POLICY_POWER_SAVE = 1,
    /** Policy optimized for performance, better throughput */
    SID_LINK_MULTI_LINK_POLICY_PERFORMANCE = 2,
    /** Policy optimized for latency, faster uplinks */
    SID_LINK_MULTI_LINK_POLICY_LATENCY = 3,
    /** Policy optimized for reliability, messages have a longer Time to live
     *  and more retries */
    SID_LINK_MULTI_LINK_POLICY_RELIABILITY = 4,
    /** Delimiter to enum sid_link_power_policy_type */
    SID_LINK_MULTI_LINK_POLICY_LAST
};

// IOCTL to set multi-link policy

// Set multi-link policy to SID_LINK_MULTI_LINK_POLICY_PERFORMANCE
enum sid_link_multi_link_policy policy =
     SID_LINK_MULTI_LINK_POLICY_PERFORMANCE;
sid_error_t ret = sid_option(handle,
                             SID_OPTION_SET_LINK_POLICY_MULTI_LINK_POLICY,
                             &policy, sizeof(policy));

// IOCTL to get multi-link policy
// Get current configured multi-link policy
enum sid_link_multi_link_policy get_policy;
sid_error_t ret = sid_option(handle,
                             SID_OPTION_GET_LINK_POLICY_MULTI_LINK_POLICY,
                             &get_policy, sizeof(get_policy));
```

### 2.9.2 Amazon Sidewalk stack statistics

The Amazon Sidewalk stack records the statistics from the time the user initializes and starts the Amazon Sidewalk stack to the time the Amazon Sidewalk stack is stopped and deinitialized. The Amazon Sidewalk stack does not store the statistics in non-volatile memory. The following are the statistics that are collected by the Amazon Sidewalk stack.

1. `Number of Messages sent`: The number of messages sent successfully by the Amazon Sidewalk stack when the developer's application calls `sid_put_msg` API to send a message. See section 3.8.
2. `Number of Acknowledgements sent`: The number of Acknowledgements sent by the Amazon Sidewalk stack when AWS IoT core for Amazon Sidewalk requests Acknowledgement to be sent for a downlink.
3. `Number of Messages failed to send`: The number of messages failed to be sent by the Amazon Sidewalk stack when the developer's application calls `sid_put_msg` API to send a message. See section 3.8.
4. `Number of Retries`: The number of retransmissions attempted by Amazon Sidewalk stack whent the developer's application calls `sid_put_msg` API to send a message with retries. See section 3.8.
5. `Number of Duplicates`: The number of duplicates received by Amazon Sidewalk stack from AWS IoT core for Amazon Sidewalk. Irrespective of the configuration setting of the filter duplicates, the duplicates statistic is incremented whenever the Amazon Sidewalk stack receives a duplicate.
6. `Number of Messages received`: The number of messages received by the Amazon Sidewalk stack from AWS IoT core for Amazon Sidewalk and notified to the developer using the `on_msg_received` callback. See section 2.10.
7. `Number of Acknowledgements received`: The number of acknowledgements received by the Amazon Sidewalk stack from AWS IoT core for Amazon Sidewalk and notified to the developer using the `on_msg_received` callback. See section 2.10.

```
/** The cumulative statistics of all links configured on the device */
struct sid_msg_statistics {
    /** number of messages sent successfully. */
    uint32_t tx;
    /** number of acknowledments sent to AWS-IOT core. */
    uint32_t acks_sent;
    /** number of messages failed transmissions. */
    uint32_t tx_fail;
    /** number of message retries. */
    uint32_t retries;
    /** number of duplicates received. */
    uint32_t duplicates;
    /** number of messages received. */
    uint32_t rx;
    /** number of acknowledments received from AWS-IOT core. */
    uint32_t acks_recv;
};

/** The metrics collected by the Sidewalk stack*/
struct sid_statistics {
    struct sid_msg_statistics msg_stats;
};

// Get statistics
struct sid_statistics stats = {0};
sid_error_t ret = sid_option(handle, SID_OPTION_GET_STATISTICS,
                             &stats, sizeof(stats));
// Clear statistics
ret = sid_option(handle, SID_OPTION_CLEAR_STATISTICS, NULL, 0);

/**
 * The set of options to be used with sid_option API.
 */
enum sid_option {
    /** Option to configure the advertised battery
    * level. Value is a uint8_t, 0-100 */
    SID_OPTION_BLE_BATTERY_LEVEL = 0,
    /** Option to configure the device profile.
     * Value is of type struct sid_device_profile */
    SID_OPTION_900MHZ_SET_DEVICE_PROFILE = 1,
    /** Option to get the device profile configuration.
     * Value is of type struct sid_device_profile */
    SID_OPTION_900MHZ_GET_DEVICE_PROFILE = 2,
    /** Option to set the message policy to filter duplicates,
      * value is 0 or 1, when set to 0, the default  setting,
      * duplicates are filtered and are not propagated using
      * #on_msg_received, when set to 1,  duplicates are detected
      * and are propagated using #on_msg_receive,
      * see #sid_msg_desc_rx_attributes
      */
    SID_OPTION_SET_MSG_POLICY_FILTER_DUPLICATES = 3,
    /** Option to get the configured policy of filtering duplicates,
     * 0 - Filter duplicates, 1 - Allow duplicates */
    SID_OPTION_GET_MSG_POLICY_FILTER_DUPLICATES = 4,
    /** Option to set the link connection policy setting*/
    SID_OPTION_SET_LINK_CONNECTION_POLICY = 5,
    /** Option to get the current link connection policy setting*/
    SID_OPTION_GET_LINK_CONNECTION_POLICY = 6,
    /** Option to allow the stack to attempt connection of the link
     * when the message is sent using sid_put_msg. This option is valid
     * only when the message is sent not using #SID_MSG_LINK_TYPE_ANY
     * #SID_MSG_LINK_TYPE_ANY enables multi-link policy and connection policy
     * is determined by the configured multi-link policy. see
     * #sid_link_auto_connect_params
     */
    SID_OPTION_SET_LINK_POLICY_AUTO_CONNECT_PARAMS = 7,
    /** Option to get the configured auto connect params for a
     * particular link type. see #sid_link_auto_connect_params.
       The link_type needs to filled when getting the auto connect params */
    SID_OPTION_GET_LINK_POLICY_AUTO_CONNECT_PARAMS = 8,
    /** Option to configure mlm link policy settings for the device */
    SID_OPTION_SET_LINK_POLICY_MULTI_LINK_POLICY = 9,
    /** Option to configure mlm link policy settings for the device */
    SID_OPTION_GET_LINK_POLICY_MULTI_LINK_POLICY = 10,
    /** Get msg statistics for the device */
    SID_OPTION_GET_STATISTICS = 11,
    /** Clear msg statistics for the device */
    SID_OPTION_CLEAR_STATISTICS = 12,
    /** OPTFDF*/
    SID_OPTION_GET_LINK_2_GW_DISCOVERY_POLICY_PARAMS = 13,
    SID_OPTION_SET_LINK_2_GW_DISCOVERY_POLICY_PARAMS = 14,
    /** Delimiter to enum sid_option*/
    SID_OPTION_LAST,
};
```

## 2.10 Stack Notifications

The Sid API notifies the developer's application through a set of callbacks. The developer is required to register these calllbacks with the Sid API when initializing the Amazon Sidewalk stack. The following events are notified by the Amazon Sidewalk stack:

1. `on event`: the stack does not create tasks for processing its internal events. The stack processes its internal events in the context supplied by the developer's application. In order to process its pending internal events, the stack notifies the developer's application through the `on event` callback. On getting the `on event` callback, the developer's application is required to call the `sid_process` API to cause the stack to process the pending events. The `on event` callback can be called in ISR context, and the arguments in the callback indicate whether the `on event` callback is being called in an ISR context or not.
2. `on msg received`: The stack uses this callback to notify the developer's application that a message has arrived. The `on msg received` callback is always called in the context in which `sid_process` is called.
3. `on msg sent`: When the stack sends the message over the desired link, the stack notifies the developer's application that it has sent the message successfully using the on `msg sent callback`. Note that the `on msg sent` callback does not indicate the message has been received by the AWS IoT application. The `on msg sent` callback only indicates that the message has been sent over the link successfully and no over the air transmission errors have occurred. The notification trigger for the `on msg sent` callback to the developer's application by the stack is different for each link type that is used to send the message:
    (a) **BLE**: on a BLE link, when the BLE Endpoint receives an acknowledgement from a BLE Gateway, the `on msg sent` callback is called.
    (b) **FSK**: on an FSK link, when the FSK Endpoint receives a link layer acknowledgement from an FSK Gateway, the `on msg sent` callback is called.
    (c) **LoRa**: on a LoRa link, there is no link layer acknowledgement, and the `on msg sent callback` is called as soon as the message is transferred over the air by the low level LoRa radio driver.
4. `on send error`: The `on send error` message is sent if the stack cannot send the message for any of the following reasons.
    (a) `Link is not connected`: If the link on which the message is being sent is disconnected, the `on send error` callback is called.
    (b) `AWS IoT Core for Amazon Sidewalk has not sent Acknowledgement`: If the message requires a `Transport Ack` from AWS IoT Core for Amazon Sidewalk and the `Transport Ack` was not received in the configured `Time to live` period, the `on send error` callback is called.

Note that `on msg sent` and `on send error` are not mutually exclusive. There could be instances where the message could be sent over the air to the Gateway, but the message might not make it to the destination of AWS IoT Core for Amazon Sidewalk. In this scenario, `on send error` and `on msg sent` are called with the same `message ID`. This indicates that even though the message was able to be sent over the air to the

Gateway, the message did not reach AWS IoT Core for Amazon Sidewalk within the `Time to live` period configured for that message.

1. `on status changed`: The `on status changed` callback is called when the Amazon Sidewalk stack state changes. The Amazon Sidewalk state is comprised of the following as explained in the previous sections.
   (a) Connection status of the link
   (b) Time sync status
   (c) Registration status

When any of these items change status, the `on status changed` callback is called.

1. `on factory reset` This callback is called when the AWS IoT application has de-registered the device. The Amazon Sidewalk stack removes all the non-volatile data stored during registration, and then uses the `on factory reset` callback to notify the developer's application that the Endpoint is de-registered from the Amazon Sidewalk.

```
/**
 * The set of callbacks a developer's application can register through
 * sid_init().
 */
struct sid_event_callbacks {
    /** Object used to store user data */
    void *context;
    /**
     * Callback to invoke when any Sidewalk event occurs.
     *
     * The Sidewalk stack invokes this callback when there is at least one
     * event to process, including internal events. Upon receiving this
     * callback the sid api user is required to schedule
     * a call to sid_process() within the application's
     * main loop or running context.
     *
     * @warning sid_process() MUST NOT be called from within the
     * #on_event callback to avoid re-entrancy and recursion problems.
     *
     * @see sid_process
     *
     * @param[in] in_isr  true if invoked from within an ISR context,
     * false otherwise.
     * @param[in] context The context pointer given in
     * sid_event_callbacks.context
     */
    void (*on_event)(bool in_isr, void *context);
    /**
     * Callback to invoke when a message from Sidewalk is received.
     *
     * @warning sid_put_msg() MUST NOT be called from within the
     * #on_msg_received callback to avoid re-entrancy and recursion problems.
     *
     * @param[in] msg_desc A pointer to the received message descriptor, which
     * is never NULL.
     * @param[in] msg      A pointer to the received message payload, which is
     * never NULL.
     * @param[in] context  The context pointer given in
     * sid_event_callbacks.context
     */
    void (*on_msg_received)(const struct sid_msg_desc *msg_desc,
                            const struct sid_msg *msg, void *context);
    /**
     * Callback to invoke when a message was successfully delivered to
     * Sidewalk.
     *
     * @param[in] msg_desc A pointer to the sent message descriptor,
     * which is never NULL.
     * @param[in] context  The context pointer given in
     * sid_event_callbacks.context
     */
    void (*on_msg_sent)(const struct sid_msg_desc *msg_desc, void *context);
    /**
     * Callback to invoke when a queued message failed to be delivered to
     * Sidewalk.
     *
     * A developer's application can use this notification to schedule
     * retrying sending a message, or invoke other error handling.
     *
     * @see sid_put_msg
     *
     * @warning sid_put_msg() MUST NOT be called from within the
     * #on_send_error callback to avoid re-entrancy and recursion problems.
     *
     * @param[in] error    The error code associated with the failure
     * @param[in] msg_desc A pointer to the unsent message descriptor,
     * which is never NULL.
     * @param[in] context  The context pointer given in
     * sid_event_callbacks.context
     */
    void (*on_send_error)(sid_error_t error,
                          const struct sid_msg_desc *msg_desc,
                          void *context);
    /**
     * Callback to invoke when the Sidewalk stack status changes.
     *
     * Once sid_start() is called, a #SID_STATE_READY status indicates
     * the stack is ready to accept messages sid_put_msg().
     *
     * When receiving #SID_STATE_ERROR, sid api user can
     * call sid_get_error() from within the #on_status_changed callback
     * context to obtain more detail about the error condition.
     * Receiving this status means the Sidewalk stack encountered a fatal
     * condition and won't be able to proceed. Hence, this notification is
     * mostly for diagnostic purposes.
     *
     * @param[in] status  The current status, valid until the next invocation
     * of this callback.
     * @param[in] context The context pointer given in
     * sid_event_callbacks.context
     */
    void (*on_status_changed)(const struct sid_status *status, void *context);
    /**
     * Callback to invoke when the Sidewalk stack
     * receives a Sidewalk factory reset from the cloud service.
     *
     * When the factory reset is triggered from cloud service
     * or #sid_set_factory_reset api call, the Sidewalk stack clears
     * its configuration from the non volatile storage
     * and resets its state accordingly.
     * This callback is then called by the Sidewalk stack
     * to notify the sid api user of Sidewalk network factory reset
     *
     * The device needs to successfully complete device registration with the
     * cloud services for the Sidewalk stack to send and receive messages
     *
     * @param[in] context The context pointer given in
     * sid_event_callbacks.context
     */
    void (*on_factory_reset)(void *context);
};
```

## 2.11 Stack Configuration

The Amazon Sidewalk stack requires configuration to be passed during initialization. Configuration includes the following items:

1. `link_mask`: The Amazon Sidewalk stack can be initialized to operate as one link or a combination of more than one link. The `link_mask` is a bit mask that allows the developer's application to specify a single link or a combination of more than one link. The following code shows the link combinations that are currently supported and the value the `link_mask` takes for these combinations. FSK and LoRa links are mutually exclusive, and the Amazon Sidewalk stack cannot support these links simultaneously. The developer's application has to de-initialize and re-initialize the Amazon Sidewalk stack to switch between FSK and LoRa links.

```
// BLE only
uint32_t link_mask = (1 << SID_LINK_TYPE_1) ; // link_mask = 1;
// FSK only
uint32_t link_mask = (1 << SID_LINK_TYPE_2) ; // link_mask = 2;
// LoRa only
uint32_t link_mask = (1 << SID_LINK_TYPE_3) ; // link_mask = 4;
// BLE and LoRa
uint32_t link_mask = (1 << SID_LINK_TYPE_1)
                   | (1 << SID_LINK_TYPE_3) ; // link_mask = 5;
// BLE and FSK
uint32_t link_mask = (1 << SID_LINK_TYPE_1)
                   | (1 << SID_LINK_TYPE_2) ; // link_mask = 3;
// FSK and LoRa
uint32_t link_mask = (1 << SID_LINK_TYPE_2)
                   | (1 << SID_LINK_TYPE_3) ; // link_mask = 6;
// BLE and FSK and LoRa
uint32_t link_mask = (1 << SID_LINK_TYPE_1)
                   | (1 << SID_LINK_TYPE_2)
                   | (1 << SID_LINK_TYPE_3) ; // link_mask = 7;
```

2. **callbacks**: a pointer to the callback structure documented above.
3. **BLE link config**: static configuration for BLE link. The Amazon Sidewalk stack expects a pointer to a statically allocated structure. The Amazon Sidewalk stack does not copy the structure contents to its own context. This requires the developer's application to statically allocate memory for this link config structure. The memory allocated to the BLE link config structure shall only be freed during de-initialization of the Amazon Sidewalk stack.
4. **Sub-GHz link config**: static configuration for the FSK link. The Amazon Sidewalk stack expects a pointer to a statically allocated structure. The Amazon Sidewalk stack does not copy the structure contents to its own context. This requires the developer's application to statically allocate memory for this link config structure. The memory allocated to the Sub-Ghz link config structure shall only be freed during de-initialization of the Amazon Sidewalk stack.

## 2.12 Sidewalk Handle

The Sidewalk handle is a pointer to an internal structure that is returned to the developer's application as part of the Amazon Sidewalk stack initialization. The developer's application is required to pass this handle when making Sid API calls. The Sidewalk handle is set to NULL when the Amazon Sidewalk stack is de-initialized.

## 2.13 Sidewalk Bulk Data Transfer

Sidewalk Bulk Data Transfer allows developers to transfer files to a device over BLE link only. These files could be firmware files for OTA upgrades. The developer's application interacts with the Amazon Sidewalk bulk data transfer functionality through the Amazon Sidewalk Bulk Data Transfer API (SBDT API). Broadly the following operations are supported:

1. Initialize and de-initialize SBDT API
2. Accept and reject incoming transfers
3. Cancel ongoing transfers
4. Verify the received file at the end of the transfer

### 2.13.1 Flow

Internally the file that is transferred is divided into fragments, the size of the fragments is configured by the developer when the transfer is setup. This fragment is further divided into chunks of size limited by the link on which the transfer is being done. The maximum size of the chunks for `SID_LINK_TYPE_1` (Bluetooth low power energy) is 216 bytes.

The device receives each of the chunks that makes up the fragment, the fragment is then verified if the verification fails the block is retransmitted, if not it sends an acknowledgement indicating that the fragment was successfully received. Then, the device receives the next fragment. The procedure continues until the whole file is transferred. Once the whole file is transferred, the application has the opportunity to validate the file. The file validation can be achieved by validating the CRC of the file, or by validating the signature.

### 2.13.2 File Id

Each transfer to the device is identified by a unique 4 byte id, called as the `file_id`.

### 2.13.3 Identifying an incoming transfer

An incoming transfer can be identified via `file_descriptor` field that developer provides when the transfer is created. The `file_descriptor` format is developer-defined, and is restricted to size not exceeding 40 bytes.

### 2.13.4 Memory Usage

The SBDT feature requires buffers to be allocated to handle the file transfer. The amount of memory needed is 2 times the chosen file transfer fragment size + context_memory (memory needed to hold the state). This memory is requested from the developer from the `on_transfer_request` callback. Some part of the memory is used for storing the transfer context, 1 fragment is used to store the received encrypted data and 1 fragment is used to hold the decrypted data. This memory is owned by SBDT until `on_release_scratch` is called.

### 2.13.5 Flash Storage

To allow for seamless functioning of the file transfer on reboots or planned device shutdown's, the SBDT stack stores information that it receives at file transfer request to kv_store at the various stages of the transfer.

### 2.13.6 Requesting a transfer

When a transfer is started, the application has the option to reject or accept a transfer. An application can choose to reject transfers based on multiple reasons. The currently available options are listed below.

```
enum sid_bulk_data_transfer_reject_reason {
    /** No reject reason */
    SID_BULK_DATA_TRANSFER_REJECT_REASON_NONE = 0x0,
    /** Rejected for generic reasons */
    SID_BULK_DATA_TRANSFER_REJECT_REASON_GENERIC = 0x1,
    /** Rejected because file is too big */
    SID_BULK_DATA_TRANSFER_REJECT_REASON_FILE_TOO_BIG = 0x3,
    /** Rejected because device has not space */
    SID_BULK_DATA_TRANSFER_REJECT_REASON_NO_SPACE = 0x4,
    /** Rejected bacause low battery to complete transfer */
    SID_BULK_DATA_TRANSFER_REJECT_REASON_LOW_BATTERY = 0x5,
    /** Rejected because file verification failed */
    SID_BULK_DATA_TRANSFER_REJECT_REASON_FILE_VERIFICATION_FAILED = 0x9,
    /** Rejected because file already exist */
    SID_BULK_DATA_TRANSFER_REJECT_REASON_FILE_ALREADY_EXISTS = 0xB,
    /** Rejected because fragment size is not supported */
    SID_BULK_DATA_TRANSFER_REJECT_REASON_INVALID_FRAGMENT_SIZE = 0xE,
};
```

The developer is provided with multiple information points such as `fragment_size`, `file_size`, `file_descriptor` (Identifying an incoming transfer) that can help them determining the reason for rejection.

### 2.13.7 Receiving data

Once each fragment is received, the developer's application is given fragment via `on_data_received` callback. The additional information that is provided includes `file_id`, `file_offset`, and the link that fragment the data is received on. Once the data is received, the developer's application can then write the data to flash and release the buffer so that the next buffer can be received.

### 2.13.8 SBDT Callbacks

The SBDT API notifies the developer's application through a set of callbacks. The developer is required to register these callbacks with SBDT API when initializing the SBDT feature. The following events are notified by SBDT:

1. `on transfer request`: SBDT uses this callback to notify the developer's application that it received a request to start a file transfer. The callback contains various information about the incoming transfer

such as the size of the file, fragment size used for the transfer, the link over which the transfer is going to happen, file descriptor if provided when the transfer was created. This API allows developer's application to reject the incoming transfer with a reject reason in Sidewalk Bulk Data Transfer Status Codes.

2. `on data received`: SBDT uses this callback to notify the developer's application that it received a fragment. The developer's application now owns the buffer that contains fragment data and can only be released via `sid_bulk_data_transfer_release_buffer` API call.

3. `on finalize request`: Once all the fragments are transferred, the SBDT triggers this callback to notify the developer that the file has been transferred. The developer's application can indicate to SBDT if the recieved file is as expected via the `sid_bulk_data_transfer_finalize` API call.

4. `on cancel request`: SBDT uses this callback to notify the developer's application that it received a cancel request from AWS cloud. This API is also called when the developer's application calls `sid_bulk_data_transfer_cancel` to cancel an ongoing transfer.

5. `on error`: SBDT uses this callback to notify the developer that it has encountered an unrecoverable error.

6. `on release scratch`: This callback is called by the SBDT to release memory it acquired from the developer's application in `on transfer request`. This callback is called after `on finalize request`, `on cancel request`, `on error` and `sid_bulk_data_transfer_deinit`.

```
struct sid_bulk_data_transfer_event_callbacks {
    /** User context data */
    void *context;

    /**
     * Callback that is invoked when a transfer request is recieved
     *
     * The Sidewalk stack invokes this callback to let the user know about
     * an Incoming transfer request. The user can then choose to accept or
     * reject said transfer request
     *
     * @param[in] transfer_request A pointer to the data transfer request,
     * which is never NULL
     * @param[out] transfer_response A pointer to the data transfer response,
     * which is never NULL
     * @param[in] context The context pointer given in
     * sid_bulk_data_transfer_event_callbacks.context
     *
     */
    void (*on_transfer_request)
         (const struct sid_bulk_data_transfer_request *const transfer_request,
          struct sid_bulk_data_transfer_response *const transfer_response,
          void *context);

    /**
       * Callback that is invoked when data has been received
       *
       * @param[in] desc A pointer to the bulk transfer data,
       * which is never NULL
       * @param[in] buffer A pointer to the buffer that contains the received
       * data, which is never NULL
       * @param[in] context The context pointer given in
       * sid_bulk_data_transfer_event_callbacks.context
```

```
        */
    void (*on_data_received)
         (const struct sid_bulk_data_transfer_desc *const desc,
          const struct sid_bulk_data_transfer_buffer *const buffer,
          void *context);

    /**
     * Callback that is invoked when data transfer is done
     *
     * @param[in] file_id Identifier of the file being transferred
     * @param[in] context The context pointer given in
     * sid_bulk_data_transfer_event_callbacks.context
     */
    void (*on_finalize_request)(uint32_t file_id, void *context);

    /**
        * Callback that is invoked when data transfer is cancelled from cloud
        *
        * @param[in] file_id Identifier of the file being transferred
        * @param[in] context The context pointer given in
        * sid_bulk_data_transfer_event_callbacks.context
        */
    void (*on_cancel_request)(uint32_t file_id, void *context);

    /**
     * Callback that is invoked when an error is encountered during data
     * transfer
     *
     * The Sidewalk stack will invoke this API to let the user know an error
     * was encountered during file transfer. Internally Sidewalk stack
     * cleans up it's state and goes to the same state as when
     * sid_bulk_data_transfer_start() is called.
     *
     * @param[in] file_id Identifier of the file being transferred
     * @param[in] context The context pointer given in
     * sid_bulk_data_transfer_event_callbacks.context
     */
    void (*on_error)(uint32_t file_id, void *context);

    /**
     * Callback that is invoked when Sidewalk stack wants to release scratch
     * buffer
     *
     * The Sidewalk stack will invoke this API to let the user cleanup the
     * scratch buffer that was allocated to during on_transfer_request call
     *
     * @param[in] file_id Identifier of the file being released
     * @param[in] context The context pointer given in
     * sid_bulk_data_transfer_event_callbacks.context
     */
    void (*on_release_scratch)(uint32_t file_id, void *context);
};
```

# Chapter 3

# Amazon Sidewalk APIs

Amazon Sidewalk provides APIs for the developer's application to configure, control and use the services of the Amazon Sidewalk stack on an Endpoint.

For the list of error codes and their values see chapter 5.

## 3.1 `sid_platform_init`

`sid_platform_init` initializes the platform specific components. These are platform abstraction Layer(PAL) routines that are required by the Sidewalk stack

The `sid_platform_init` is the first API that is required be invoked before invoking any other Sidewalk API. The `sid_platform_init` invocation can be avoided if the each PAL component's initialization routine is explicitly invoked in the developer's application prior to calling the `sid_init` API. The `sid_platform_init` API accepts a pointer to the platform specific data. Invoking `sid_platform_init` API multiple times without invocation of `sid_platform_deinit` would not re-initialize the PALs but would return `SID_ERROR_NONE`

`SID_ERROR_INCOMPATIBLE_PARAMS` is returned if the argument passed is invalid for a specific PAL initialization's implementation.

`SID_ERROR_NO_SUPPORT` is returned if a specific PAL is not supported by the platform.

`SID_ERROR_GENERIC` is returned if the error code is specific to a PAL implementation.

`SID_ERROR_NONE` is returned if the PALs are initialized successfully when `sid_platform_init` is invoked or PALs are already initialized before the invocation of `sid_platform_init`.

```
/**
 * Initializes the Sidewalk stack platform specific parameters.
 *
 * sid_platform_init() shall only be called once before other Sidewalk
 * stack functions e.g. sid_init().
 *
 * @param[in] platform_init_parameters  The required configuration in order
 * to properly initialize platform specific
 * parameters
 *
 * @returns #SID_ERROR_NONE in case of success.
 */
sid_error_t sid_platform_init(const void *platform_init_parameters);

platform_parameters_t platform_parameters = {
    .mfg_store_region.addr_start = 0x71000,
    .mfg_store_region.addr_end = 0x72000,
    .platform_init_parameters.radio_cfg =
        (radio_sx126x_device_config_t *)get_radio_cfg(),
};
sid_error_t ret_code = sid_platform_init(&platform_parameters);
```

## 3.2 sid_platform_deinit

`sid_platform_deinit` deinitializes the platform specific components.

The `sid_platform_deinit` shall not be called before calling `sid_deinit`. The `sid_platform_deinit` invocation can be avoided if each PAL component's deinitialization routine is explicitly invoked in the developer's application after calling the `sid_deinit` API. Invoking `sid_platform_deinit` API multiple times without invocation of `sid_platform_init` would not deinitialize the PALs multiple times but returns `SID_ERROR_NONE`

`SID_ERROR_GENERIC` is returned if the error code is specific to a PAL implementation.

`SID_ERROR_NONE` is returned if the PALs are deinitialized successfully when `sid_platform_deinit` is invoked or PALs are already deinitialized before the invocation of `sid_platform_deinit`.

```
/**
 * Deinitializes the Sidewalk stack platform specific parameters.
 *
 * sid_platform_deinit() shall only be called once after
 * using of Sidewalk stack, i.e. after sid_deinit().
 *
 * @returns #SID_ERROR_NONE in case of success.
 */
sid_error_t sid_platform_deinit(void);

sid_error_t ret_code = sid_platform_deinit();
```

## 3.3 sid_init

The Amazon Sidewalk stack can be initialized using the `sid_init` API. The `sid_init` API accepts a pointer to the Amazon Sidewalk configuration see section 2.1.1. The `sid_init` API returns a pointer to its internal handle. This handle needs to be passed to all other APIs. On initialization, the Amazon Sidewalk stack and the link specific initialization sequences are executed. On initialization, message buffers and other control structures are allocated however no radio activity occurs in this state. All features supported by the Amazon Sidewalk stack (such as FFS, Time Synchronization, Security, etc.) are available without requiring further feature-specific initialization.

`sid_init` should not be called in the context of any sid_event_callbacks to avoid re-entrancy and recursion issues. `sid_init` does not trigger any notifications to the callbacks that the developer has registered with the Amazon Sidewalk stack.

Re-initializing the Amazon Sidewalk stack without de-initializing causes the `sid_init` to fail with the API return error code `SID_ERROR_ALREADY_INITIALIZED`.

The `sid_init` API returns error code `SID_ERROR_INVALID_ARGS` if any of the following conditions are met:

1. Pointers to `config` and `handle` are `NULL`.
2. The FFS over FSK periodicity is configured with a value which is not in the valid range.

3. `link_mask` has `SID_LINK_TYPE_2` (FSK) or `SID_LINK_TYPE_3` (LoRa) support but the `sub_ghz_config` is `NULL`.

The `sid_init` API returns error code `SID_ERROR_NOSUPPORT` if any of the following conditions are met:

1. `link_mask` in the config structure has an unsupported combination of link types. The following list are unsupported combinations. Note for FSK and LoRa a device may have the capability to support both link types, but FSK and LoRa cannot be active at the same time.

   (a) `SID_LINK_TYPE_3`(LoRa) only, if the Endpoint is not registered yet with Amazon Sidewalk cloud

2. `link_mask` is configured to `SID_LINK_TYPE_2` (FSK) only, and FFS registration over FSK is disabled either through run time config or through a compilation flag. Please note that FFS registration over FSK can be disabled if `link_mask` includes BLE. For instance `link_mask` configured to `SID_LINK_TYPE_1` (BLE) and `SID_LINK_TYPE_2` (FSK) can have FFS registration over FSK disabled.

3. `link_mask` is configured to `SID_LINK_TYPE_2` (FSK) and `SID_LINK_TYPE_2` (LoRa), the Sidewalk stack switches between the two links for every one minute until either one of the link is able to connect to Amazon Sidewalk cloud

4. `link_mask` is configured to `SID_LINK_TYPE_3` (LoRa) only when the Endpoint is registered. In an unregistered state, the developer's application should configure `link_mask` with atleast one link that supports registration.

The `sid_init` API returns error code `SID_ERROR_NONE` when the Amazon Sidewalk stack initialization with the requested configuration is successful.

```
/**
 * Initializes the Sidewalk library for the chosen link type.
 *
 * sid_init() can only be called once for the given sid_config.link_type
 * unless sid_deinit() is called first.
 *
 * @see sid_deinit
 *
 * @param[in] config The required configuration in order to properly
 * initialize Sidewalk for the chosen link type.
 * @param[out] handle A pointer where the the opaque handle type will be
 * stored. ‘handle‘ is set to NULL on error.
 *
 * @returns #SID_ERROR_NONE on success.
 * @returns #SID_ERROR_ALREADY_INITIALIZED if Sidewalk was already initialized
 * for the given link type.
 */
sid_error_t sid_init(const struct sid_config *config,
                     struct sid_handle **handle);

// Initialize BLE and LoRa links
struct sid_config config = {
     .link_mask = (1 << SID_LINK_TYPE_1 | 1 << SID_LINK_TYPE_3),
    .callbacks = &event_callbacks,
    .link_config = app_get_ble_config(), // update the BLE config
    .sub_ghz_link_config = app_get_sub_ghz_config(), // update sub GHz config
};
static struct sid_handle *sidewalk_handle;
sid_error_t ret = sid_init(config, &sid_handle);
```

## 3.4 sid_deinit

The Amazon Sidewalk stack can be de-initialized using the sid_deinit API. The sid_deinit API requires the pointer to the internal handle that was provided to the developer's application during initialization. The sid_deinit API stops the links that are initialized during sid_init by calling sid_stop. The time is reset to zero which means that when the the Amazon Sidewalk stack is initialized with the sid_init again, the Amazon Sidewalk stack is required to acquire time synchronization again. There is no separate API required to de-initialize any features, calling the sid_deinit API causes all the routines that de-initialize features to be called.

sid_deinit does not trigger any notifications to the callbacks that the developer has registered with the Amazon Sidewalk stack.

sid_init can be called with a different or the same config after sid_deinit is called. sid_deinit should not be called in the context of any sid_deinit to avoid re-entrancy and recursion issues.

The pointer to the internal handle that the developer's application receives is set to NULL.

1. The sid_deinit API call returns SID_ERROR_INVALID_ARGS, if the handle passed is NULL.
2. The sid_deinit API call returns SID_ERROR_INVALID_STATE, if the Amazon Sidewalk stack is already de-initialized.
3. The sid_deinit API call return SID_ERROR_NONE, if the Amazon Sidewalk stack is successfully de-initialized.

```
/**
 * De-initialize the portions of the Sidewalk library associated with the
 * given handle.
 *
 * @see sid_init
 *
 * @param[in] handle A pointer to the handle returned by sid_init()
 *
 * @returns #SID_ERROR_NONE in case of success
 */
sid_error_t sid_deinit(struct sid_handle *handle);

sid_error_t ret = sid_deinit(handle);
```

## 3.5 sid_start

The Amazon Sidewalk stack can be started using the sid_start API. The Sidewalk handle provided to the developer's application during initialization is required to be passed to the sid_start API. The link_mask passed to sid_start can be the same as the link_mask passed to the sid_init API or a subset of the links passed to the sid_init API. The links that are not initialized cannot be started. For example, if SID_LINK_TYPE_1 and SID_LINK_TYPE_2 are initialized, the developer's application can start only one link or both the links as follows:

1. Setting the link_mask to SID_LINK_TYPE_1 or SID_LINK_TYPE_2 starts one link.
2. Setting the link_mask to SID_LINK_TYPE_1 and SID_LINK_TYPE_2 starts both the links.

A call to sid_start triggers the following behavior:

1. If the Amazon Sidewalk stack is not registered, the Amazon Sidewalk stack starts registration based on the links that have started. If both SID_LINK_TYPE_1 and SID_LINK_TYPE_2 have started, and if FFS over FSK is enabled, the registration procedure starts in SID_LINK_TYPE_2 (FSK), and the registration procedure also starts on SID_LINK_TYPE_1 (BLE). The registration procedure is completed based on

whichever Gateway is in range: if an Amazon Sidewalk BLE Gateway is in range, the registration procedure is started and completed on BLE link and if an Amazon Sidewalk FSK Gateway is in range, registration procedure is started and completed on the FSK link. The developer's application can force the Amazon Sidewalk stack to perform registration on the desired link by starting only that link. The developer's application can initialize BLE and FSK links, but may then start just the BLE link if the developer requires registration to complete on the BLE link or the developer's application may start just the FSK link if the developer requires registration to complete on the FSK link. Note that registration is not supported on the LoRa link.

2. If the Amazon Sidewalk stack does not have time synhcronization, the Amazon Sidewalk stack acquires time based on the links that are started. If more than one link has started, the time acquisition procedure is started simultaneously on all links that have started.
   (a) On the BLE link, the Amazon Sidewalk stack updates the beacon's payload requesting the GPS time.
   (b) On the FSK link, the Amazon Sidewalk stack synchronizes to an FSK Gateway in range and sends a time request message to Amazon Sidewalk cloud services.
   (c) on the LoRa link, the Amazon Sidewalk stack immediately sends the time request message to Amazon Sidewalk cloud services.
3. After `sid_start` is called the link behaves as follows
   (a) on BLE, the Endpoint is always beaconing. If the device has an uplink message to send, the beacon's payload is updated with the intent to connect with a BLE Gateway to send the uplink message. If there is no uplink message to send, the Endpoint advertises its presence to the Amazon Sidewalk BLE Gateways to receive downlink messages from the Amazon Sidewalk cloud services.
   (b) on FSK, the Endpoint synchronizes with the Amazon Sidewalk FSK Gateway and, based on the device profile configured, performs a join procedure through the Amazon Sidewalk FSK Gateway with the Amazon Sidewalk cloud services. For more details on the join procedure and device profiles see the device profile document.
   (c) on LoRa, the device performs the join procedure with the Amazon Sidewalk cloud services. For more details on join procedure and device profile, see the device profile document.
4. `sid_start` should not be called in the context of any sid_event_callbacks to avoid re-entrancy and recursion issues.
5. `sid_start` uses the `on_status_changed` callback to notify the developer of the Amazon Sidewalk stack status. The `on_status_changed` callback notifies the developer's application of the Sidewalk state on the Endpoint, whether it is registered, whether it has acquired GPS time and the connection status of the links that are being started.
6. If a link that has already started, is started again, `SID_ERROR_NONE` is returned.
7. The Amazon Sidewalk stack returns `SID_ERROR_INVALID_ARGS` if any of the following is true:
   (a) `sid_start` is called with a handle that is `NULL`.
   (b) `sid_start` is called before `sid_init` has been called at least once
   (c) `link_mask` contains links that are not initialized but are being requested to start.
8. The Amazon Sidewalk stack returns `SID_ERROR_NONE`, if the requested links can be started successfully.

```
/**
 * Makes the Sidewalk library start operating.
 *
 * The notifications registered during sid_init() are invoked once sid_start()
 * is called.
 *
 * On an unregistered Sidewalk Endpoint sid_start() will put the Sidewalk
 * library into a state ready for device registration, typically the device is
 * registered with a separate mobile application or MCU SDK tool.
 *
 * After registration the Sidewalk Endpoint will receive a time sync message
 * from an accessible Sidewalk gateway. This is a prerequisite to establish
 * up-link or down-link data connectivity.This is visible to the developer by
 * #SID_STATE_READY and #SID_STATUS_TIME_SYNCED.
 *
 * If sid_stop() is called after receiving #SID_STATE_READY the Sidewalk
 * library will cache the time sync for a minimum of SID_TIME_SYNC_MIN_PERIOD
 * seconds. This value should not be modified by the developer using the API.
 *
 * sid_start() can be used to start one or more links at once.
 * To start a single link, eg #SID_LINK_TYPE_1, link_mask = SID_LINK_TYPE_1,
 * To start more than one link, eg #SID_LINK_TYPE_1 and #SID_LINK_TYPE_3 ,
 * link_mask = SID_LINK_TYPE_1 | SID_LINK_TYPE_3
 *
 * @note Developer can only start a link_type that was initialized in
 * sid_init()
 *
 * @see sid_stop
 *
 * @param[in] handle A pointer to the handle returned by sid_init()
 * @param[in] link_mask The links that need to started.
 *
 * @returns #SID_ERROR_NONE in case of success.
 */
sid_error_t sid_start(struct sid_handle *handle, uint32_t link_mask);

// To start BLE and FSK links and initializing both BLE and FSK links
uint32_t link_mask = (SID_LINK_TYPE_1 | SID_LINK_TYPE_2);

// To start BLE link only after initializing BLE and FSK links
uint32_t link_mask = SID_LINK_TYPE_1;

// To start FSK link only after initializing BLE and FSK links
uint32_t link_mask = SID_LINK_TYPE_2;

sid_error_t ret = sid_start(handle, link_mask);
```

## 3.6 sid_stop

The Amazon Sidewalk stack can be stopped by calling the `sid_stop` API. The Sidewalk handle provided to the developer's application during initialization is required to be passed to the `sid_stop` API.

The `link_mask` passed to `sid_stop` can be the same as the `link_mask` passed to the `sid_init` API or a subset of the links passed to the `sid_init` API. The links that are not initialized cannot be stopped. For example, if `SID_LINK_TYPE_1` and `SID_LINK_TYPE_2` are initialized, the developer's application can stop one link or both links as follows:

1. To stop one link, set the `link_mask` to `SID_LINK_TYPE_1` or `SID_LINK_TYPE_2`.
2. To stop both links, set the `link_mask` to `SID_LINK_TYPE_1` and `SID_LINK_TYPE_2`.

A call to `sid_stop` triggers the following behavior:

1. The links that are requested to be stopped are handled as follows:
   (a) on the BLE link
      i. If the Endpoint was in connected state, calling `sid_stop` causes the Amazon Sidewalk stack to disconnect
      ii. If the Endpoint was in advertisement state, calling `sid_stop` causes the Endpoint to stop beaconing.

   Note that the BLE radio is not set to sleep on called `sid_stop`.

   (a) on the FSK link, calling `sid_stop` causes the Amazon Sidewalk stack to disconnect from the Amazon Sidewalk FSK Gateway and sets the radio to sleep.
   (b) on the LoRa link, calling `sid_stop` causes the Amazon Sidewalk stack to set the radio to sleep.
2. `sid_stop` notifies the Amazon Sidewalk stack status to the developer through the `on_status_changed` callback. The `on_status_changed` callback notifies the developer's application of the Sidewalk state on the Endpoint, whether it is registered, whether it has time synchronization and the connection status of the links that are started.
3. `sid_stop` should not be called in the context of any sid_event_callbacks to avoid re-entrancy and recursion issues.
4. If a link that is already stopped is stopped again, `SID_ERROR_NONE` is returned.
5. The Amazon Sidewalk stack returns `SID_ERROR_INVALID_ARGS` if any of the following is true:
   (a) `sid_stop` is called with a handle that is `NULL`.
   (b) `sid_stop` is called before `sid_init` has been called at least one time.
   (c) `link_mask` is set to request links to stop that are not initialized.
6. The Amazon Sidewalk stack returns `SID_ERROR_NONE`, if the requested links can be stopped successfully.

```
/**
 * Makes the Sidewalk library stop operating.
 *
 * No messages will be sent or received and no notifications will occur after
 * sid_stop() is called.
 * Link status will be changed to disconnected and time sync status is cached
 * after sid_stop() is called.
 *
 * sid_stop() can be used to stop one or more links at once.
 * To stop a single link, eg #SID_LINK_TYPE_1, link_mask = SID_LINK_TYPE_1,
 * To stop more than one link, eg #SID_LINK_TYPE_1 and #SID_LINK_TYPE_3 ,
 * link_mask = SID_LINK_TYPE_1 | SID_LINK_TYPE_3
 *
 * @note Developer can only stop a link_type that was initialized in
 * sid_init()
 *
 * @see sid_start
 *
 * @warning sid_stop() should be called in the same caller context as
 * sid_process().
 *
 * @warning sid_stop() must not be called from within the caller context of
 * any of the sid_event_callbacks registered during sid_init() to avoid
 * re-entrancy and recursion problems.
 *
 * @warning RTC should not be stopped after sid_stop() is called.
 *
 * @param[in] handle A pointer to the handle returned by sid_init()
 * @param[in] link_mask The links that need to be stopped.
 *
 * @returns #SID_ERROR_NONE in case of success.
 */
sid_error_t sid_stop(struct sid_handle *handle, uint32_t link_mask);

/** To stop BLE and FSK links and initializing and starting both BLE
 *  and FSK links  */
uint32_t link_mask = (SID_LINK_TYPE_1 | SID_LINK_TYPE_2);

// To stop BLE link only after initializing BLE and FSK links
uint32_t link_mask = SID_LINK_TYPE_1;

// To start FSK link only after initializing BLE and FSK links
uint32_t link_mask = SID_LINK_TYPE_2;

sid_error_t ret = sid_stop(handle, link_mask);
```

## 3.7 sid_process

When the Amazon Sidewalk stack calls the developer's registered callback of on_event, the developer's application is required to call sid_process. The event sources for the Amazon Sidewalk stack are timers and the radio. The timer events are generated when the Amazon Sidewalk stack schedules events to occur in the future to maintain its states. The radio events are generated by the radio when a particular low level radio process is complete, for example transmitting a message over the air, receiving a message, radio transmit and receive errors etc. The timer and radio events are generated in the Interrupt Service Routine (ISR) context. sid_process should not be called in the same context in which on_event is notified to the developer's application. This is because on_event notifies internal events that run and maintain Sidewalk states to the developer's application. This notification process is a slow operation that must not be performed in ISR context. Note that these internal events are not meaningful to the developer.

SID_ERROR_STOPPED is returned if all the links are stopped.

SID_ERROR_INVALID_ARGS is returned if the Amazon Sidewalk stack is not initialized.

SID_ERROR_NONE is returned if the Amazon Sidewalk stack can successfully process its internal events.

Failure to call sid_process after being notified of an event by on_event causes events of the Amazon Sidewalk stack to remain in an unprocessed state. sid_process should be called before calling sid_stop or sid_deinit.

```
/**
 * Process Sidewalk events.
 *
 * When there are no events to process, the function returns immediately.
 * When events are present, sid_process() invokes the sid_event_callbacks
 * registered during sid_init() within its calling context. You may not
 * receive any callbacks for internal events.
 *
 * You are required to schedule sid_process() to run within your main-loop or
 * running context when the sid_event_callbacks.on_event callback is received.
 *
 * Although not recommended for efficiency and power usage reasons,
 * sid_process() can also be called even if sid_event_callbacks.on_event has
 * not been received, to support main loops that operate in a polling manner.
 *
 * @warning sid_process() must not be called from within the caller context of
 * any of the sid_event_callbacks registered during sid_init() to avoid
 * re-entrancy and recursion problems.
 *
 * @see sid_init
 * @see sid_start
 * @see sid_event_callbacks
 *
 * @param[in] handle A pointer to the handle returned by sid_init()
 *
 * @returns #SID_ERROR_NONE in case of success.
 * @returns #SID_ERROR_STOPPED if sid_start() has not been called.
 */
sid_error_t sid_process(struct sid_handle *handle);

sid_error_t ret = sid_process(handle);
```

## 3.8 sid_put_msg

Messages can be sent over Amazon Sidewalk from an Endpoint by calling the `sid_put_msg` API. For more details on message's metadata, attributes to configure message's metadata, and MTU, see section 2.8.

The Amazon Sidewalk stack generates a unique ID for a every message that it accepts to send. The `message ID` starts from the value zero at the time of reset and incremented by one on every message sent including Amazon Sidewalk internal messages. The maximum value that this unique ID is incremented to is 16383, after which it resets to zero. The `message ID` is used as follows:

1. For the LoRa link, the Amazon Sidewalk stack uses the `message ID` to notify the developer's application of the sent status of the message over the air.
2. For BLE and FSK links, the `message ID` is used for Gateway acknowledgements.
3. if the call to `sid_put_msg` configures the message to request an acknowledgement from AWS IoT Core for Amazon Sidewalk. AWS IoT Core for Amazon Sidewalk uses the `message ID` to notify the message's acknowledgement status. The notification is as follows:
   (a) If the message is sent successfully over the air. This is notified to the developer's application using the `on_msg_sent` callback. The `on_msg_sent` callback has a pointer to the message descriptor that has the same `message ID` that was provided to it at the call made to `sid_put_msg`. The `on_msg_sent` callback does not guarantee that the message has been received successfully by AWS IoT Core for Amazon Sidewalk. For messages sent by BLE and FSK, the `on_msg_sent` callback only notifies the developer's application that the message has been sent over the air, and that the mesage was acknowledged by a Gateway. For messages sent by LoRa, the `on_msg_sent` only indicates that the message has been sent over the air successfully.
   (b) If the message can not be sent successfully over the air, the developer's application is notified using the `on_send_error` callback. The `on_send_error` callback has a pointer to the message descriptor that has the same `message ID` that was provided to it at the call made to `sid_put_msg`. Note that `on_msg_sent` and `on_send_error` notifications are mutually exclusive. For a given `message ID`, both callbacks will not be notified, only one of them is notified to the developer. `on_msg_sent` is called for successful transmission and `on_send_error` is called for failure to transmit.

(c) If the message has been acknowledged by AWS IoT Core for Amazon Sidewalk, if it was configured to receive an acknowledgement in the message's transmit attributes. A message which has the acknowledgement field set to true in its transmit attributes gets the first notification that the message has been successfully sent using the `on_msg_sent` callback and also another notification that the message has been successfully acknowledged by AWS IoT Core for Amazon Sidewalk through `on_msg_received` callback.

The message attributes passed define the behavior the Amazon Sidewalk stack applies to the message. The messages' transmit attributes play a role only in this API. The developer's application uses this API to inform the Amazon Sidewalk stack of the following attributes:

1. The message is fire and forget, this message has acknowledgement set to false.
2. The Amazon Sidewalk stack is required to retry the message. The number of retries field governs this setting.
3. The period that the Amazon Sidewalk stack is required to hold the message before reporting error code `SID_ERROR_TIMEOUT` using the `on_msg_sent_error` callback.
4. The retry periodicity is linear and is derived from time to live and number of retries. For example: for a message configured with time to live as 60 seconds and number of retries as 3, the Amazon Sidewalk stack attempts to retry the message once every 20 seconds until an acknowledgement is received from the AWS IoT Core for Amazon Sidewalk. Once the acknowledgement is received, the developer's application is notified using `on_msg_received` that an acknowledgement has been received for the message. Acknowledgements can be differentiated from regular messages notified through the `on_msg_received` callback by using the `is_msg_ack` field in the receive attributes in the message descriptor.

When a call to `sid_put_msg` is made, the receive message attributes of the message descriptor are disregarded by the Amazon Sidewalk stack. This is because `sid_put_msg` is used to send messages and receive attributes in the message descriptor are only valid for the downlink messages or for Acknowledgements sent by AWS IoT and notified using the `on_msg_received` callback.

For fire and forget messages, if the time to live field in the transmit attributes is set to zero, the Amazon Sidewalk stack holds the message in its transmit queue for a maximum period of two minutes. In this period, the Amazon Sidewalk stack tries to send the message over the air. If the message cannot be sent over the air within two minutes, the developer's application is notified with the error code `SID_ERROR_TIMEOUT`.

A message can be configured to be sent on a specific link or a combination of the links. This setting is in the `link_mask`field of the message descriptor. The rules are as follows:

1. If only one link is specified in the `link_mask`:
   (a) If the link is not connected, the Amazon Sidewalk stack will fail to send messages with the error `SID_ERROR_INVALID_STATE`.
   (b) If the link is connected the Amazon Sidewalk stack will accept the message.
2. If more than one link is specified in the `link_mask` the following rules apply:
   (a) If only one link is in the connected state, the Amazon Sidewalk stack will send the message over the link which is in connected state.
   (b) If all the links in the `link_mask` are in disconnected state, `sid_put_msg` fails with the error code `SID_ERROR_INVALID_STATE`.
   (c) If more than one link specified in the `link_mask` is in the connected state, then the Amazon Sidewalk stack follows a pre-defined order: BLE has the highest priority, followed by FSK and then LoRa has the lowest priority.

`sid_put_msg` returns `SID_ERROR_OOM` when the Amazon Sidewalk stack does not have enough buffers to send the message. Whenever the `sid_put_msg` is called, the stack allocates a buffer from its pre-allocated memory pool and queues the message for transmission. If the developer's application requests transmissions of messages at a rate higher than the Amazon Sidewalk stack can transmit messages, the Amazon Sidewalk stack runs out of buffers and messages start to fail with the SID_ERROR_OOM error code. It is recommended that the developer's application should control the rate at which the messages are requested to be transmitted.

When sid_put_msg starts to fail messages with SID_ERROR_OOM, callbacks indicate whether the message was transmitted as follows:

1. For messages that do not require retries the on_msg_sent or on_msg_sent_error callbacks indicate whether the message was transmitted.
2. For messages that are configured to expect an acknowledgement from AWS IoT Core for Amazon Sidewalk the on_msg_received callback gives an additional indication of whether the message was received by AWS IoT Core for Amazon Sidewalk.

sid_put_msg returns SID_ERROR_INVALID_ARGS when one or more of the following conditions occurs:

1. The pointer to the msg structure or the pointer to the Sidewalk handle, or the pointer to the message descriptor is NULL.
2. The message size is zero or the message size is greater than 255 bytes.
3. An acknowledgement was requested from AWS IoT Core for Amazon Sidewalk, but either the number of retries is zero or the time to live is zero.
4. The message type is RESP, and the message id is greater than 16383.

sid_put_msg returns SID_ERROR_INVALID_STATE if any of the following conditions occur:

1. The Sidewalk state is not ready. The Amazon Sidewalk stack can send and receive messages only when it's state is ready. State ready requires, the Endpoint to be registered, acquired GPS time and at least one link is connected.
2. The link_mask specifies a link that is in disconnected state. For example, when the Amazon Sidewalk stack's BLE and FSK links are started. and only the FSK link is in connected state, and the BLE link is in disconnected state, if the developer's application sets the link_mask field in the message descriptor to BLE only, sid_put_msg fails with the SID_ERROR_INVALID_STATE error code.

sid_put_msg returns SID_ERROR_NOSUPPORT if a link mode is not supported on the link the message is being requested to be sent on.

sid_put_msg returns SID_ERROR_NONE when the Amazon Sidewalk stack has successfully accepted the message for transmission. When sid_put_msg returns SID_ERROR_NONE this does not imply that the Endpoint's Amazon Sidewalk stack has already successfully transferred the message to AWS IoT Core for Amazon Sidewalk, instead it implies that the message has been successfully queued in the Endpoint's internal transmit queue. The sent status of the message is notified to the developer through on_msg_sent and on_msg_sent_error for fire and forget messages and for messages requiring acknowledgement from AWS IoT Core for Amazon Sidewalk.

```
/**
 * Queues a message.
 *
 * @note msg_desc can be used to correlate this message with the
 * sid_event_callbacks.on_msg_sent and sid_event_callbacks.on_send_error
 * callbacks.
 *
 * @note When sending #SID_MSG_TYPE_RESPONSE in response to #SID_MSG_TYPE_GET,
 * the developer is expected to fill the id field of message descriptor with
 * the id from the corresponding #SID_MSG_TYPE_GET message descriptor.
 * This allows the sid_api to match each unique #SID_MSG_TYPE_RESPONSE with
 * #SID_MSG_TYPE_GET.
 *
 * @param[in]  handle   A pointer to the handle returned by sid_init()
 * @param[in]  msg      The message data to send
 * @param[out] msg_desc The message descriptor this function fills which
 *                      identifies this message.
 *                      Only valid when #SID_ERROR_NONE is returned.
 *
 * @returns #SID_ERROR_NONE when the message is successfully placed in the
 * transmit queue.
 * @returns #SID_ERROR_TRY_AGAIN when there is no space in the transmit queue.
 */
sid_error_t sid_put_msg(struct sid_handle *handle,
                        const struct sid_msg *msg,
                        struct sid_msg_desc *msg_desc);

/* send a fire and forget message of length 10 bytes and type notify to
 * AWS IoT, over BLE link with time to live as 60 seconds */
#define MAX_MSG_PAYLOAD_SIZE 255
static uint8_t payload[MAX_MSG_PAYLOAD_SIZE];
struct sid_msg_desc msg_desc = {
   .link_type = SID_LINK_TYPE_1,
   .type = SID_MSG_TYPE_NOTIFY,
   .mode = SID_LINK_MODE_CLOUD,
   .msg_desc_attr = {
       .tx_attr = {
           .request_ack = false,
           .num_retires = 0,
           .ttl_in_seconds = 60,
       },
   },
};

struct sid_msg msg = {
    .data = payload,
    .size = 10,
};

sid_error_t ret;
ret = sid_put_msg(handle, &msg, &msg_desc);

/* send message of length 100 bytes and type set to AWS IoT, over BLE or FSK
 * link with request_ack set to true, number of retries to 3 and time to live
 * as 120 seconds  */
#define MAX_MSG_PAYLOAD_SIZE 255
static uint8_t payload[MAX_MSG_PAYLOAD_SIZE];
struct sid_msg_desc msg_desc = {
   .link_type = (SID_LINK_TYPE_1 | SID_LINK_TYPE_2),
   .type = SID_MSG_TYPE_SET,
   .mode = SID_LINK_MODE_CLOUD,
   .msg_desc_attr = {
       .tx_attr = {
           .request_ack = true,
           .num_retires = 3,
           .ttl_in_seconds = 120,
        },
    },
};

struct sid_msg msg = {
    .data = payload,
    .size = 100,
};

sid_error_t ret;
ret = sid_put_msg(handle, &msg, &msg_desc);
```

## 3.9 sid_get_error

sid_get_error returns the error code, and is used after the on_status_changed callback has returned with SID_STATE_ERROR.

The sid_get_error API call returns the detail error code. The sid_get_error API call is only valid in the context in which the on_status_changed callback was called

SID_ERROR_INVALID_ARGS is returned if the Amazon Sidewalk stack is not initialized.

```
/**
 * Get the current error code.
 *
 * When the sid_event_callbacks on_status_changed callback is called
 * with a #SID_STATE_ERROR you can use this function to retrieve the
 * detailed error code. The error code will only be valid in the
 * calling context of the sid_event_callbacks.on_status_changed callback.
 *
 * @param[in] handle A pointer to the handle returned by sid_init()
 *
 * @returns The current error code
 */
sid_error_t sid_get_error(struct sid_handle *handle);

sid_error_t ret = sid_get_error(handle);
```

## 3.10 sid_get_mtu

The sid_get_mtu API call gets the MTU of the requested link.

SID_ERROR_INVALID_ARGS is returned if the Amazon Sidewalk stack is not initialized. Note that a specific link type has to be requested to get the MTU and not a combination of link types

SID_ERROR_NOSUPPORT is returned if the link type requested is not initialized. For example: if the BLE link is the only link that is initialized but the MTU of the FSK link is requested, SID_ERROR_NOSUPPORT is returned

SID_ERROR_NONE is returned if the link MTU can be successfully retrieved.

```
/**
 * Gets the MTU associated with the given link_type.
 *
 * @param[in]  handle    A pointer to the handle returned by sid_init()
 * @param[in]  link_type The link type to query
 * @param[out] mtu       A pointer to store the MTU size for the given
 *                       link_type
 *
 * @returns #SID_ERROR_NONE on success.
 */
sid_error_t sid_get_mtu(struct sid_handle *handle,
                        enum SID_LINK_TYPE link_type,
                        size_t *mtu);

size_t mtu = 0;
sid_error_t ret = sid_get_mtu(handle, SID_LINK_TYPE_1, &mtu);
```

## 3.11 sid_option

The `sid_option` API call is used to configure settings of the Amazon Sidewalk stack. For details of configuration see section 2.9

```
/**
 * Set an option
 *
 * @see sid_option
 * @param[in] handle      A pointer to the handle returned by sid_init
 * @param[in] option      The option to set
 * @param[in/out] data    A pointer to the memory for input/output data
 *                        associated with the option
 * @param[in] len         The size of the data array
 *
 * @returns #SID_ERROR_NONE on success.
 */
sid_error_t sid_option(struct sid_handle *handle,
                       enum sid_option option,
                       void *data, size_t len);

// Set battery level to 50%
uint8_t battery_level_in_percentage = 50;
sid_error_t ret = sid_option(handle,
                             SID_OPTION_BLE_BATTERY_LEVEL,
                             &battery_level_in_percentage,
                             sizeof(battery_level_in_percentage));

// Set FSK device profile to Sync profile with the following settings
struct sid_device_profile target_dev_cfg = {
   .unicast_params = {
      .device_profile_id = SID_LINK2_PROFILE_2,
      .rx_window_count = SID_RX_WINDOW_CNT_INFINITE,
      .unicast_window_interval = {
         .sync_rx_interval_ms = SID_LINK2_RX_WINDOW_SEPARATION_1,
      },
      .wakeup_type = SID_TX_AND_RX_WAKEUP,
   },
};

// Get current configured for device profile for FSK
sid_error_t ret = sid_option(sid_handle,
                             SID_OPTION_900MHZ_SET_DEVICE_PROFILE,
                             &target_dev_cfg,
                             sizeof(target_dev_cfg));

struct sid_device_profile curr_dev_cfg = {
   .unicast_params = {
      .device_profile_id = SID_LINK2_PROFILE_2,
   },
};
sid_error_t ret = sid_option(sid_handle,
                             SID_OPTION_900MHZ_GET_DEVICE_PROFILE,
                             &curr_dev_cfg,
                             sizeof(curr_dev_cfg));
```

```
// Set filter duplicates settings to allow all duplicates
bool filter_duplicates = true;
sid_error_t ret = sid_option(sid_handle,
                             SID_OPTION_SET_MSG_POLICY_FILTER_DUPLICATES,
                             &filter_duplicates,
                             sizeof(filter_duplicates));


// Get current configured duplicate setting
bool filter_duplicates;
sid_error_t ret = sid_option(sid_handle,
                             SID_OPTION_GET_MSG_POLICY_FILTER_DUPLICATES,
                             &filter_duplicates,
                             sizeof(filter_duplicates));
```

## 3.12 sid_set_factory_reset

The `sid_set_factory_reset` API shall be called when the application decides to clear the Amazon Sidewalk stack's configuration and keys that were established during registration from non-volatile storage. The `sid_set_factory_reset` API places the Amazon Sidewalk stack in pre-registration state.

Application usage of this API shall be tied to a button press or other event to trigger a de-registration process. When this API is invoked, if a link is available, the SDK attempts to send notification to the Amazon Sidewalk cloud before clearing the Amazon Sidewalk stack's configuration and keys.

On completion of clearing the Amazon Sidewalk stack's configuration and keys, `on_factory_reset` callback shall be triggered from the SDK.

`SID_ERROR_INVALID_ARGS` is returned by the API if the Amazon Sidewalk stack is not initialized or if the Sidewalk handle is `NULL`.

`SID_ERROR_STOPPED` is returned by the API if the Amazon Sidewalk stack is not started.

```
/**
 * set factory reset
 *
 * Inform the sidewalk stack the factory reset event.
 * The sidewalk library clears its configuration from the non volatile storage
 * and resets its state accordingly.
 * The sidewalk link status resets to #SID_STATE_DISABLED.
 * The device needs to successfully complete device registration with the
 * cloud services for the sidewalk library to sned and receive messages
 *
 * @param[in] handle A pointer to the handle returned by sid_init
 *
 * @returns #SID_ERROR_NONE on success.
 */
sid_error_t sid_set_factory_reset(struct sid_handle *handle);
```

## 3.13 sid_ble_connection_request

The `sid_ble_connection_request` API can only be used on the BLE link.

The `sid_ble_connection_request` API is called by the developer's application to request connection from a BLE Gateway, or to cancel a request for a connection.

When the `sid_ble_connection_request` API is called by the developer's application to request connection, the Endpoint requests an uplink connection on BLE through a BLE Gateway by updating its beacon's advertisement payload, requesting a BLE Gateway to connect.

The `sid_ble_connection_request` API must be called again to stop the request to connect. This is because when the `sid_ble_connection_request` has requested a connection, the beacons have `connect request` set to `TRUE`. The value of `connect request` then remains set to `TRUE` until set to `FALSE` by another call to `sid_ble_connection_request`.

`SID_ERROR_NOSUPPORT` is returned by the API if the BLE link is not compiled into the image.

`SID_ERROR_INVALID_ARGS` is returned if one or more of the following conditions is True:

1. BLE is not initialized.
2. BLE is not started.
3. The Sidewalk handle is `NULL`.

`SID_ERROR_INVALID_STATE` is returned if one of the following conditions occurs:

1. The Sidewalk state is un-registered
2. The Sidewalk time sync state is `FALSE` (indicating that the Endpoint does not have time synchronization)

`SID_ERROR_ALREADY_EXISTS` is returned if the BLE is already beaconing with the request to connect.

`SID_ERROR_NONE` is returned if the Amazon Sidewalk stack can successfully handle the request.

```
/**
 * Using this API, the device can request that the Sidewalk gateway initiates
 * a connection to the device while the device is advertising via BLE
 * (Sidewalk beaconing). After a connection is dropped the developer's
 * application has to set this beacon state again. A Gateway may not always be
 * able to honor this request depending on the number of devices connected to
 * it.
 *
 * @param[in]  handle    A pointer to the handle returned by sid_init().
 * @param[in]  set       Set or clear the connection request in BLE
 *                       advertising packet.
 *
 * @returns #SID_ERROR_NONE on success.
 * @returns #SID_ERROR_ALREADY_EXISTS when device is already connected to a
 *                                    Sidewalk gateway and connection request
 *                                    is set.
 */
sid_error_t sid_ble_bcn_connection_request(struct sid_handle *handle,
                                           bool set);

// Enable connection request
sid_error_t ret = sid_ble_bcn_connection_request(struct sid_handle *handle,
                                                 true);

// Disable connection request
sid_error_t ret = sid_ble_bcn_connection_request(struct sid_handle *handle,
                                                 false);
```

## 3.14 sid_set_msg_dest_id

Amazon Sidewalk currently supports only one destination ID: AWS IoT Core for Amazon Sidewalk When the developer's application sends a message using the Amazon Sidewalk stack, using link mode cloud, the destination ID is set to 32 (the ID of AWS IoT Core for Amazon Sidewalk). The `sid_set_msg_dest_id` API configures the destination IDs, for future use when Amazon Sidewalk supports destination IDs other than AWS IoT Core for Amazon Sidewalk.

`SID_ERROR_INVALID_ARGS` is returned if handle is `NULL`. `SID_ERROR_NONE` is returned if the destination ID can be set successfully.

```
/**
 * Set destination ID for messages.
 *
 * By default, the destination ID is set to #SID_MSG_DESTINATION_AWS_IOT_CORE
 * unless changed by sid_set_msg_dest_id(). The destination ID is retained
 * until the device resets or its changed by another invocation of
 * sid_set_msg_dest_id().
 *
 * @see sid_put_msg().
 *
 * @param[in] handle A pointer to the handle returned by sid_init().
 * @param[in] id     The new destination ID.
 *
 * @returns #SID_ERROR_NONE on success.
 */
sid_error_t sid_set_msg_dest_id(struct sid_handle *handle, uint32_t id);

sid_error_t  ret = sid_set_msg_dest_id(handle, 32); //only AWS IoT destination
                                                    //is currently supported
```

## 3.15 sid_get_status

The status of the Amazon Sidewalk stack can be polled by calling the `sid_get_status` API. For details of the `sid_status` structure, see section 2.6.

`SID_ERROR_INVALID_ARGS` is returned if the Sidewalk handle passed is `NULL` or if the Amazon Sidewalk stack is not initialized.

`SID_ERROR_NONE` is returned when the current status can be successfully returned to the developer's application.

```
/**
 * Get current status from Sidewalk library.
 *
 * @warning sid_get_status() should be called in the same caller context as
 * sid_process().
 *
 * @warning sid_get_status() must not be called from within the caller context
 * of any of the sid_event_callbacks registered during sid_init() to avoid
 * re-entrancy and recursion problems.
 *
 * @param[in] handle A pointer to the handle returned by sid_init()
 * @param[out] current_status A pointer to store the sdk current status
 *
 * @returns #SID_ERROR_NONE in case of success.
 * @returns #SID_ERROR_INVALID_ARGS when Sidewalk library is not initialized.
 */
sid_error_t sid_get_status(struct sid_handle *handle,
                              struct sid_status *current_status);

struct sid_status current_status;
sid_error_t ret = sid_get_status(handle, &current_status);
```

## 3.16 sid_get_time

sid_get_time provides the caller with the GPS time. UTC and local time are not supported.

SID_ERROR_NOSUPPORT is returned when the sid_get_time call requests time other than GPS time.

SID_ERROR_INVALID_ARGS is returned if one or more of the following conditions occurs:

1. The handle is NULL.
2. The pointer to hold the time is NULL
3. The Amazon Sidewalk stack is not initialized

SID_ERROR_NONE is returned if the API can successfully provide GPS time.

```
/**
 * Get time from the Sidewalk library with the requested format.
 *
 * @warning sid_get_time() should be called in the same caller context as
 * sid_process().
 *
 * @warning sid_get_time() must not be called from within the caller context
 * of any of the sid_event_callbacks registered during sid_init() to avoid
 * re-entrancy and recursion problems.
 *
 * @param[in] handle     A pointer to the handle returned by sid_init().
 * @param[in] format     The time format to query
 * @param[out] curr_time A pointer to store the current time in
 *                       sid_timespec format.
 *
 * @returns #SID_ERROR_NONE in case of success.
 * @returns #SID_ERROR_INVALID_ARGS when sidewalk is not initialized or not
 *          registered or invalid format is supplied.
 * @returns #SID_ERROR_UNINITIALIZED when time is not available.
 */
sid_error_t sid_get_time(struct sid_handle *handle,
                          enum sid_time_format format,
                          struct sid_timespec *curr_time);

struct sid_timespec curr_time;

sid_error_t ret = sid_get_time(handle, SID_GET_GPS_TIME, &curr_time);
```

## 3.17 sid_bulk_data_transfer_init

SBDT can be initialized using sid_bulk_data_transfer_init API. The sid_bulk_data_transfer_init API accepts a pointer to the SBDT configuration and sid_handle. On initialization, SBDT registers for SBDT messages from the cloud, and loads any existing ongoing transfers. If there was an ongoing transfer, the memory is requested from the developer's application to handle said transfers see section 2.13.4.

`sid_bulk_data_transfer_init` should not be called in the context of SBDT Callbacks to avoid re-entrancy and recursion issues. If there was an ongoing transfer `sid_bulk_data_transfer_init` API will trigger `on_transfer_request` API call to request memory and to allow developer's application to either reject or accept the ongoing transfer. If the developer's application chooses to reject the ongoing transfer then ongoing transfer information is wiped away from the KV store API.

Re-initializing SBDT without de-initializing causes the `sid_bulk_data_transfer_init` to fail with the API return error code `SID_ERROR_ALREADY_INITIALIZED`.

The `sid_bulk_data_transfer_init` API returns error code `SID_ERROR_INVALID_ARGS` if any of the following conditions are met:

- Pointers to config and handle are `NULL`.
- Any of the SBDT Callbacks functions are set to `NULL`.
- Amazon Sidewalk stack is not initialized via the `sid_init` API.

The `sid_bulk_data_transfer_init` API will return error code `SID_ERROR_STORAGE_READ_FAIL` if it is not able to read the status of any ongoing transfer's from the KV store API.

```
/**
 * Initializes Sidewalk bulk transfer
 *
 * @see sid_bulk_data_transfer_init
 * @see sid_init
 *
 * @param[in] config Required configuration needed to initialize Sidewalk
 * bulk transfer
 * @param[in] handle A pointer to the handle returned by sid_init()
 *
 * @returns #SID_ERROR_NONE in case of success
 */
sid_error_t sid_bulk_data_transfer_init
            (const struct sid_bulk_data_transfer_config *config,
             struct sid_handle *handle);


// Initialize BLE link
struct sid_config config = {
    .link_mask = (1 << SID_LINK_TYPE_1),
    .callbacks = &event_callbacks,
    .link_config = app_get_ble_config(), // update the BLE config
};
static struct sid_handle* sidewalk_handle;
sid_error_t ret = sid_init(config, &sid_handle);

struct sid_bulk_data_transfer_config sbdt_config = {
    .callbacks = &sbdt_event_callbacks,
};
ret = sid_bulk_data_transfer_init(&sbdt_config, sidewalk_handle);
```

## 3.18 sid_bulk_data_transfer_deinit

SBDT can be de-initialized using sid_bulk_data_transfer_deinit API.
The sid_bulk_data_transfer_deinit API requires the pointer to the internal handle that was provided to the developer's application during initialization. The sid_bulk_data_transfer_deinit API deregisters SBDT from handling any incoming SBDT commands and releases the memory that was acquired via on_transfer_request callback using the on_release_scratch callback. Calling sid_deinit API will not de initialize SBDT and it needs to be done separately as well.

- The sid_bulk_data_transfer_deinit return's SID_ERROR_INVALID_ARGS if the handle passed is NULL or is not initialized.

```
/**
 * De-initialize Sidewalk bulk transfer
 *
 * @see sid_bulk_data_transfer_init
 *
 * @param[in] handle A pointer to the handle returned by sid_init()
 *
 * @returns #SID_ERROR_NONE in case of success
 */
sid_error_t sid_bulk_data_transfer_deinit(struct sid_handle* handle);

sid_error_t ret = sid_bulk_data_transfer_deinit(sidewalk_handle);
```

## 3.19 sid_bulk_data_transfer_release_buffer

The sid_bulk_data_transfer_release_buffer API is called to indicate to SBDT that the developer's application is done processing the data buffer containing the fragment it received in on_data_recieved callback, and it's ready to receive the next one. The sid_bulk_data_transfer_release_buffer API in conjunction with on_data_recieved callback can handle the below cases in respect to flash write. The developer's application can write the data received in on_data_recieved to flash and release the buffer via sid_bulk_data_transfer_release_buffer within the on_data_recieved callback itself. If the write takes too long that it impacts the performance of the system, the write can be scheduled to a later time and sid_bulk_data_transfer_release_buffer can be called once the write is done. This approach gives the developer enough flexibility to handle both cases with respect to writes.

- sid_bulk_data_transfer_release_buffer returns SID_ERROR_INVALID_ARGS if
  - sid_handle is NULL
  - If Sidewalk stack is not initialized
- sid_bulk_data_transfer_release_buffer returns SID_ERROR_NOT_FOUND if
  - file_id is incorrect or it does not find an ongoing transfer corresponding to the given file_id
  - The given buffer does not match the one received in the on_recieved_callback
- sid_bulk_data_transfer_release_buffer returns SID_ERROR_INVALID_STATE if
  - SBDT was not initialized via the sid_bulk_data_transfer_init API call
  - If developer's application is trying to release a buffer that was never given or already released

```
/**
 * Release data buffer
 *
 * This API is called to release ownership of the data buffer back to bulk
 * transfer
 *
 * @param[in] handle A pointer to the handle returned by sid_init()
 * @param[in] file_id Identifier of the file transfer whose buffer is being
 * released
 * @param[in] buffer A pointer to the buffer that needs to be released, which
 * is never NULL
 *
 * @returns #SID_ERROR_NONE in case of success
 */
sid_error_t sid_bulk_data_transfer_release_buffer(struct sid_handle* handle,
    uint32_t file_id,
    const struct sid_bulk_data_transfer_buffer* const buffer);

void sbdt_on_data_recieved
     (const struct sid_bulk_data_transfer_desc *const desc,
      const struct sid_bulk_data_transfer_buffer *const buffer,
      void *context) {

    // Illustrative API indicating write to flash
    write_to_flash(buffer->data, buffer->size);

    sid_error_t ret = sid_bulk_data_transfer_release_buffer(sidewalk_handle,
                                                            desc->file_id,
                                                            buffer);
}
```

## 3.20 sid_bulk_data_transfer_finalize

The developer's application calls `sid_bulk_data_transfer_finalize` to indicate weather the file was fully transferred and it passed any of user's validation checks, such as CRC32 check or signature verification checks. This API is used in conjunction with `on_finalize_request`, where once the developer's application receives the `on_finalize_request` callback the developer's application verifies the received file and calls `sid_bulk_data_transfer_finalize` with the appropriate success or failure response.
The `sid_bulk_data_transfer_finalize` API can be called within any context.

- `sid_bulk_data_transfer_finalize` returns `SID_ERROR_INVALID_ARGS` if
  - `sid_handle` is `NULL`
  - If Sidewalk stack is not initialized
- `sid_bulk_data_transfer_finalize` returns `SID_ERROR_NOT_FOUND` if
  - `file_id` is incorrect or it does not find an ongoing transfer corresponding to the given `file_id`
- `sid_bulk_data_transfer_finalize` returns `SID_ERROR_INVALID_STATE` if
  - SBDT was not initialized via the `sid_bulk_data_transfer_init` API call
  - If developer's application is trying to finalize a transfer before all of the file is received

```
/**
 * Complete an ongoing Sidewalk bulk transfer
 *
 * This function is called when on_finalize_request callback is invoked. The
 * purpose of the API is to inform bulk transfer if the transfer was
 * successful (CRC checks/signature is verified), or it was a failure.
 *
 * @see sid_bulk_data_transfer_event_callbacks.on_finalize_request
 *
 * @param[in] handle A pointer to the handle returned by sid_init()
 * @param[in] file_id Identifier of the file being transferred
 * @param[in] status Indicates if the transfer was completed successfully from
 * the user end
 *
 * @returns #SID_ERROR_NONE in case of success
 */
sid_error_t sid_bulk_data_transfer_finalize
            (struct sid_handle *handle, uint32_t file_id,
             enum sid_bulk_data_transfer_final_status status);

void sbdt_on_finalize_request(uint32_t file_id, void *context)
{
    // Illustrative API indicating verification of file
    bool success = validate_recieved_file(file_id);

    sid_error_t ret = sid_bulk_data_transfer_finalize
                        (sidewalk_handle, file_id,
                         success ? SID_BULK_DATA_TRANSFER_FINAL_STATUS_SUCCESS
                                 : SID_BULK_DATA_TRANSFER_FINAL_STATUS_FAILURE);
}
```

## 3.21 sid_bulk_data_transfer_cancel

The developer's application can use `sid_bulk_data_transfer_cancel` API to cancel an ongoing transfer. When the developer's application calls the `sid_bulk_data_transfer_cancel`, SBDT informs AWS of the cancellation request and calls `on_cancel_request` callback and `on_release_scratch`.

- `sid_bulk_data_transfer_cancel` returns `SID_ERROR_INVALID_ARGS` if
  - `sid_handle` is `NULL`
  - If Sidewalk stack is not initialized
- `sid_bulk_data_transfer_cancel` returns `SID_ERROR_NOT_FOUND` if
  - `file_id` is incorrect or it does not find an ongoing transfer corresponding to the given `file_id`
- `sid_bulk_data_transfer_cancel` returns `SID_ERROR_INVALID_STATE` if
  - SBDT was not initialized via the `sid_bulk_data_transfer_init` API call
- `sid_bulk_data_transfer_cancel` returns `SID_ERROR_NO_ROUTE_AVAILABLE` if the transfer link is not connected. The developer's application will then need to re-establish the link again before calling the `sid_bulk_data_transfer_cancel` again.

```
/**
 * Cancels any ongoing bulk transfer
 *
 * @param[in] handle A pointer to the handle returned by sid_init()
 * @param[in] file_id Identifier of the file transfer that needs to be
 * cancelled
 * @param[in] reason Reason for cancelling the transfer
 *
 * @return #SID_ERROR_NONE in case of success
 */
sid_error_t sid_bulk_data_transfer_cancel
            (struct sid_handle *handle, uint32_t file_id,
             enum sid_bulk_data_transfer_reject_reason reason);

sid_error_t ret = sid_bulk_data_transfer_cancel
                  (sidewalk_handle, file_id,
                   SID_BULK_DATA_TRANSFER_REJECT_REASON_NONE);
```

## 3.22 sid_bulk_data_transfer_get_stats

The developer's application can use `sid_bulk_data_transfer_get_stats` API to get the current progress of the file transfer. It returns the fragment size, size of the file transferred and total size of the file. The developer's application can call this API anytime during the duration of the transfer.

- `sid_bulk_data_transfer_get_stats` returns `SID_ERROR_INVALID_ARGS` if
  - `sid_handle` is `NULL`
  - If Sidewalk stack is not initialized
  - If the `stats` argument is `NULL`
- `sid_bulk_data_transfer_get_stats` returns `SID_ERROR_NOT_FOUND` if
  - `file_id` is incorrect or it does not find an ongoing transfer corresponding to the given `file_id`
- `sid_bulk_data_transfer_get_stats` returns `SID_ERROR_INVALID_STATE` if
  - SBDT was not initialized via the `sid_bulk_data_transfer_init` API call

```
/**
 * Get the stats of an ongoing Sidewalk bulk transfer
 *
 * This function is called to get the current stats/progress of an ongoing
 * file transfer
 *
 * @param[in] handle A pointer to the handle returned by sid_init()
 * @param[in] file_id Identifier of the file being transferred
 * @param[in] stats A pointer to the structure that holds the stats
 * information
 *
 * @returns #SID_ERROR_NONE in case of success
 */
sid_error_t sid_bulk_data_transfer_get_stats
            (struct sid_handle *handle, uint32_t file_id,
             struct sid_bulk_data_transfer_stats *const stats);

struct sid_bulk_data_transfer_stats stats = {};
sid_error_t ret = sid_bulk_data_transfer_get_stats(sidewalk_handle, file_id,
                                                    &stats);
```

# Chapter 4

# Sidewalk Bulk Data Transfer Status Codes

For the values of the Sidewalk Bulk Data Transfer status codes see table 4.1.

For the description of the Sidewalk Bulk Data Transfer status codes see table 4.2.

For the FuotaDeviceStatus of the Sidewalk Bulk Data Transfer status codes see table 4.3.

| **Sidewalk Bulk Data Transfer Status Code** | **Value** |
|---|---|
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_NONE` | 0x00 |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_GENERIC` | 0x01 |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_VERSION_MISMATCH` | 0x02 |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_FILE_TOO_BIG` | 0x03 |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_NO_SPACE` | 0x04 |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_LOW_BATTERY` | 0x05 |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_FRAGMENT_VERIFICATION_FAILED` | 0x06 |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_MISSING_CHUNKS` | 0x07 |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_WRITE_FAILED` | 0x08 |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_FILE_VERIFICATION_FAILED` | 0x09 |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_TRANSFER_RESTART` | 0x0A |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_FILE_ALREADY_EXISTS` | 0x0B |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_LINK_NOT_SUPPORTED` | 0x0C |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_INVALID_FILE_ID` | 0x0D |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_INVALID_FRAGMENT_SIZE` | 0x0E |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_INVALID_FRAGMENT_ID` | 0x0F |

Table 4.1: Values of the Sidewalk Bulk Data Transfer Status Codes.

| Sidewalk Bulk Data Transfer Status Code | Description |
|---|---|
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_NONE` | Success |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_GENERIC` | Generic Error |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_VERSION_MISMATCH` | SBDT version mismatch |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_FILE_TOO_BIG` | File Too Big |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_NO_SPACE` | No Space |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_LOW_BATTERY` | Battery power low |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_FRAGMENT_VERIFICATION_FAILED` | Fragment Verification Failed |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_MISSING_CHUNKS` | Missing Chunks |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_WRITE_FAILED` | Write Failed |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_FILE_VERIFICATION_FAILED` | File Verification Failed |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_TRANSFER_RESTART` | Restart Transfer |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_FILE_ALREADY_EXISTS` | File already exists |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_LINK_NOT_SUPPORTED` | Link type not supported |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_INVALID_FILE_ID` | Invalid File ID |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_INVALID_FRAGMENT_SIZE` | Invalid fragment Size |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_INVALID_FRAGMENT_ID` | Invalid fragment ID |

Table 4.2: Description of the Sidewalk Bulk Data Transfer Status Codes Description.

| Sidewalk Bulk Data Transfer Status Code | FuotaDeviceStatus |
|---|---|
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_NONE` | `Successful` |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_GENERIC` | `Wrong_descriptor` |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_VERSION_MISMATCH` | `Wrong_descriptor` |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_FILE_TOO_BIG` | `Package_Not_Supported` |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_NO_SPACE` | `Not_enough_memory` |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_LOW_BATTERY` | `Low_Battery` |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_FRAGMENT_VERIFICATION_FAILED` | `Security_Error` |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_MISSING_CHUNKS` | `MissingFrag` |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_WRITE_FAILED` | `File_System_Error` |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_FILE_VERIFICATION_FAILED` | `Security_Error` |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_TRANSFER_RESTART` | `Restart_Transfer` |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_FILE_ALREADY_EXISTS` | `Already_Available` |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_LINK_NOT_SUPPORTED` | `Link_Not_Supported` |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_INVALID_FILE_ID` | `Internal_Error` |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_INVALID_FRAGMENT_SIZE` | `Internal_Error` |
| `SID_BULK_DATA_TRANSFER_STATUS_ERROR_INVALID_FRAGMENT_ID` | `MissingFrag` |

Table 4.3: FuotaDeviceStatus of the Sidewalk Bulk Data Transfer Status Codes.

# Chapter 5

# Amazon Sidewalk Error Codes

For the values of the Amazon Sidewalk error codes see table 5.1.

| **Sidewalk Error Code** | **Value** |
|---|---|
| SID_ERROR_NONE | 0 |
| SID_ERROR_GENERIC | -1 |
| SID_ERROR_TIMEOUT | -2 |
| SID_ERROR_OUT_OF_RESOURCES | -3 |
| SID_ERROR_OOM | -4 |
| SID_ERROR_OUT_OF_HANDLES | -5 |
| SID_ERROR_NOSUPPORT | -6 |
| SID_ERROR_NO_PERMISSION | -7 |
| SID_ERROR_NOT_FOUND | -8 |
| SID_ERROR_NULL_POINTER | -9 |
| SID_ERROR_PARAM_OUT_OF_RANGE | -10 |
| SID_ERROR_INVALID_ARGS | -11 |
| SID_ERROR_INCOMPATIBLE_PARAMS | -12 |
| SID_ERROR_IO_ERROR | -13 |
| SID_ERROR_TRY_AGAIN | -14 |
| SID_ERROR_BUSY | -15 |
| SID_ERROR_DEAD_LOCK | -16 |
| SID_ERROR_DATA_TYPE_OVERFLOW | -17 |
| SID_ERROR_BUFFER_OVERFLOW | -18 |
| SID_ERROR_IN_PROGRESS | -19 |
| SID_ERROR_CANCELED | -20 |
| SID_ERROR_OWNER_DEAD | -21 |
| SID_ERROR_UNRECOVERABLE | -22 |
| SID_ERROR_PORT_INVALID | -23 |
| SID_ERROR_PORT_NOT_OPEN | -24 |
| SID_ERROR_UNINITIALIZED | -25 |
| SID_ERROR_ALREADY_INITIALIZED | -26 |
| SID_ERROR_ALREADY_EXISTS | -27 |
| SID_ERROR_BELOW_THRESHOLD | -28 |
| SID_ERROR_STOPPED | -29 |
| SID_ERROR_STORAGE_READ_FAIL | -30 |
| SID_ERROR_STORAGE_WRITE_FAIL | -31 |
| SID_ERROR_STORAGE_ERASE_FAIL | -32 |
| SID_ERROR_STORAGE_FULL | -33 |
| SID_ERROR_AUTHENTICATION_FAIL | -34 |
| SID_ERROR_ENCRYPTION_FAIL | -35 |
| SID_ERROR_DECRYPTION_FAIL | -36 |
| SID_ERROR_ID_OBFUSCATION_FAIL | -37 |
| SID_ERROR_NO_ROUTE_AVAILABLE | -38 |
| SID_ERROR_INVALID_RESPONSE | -39 |
| SID_ERROR_INVALID_STATE | -40 |

Table 5.1: Amazon Sidewalk Error Codes.

# Glossary

| | |
|---|---|
| BLE | Bluetooth low energy. |
| CSS | Chirp Spread Spectrum. |
| Downlink | Data sent from a Gateway to an Endpoint. |
| Endpoint | A device that accesses services of Amazon Sidewalk to transport messages to and from the application services through AWS IoT. |
| FFN | Frustration Free Networking. |
| FFS | Frustration Free Setup. |
| FSK | 2-Gaussian Frequency Shift Keying modulation scheme. |
| Gateway | Connections between Endpoints and Amazon Sidewalk are established and maintained through these devices. Gateways are Amazon and Ring devices. Amazon Sidewalk messages are transported between Amazon Sidewalk and the AWS IoT service through these devices. |
| ISR | Interrupt Service Routine. |
| LoRa | Long Range proprietary protocol based on Chirp Spread Spectrum modulation scheme (CSS). |
| MTU | Maximum Transmission Unit. |
| PAL | Platform abstraction Layer. |
| SBDT | Sidewalk Bulk Data Transfer. |
| Sid API | Amazon Sidewalk Application Programming Interface. |
| Uplink | Data sent from an Endpoint to a Gateway. |

# Chapter 6

# Change History

| **Version** | **Summary of Changes** |
|---|---|
| Protocol Stack 1.0, Document Revision A | First release of Sid API User Guide. |
| Protocol Stack 1.0, Document Revision A.1 | Add Sidewalk File transfer and Multi-link feature support. |