Talaria TWO™ Host API Reference Manual

Release: 06-10-2021

InnoPhase, Inc.
6815 Flanders Drive
San Diego, CA 92121
innophaseinc.com

Revision History

Version	Date	Comments
1.0	06-29-2020	First release
1.1	09-03-2020	Updated for SDK 2.1.1 release
2.0	06-10-2021	Updated for SDK 2.2 release

Contents

1 Figures

2 Terms & Definitions

AES	Advanced Encryption Standard
A-MPDU	Aggregate MAC Protocol Data Unit
AP	Access Point
API	Application Programming Interface
BLE	Bluetooth Low Energy
BSD	Berkeley Software Distribution
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
EAP	Extensible Authentication Protocol
FAST	Flexible Authentication via Secure Tunneling
GCM	Galois/Counter Mode
GTC	Generic Token Card
HAPI	Host Application Processor Interface
HIO	Host Interface Operation
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IoT	Internet of Things
IP	Internet Protocol
IRQ	Interrupt Request Line
LEAP	Lightweight Extensible Authentication Protocol
MAC	Media Access Control

MQTT	Message Queuing Telemetry Transport
MS-CHAP	Microsoft version of the Challenge-Handshake Authentication Protocol
OS	Operating System
PEAP	Protected Extensible Authentication Protocol
PHY	Physical Layer
PSK	Pre Shared Key
PUF	Physically Unclonable Function
RC4	Rivest Cipher 4
RF	Radio Frequency
RTOS	Real Time Operating System
Rx	Receive
SHA1/2	Secure Hash Algorithm 1/2
SPI	Serial Peripheral Interface
SSID	Service Set Identifier
SSL	Secure Sockets Layer
T2	Talaria TWO™
TCP	Transmission Control Protocol
TDES	Triple Data Encryption Algorithm
TLS	Transport Layer Security
TTLS	Tunneled Transport Layer Security
UART	Universal Asynchronous Receiver-Transmitter
UDP	User Datagram Protocol
WLAN	Wireless Local Area Network
WPA	Wireless Access Point
XEX	Ciphertext Stealing

3 Introduction

The InnoPhase Talaria TWO™ Multi-Protocol Platform is a highly integrated, single-chip wireless solution offering ultimate size, power, and cost advantages for a wide range of low-power IoT designs. The Talaria TWO™ system was designed for power efficiency and intelligent integration from the beginning for the unique demands of IoT applications.

4 Architecture

4.1 Overview

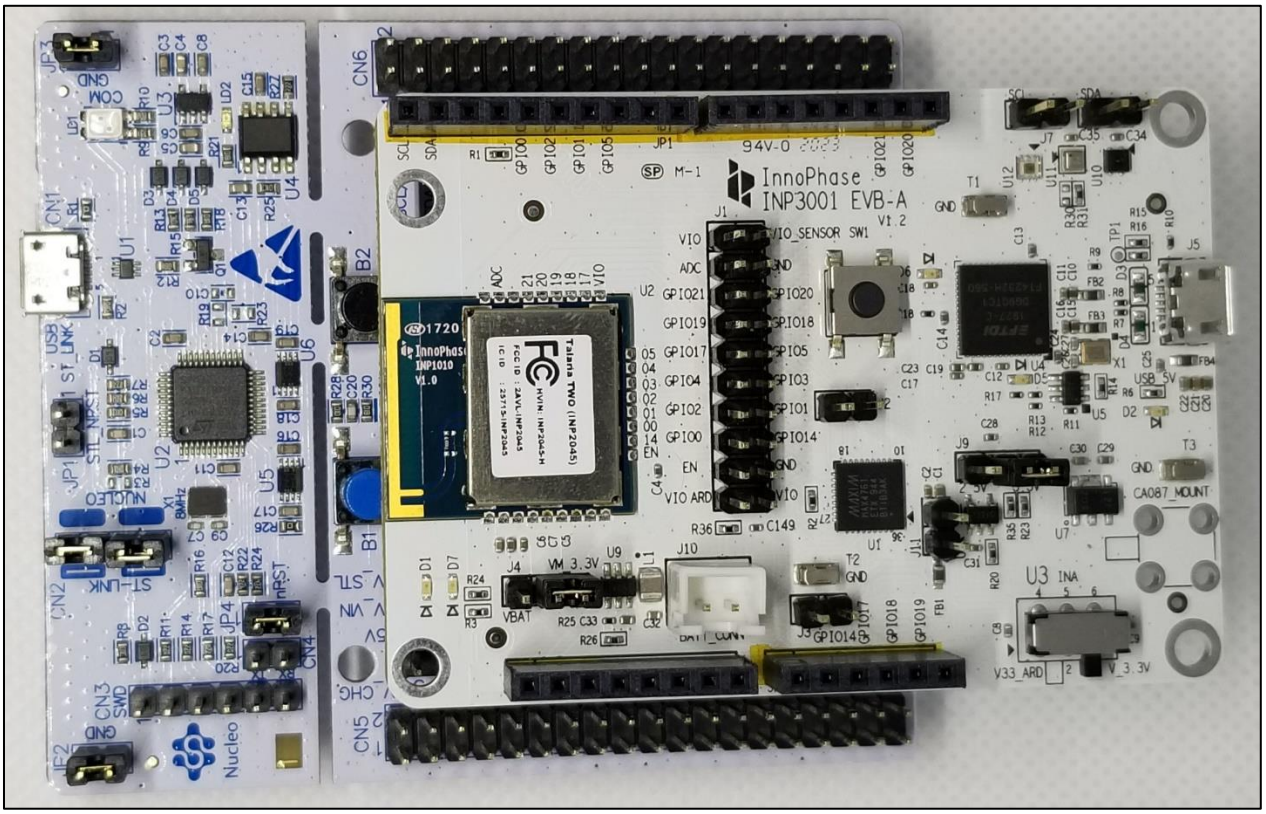

Figure 1: Talaria TWO™ Multi-Protocol Platform shown as a shield to STM32L433RC-P board

5 Talaria TWO™ System on Chip (SoC)

Talaria TWO™ performs the following based on commands from the Host processor.

1. Provides wireless (802.11b/g/n) link between the Host processor and AP or Hotspot
2. Scan and Connect to the AP specified by the Host
3. Performs WPA2 security handshake
4. Enables IP supports like TCP, UDP and DHCP
5. Adds network protocols like MQTT and HTTP
6. Supports transport protocols like SSL and TLS
7. Supports data scrambles on Serial interface
8. Provides BLE connectivity for provisioning

The major components in Talaria TWO™ are shown in Figure 2.

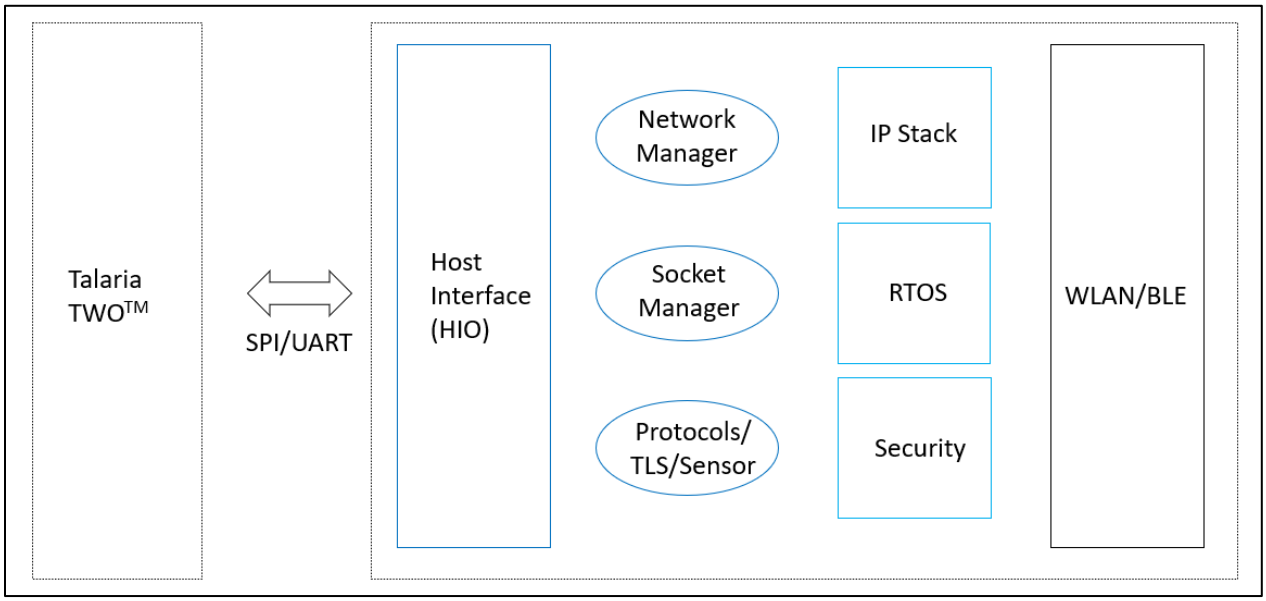

Figure 2: Major components in Talaria TWO™

5.1 Wi-Fi Connection Manager

This is the network connection manager which handles all the Wi-Fi connection/disconnection.

5.2 Socket Manager

HIO handles all socket operations. It supports TCP, UDP, and raw sockets.

5.3 RTOS

Highly efficient, low footprint, real-time OS for low power applications.

5.4 IPSTACK

1. IPv4
2. ICMP
3. UDP
4. TCP
5. DHCP
6. DNS Resolver
7. BSD Sockets Interface
8. TLS
9. MQTT
10. IPv6

6 STM32 Host Processor

Host processor consists of the Host Application Processor (HAPI) Interface Layer and Host Applications. Host Applications may vary and will interact with Talaria TWO™ via APIs in the interface layer. HAPI provides APIs for Host Application to facilitate communication with the Talaria TWO™.

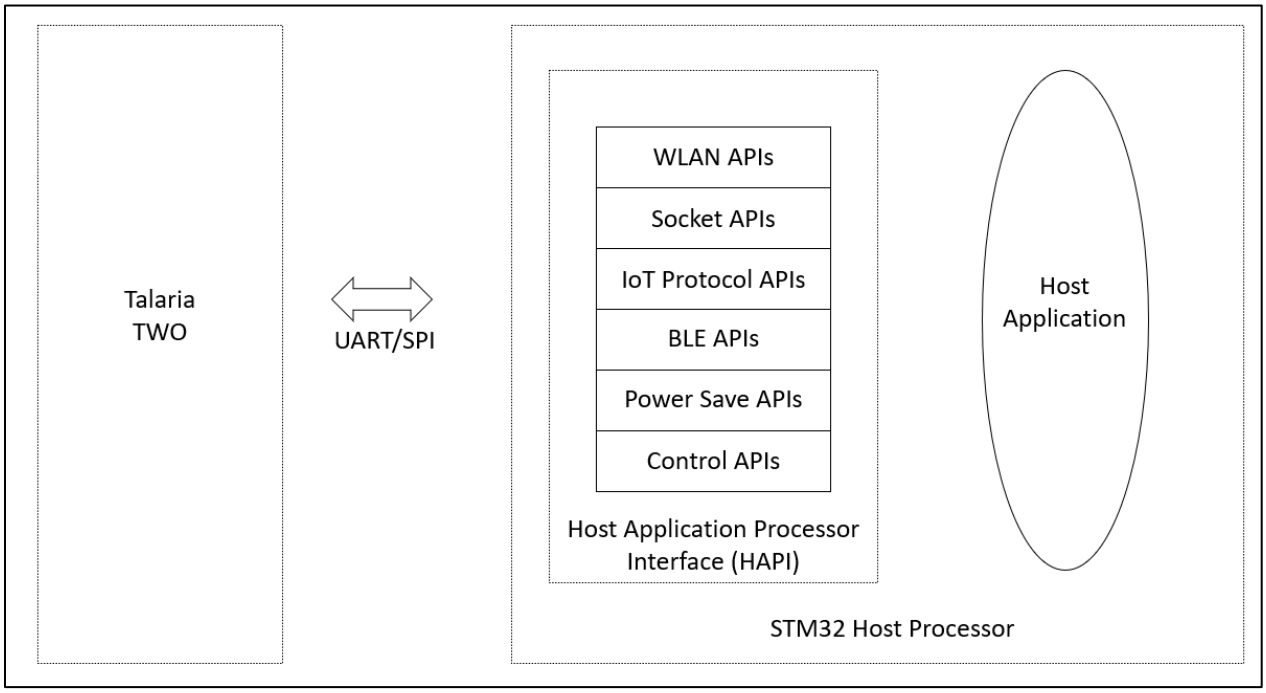

Figure 3: Communication between Host and Talaria TWO™ via UART/SPI

7 Talaria TWO™ – Host Processor Interface

Host processor communicates with Talaria TWO™ via a SPI or UART and follows a protocol to exchange command and data. This protocol is implemented on the host side and are provided as APIs. The host application can then use these APIs to access and control Talaria TWO™.

7.1 Talaria TWO™ – Host APIs (HAPI)

APIs are grouped into:

1. WLAN APIs
2. Socket APIs
3. BLE APIs
4. IoT Protocols
5. Interface Port APIs

Host applications use HAPI WLAN and Socket APIs, which internally use interface port APIs to transfer data between the wireless network and host processor.

7.1.1 Port APIs

These APIs provides basic read/write over the hardware interface (SPI/UART) between the host and Talaria TWO™ where each API must be defined for each port.

7.1.1.1 hapi_uart_init

API to initialize HAPI UART interface.

```
void hapi_uart_init(struct hapi * hapi, void* hapi_uart_ptr, int
baudrate)
```

Arguments:

1. hapi: HAPI instance pointer.
2. hapi_uart_ptr: Pointer to the HAPI UART instance.
3. baudrate: Baud rate to be configured.

Return: None

7.1.1.2 hapi_uart_write

API to write data to Talaria TWO™ over UART HAPI.

```
ssize_t hapi_uart_write(struct hapi *hapi, void *data, size_t
length)
```

Arguments:

1. hapi: HAPI instance pointer.
2. data: Source buffer address.
3. length: Number of bytes to be written.

Return: number of bytes >0 on success, -1 on error.

7.1.1.3 hapi_uart_read

API to read data from Talaria TWO™ over UART HAPI.

```
ssize_t hapi_uart_read(struct hapi *hapi, void *data, size_t
length)
```

Arguments:

1. hapi: HAPI instance pointer.
2. data: Source buffer address.
3. length: Number of bytes to be read.

Return: >=0 on success. -1 on error.

7.1.1.4 hapi_spi_init

API to initialize HAPI SPI interface.

```
void hapi_spi_init(struct hapi * hapi, void* hapi_spi_ptr)
```

Arguments:

1. hapi: HAPI instance pointer.
2. hapi_spi_ptr: Pointer to the HAPI SPI instance.

Return: None.

7.1.1.5 hapi_spi_write

API to write data to Talaria TWO™ over SPI HAPI.

```
ssize_t hapi_spi_write(struct hapi *hapi, void *data, size_t
length)
```

Arguments:

1. hapi: HAPI instance pointer.
2. data: Source buffer address.
3. length: Number of bytes to be written.

Return: number of bytes >0 on success, -1 on error.

7.1.1.6 hapi_spi_read

API to read data from Talaria TWO™ over SPI HAPI.

```
ssize_t hapi_spi_read(struct hapi *hapi, void *data, size_t
length)
```

Arguments:

1. hapi: HAPI instance pointer.
2. data: Source buffer address.
3. length: Number of bytes to be read.

Return: >=0 on success. -1 on error.

7.1.2 WLAN APIs

7.1.2.1 hapi_wcm_create

Creates the HAPI WLAN manager interface and should be called before any WLAN APIs.

```
struct hapi_wcm * hapi_wcm_create(struct hapi *hapi)
```

Arguments:

1. hapi: HAPI instance pointer.

Return: a valid pointer points to the HAPI WLAN instance on success. -1 on error.

7.1.2.2 hapi_wcm_autoconnect

Triggers the scan and connects/disconnects to the AP specified by the SSID and uses the passphrase for the WPA2 security.

```
Bool      hapi_wcm_autoconnect(struct      hapi_wcm      *hapi_wcm,
int flag, const char *ssid,const char *passphrase);
```

Arguments:

1. hapi_wcm: HAPI WLAN instance pointer.
2. Flag: Allow to connect/disconnect, 1=connection enabled, 0=disconnect.
3. SSID: SSID of the AP to scan and connect.
4. Passphrase: Password for the WPA2-PSK security.

Return: 0(OK) On success. -1(FAIL) on error.

7.1.2.3 hapi_wcm_set_link_cb

Registers the callback function to the HAPI WLAN interface for the asynchronous WLAN link change notification.

```
void hapi_wcm_set_link_cb(struct hapi_wcm *hapi_wcm,

hapi_wcm_link_cb cb, void *context)
```

Arguments:

1. hapi_wcm: HAPI WLAN instance pointer.
2. cb: The call back function to be registered for link change notification.
3. context: context pointer to be passed when the call back is getting called.

Return: None.

7.1.2.4 hapi_wcm_destroy

Removes the HAPI WLAN manager interface created.

```
bool hapi_wcm_destroy(struct hapi_wcm *hapi_wcm)
```

Arguments:

1. hapi_wcm: HAPI instance pointer.

Return: A valid pointer points to the HAPI WLAN instance on success. -1 on error.

7.1.2.5 hapi_wcm_get_handle

Returns the WCM handle address from hapi_wcm.

```
uint32_t

hapi_wcm_get_handle(struct hapi_wcm *hapi_wcm);
```

Arguments:

1. hapi_wcm: HAPI instance pointer.

Return: a valid pointer points to the HAPI WLAN instance on success. -1 on error.

7.1.2.6 hapi_wcm_scan

Starts the Wi-Fi scan. The scan can be SSID based and/or channel based. Depends on the parameters provided.

```
bool hapi_wcm_scan(struct hapi_wcm *hapi_wcm, const char *ssid, char
channel, int *num)
```

Arguments:

1. hapi_wcm: HAPI WLAN instance pointer.
2. ssid: The SSID to be scanned.
3. Channel: The channel number to be scanned.
4. Num: The pointer to the variable that stores the number scanned results.

Return: 0(OK) On success. -1(FAIL) on error.

7.1.2.7 hapi_wcm_set_scan_cb

Registers the callback function for scan operation. The callback function is getting called when the required number of entries available once the scan starts.

```
void hapi_wcm_set_scan_cb(struct hapi_wcm *hapi_wcm,
hapi_wcm_scan_cb cb, void *context)
```

Arguments:

1. hapi_wcm: HAPI WLAN instance pointer.
2. cb: The callback function to be registered.
3. Context: The context to be passed along when the call back getting called.

Return: None.

7.1.2.8 hapi_wcm_setaddr_4

Sets the ipv4 address to Talaria TWO™ device. This APIs is normally called for setting the static IP.

```
bool hapi_wcm_setaddr_4(struct hapi_wcm *hapi_wcm, unsigned int
*ipaddr, unsigned int *netmask, unsigned int *gw, unsigned int *dns)
```

Arguments:

1. hapi_wcm: HAPI WLAN instance pointer.
2. ipaddr: Pointer contains IP address.
3. netmask: Pointer contains netmask address.
4. gw: Pointer contains gate way address.
5. dns: Pointer contains DNS address.

Return: 0(OK) On success. -1(FAIL) on error.

7.1.2.9 hapi_wcm_getaddr_4

Returns the ipv4 address from Talaria TWO™ device.

```
bool hapi_wcm_getaddr_4(struct hapi_wcm *hapi_wcm, unsigned int
*ipaddr, unsigned int *netmask, unsigned int *gw, unsigned int *dns)
```

Arguments:

1. hapi_wcm: HAPI WLAN instance pointer.
2. ipaddr: pointer to update IP address.
3. netmask: pointer to update netmask address.
4. gw: pointer to update gate way address.
5. dns: pointer to update DNS address.

Return: 0(OK) On success. -1(FAIL) on error.

7.1.3 BLE APIs

7.1.3.1 hapi_bt_host_create

Creates the HAPI BLE interface and should be called before any BLE APIs.

```
struct hapi_bt_host *hapi_bt_host_create(struct hapi *hapi)
```

Arguments:

1. hapi: HAPI instance pointer.

Return: a valid pointer points to the HAPI BLE instance on success. -1 on error.

7.1.3.2 hapi_bt_host_gap_addr_set

Used to set the address of the ble/bt of Talaria TWO™.

```
bool hapi_bt_host_gap_addr_set(struct hapi_bt_host *hapi_bt_host,
uint8_t addr_type, uint8_t *addr)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. addr_type: Type of address set. 0=public, 1=random.
3. addr: Address.

Return: true (0) on success. -1 on error.

7.1.3.3 hapi_bt_host_bt_gap_create

Used to set create the BLE gap device.

```
bool hapi_bt_host_bt_gap_create(struct hapi_bt_host *hapi_bt_host)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.

Return: true (0) on success. -1 on error.

7.1.3.4 hapi_bt_host_bt_gap_destroy

Used to remove the BLE gap service.

```
bool hapi_bt_host_bt_gap_destroy(struct hapi_bt_host
*hapi_bt_host)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.

Return: true (0) on success. -1 on error.

7.1.3.5 hapi_bt_host_gap_cfg_conn

Used to configure the parameter of the BLE gap connection.

```
bool hapi_bt_host_gap_cfg_conn(
        struct hapi_bt_host *hapi_bt_host, uint16_t conn_interval,
        uint16_t conn_latency, uint16_t conn_timeout,
        uint16_t conn_params_reject, uint16_t conn_params_int_min,
        uint16_t conn_params_int_max )
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. conn_interval: The BLE connection interval, in 1.25 ms, range: 0x0006 to 0x0C80 (default: 80).
3. conn_latency: In intervals, range: 0x0000 to 0x01F3 (default: 0).
4. conn_timeout: In ms, range: 0x000A to 0x0C80 (default: 2000).
5. conn_params_reject: Reject parameter update, 1=True, 0=False (default: 0).
6. conn_params_int_min: In 1.25 ms, parameter update min connection interval (default: 6)
7. conn_params_int_max: In 1.25 ms, parameter update max connection interval (default: 8in 1.25 ms, parameter update min connection interval (default: 6)00)

Return: true (0) on success. -1 on error.

7.1.3.6 hapi_bt_host_gap_cfg_smp

Used to configure the parameter of the secure BLE gap connection.

```
bool hapi_bt_host_gap_cfg_smp(struct hapi_bt_host *hapi_bt_host,
        uint8_t io_cap, uint8_t oob, uint8_t bondable,
        uint8_t mitm, uint8_t sc, uint8_t keypress,
        uint8_t key_size, uint8_t encrypt)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. Io_cap: I/O-capabilities: 0-display_only, 1-display_yes_no, 2-keyboard_only, 3-no_input_no_output, 4-keyboard_display (default: 0)
3. oob: OOB exists: 1=True, 0=False (default: 0).
4. bondable: Enable bondable feature: 1=True, 0=False (default: 0).
5. mitm: MITM protection: 1=True, 0=False (default: 0).
6. sc: Secure connection: 1=True, 0=False (default: 0)
7. keypress: Send keypress: 1=True, 0=False (default: 0).
8. keysize: Smallest key size (7..16 octets) (default: 16).
9. encrypt: Automatically encrypt link at connection setup if key exists: 1=True, 0=False (default: 0).

Return: true (0) on success. -1 on error.

7.1.3.7 hapi_bt_host_gap_connectable

Used to configure the connectable mode when it used as peripheral.

```
bool hapi_bt_host_gap_connectable(struct hapi_bt_host
        *hapi_bt_host, uint8_t mode, uint8_t own_type,
        uint8_t peer_type, uint8_t *peer_addr)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. mode: Connectible mode 0=disable, 1=non, 2=direct, 3=undirect.
3. own_type: Type of own address: 0=public, 1=random, 2=resolvable (or public), 3=resolvable (or random).
4. peer_type: Peer address type: 0=public, 1=random.
5. peer_addr: Peer address.

Return: true (0) on success. -1 on error.

7.1.3.8 hapi_bt_host_gap_authenticate

Used to configure the parameter of the secure BLE gap connection.

```
bool hapi_bt_host_gap_cfg_smp(struct hapi_bt_host *hapi_bt_host,
        uint8_t handle, uint8_t oob, uint8_t bondable,
        uint8_t mitm, uint8_t sc, uint8_t key128)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. handle: Connection handle.
3. oob: OOB exists: 1=True, 0=False (default: 0).
4. bondable: Enable bondable feature: 1=True, 0=False (default: 0).
5. mitm: MITM protection: 1=True, 0=False (default: 0).
6. sc: Secure connection: 1=True, 0=False (default: 0)
7. key128: 128-bits key required: 1=True, 0=False.

Return: true (0) on success. -1 on error.

7.1.3.9 hapi_bt_host_gap_set_adv_data

Used to set the advertisement data for the BLE peripheral advertisement.

```
bool hapi_bt_host_gap_set_adv_data(struct hapi_bt_host
            *hapi_bt_host, uint8_t length, uint8_t *data)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. length: The number of significant octets in the advertising data (1 to 31).
3. data: Advertising data.

Return: true (0) on success. -1 on error.

7.1.3.10 hapi_bt_host_gap_broadcast

Used to start the BLE advertisement.

```
bool hapi_bt_host_gap_broadcast(struct hapi_bt_host *hapi_bt_host,
        uint8_t mode, uint8_t own_type, uint8_t peer_type,
        uint8_t *peer_addr)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. mode: Mode, 0=disable, 1=enable.
3. own_type: Type of own address: 0=public, 1=random, 2=resolvable (or public), 3=resolvable (or random).
4. peer_type: Peer address type: 0=public, 1=random.
5. peer_addr: Peer address.

Return: true (0) on success. -1 on error.

7.1.3.11 hapi_bt_host_gap_terminate

Used to terminate the established BLE connection.

```
bool hapi_bt_host_gap_terminate(struct hapi_bt_host *hapi_bt_host,
                                uint8_t handle)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. handle: Connection handle.

Return: true (0) on success. -1 on error.

7.1.3.12 hapi_bt_host_gap_discoverable

Used to configure the discoverable parameter of the BLE device.

```
bool hapi_bt_host_gap_discoverable(struct hapi_bt_host
        *hapi_bt_host, uint8_t mode, uint8_t own_type,
        uint8_t peer_type, uint8_t *peer_addr )
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. mode: Mode, 0=disable, 1=non, 2=limited, 3=general.
3. own_type: Type of own address: 0=public, 1=random, 2=resolvable (or public), 3=resolvable (or random).
4. peer_type: Peer address type: 0=public, 1=random.
5. peer_addr: Peer address.

Return: true (0) on success. -1 on error.

7.1.3.13 hapi_bt_host_gap_discovery

Used to start the discovery of BLE devices.

```
bool hapi_bt_host_gap_discovery(struct hapi_bt_host *hapi_bt_host,
        uint8_t mode, uint8_t own_type, uint8_t peer_type,
        uint8_t *peer_addr)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. Mode: Mode, 0=disable, 1=limited, 2=general, 3=name.
3. own_type: Own address type: 0=public, 1=random, 2=resolvable (or public), 3=resolvable (or random).
4. peer_type: Peer address type (only for mode "name"): 0=public, 1=random, 2=public identity, 3=random identity.
5. peer_addr: Peer address (only for mode "name").

Return: true (0) on success. -1 on error.

7.1.3.14 hapi_bt_host_gap_connection

Used to connect to the BLE peripheral.

```
bool hapi_bt_host_gap_connection( struct hapi_bt_host
*hapi_bt_host, uint8_t mode,uint8_t own_type, uint8_t peer_type,
uint8_t *peer_addr)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. mode: The mode of connection. 0=disable, 1=auto, 2=general, 3=selective, 4=direct ("auto" and "selective" require a white list).
3. own_type: Own address type: 0=public, 1=random, 2=resolvable (or public), 3=resolvable (or random).
4. peer_type: Peer address type (only for mode "name"): 0=public, 1=random, 2=public identity, 3=random identity.
5. peer_addr: Peer address (only for mode "name").

Return: true (0) on success. -1 on error.’

7.1.3.15 hapi_bt_host_gap_connection_update

Used to update the existing BLE connection parameters when it is configured as a peripheral.

```
bool hapi_bt_host_gap_connection_update(
        struct hapi_bt_host *hapi_bt_host, uint16_t handle,
        uint16_t interval_min, uint16_t interval_max,
        uint16_t latency, uint16_t timeout)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. handle: The connection handle.
3. Interval_min: In 1.25 ms, range: 0x0006 to 0x0C80.
4. Interval_max: In 1.25 ms, range: 0x0006 to 0x0C80.
5. latency: In intervals, range: 0x0000 to 0x01F3.
6. timeout: In ms, range: 0x000A to 0x0C80.

Return: true (0) on success. -1 on error.

7.1.3.16 hapi_bt_host_gap_add_device_to_white_list

Used to update the white list with the device.

```
bool hapi_bt_host_gap_add_device_to_white_list(
      struct hapi_bt_host *hapi_bt_host, uint8_t addr_type,
      uint8_t *addr)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. addr_type: The address type: 0=public, 1=random.
3. addr: public or random device address.

Return: true (0) on success. -1 on error.

7.1.3.17 hapi_bt_host_gap_remove_device_from_white_list

Used to remove the device addressed from the white list.

```
bool hapi_bt_host_gap_remove_device_from_white_list(
        struct hapi_bt_host *hapi_bt_host,
        uint8_t addr_type, uint8_t *addr)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. addr_type: The address type: 0=public, 1=random.
3. addr: public or random device address.

Return: true (0) on success. -1 on error.

7.1.3.18 hapi_bt_host_gap_clear_white_list

Used to clear the white list.

```
bool hapi_bt_host_gap_clear_white_list(
              struct hapi_bt_host *hapi_bt_host)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.

Return: true (0) on success. -1 on error.

7.1.3.19 hapi_bt_host_gap_add_device_to_resolving_list

Used to update the resolving list with the device.

```
bool hapi_bt_host_gap_add_device_to_resolving_list(
        struct hapi_bt_host *hapi_bt_host, uint8_t addr_type,
        uint8_t *addr, uint8_t *peer_irk, uint8_t *local_irk)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. addr_type: The address type: 0=public, 1=random.
3. addr: public or random device address.
4. peer_irk: IRK of the peer device.
5. local_irk: IRK of the local device.

Return: true (0) on success. -1 on error.

7.1.3.20 hapi_bt_host_gap_remove_device_from_resolving_list

Used to remove the device from the resolving list.

```
bool hapi_bt_host_gap_remove_device_from_resolving_list(
        struct hapi_bt_host *hapi_bt_host, uint8_t addr_type,
        uint8_t *addr)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. addr_type: The address type: 0=public, 1=random.
3. addr: public or random device address.

Return: true (0) on success. -1 on error.

7.1.3.21 hapi_bt_host_gap_clear_resolving_list

Used to update the white list with the device.

```
bool hapi_bt_host_gap_clear_resolving_list(
        struct hapi_bt_host *hapi_bt_host
)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.

Return: true (0) on success. -1 on error.

7.1.3.22 bt_host_gap_set_address_resolution_enable

Used to enable/disable the address resolution of the device addressed.

```
bool hapi_bt_host_gap_set_address_resolution_enable(
        struct hapi_bt_host *hapi_bt_host, uint16_t timeout,
        uint8_t)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. timeout: The Resolvable private address timeout in s (default: 900s).
3. enable:Enable: 1=True, 0=False (default: 0).

Return: true (0) on success. -1 on error.

7.1.3.23 hapi_bt_host_common_server_create

Used create the common server functionality when it configured as a BLE peripheral.

```
bool hapi_bt_host_common_server_create(struct hapi_bt_host
          *hapi_bt_host, char *name, uint16_t appearance,
          char *manufacture_name)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. name: Name of the server.
3. appearance: Appearance of the server.
4. manufacture_name: Server manufacturer name.

Return: true (0) on success. -1 on error.

7.1.3.24 hapi_bt_host_gatt_add_service

Used to add a BLE service when configured as a server.

```
bool hapi_bt_host_gatt_add_service(struct hapi_bt_host
            *hapi_bt_host, uint32_t handle)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. handle: The handle of the service.

Return: true (0) on success. -1 on error.

7.1.3.25 hapi_bt_host_gatt_destroy_service

Used to destroy an added BLE service.

```
bool hapi_bt_host_gatt_destroy_service(
        struct hapi_bt_host *hapi_bt_host, uint32_t handle)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. handle: The handle of the service.

Return: true (0) on success. -1 on error.

7.1.3.26 hapi_bt_host_comon_server_destroy

Used to destroy the common BLE server created.

```
bool hapi_bt_host_comon_server_destroy(
                        struct hapi_bt_host *hapi_bt_host)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.

Return: true (0) on success. -1 on error.

7.1.3.27 hapi_bt_host_gatt_exchange_mtu

Used to exchange the BLE MTU size when it tries to connect to a peripheral device.

```
bool hapi_bt_host_gatt_exchange_mtu(
        struct hapi_bt_host *hapi_bt_host, uint16_t size)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. size: Client RX MTU size (23 - 251) (default: 23).

Return: true (0) on success. -1 on error.

7.1.3.28 hapi_bt_host_gatt_create_service_128

Used to create a BLE service (128-bit UUID) when it acts as a peripheral with a GATT server.

```
void* hapi_bt_host_gatt_create_service_128(
     struct hapi_bt_host *hapi_bt_host, uint8_t *uuid)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. uuid: The uuid of service.

Return: Handle of newly created service or NULL pointer if it failed.

7.1.3.29 bt_host_gatt_create_service_16

Used to create a BLE service (16-bit) when it acts as a peripheral with a GATT server.

```
void* hapi_bt_host_gatt_create_service_16(
       struct hapi_bt_host *hapi_bt_host, uint16_t uuid16)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. uuid16: The uuid of service.

Return: Handle of newly created service or NULL pointer if it failed.

7.1.3.30 hapi_bt_host_gatt_notification

Used to create a BLE GATT notification.

```
bool hapi_bt_host_gatt_notification(
        struct hapi_bt_host *hapi_bt_host, uint8_t value)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. value: The value in notification.

Return: true (0) on success. -1 on error.

7.1.3.31 hapi_bt_host_gatt_indication

Used to create a BLE GATT notification.

```
bool hapi_bt_host_gatt_indication(
        struct hapi_bt_host *hapi_bt_host, uint8_t value)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. value: The value in indication.

Return: true (0) on success. -1 on error.

7.1.3.32 hapi_bt_host_gatt_write_characteristic_descriptor

Used to write the BLE characteristics value.

```
bool hapi_bt_host_gatt_write_characteristic_descriptor(
        struct hapi_bt_host *hapi_bt_host, uint16_t handle,
        uint32_t len, uint8_t *value)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. handle: The handle for the characteristic descriptor.
3. length: The length of value to write.
4. value: The value to write.

Return: true (0) on success. -1 on error.

7.1.3.33 hapi_bt_host_gatt_discover_all_primary_services

Used to discover all the supported BLE primary services.

```
bool hapi_bt_host_gatt_discover_all_primary_services(
        struct hapi_bt_host *hapi_bt_host)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.

Return: true (0) on success. -1 on error.

7.1.3.34 hapi_bt_host_gatt_discover_all_characteristic_descriptors

Used to discover all BLE characteristics descriptors of a service.

```
bool hapi_bt_host_gatt_discover_all_characteristic_descriptors(
        struct hapi_bt_host *hapi_bt_host, uint16_t start_handle,
        uint16_t end_handle)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. start_handle: The starting handle of the specified service.
3. end_handle: The ending handle of the specified service.

Return: true (0) on success. -1 on error.

7.1.3.35 hapi_bt_host_gatt_discover_all_characteristics_of_a_service

Used to discover all BLE characteristics of a service.

```
bool hapi_bt_host_gatt_discover_all_characteristic_descriptors(
        struct hapi_bt_host *hapi_bt_host, uint16_t start_handle,
        uint16_t end_handle)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. start_handle: The starting handle of the specified service.
3. end_handle: The ending handle of the specified service.

Return: true (0) on success. -1 on error

7.1.3.36 hapi_bt_host_gatt_discover_characteristics_by_uuid

Used to discover BLE characteristics by a specified UUID.

```
bool hapi_bt_host_gatt_discover_characteristics_by_uuid(
        struct hapi_bt_host *hapi_bt_host, uint16_t start_handle,
        uint16_t end_handle, uint16_t size, uint8_t *uuid)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. start_handle: Starting handle of the specified service.
3. end_handle: Ending handle of the specified service.
4. size: The UUID size in bytes, 2-uuid16, 16-uuid128.
5. uuid: The UUID - 16 or 128 bits.

Return: true (0) on success. -1 on error

7.1.3.37 hapi_bt_host_gatt_discover_primary_service_by_service_uuid

Used to discover the primary service supported with the specified UUID.

```
bool hapi_bt_host_gatt_discover_primary_service_by_service_uuid(
        struct hapi_bt_host *hapi_bt_host, uint16_t size
        uint8_t *uuid )
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. size: Uuid size in bytes, 2-uuid16, 16-uuid128.
3. uuid: The uuid - 16 or 128 bits.

Return: true (0) on success. -1 on error.

7.1.3.38 hapi_bt_host_gatt_read_characteristic_value

Used to read the characteristics value using a handle.

```
bool hapi_bt_host_gatt_read_characteristic_value(
       struct hapi_bt_host *hapi_bt_host, uint16_t value_handle)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. value_handle: The value_handle to be read from remote server.

Return: true (0) on success. -1 on error.

7.1.3.39 hapi_bt_host_gatt_read_using_characteristic_uuid

Used to read the characteristics value using a specified UUID.

```
bool hapi_bt_host_gatt_read_using_characteristic_uuid(
        struct hapi_bt_host *hapi_bt_host, uint16_t start_handle,
        uint16_t end_handle, uint16_t size, uint8_t *uuid )
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. start_handle: The starting handle of the service handle range.
3. end_handle: The ending handle of the service handle range.
4. size: The uuid size in bytes, 2-uuid16, 16-uuid128.
5. uuid: The uuid - 16 or 128 bits.

Return: true (0) on success. -1 on error.

7.1.3.40 hapi_bt_host_gatt_read_long_characteristic_value

Used to read the characteristics value using a service handle from an offset.

```
bool hapi_bt_host_gatt_read_long_characteristic_value(
        struct hapi_bt_host *hapi_bt_host, uint16_t value_handle,
        uint16_t value_offset)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. value_handle: The value_handle to be read from remote server.
3. value_offset: The value_offset to be read.

Return: true (0) on success. -1 on error.

7.1.3.41 hapi_bt_host_gatt_read_multiple_characteristic_values

Used to read multiple characteristics value using service handle.

```
bool hapi_bt_host_gatt_read_multiple_characteristic_values(
        struct hapi_bt_host *hapi_bt_host, uint16_t nof_handles,
        uint8_t *handles)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. nof_handle: The number of handles to be read.
3. handles: The handles to be read (two bytes per handle (lsb,msb)).

Return: true (0) on success. -1 on error.

7.1.3.42 hapi_bt_host_gatt_read_characteristic_descriptor

Used to read multiple characteristics descriptor using handle.

```
bool hapi_bt_host_gatt_read_characteristic_descriptor(
        struct hapi_bt_host *hapi_bt_host, uint16_t handle )
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. handle: The handle of the characteristics descriptor to read.

Return: true (0) on success. -1 on error.

7.1.3.43 hapi_bt_host_gatt_write_without_response

Used to write the characteristics value using a handle. This API will not generate any response from the remote.

```
bool hapi_bt_host_gatt_write_without_response(
        struct hapi_bt_host *hapi_bt_host, uint16_t value_handle,
        uint8_t *value,int len)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. value_handle: The value_handle to be write on the remote server.
3. value: The value to write.
4. len: The length of the data to be written.

Return: true (0) on success. -1 on error.

7.1.3.44 hapi_bt_host_gatt_write_characteristic_value

Used to write the characteristics value using a handle.

```
bool hapi_bt_host_gatt_write_characteristic_value(
      struct hapi_bt_host *hapi_bt_host, uint16_t value_handle,
      uint8_t *value, int len)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. value_handle: The value_handle to be write on the remote server.
3. value: The value to write.
4. len: The length of the data to be written.

Return: true (0) on success. -1 on error.

7.1.3.45 hapi_bt_host_smp_passkey

Used to set the key for secure BLE connection.

```
bool hapi_bt_host_smp_passkey(
        struct hapi_bt_host *hapi_bt_host, uint32_t key0,
        uint32_t oob1, uint32_t oob2, uint32_t oob3)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. Key0: The 20 bits passkey or OOB0 (bits 0..31).
3. oob1: OOB1 (bits 32..63).
4. oob2: OOB2 (bits 64..95).
5. oob3: OOB3 (bits 96..127).

Return: true (0) on success. -1 on error.

7.1.3.46 hapi_bt_host_gatt_char_rd_data_update

Used to update the data for read operation.

```
bool hapi_bt_host_gatt_char_rd_data_update(
        struct hapi_bt_host *hapi_bt_host, uint32_t ctx,
        uint8_t uuid_len, uint8_t *uuid, uint16_t len,
        uint8_t *data)
```

Arguments:

6. hapi_bt_host: BLE HAPI instance pointer.
7. ctx: The context of read.
8. uuid_len: The length of UUID.
9. uuid: The uuid of service.
10. len: The length of data.
11. data: The data to give caller.

Return: true (0) on success. -1 on error.

7.1.3.47 hapi_bt_host_gatt_char_wr_data_update

Used to update that data is written.

```
bool hapi_bt_host_gatt_char_wr_data_update(
        struct hapi_bt_host *hapi_bt_host, uint32_t ctx,
        uint8_t uuid_len, uint8_t *uuid, uint32_t status)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.
2. ctx: The context of write.
3. uuid_len: The length of UUID.
4. uuid: The UUID of service.
5. status: The status of write operation.

Return: true (0) on success. -1 on error.

7.1.3.48 hapi_bt_host_gatt_add_chr_16

Used to add a characteristic for a created BLE service.

```
Bool hapi_bt_host_gatt_add_chr_16(
     struct hapi_bt_host *hapi_bt_host, uint32_t handle,
     uint16_t uuid16, uint8_t permission, uint8_t property)
```

Arguments:

1. hapi_ble: BLE HAPI instance pointer.
2. handle: The handle of service.
3. uuid16: The UUID of service.
4. Permission: The Permission of service.
5. Property: The Property of service.

Return: true (0) on success. -1 on error.

7.1.4 Power Save APIs

7.1.4.1 hapi_send_sleep

Requests to enable sleep in Talaria TWO™.

```
void hapi_send_sleep(struct hapi *hapi)
```

Arguments:

1. hapi: HAPI instance pointer.

Return: None.

7.1.5 Socket APIs

7.1.5.1 socket_create

Creates a socket according to the parameter passed.

```
int socket_create(struct hapi *hapi, int proto, char *server, char
*port)
```

Arguments:

1. hapi: HAPI instance pointer
2. proto: specifies the protocol used for the socket to create. The valid combinations are TCP client, UDP client, TCP server and UDP server
3. server: The server URL for the TCP or UDP client connection
4. port: the port number to connect. If the proto is TCP/UDP server this is the port on which the Talaria TWO™ waits for connection

Return: integer value > = 0 on success or -1 on failure.

7.1.5.2 socket_send

Used to send data on a socket.

```
bool socket_send (struct hapi *hapi, uint32_t socket, const void
*data, size_t len)
```

Arguments:

1. hapi: HAPI instance pointer
2. socket: The socket ID which has been created
3. data: The data to be sent on the socket
4. len: The length of the data to be sent

Return: TRUE (0) on success and FALSE (-1) on failure.

7.1.5.3 socket_receive

Used to receive data from a socket.

```
size_t socket_receive(struct hapi *hapi, uint32_t socket, void
*data, size_t len)
```

Arguments:

1. hapi: HAPI instance pointer.
2. socket: The socket ID which has been created.
3. data: The data pointer on which the data is to be received from the socket.
4. len: The length of the data to be received.

Return: the length of the actual data received.

7.1.5.4 socket_getavailable

Used to check received data available on a socket.

```
int socket_getavailable(struct hapi *hapi, uint32_t socket)
```

Arguments:

1. hapi: HAPI instance pointer.
2. socket: The socket ID which has been created.

Return: The length of the data available on the socket which can be read.

7.1.5.5 socket_close

Used to close a socket which has been opened.

```
void socket_close(struct hapi *hapi, uint32_t socket)
```

Arguments:

1. hapi: HAPI instance pointer.
2. socket: The socket ID which has been created.

Return: none.

7.1.6 MDNS APIs

7.1.6.1 hapi_setup_mdns

Used to setup the MDNS service.

```
struct hapi_mdns*

hapi_setup_mdns(struct hapi *hapi, struct hapi_wcm *hapi_wcm,

                                const char *host_name)
```

Arguments:

1. hapi: HAPI instance pointer.
2. host_name: The hostname to be used for the MDNS service.

Return: On success hapi_mdns pointer, else NULL

7.1.6.2 hapi_mdns_set_ind_cb

Used to set MDNS notification callback function. This callback is getting called when there is a notification from MDNS service.

```
Void hapi_mdns_set_ind_cb(struct hapi_mdns *hapi_mdns,

                              hapi_mdns_ind_cb cb,void *context)
```

Arguments:

1. hapi: HAPI instance pointer.
2. cb: The callback function to be set.
3. context: The context pointer to be passed along when the callback is getting called.

Return: none

7.1.6.3 hapi_add_mdns_service

Used to add a MDNS service so that the MDNS operation get started.

```
bool hapi_add_mdns_service(struct hapi *hapi, *hapi_wcm,const char
                *host_name, const char *type,uint32_t proto,uint32_t
                 port, char *description, uint32_t *serviceId)
```

Arguments:

1. hapi: HAPI instance pointer.
2. hapi_wcm: HAPI WCM pointer.
3. host_name: The MDNS host name.
4. type: The host type.
5. proto: The protocol type.
6. port: The port number.
7. description: Description about the service.
8. serviceid: The MDNS service identifier of the service getting added.

Return: TRUE (0) on success and FALSE (-1) on failure.

7.1.6.4 hapi_remove_mdns_service

Used to remove a MDNS service being added.

```
bool hapi_remove_mdns_service(struct hapi *hapi, struct hapi_wcm
                              *hapi_wcm, uint32_t service_id)
```

Arguments:

1. hapi: HAPI instance pointer.
2. hapi_wcm: HAPI WCM pointer.
3. serviceid: The MDNS service identifier, being added with hapi_add_mdns_service API.

Return: TRUE (0) on success and FALSE (-1) on failure.

7.1.6.5 hapi_stop_mdns

Used to stop the MDNS service.

```
bool hapi_stop_mdns(struct hapi *hapi, struct hapi_wcm *hapi_wcm)
```

Arguments:

1. hapi: HAPI instance pointer.
2. hapi_wcm: Hapi wcm pointer.

Return: TRUE (0) on success and FALSE (-1) on failure.

7.1.6.6 hapi_resolve_mdns

Used to resolve the MDNS host name to get the IP address.

```
bool hapi_resolve_mdns(struct hapi *hapi, const char *host_name,
            uint8_t addrtype, uint8_t *ipaddr, uint16_t* addrlen)
```

Arguments:

1. hapi: HAPI instance pointer.
2. host_name: The MDNS host name.
3. addrtype: The address type.
4. ipaddr: The pointer that will contain the IP address to be filled.
5. addrlen: The length of the IP address to be resolved.

Return: TRUE (0) on success and FALSE (-1) on failure

7.1.7 HTTP Client APIs

7.1.7.1 hapi_http_client_setup

Used to setup the HTTP client service.

```
Void hapi_http_client_setup(struct hapi *hapi_p,
                hapi_http_client_resp_cb cb, void *cb_ctx)
```

Arguments:

1. hapi_p: HAPI instance pointer.
2. cb: The http callback function pointer.
3. cb_ctx: The callback context.

Return: None.

7.1.7.2 hapi_http_client_start

Used to start the HTTP client connection.

```
Bool hapi_http_client_start(struct hapi *hapi_p, char* serverName,
          uint32_t port, char* certName, uint32_t* clientID)
```

Arguments:

1. hapi_p: HAPI instance pointer.
2. serverName: The server domain name or IP address.
3. port: The port number of the http server.
4. certName: The SSL certificate name.
5. clientID: Pointer to integer used for returning client ID.

Return: TRUE (0) on success and FALSE (-1) on failure

7.1.7.3 hapi_http_client_send_req

Used to send HTTP request to the server. The HTTP server connection should exist to succeed this API.

```
bool hapi_http_client_send_req(struct hapi *hapi_p,
            uint32_t clientID, uint32_t method, char* req_uri,
            uint32_t dataLen, char* dataToSend)
```

Arguments:

1. hapi_p: HAPI instance pointer
2. clientID: The valid client id created with the HTTP connection.
3. method: The GET(1) and POST(0) methods.
4. Req_uri: The URI to request.
5. dataLen: The length of the data to request.
6. dataToSend: Pointer to the data.

Return: TRUE (0) on success and FALSE (-1) on failure

7.1.7.4 hapi_http_client_hdr_set

Used to set HTTP request header.

```
Bool hapi_http_client_hdr_set(struct hapi *hapi_p,
                uint32_t headerID, char* headerVal)
```

Arguments:

1. hapi_p: HAPI instance pointer.
2. headerID: The header id as per the httphdrtype definition.

   The httphdrtype is defined as:

   typedef enum {

   STW_HTTP_HDR_INVAL, /* special value for invalid header */

   STW_HTTP_HDR_ALLOW,

   STW_HTTP_HDR_AUTHORIZATION,

   STW_HTTP_HDR_CONNECTION,

   STW_HTTP_HDR_CONTENT_ENCODING,

   STW_HTTP_HDR_CONTENT_LENGTH,

   STW_HTTP_HDR_CONTENT_RANGE,

   STW_HTTP_HDR_CONTENT_TYPE,

   STW_HTTP_HDR_COOKIE,

   STW_HTTP_HDR_COOKIE2,

   STW_HTTP_HDR_DATE,

   STW_HTTP_HDR_EXPIRES,

   STW_HTTP_HDR_FROM,

   STW_HTTP_HDR_HOST,

   STW_HTTP_HDR_IF_MODIFIED_SINCE,

   STW_HTTP_HDR_LAST_MODIFIED,

   STW_HTTP_HDR_LOCATION,

   STW_HTTP_HDR_PRAGMA,


```
    STW_HTTP_HDR_RANGE,
    STW_HTTP_HDR_REFERER,
    STW_HTTP_HDR_SERVER,
    STW_HTTP_HDR_SET_COOKIE,
    STW_HTTP_HDR_TRANSFER_ENCODING,
    STW_HTTP_HDR_USER_AGENT,
    STW_HTTP_HDR_WWW_AUTHENTICATE,
    STW_HTTP_HDR_COUNT,
    STW_HTTP_HDR_CUSTOM      /* Value indicating the start of custom headers
*/
} httphdrtype;
```

3. headerVal: The header value to set.

Return: TRUE (0) on success and FALSE (-1) on failure.

7.1.7.5 hapi_http_client_hdr_delete

Used to delete HTTP request header.

```
Bool hapi_http_client_hdr_delete(struct hapi *hapi_p,
                                 uint32_t headerID)
```

Arguments:

1. hapi_p: HAPI instance pointer
2. headerID: The header ID as per the httphdrtype definition.

Return: TRUE (0) on success and FALSE (-1) on failure

7.1.7.6 hapi_http_cert_store

Used to store SSL/TLS certificate for HTTPS connection.

```
Bool hapi_http_cert_store(struct hapi *hapi_p, char* certName,
                     uint32_t certLen, char* certData)
```

Arguments:

1. hapi_p: HAPI instance pointer.
2. certName: The certificate name.
3. certData: The certificate content data pointer.

Return: TRUE (0) on success and FALSE (-1) on failure

7.1.7.7 hapi_http_cert_delete

Used to delete SSL/TLS certificate for HTTPS.

```
Bool hapi_http_cert_delete(struct hapi *hapi_p, char* certName)
```

Arguments:

1. hapi_p: HAPI instance pointer.
2. certName: The certificate name to delete.

Return: TRUE (0) on success and FALSE (-1) on failure

7.1.7.8 hapi_http_close

Used to close the HTTP connection opened.

```
Bool hapi_http_close(struct hapi *hapi_p, uint32_t clientId)
```

Arguments:

1. hapi_p: HAPI instance pointer.
2. clientID: The valid client id created with the http connection.

Return: TRUE (0) on success and FALSE (-1) on failure

7.1.8 MQTT APIs

7.1.8.1 hapi_mqtt_nw_init

Used to initialize the MQTT network. This is the first API to be called to use MQTT protocol.

```
struct hapi_mqtt* hapi_mqtt_nw_init(struct hapi *hapi,
        char* serverName, uint16_t port, char* certName,
        uint16_t *sockId, uint32_t *status)
```

Arguments:

1. hapi: HAPI instance pointer.
2. serverName: The MQTT server (Broker) name.
3. port: The MQTT port number.
4. certName: The certificate name in case of MQTT with TLS.

Return: TRUE (0) on success and FALSE (-1) on failure

7.1.8.2 hapi_mqtt_set_ind_cb

Used to set the MQTT notification callback.

```
Void hapi_mqtt_set_ind_cb(struct hapi_mqtt *hapi_mqtt,
             hapi_mqtt_ind_cb cb, void *context)
```

Arguments:

1. hapi_mqtt: MQTT instance pointer.
2. cb: The callback function.
3. context: The context pointer pass along with the callback.

Return: TRUE (0) on success and FALSE (-1) on failure

7.1.8.3 hapi_mqtt_nw_connect

Used to connect to the MQTT network.

```
bool hapi_mqtt_nw_connect(struct hapi *hapi,
                          struct hapi_mqtt *hapi_mqtt,
                          char* mqtt_server_name, uint16_t mqtt_port)
```

Arguments:

1. hapi: HAPI instance pointer.
2. hapi_mqtt: The MQTT instance pointer.
3. mqtt_server_name: The server's name or IP address of the MQTT broker.
4. mqtt_port: The MQTT port number to connect.

Return: TRUE (0) on success and FALSE (-1) on failure.

7.1.8.4 hapi_mqtt_client_init

Used to initialize the MQTT client.

```
bool hapi_mqtt_client_init(struct hapi *hapi,
                                      struct hapi_mqtt *hapi_mqtt,
                                      uint16_t timeout_ms)
```

Arguments:

1. hapi: HAPI instance pointer.
2. hapi_mqtt: The MQTT instance pointer
3. timeout_ms: The connection timeout in milli-seconds.

Return: TRUE (0) on success and FALSE (-1) on failure.

7.1.8.5 hapi_mqtt_connect

Used to connect the MQTT broker with the username and password provided.

```
bool hapi_mqtt_connect(struct hapi *hapi, struct hapi_mqtt
                    *hapi_mqtt, uint32_t mqtt_version,
                    char* clientId, char* userName, char* passWord)
```

Arguments:

1. hapi: HAPI instance pointer.
2. hapi_mqtt: The MQTT instance pointer.
3. mqtt_version: The current supported MQTT version.
4. clientId: The ID of the client, trying to get connected to.
5. userName: The username for the MQTT connection.
6. password: The password for the MQTT connection.

Return: TRUE (0) on success and FALSE (-1) on failure.

7.1.8.6 hapi_mqtt_publish

Used to publish data to the broker in the existing MQTT connection.

```
bool hapi_mqtt_publish(struct hapi *hapi, struct hapi_mqtt
                *hapi_mqtt, char* topic_to_publish, char* topic)
```

Arguments:

1. hapi: HAPI instance pointer.
2. hapi_mqtt: The MQTT instance pointer.
3. topic_to_publish: Topic of the MQTT to publish.
4. topic: The data to publish.

Return: TRUE (0) on success and FALSE (-1) on failure.

7.1.8.7 hapi_mqtt_subscribe

Used to subscribe to a particular topic.

```
bool hapi_mqtt_subscribe(struct hapi *hapi,
                  struct hapi_mqtt *hapi_mqtt,
                  char* topic_to_sub, uint16_t qos)
```

Arguments:

1. hapi: HAPI instance pointer.
2. hapi_mqtt: The MQTT instance pointer.
3. topic_to_sub: Topic of the MQTT to subscribe.
4. qos: The qos of the MQTT connection.

Return: TRUE (0) on success and FALSE (-1) on failure.

7.1.8.8 hapi_mqtt_unsubscribe

Used to resolve the MDNS host name to get the IP address.

```
bool hapi_mqtt_unsubscribe(struct hapi *hapi,
                     struct hapi_mqtt *hapi_mqtt, char* topic)
```

Arguments:

1. hapi: HAPI instance pointer.
2. hapi_mqtt: The MQTT instance pointer.
3. topic: Topic of the MQTT to un-subscribe.

Return: TRUE (0) on success and FALSE (-1) on failure.

7.1.9 TLS APIs

7.1.9.1 hapi_tls_create

Create the TLS socket and do the handshake to support the TLS functionality.

```
struct hapi_tls * hapi_tls_create(struct hapi *hapi,const char
*server, const char *port, uint16_t maxfraglen, uint16_t cacertlen,
uint16_t owncertlen, uint16_t pkeylen)
```

Arguments:

1. hapi: HAPI instance pointer.
2. server: Server URI string.
3. port: Server port.
4. maxfraglen: Max fragmentation size.
5. cacertlen: The CA certificate length.
6. owncertlen: Own certificate length.
7. pkeylen: The key length.

Return: TLS HAPI instance pointer on success, NULL on failure.

7.1.9.2 hapi_tls_set_dataready_cb

Registers the callback function when the TLS data available.

```
void hapi_tls_set_dataready_cb(struct hapi_tls *hapi_tls,
hapi_tls_dataready_cb dataready_cb, void *context)
```

Arguments:

1. hapi_tls: HAPI TLS instance pointer.
2. dataready_cb: Call back function.
3. context: The context to pass when the callback getting called.

Return: None.

7.1.9.3 hapi_tls_upload_cert

To store the certificate passed.

```
Bool hapi_tls_upload_cert(struct hapi_tls *hapi_tls, enum
hapi_tls_cert_type cert_type, const char * cert, size_t cert_size)
```

Arguments:

1. hapi_tls: HAPI TLS instance pointer.
2. hapi_tls_cert_type cert_type: Type of the certificate to load.
3. cert: The certificate start pointer.
4. cert_size: The size of the certificate in bytes.

Return: Bool, true on success, false on failure.

7.1.9.4 hapi_tls_handshake

Triggers the TLS handshake operation.

```
Bool hapi_tls_handshake(struct hapi_tls *hapi_tls, enum
hapi_tls_auth_mode auth_mode)
```

Arguments:

1. hapi_tls: HAPI TLS instance pointer.
2. auth_mode: The authentication mode supported.

Return: Bool, true on success, false on failure.

7.1.9.5 hapi_tls_write

Sends the data on the TLS connection.

```
ssize_t hapi_tls_write(struct hapi_tls *hapi_tls, const void *
data, size_t size)
```

Arguments:

1. hapi_tls: HAPI TLS instance pointer.
2. data: Data to be sent.
3. Size: Size of the data in bytes to be sent.

Return: the number of bytes sent, on success, 0 on failure.

7.1.9.6 hapi_tls_read

To read data from the TLS socket.

```
ssize_t hapi_tls_read(struct hapi_tls *hapi_tls, void * buf,
size_t size)
```

Arguments:

1. hapi_tls: HAPI TLS instance pointer.
2. buf: Data buffer to which the reception happens.
3. Size: Size of the data in bytes TPO read.

Return: the number of bytes received, on success, 0 on failure.

7.1.9.7 hapi_tls_close

To close the TLS socket and release all the resources allocated.

```
Bool hapi_tls_close(struct hapi_tls *hapi_tls)
```

Arguments:

1. hapi_tls: HAPI TLS instance pointer.

Return: true on success, false on failure.

7.1.10 Common APIs

7.1.10.1 hapi_init

Initializes and starts HAPI on the host micro controller. This should be called before accessing any HAPI API.

```
void* hapi_init(void *console_uart_ptr,  void* hapi_uart_ptr, int
hapi_baudrate, void* hapi_spi_ptr)
```

Arguments:

1. console_uart_ptr : Debug UART pointer.
2. hapi_uart_ptr : HAPI interface UART pointer.
3. hapi_baudrate: UART baudrate.
4. hapi_spi_ptr: HAPI interface SPI pointer.

Return: HAPI instance pointer on success, NULL on failure.

7.1.10.2 hapi_get_error_code

Returns the currently set error code in HAPI layer.

```
int hapi_get_error_code(struct hapi *hapi)
```

Arguments:

1. hapi: HAPI instance pointer.

Return: integer value corresponding to the error code.

7.1.10.3 hapi_get_error_message

Returns the currently set error message in HAPI layer.

```
const char*hapi_get_error_message(struct hapi *hapi)
```

Arguments:

1. hapi: HAPI instance pointer.

Return: error message in string format corresponding to the error code.

7.1.10.4 set_hapi_scarmbling_mode

Sets the scrambling enable/disable in serial communication.

```
void hapi_set_hio_scrambling(struct hapi *hapi, int enable,
        void* scrambling_ctx, void* key, scrambling_fn
        scrambling_fn, descrambling_fn descrambling_fn);
```

Arguments:

1. hapi: HAPI instance pointer.
2. enable: 1 to enable the pass or 0 to disable.
3. scrambling_ctx: Context pointer passed along with scrambling/descrambling callback function.
4. key: Scrambling/descrambling key.
5. scrambling _fn: Scrambling callback function.
6. descrambling _fn: De-scrambling callback function.

Return: None.

7.1.10.5 hapi_add_ind_handler

Request to add an indication handler for a message in a group.

```
struct hapi_ind_handler * hapi_add_ind_handler(
        struct hapi *hapi,
        uint8_t group_id,
        uint8_t msg_id,
        hapi_ind_callback ind_cb,
        void * context);
```

Arguments:

1. hapi: HAPI instance pointer.
2. group_id: The group id to which it the handler registered.
3. msg_id: The message id to which it the handler registered.
4. ind_cb: The callback function to be called.
5. context: The context to be passed when the call back is getting called.

Return: The valid pointer on success or -1 on failure.

7.1.10.6 hapi_config

To configure the HAPI interface.

```
void hapi_config(struct hapi *hapi, bool suspend_enable, uint8_t
wakeup_pin, uint8_t wakeup_level, uint8_t irq_pin, uint8_t
irq_mode)
```

Arguments:

1. hapi: HAPI instance pointer.
2. suspend_enable: suspend enabled or not.
3. wakeup_pin: The pin used to wake up from suspend.
4. wakeup_level: The level of the wake pin state.
5. irq_pin: The interrupt request pin.
6. irq_mode: The IRQ mode to be configured.

Return: None.

7.1.10.7 set_hapi_default_interface

This API sets the default interface as specified.

```
void set_hapi_default_interface(struct hapi *hapi, int
def_interface)
```

Arguments:

1. hapi: HAPI instance pointer.
2. def_interface: Default interface index, 0->uart,1->spi.

Return: None.

7.1.10.8 hapi_hio_query

Checks if Talaria TWO™ is ready to accept the HIO commands from the host.

```
hapi_hio_query(struct hapi *hapi)
```

Arguments:

1. hapi: HAPI instance pointer.

Return: None.

7.1.10.9 hapi_hio_get_time

To get the current time that can be used for any time synced applications.

```
bool hapi_hio_get_time(struct hapi *hapi, uint64_t *time_now)
```

Arguments:

1. hapi: HAPI instance pointer.
2. time_now: Pointer which contain the current time.

Return: True on success, False on failure.

7.1.10.10 hapi_nw_misc_app_time_get

To get the network time that can be used for any time synced applications.

```
bool hapi_nw_misc_app_time_get(struct hapi *hapi, uint64_t
*current_time)
```

Arguments:

1. hapi: HAPI instance pointer.
2. current_time: Pointer which contain the current network time.

Return: True on success, False on failure.

7.1.10.11 hapi_get_dbg_info

To get more debug information from Talaria TWO™.

```
bool hapi_get_dbg_info(struct hapi *hapi, struct

hapi_demo_dbg_info_get_rsp *dbg_info)
```

Arguments:

1. hapi: HAPI instance pointer.
2. dbg_info: Debug information received from Talaria TWO™ to be copied here.

Return: True on success, False on failure.

8 Support

1. Sales Support: Contact an InnoPhase sales representative via email – sales@innophaseinc.com
2. Technical Support:
    a. Visit: https://innophaseinc.com/contact/
    b. Also Visit: https://innophaseinc.com/talaria-two-modules
    c. Contact: support@innophaseinc.com

InnoPhase is working diligently to provide outstanding support to all customers.

9 Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, InnoPhase Incorporated does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and assumes no liability associated with the use of such information. InnoPhase Incorporated takes no responsibility for the content in this document if provided by an information source outside of InnoPhase Incorporated.

InnoPhase Incorporated disclaims liability for any indirect, incidental, punitive, special or consequential damages associated with the use of this document, applications and any products associated with information in this document, whether or not such damages are based on tort (including negligence), warranty, including warranty of merchantability, warranty of fitness for a particular purpose, breach of contract or any other legal theory. Further, InnoPhase Incorporated accepts no liability and makes no warranty, express or implied, for any assistance given with respect to any applications described herein or customer product design, or the application or use by any customer's third-party customer(s).

Notwithstanding any damages that a customer might incur for any reason whatsoever, InnoPhase Incorporated' aggregate and cumulative liability for the products described herein shall be limited in accordance with the Terms and Conditions of identified in the commercial sale documentation for such InnoPhase Incorporated products.

Right to make changes — InnoPhase Incorporated reserves the right to make changes to information published in this document, including, without limitation, changes to any specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — InnoPhase Incorporated products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an InnoPhase Incorporated product can reasonably be expected to result in personal injury, death or severe property or environmental damage. InnoPhase Incorporated and its suppliers accept no liability for inclusion and/or use of InnoPhase Incorporated products in such equipment or applications and such inclusion and/or use is at the customer's own risk.