Talaria TWO™ (INP2045)

Low Power Multi-Protocol Wireless Platform SoC

IEEE 802.11 b/g/n, BLE 5.0

Host API Reference Guide

Release: 08-03-2022

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

Revision History

Version	Date	Comments
1.0	06-29-2020	First version
1.1	09-03-2020	Updated for SDK 2.1.1 release
2.0	05-19-2021	Updated for SDK 2.2 release
2.1	09-06-2021	Updated details for hapi_mqtt_unsubscribe and hapi_resolve_mdns APIs
3.0	12-07-2021	Updated for SDK 2.4alpha release - Updated the following APIs: Port, WLAN, Socket, MQTT and Common
3.1	04-07-2022	Included the following API details: SPI Interface, UART Interface, Power Save and Unassoc. Updated Common APIs list.
3.2	05-24-2022	Updated details for hapi_wcm_network_profile_add_enterprise API.
3.3	07-04-2022	Included additional WLAN, Power Save, Common APIs.
3.4	07-07-2022	Updated hapi_wcm_network_profile_add_ext API.
3.5	07-28-2022	Updated Talaria TWO Host APIs, UART and Power Save APIs.
3.6	08-03-2022	Updated return values for all APIs.

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 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 1.

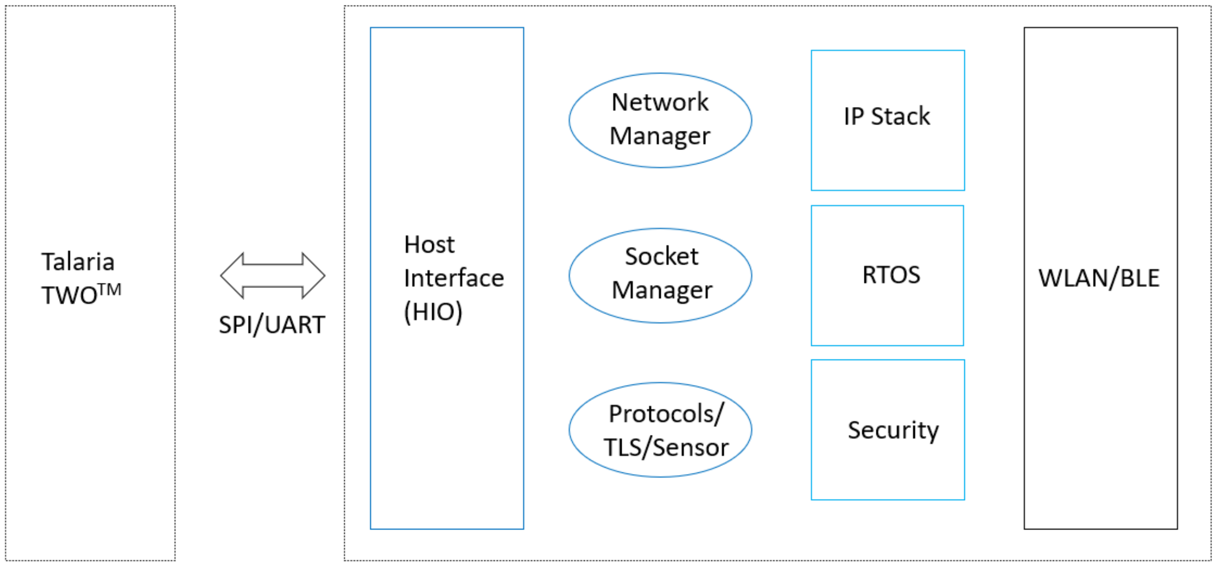

Figure 1: Major components in Talaria TWO

4.1 Wi-Fi Connection Manager

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

4.2 Socket Manager

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

4.3 RTOS

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

4.4 IPSTACK

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

5 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.

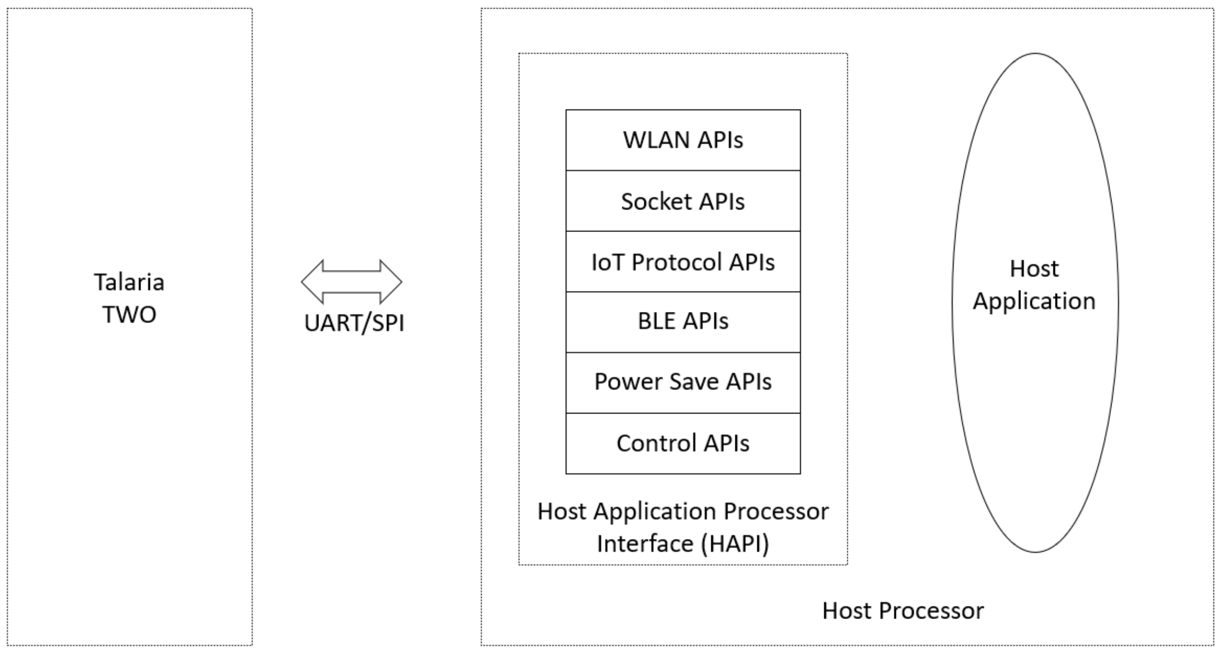

Figure 2: Communication between Host and Talaria TWO via UART/SPI

6 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.

6.1 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
6. SPI Interface APIs
7. UART Interface APIs
8. Unassociation APIs
9. Common 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.

6.1.1 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.

6.1.1.1 hapi_serial_open

Initializes HAPI serial interface. This function initializes the serial device and creates the HAPI interface. This is specific to each platform. This function also registers the platform specific read/write/close APIs to the HAPI interface.

```
struct hapi * hapi_serial_open(const char *devname,int baudrate)
```

Arguments:

1. devname: Pointer to HAPI serial device context.
2. Baudrate: SPI clock speed.

Return: HAPI context.

6.1.1.2 hapi_serial_write

Writes data to Talaria TWO over HAPI interface.

```
ssize_t hapi_serial_write(void *dev, const void *data, size_t

length)
```

Arguments:

1. dev: Pointer to interface device.
2. data: Source buffer address.
3. length: Number of bytes to be written.

Return: number of bytes written on Success else Error.

6.1.1.3 hapi_serial_read

Reads data from Talaria TWO over HAPI interface.

```
ssize_t hapi_serial_read(void *dev, void *data, size_t length)
```

Arguments:

1. dev: Pointer to interface device.
2. data: Source buffer address.
3. length: Number of bytes to be read.

Return: number of bytes read. -1 on Error.

6.1.1.4 hapi_serial_close

Closes HAPI interface.

```
void hapi_serial_close(void* dev)
```

Arguments:

1. dev: Pointer to the interface device.

Return: None.

6.1.1.5 hapi_serial_break

Used to wakeup Talaria TWO. Sends break to Talaria TWO.

```
void hapi_serial_break(void *dev, bool on)
```

Arguments:

1 dev: Pointer to the interface device.

2. on: Send break to Talaria TWO is this set to TRUE.

Return: None.

6.1.2 WLAN APIs

6.1.2.1 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. NULL pointer on Error.

6.1.2.2 hapi_wcm_network_profile_add

Adds a network profile to connect. This API should be called before the HAPI autoconnect API that starts the WLAN connection.

```
bool

hapi_wcm_network_profile_add(struct hapi_wcm *hapi_wcm,

         const char *ssid, const char *bssid,

         const char *passphrase, const char *passphrase_id)
```

Arguments:

1. hapi_wcm: HAPI WLAN instance pointer.
2. ssid : SSID of the network or empty string if BSSID is set.
3. bssid : BSSID of the network, set to all zeroes if SSID is set.
4. passphrase: passphrase for RSN, key for WEP or empty string for unencrypted connection.
5. Passphrase_id : passphrase ID.

Return: Status of add network profile operation. True=Success, False otherwise.

6.1.2.3 hapi_wcm_network_profile_add_ext

Adds a network profile to connect in enterprise mode. This API should be called before the HAPI `autoconnect` API which starts the WLAN connection.

```
bool

hapi_wcm_network_profile_add_ext(struct hapi_wcm *hapi_wcm,struct

wcm_connect_param *wcm_param)
```

Arguments:

1. hapi_wcm: HAPI WLAN instance pointer.
2. wcm_param : Pointer to WCM configuration structure. `wcm_connect_param` consists of the following parameters:
    a. ssid: Pointer to the name of the Access Point string
    b. passphrase: Pointer to the AP passphrase string
    c. security_type: Type of enterprise security, which can have anyone of the following values:

        0: Open
        1: Personal WPA2/3
        2: Enterprise PSK
        3: Enterprise TLS
        4: Enterprise PEAP

    d. eap_identity: Pointer to identity string
    e. eap_ca_path: Pointer to the path of CA certificate in Talaria TWO files system
    f. eap_cert_path: Pointer to the path of client certificate in Talaria TWO file system
    g. eap_pkey_path: Pointer to the path of private key file in Talaria TWO file system
    h. eap_pkey_pwd: Pointer to the password of private key
    i. eap_identity2: Pointer to phase 2 identity
    j. eap_password: Pointer to the password of private key
    k. eap_phase2auth: Pointer to phase 2 authentication

Return: Status of add network profile operation. True=Success, False otherwise.

6.1.2.4 hapi_wcm_network_profile_remove

Removes the network profile that was added.

```
bool

hapi_wcm_network_profile_remove(struct hapi_wcm *hapi_wcm)
```

Arguments:

1. hapi_wcm: HAPI WLAN instance pointer.

Return: Status of remove network profile operation. True=Success, False otherwise.

6.1.2.5 hapi_wcm_autoconnect

Triggers the scan and connects/disconnects to the AP specified by the SSID and uses the passphrase that gets configured using the hapi_wcm_network_profile_add API.

```
bool

hapi_wcm_autoconnect(struct hapi_wcm *hapi_wcm, uint32_t enabled)
```

Arguments:

1. hapi_wcm: HAPI WLAN instance pointer.
2. enabled: flag allow to connect. 1=enabled, 0=disabled

Return: Status of auto connect operation. True=Success, False otherwise.

.

6.1.2.6 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.

6.1.2.7 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: Status of destroy operation. True=Success, False otherwise.

6.1.2.8 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 wlan instance pointer.

Return: a valid pointer points to the HAPI WLAN instance on Success. 0 on Error.

6.1.2.9 hapi_wcm_scan

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

```
Int32_t 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: 1 on Success else Error.

6.1.2.10 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.

6.1.2.11 hapi_wcm_setpmconfig

Used to set the WLAN power save parameters.

```
bool
 hapi_wcm_setpmconfig(struct hapi_wcm *hapi_wcm,
                      uint32_t listen_interval,
                      uint32_t traffic_tmo, uint32_t pm_flags)
```

Arguments:

1. hapi_wcm: HAPI WLAN instance pointer.
2. listen_interval: Listen interval in units of beacon intervals.
3. traffic_tmo: Traffic timeout (in ms)
4. pm_flags: power management flags, specified as follows:
    a. ps_poll: bit 0(0x01)
    b. dynamic_listen_intervel: bit 1(0x02)
    c. sta_rx_nap : bit 2(0x04)
    d. sta_only_broadcast : bit 3(0x08)
    e. tx_ps : bit 4(0x10)
    f. mcast_dont_care: bit 5(0x20)

    multiple options can be selected as logical 'or'-ing of above bits.

Return: Status of set pmconfig operation. True=Success, False otherwise.

6.1.2.12 hapi_wcm_regdomain_set

Used to set the WLAN regulatory domain.

```
bool

hapi_wcm_regdomain_set(struct hapi_wcm *hapi_wcm, char *domain)
```

Arguments:

1. hapi_wcm: HAPI WLAN instance pointer.
2. domain: the regulatory domain name. supported strings are
   a. "FCC",
   b. "ETSI",
   c. "TELEC",
   d. "KCC",
   e. "SRCC"

Return: Status of set regdomain operation. True=Success, False otherwise.

6.1.2.13 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:

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

Return: True(1) on Success. False(0) on Error.

6.1.2.14 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: True(1) on Success. False (0) on Error.

6.1.2.15 hapi_wcm_network_profile_add_new

Adds a network profile in personal or enterprise security mode to connect.

```
bool hapi_wcm_network_profile_add_new(struct hapi_wcm *hapi_wcm,

struct wcm_connect_param *wcm_param)
```

Arguments:

1. hapi_wcm: Pointer to HAPI WCM context.
2. wcm_param: Pointer to connection parameters.

Return: Status of add network profile operation. True=Success, False otherwise.

6.1.2.16 hapi_ wcm_scan_indhandler

Indication callback for scan response from Talaria TWO.

```
void hapi_wcm_scan_indhandler(void *context, struct hapi_packet

*pkt)
```

Arguments:

1. context: Context pointer to be passed when the call back is being called.
2. pkt: Packet to be sent. The packet should be in HAPI packet format.

Return: None.

6.1.2.17 hapi_wcm_autoconnectcfg

Enables/Disables async connect.

```
bool hapi_wcm_autoconnectcfg(struct hapi_wcm *hapi_wcm, int flag)
```

Arguments:

1. hapi_wcm: Pointer to HAPI WCM context.
2. flag: Allows WCM to connect. 1=enabled, 0=disabled.

Return: Status of auto connect operation. True=Success, False otherwise.

6.1.2.18 hapi_wcm_lastind_get

Returns last indication value.

```
int hapi_wcm_lastind_get(struct hapi_wcm *hapi_wcm)
```

Arguments:

1. hapi_wcm: Pointer to HAPI WCM context.

Return: Indication value.

6.1.2.19 hapi_wcm_reinit

Re-initializes WCM interface and returns its pointer. This will be used after host wakeup to initialize the WCM.

```
struct hapi_wcm * hapi_wcm_reinit(struct hapi *hapi,uint32_t

wcm_handle)
```

Arguments:

1. hapi: Pointer to HAPI context.

Return: Newly created WCM interface context.

6.1.2.20 hapi_wcm_set_handle

Sets WCM handle address after host wakeup.

```
void

hapi_wcm_set_handle(struct hapi_wcm *hapi_wcm, uint32_t

wcm_handle)
```

Arguments:

1.hapi_wcm: Pointer to HAPI WCM context.

2.wcm_handle: WCM handle address.

Return: None.

6.1.2.21 hapi_wcm_getpmconfig

Gets WLAN power save parameters.

```
bool
 hapi_wcm_getpmconfig(struct hapi_wcm *hapi_wcm,
                      uint32_t listen_interval,
                      uint32_t traffic_tmo, uint32_t pm_flags)
```

Arguments:

1. hapi_wcm: HAPI WLAN instance pointer.
2. listen_interval: Listen interval in units of beacon intervals.
3. traffic_tmo: Traffic timeout (in ms)
4. pm_flags: Power management flags, specified as follows:
    a. ps_poll: bit 0(0x01)
    b. dynamic_listen_intervel: bit 1(0x02)
    c. sta_rx_nap : bit 2(0x04)
    d. sta_only_broadcast : bit 3(0x08)
    e. tx_ps : bit 4(0x10)
    f. mcast_dont_care: bit 5(0x20)

    Multiple options can be selected as logical ‘or’-ing of above bits.

Return: Status of `getpmconfig` operation. True=Success, False otherwise.

6.1.3 BLE APIs

6.1.3.1 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. NULL on Error.

6.1.3.2 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 (1) on Success.   False (0) on Error.

6.1.3.3 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 (1) on Success.   False (0) on Error.

6.1.3.4 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 (1) on Success.   False (0) on Error.

6.1.3.5 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 (1) on Success. False(0) on Error.

6.1.3.6 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 (1) on Success. False (0) on Failure.

6.1.3.7 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 (1) on Success. False(0) on Failure.

6.1.3.8 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 (1) on Success. False(0) on Failure.

6.1.3.9 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 (1) on Success. False(0) on Failure.

6.1.3.10 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 (1) on Success. False(0) on Failure.

6.1.3.11 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 (1) on Success. False(0) on Failure.

6.1.3.12 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 (1) on Success. False(0) on Failure.

6.1.3.13 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 (1) on Success. False(0) on Failure.

6.1.3.14 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 (1) on Success. False(0) on Failure.

6.1.3.15 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 (1) on Success. False(0) on Failure.

6.1.3.16 hapi_bt_host_gap_add_device_to_white_list

Used to update the device in white list.

```
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 (1) on Success. False(0) on Failure.

6.1.3.17 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 (1) on Success. False(0) on Failure.

6.1.3.18 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 (1) on Success. False(0) on Failure.

6.1.3.19 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 (1) on Success. False(0) on Failure.

6.1.3.20 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 (1) on Success. False(0) on Failure.

6.1.3.21 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 (1) on Success. False(0) on Failure.

6.1.3.22 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 (1) on Success. False(0) on Failure.

6.1.3.23 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 (1) on Success. False(0) on Failure.

6.1.3.24 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 (1) on Success. False(0) on Failure.

6.1.3.25 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 (1) on Success. False(0) on Failure.

6.1.3.26 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 (1) on Success. False(0) on Failure.

6.1.3.27 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 (1) on Success. False(0) on Failure.

6.1.3.28 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.

6.1.3.29 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.

6.1.3.30 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 (1) on Success. False(0) on Failure.

6.1.3.31 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 (1) on Success. False(0) on Failure.

6.1.3.32 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 (1) on Success. False(0) on Failure.

6.1.3.33 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 (1) on Success. False(0) on Failure.

6.1.3.34 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 (1) on Success. False(0) on Failure.

6.1.3.35 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 (1) on Success. False(0) on Failure

6.1.3.36 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 (1) on Success. False(0) on Failure

6.1.3.37 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 (1) on Success. False(0) on Failure.

6.1.3.38 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 (1) on Success. False(0) on Failure.

6.1.3.39 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 (1) on Success. False(0) on Failure.

6.1.3.40 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 (1) on Success. False(0) on Failure.

6.1.3.41 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 (1) on Success. False(0) on Failure.

6.1.3.42 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 (1) on Success. False(0) on Failure.

6.1.3.43 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 (1) on Success. False(0) on Failure.

6.1.3.44 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 (1) on Success. False(0) on Failure.

6.1.3.45 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 (1) on Success. False(0) on Failure.

6.1.3.46 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 (1) on Success. False(0) on Failure.

6.1.3.47 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 (1) on Success. False(0) on Failure.

6.1.3.48 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 (1) on Success. False(0) on Failure.

6.1.3.49 hapi_bt_host_gap_cfg_scan

Used to scan the characteristics of a created BLE service.

```
bool hapi_bt_host_gap_cfg_scan(

struct hapi_bt_host *hapi_bt_host, uint16_t scan_period, uint16_t

scan_int, uint16_t scan_win, uint16_t scan_bkg_int, uint16_t

scan_bkg_win, uint8_t scan_filter_duplicates )
```

Arguments:

1. scan_period : Foreground scanning in ms (no connected link) (default: 10240).
2. scan_int: In 625 µs, range: 0x0004 to 0x4000 (default: 96)
3. scan_win: In 625 µs, range: 0x0004 to 0x4000 (default: 48)
4. scan_bkg_int: In 625 µs, range: 0x0004 to 0x4000 (default: 2048)
5. scan_bkg_win: In 625 µs, range: 0x0004 to 0x4000 (default: 18)
6. scan_filter_duplicates: 1=True, 0=False (default: 1).

Return: True on Success and False on Failure.

6.1.3.50 hapi_bt_host_gatt_service_changed

Used to message `gatt_service_changed`.

```
bool

hapi_bt_host_gatt_service_changed(struct hapi_bt_host

*hapi_bt_host)
```

Arguments:

1. hapi_bt_host: BLE HAPI instance pointer.

Return: True on Success and False on Failure.

6.1.3.51 hapi_bt_host_gatt_find_included_services

Used to message `gatt_find_included_services`.

```
bool hapi_bt_host_gatt_find_included_services(

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: Starting handle of the specified service
3. end_handle: Ending handle of the specified service.

Return: True (0) on Success. False on Failure.

6.1.4 Power Save APIs

6.1.4.1 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.

6.1.4.2 hapi_set_sleep_del

Provides a delay after initiating sleep.

```
void hapi_set_sleep_del(struct hapi *hapi, int usecs)
```

Arguments:

1. hapi: HAPI instance pointer.
2. usecs: delay in microseconds.

Return: None.

6.1.5 Socket APIs

6.1.5.1 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

   tcp_client=0, tcp_server=1, udp_client=2, udp_server=3, raw=4.
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: Socket descriptor on Success or -1 on Failure.

6.1.5.2 hapi_socket_send_tcp

Used to send data on a TCP socket.

```
bool

hapi_sock_send_tcp(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(1) on Success. False(0) on Error

6.1.5.3 hapi_sock_send_udp

Used to send data on a UDP socket.

```
bool

hapi_sock_send_udp(struct hapi *hapi, uint32_t socket,

             uint32_t *addr, uint16_t port, uint16_t addrlen,

             const void *data, size_t len)
```

Arguments:

1. hapi: HAPI instance pointer
2. socket: The socket ID which has been created
3. addr: destination IP address
4. port: destination port
5. addrlen: size of the address IPv4(4)/IPv6(16)
6. data: The data to be sent on the socket
7. len: The length of the data to be sent

Return: True(1) on Success. False(0) on Error

6.1.5.4 hapi_socket_receive

Used to receive data from a socket.

```
size_t

hapi_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.

6.1.5.5 hapi_socket_getavailable

Used to check received data available on a socket.

```
int

    hapi_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.

6.1.5.6 hapi_sock_notify

Registers notification for socket creation.

```
bool

    hapi_sock_notify(struct hapi *hapi, uint32_t socket,

                    uint32_t threshold, uint32 flags)
```

Arguments:

1. hapi: HAPI instance pointer.
2. socket: The socket ID which has been created.
3. Threshold: Threshold of data
4. flags: To read flags
    a. SOCKET_EVENT (Default):

       Data packet(s) of N bytes will arrive to the RX socket at any time.
    b. SOCKET_POLL:

       Data packet(s) with indication of N bytes available will be sent at any time. Receiver needs to use REQ/RSP to get the available data from buffer.

Return: Whether socket notification indication request was Successful. 0=Success, non-zero otherwise.

6.1.5.7 hapi_socket_close

Used to close a socket which has been opened.

```
void

    hapi_socket_close(struct hapi *hapi, uint32_t socket)
```

Arguments:

5. hapi: HAPI instance pointer.
6. socket: The socket ID which has been created.

Return: None.

6.1.5.8 hapi_sock_getavailable

Gets the number of bytes available to read in a socket.

```
int

hapi_sock_getavailable(struct hapi *hapi, uint32_t socket)
```

Arguments:

1. hapi: HAPI pointer to HAPI context.
2. socket: Socket handle.

Return: Number of bytes available at socket to read.

6.1.5.9 hapi_sock_burst_send

Writes multiple packets of data bytes into the socket.

```
bool hapi_sock_burst_send(struct hapi *hapi, uint32_t socket,

uint32_t *addr, uint16_t port, uint16_t addrlen, uint32_t num_pkt,

const void *data, size_t len)
```

Arguments:

1. hapi: HAPI pointer to HAPI context.
2. socket: Socket handle.
3. addr: Destination IP address.
4. port: Port destination.
5. addrlen: Size of the address IPv4(4)/IPv6(16).
6. num_packets: Number of packets to send to the socket.
7. data: Data to be sent.
8. len: Length of data.

Return: Socket send was Successful. True=Success, False otherwise.

6.1.5.10 hapi_sock_burst_receive

Reads multiple packets up to the size of the data bytes from the socket.

```
size_t hapi_sock_burst_receive(struct hapi *hapi, uint32_t socket,

void *data, size_t len, int *status, int *flags)
```

Arguments:

1. socket: Socket descriptor.
2. size: Number of bytes to receive.
3. flags: Reserved for future use.

Return:

1. num_pks: Number of packets to send to the socket.

6.1.6 MDNS APIs

6.1.6.1 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

6.1.6.2 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.

6.1.6.3 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(1) on Success. False(0) on Error

6.1.6.4 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(1) on Success. False(0) on Error

6.1.6.5 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(1) on Success. False(0) on Error

6.1.6.6 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(1) on Success. False(0) on Error

6.1.7 HTTP Client APIs

6.1.7.1 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.

6.1.7.2 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: Return: True(1) on Success. False(0) on Error

6.1.7.3 hapi_http_client_send_req

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

```
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: Return: True(1) on Success. False(0) on Error

6.1.7.4 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(1) on Success. False(0) on Error

6.1.7.5 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: Return: True(1) on Success. False(0) on Error

6.1.7.6 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: Return: True(1) on Success. False(0) on Error

6.1.7.7 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: Return: True(1) on Success. False(0) on Error

6.1.7.8 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: Return: True(1) on Success. False(0) on Error

6.1.8 MQTT APIs

6.1.8.1 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: Return: True(1) on Success. False(0) on Error

6.1.8.2 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: Return: True(1) on Success. False(0) on Error

6.1.8.3 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(1) on Success. False(0) on Error

6.1.8.4 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(1) on Success. False(0) on Error

6.1.8.5 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(1) on Success. False(0) on Error

6.1.8.6 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(1) on Success. False(0) on Error

6.1.8.7 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(1) on Success. False(0) on Error

6.1.8.8 hapi_mqtt_unsubscribe

Used to unsubscribe from a particular topic that has already been subscribed for.

```
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(1) on Success. False(0) on Error

6.1.8.9 hapi_mqtt_disconnect

Used to disconnect the MQTT.

```
bool

      hapi_mqtt_disconnect(struct hapi *hapi,

              struct hapi_mqtt *hapi_mqtt)
```

Arguments:

1. hapi: HAPI instance pointer.
2. Hapi_mqtt: The MQTT instance pointer

Return: True(1) on Success. False(0) on Error

6.1.8.10 hapi_mqtt_nw_disconnect

Used to disconnect from the network.

```
bool

    hapi_mqtt_nw_disconnect(struct hapi *hapi,

                     struct hapi_mqtt *hapi_mqtt)
```

Arguments:

1. hapi: HAPI instance pointer.
2. hapi_mqtt: The MQTT instance pointer

Return: True(1) on Success. False(0) on Error

6.1.8.11 hapi_mqtt_cert_store

Used to store the SSL/TLS certificate for MQTT.

```
bool
    hapi_mqtt_cert_store (struct hapi *hapi, char* certName,
                  uint32_t certLen, const unsigned char* certData)
```

Arguments:

a. hapi: HAPI instance pointer.
b. certName: Certificate name
c. certLen: Length of the certificate
d. certData: the certificate stream.

Return: True(1) on Success. False(0) on Error

6.1.8.12 hapi_mqtt_cert_delete

Used to delete the SSL/TLS certificate for MQTT.

```
bool
    hapi_mqtt_cert_delete(struct hapi *hapi, char* certName)
```

Arguments:

e. hapi: HAPI instance pointer.
f. certName: Certificate name

Return: True(1) on Success. False(0) on Error

6.1.8.13 hapi_mqtt_client_connect

Used to connect to the MQTT client.

```
struct hapi_mqtt *

hapi_mqtt_client_connect(struct hapi *hapi, struct

mqtt_client_config *config)
```

Arguments:

1. hapi: HAPI instance pointer.
2. mqtt_client_config: MQTT client configuration.

Return:

1. hapi_mqtt : Returns MQTT identifier.

6.1.8.14 hapi_mqtt_client_disconnect

Used to disconnect the MQTT client.

```
bool hapi_mqtt_client_disconnect(struct hapi *hapi, struct

hapi_mqtt *hapi_mqtt)
```

Arguments:

1. hapi: HAPI instance pointer.
2. mqtt_client_config: MQTT client configuration.

Return: True(1) on Success. False(0) on Error

6.1.9 TLS APIs

6.1.9.1 hapi_tls_create

Creates the TLS socket and does 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.

6.1.9.2 hapi_tls_set_dataready_cb

Registers the callback function when the TLS data is 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.

6.1.9.3 hapi_tls_upload_cert

Stores 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.

6.1.9.4 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.

6.1.9.5 hapi_tls_write

Sends 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.

6.1.9.6 hapi_tls_read

Reads 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.

6.1.9.7 hapi_tls_close

Closes the TLS socket and releases 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.

6.1.9.8 hapi_tls_set_notification_cb

Registers TLS set notification callback.

```
void

hapi_tls_set_notification_cb(struct hapi_tls *hapi_tls,

hapi_tls_notification_cb notification_cb, void *context)
```

Arguments:

1. hapi_tls: HAPI TLS instance pointer.
2. hapi_tls_notification_cb: TLS data ready callback function.
3. context: Context for callback.

Return: None.

6.1.10 Common APIs

6.1.10.1 hapi_start

Starts the HAPI interface. Initializes indication semaphore, resets the variables and starts the receive thread.

```
bool

hapi_start(struct hapi *hapi)
```

Arguments:

1. hapi: HAPI instance pointer.

Return: True on Success, False on Failure.

6.1.10.2 hapi_close

Stops HAPI and closes the interface. Destroys the indication semaphore, releases all indication handlers, destroys receive thread semaphore, and receives thread itself, and finally, frees the HAPI context itself.

```
void

hapi_close(struct hapi *hapi)
```

Arguments:

1. hapi: HAPI instance pointer.

Return: True on Success, False on Failure.

6.1.10.3 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.

6.1.10.4 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.

6.1.10.5 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.

6.1.10.6 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 NULL pointer on Failure.

6.1.10.7 hapi_config

Configures the HAPI interface for sleep wakeup.

```
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.

6.1.10.8 hapi_suspend

Enables/disables suspend mode. The pin settings set with `hapi_config` will be retained.

```
void

hapi_suspend(struct hapi *hapi, bool suspend_enable);
```

Arguments:

1. hapi: HAPI instance pointer.
2. Suspend_enable: enable (1)/disable (0) suspend mode.

Return: None.

6.1.10.9 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.

6.1.10.10 hapi_get_time

Gets 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.

6.1.10.11 hapi_nw_misc_app_time_get

Gets 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.

6.1.10.12 hapi_get_dbg_info

Gets 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.

6.1.10.13 hapi_get_ver

Gets the HAPI version.

```
char *

       hapi_ger_ver()
```

Arguments: None

Return: the version string.

6.1.10.14 hapi_setup

Set-up HAPI.

```
struct hapi *hapi_setup(void *hapi_uart, void *hapi_spi)
```

Arguments:

1. hapi_uart :pointer to HAPI UART.
2. hapi_spi : pointer to HAPI SPI.

Return: valid pointer pointing to hapi instance on Success.

6.1.10.15 show_hapi_ver

Shows information about the HAPI library.

```
static void show_hapi_ver(struct hapi * hapi, struct hio_query_rsp

*hio_query_rsp)
```

Arguments:

1. hapi: HAPI instance pointer.
2. hio_query_rsp: Response to HIO query

Return: True on Success, False on Failure.

6.1.10.16 hapi_console_init

Initializes HAPI console.

```
void hapi_console_init(struct hapi *hapi,CONSOLE_PRINT_FN

*console_print_fn);
```

Arguments:

1. hapi: HAPI instance pointer.
2. console_print: Print debug message on the console UART.

Return: True on Success, False on Failure.

6.1.10.17 hapi_get_scrambled_data_len

Returns scrambled data length.

```
int hapi_get_scrambled_data_len(int len)
```

Arguments:

1. len: Length of non-scrambled data.

Return: Length of scrambled data.

6.1.10.18 hapi _hio_scrambling_init

Initializes the HIO scrambling context.

```
void hapi_hio_scrambling_init(struct hapi *hapi, void

*scrambling_ctx, void* key,scrambling_fn scrambling_fn,

descrambling_fn descrambling_fn)
```

Arguments:

1. hapi: Pointer to HAPI context.
2. scrambling_ctx: Context for scrambling and descrambling.
3. key: Key for scrambling/descrambling.
4. scrambling_fn: Function implementing scrambling.
5. descrambling_fn: Function implementing descrambling.

Return: None

6.1.10.19 hapi_disp_pkt_info

Prints input output packet information.

```
void hapi_disp_pkt_info(struct hapi *hapi, int val)
```

Arguments:

1. hapi: Pointer to HAPI context.
2. val: Enables/disables packet information print.

Return: None.

6.1.10.20 hapi_init_interface

Registers interface parameters.

```
void hapi_init_interface(struct hapi *hapi, struct hapi_ops

*hapi_ops, void *dev)
```

Arguments:

1. hapi: Pointer to HAPI context.
2. hapi_ops: Device options.
3. dev: Pointer to interface device.

Return: None

6.1.10.21 hapi_custom_msg_proc

Sends the command to Talaria TWO and waits for response. Once the response is received, it reverts the response data to the sender application.

```
int hapi_custom_msg_proc(struct hapi *hapi, uint8_t *group_id,

uint8_t *msg_id,uint8_t *data, uint16_t *len, int data_max_rx_len)
```

Arguments:

1. hapi: Pointer to HAPI context.
2. group_id: Group ID.
3. msg_id: Message ID.
4. data: Message data.
5. len: Payload size of packet.
6. data_max_rx_len: Maximum reception data length.

Return: -1 if packet reception fails and 0 on success.

6.1.10.22 hapi_pkt_free

Frees the HAPI packet, and message buffer associated to packet.

```
void hapi_pkt_free(struct hapi_packet* pkt)
```

Arguments:

1. pkt: Packet to be freed.

Return: None.

6.1.10.23 hapi_rx_disable

Disables reception by killing the thread.

```
void hapi_rx_disable(struct hapi *hapi)
```

Arguments:

1. hapi: Pointer to HAPI context.

Return: None.

6.1.10.24 hapi_set_Error

Prints Error.

```
hapi_set_Error(struct hapi *hapi, int Error_code, const char *fmt,

...)
```

Arguments:

1. hapi: Pointer to HAPI context.
2. Error_code: Error code.
3. fmt: Printf style formatting arguments.

Return: None.

6.1.10.25 hapi_clear_Error

Clears Error.

```
void hapi_clear_Error(struct hapi *hapi)
```

Arguments:

1. hapi: Pointer to HAPI context.

Return: None.

6.1.10.26 hapi_suspend_enabled_get

Checks suspend status.

```
bool hapi_suspend_enabled_get(struct hapi *hapi)
```

Arguments:

1. hapi: Pointer to HAPI context.

Return: 1: if suspend mode is enabled, else 0.

6.1.10.27 hapi_sig_wakeup

Used to wake Talaria TWO from suspended state.

```
void hapi_sig_wakeup(struct hapi *hapi)
```

Arguments:

1. hapi: Pointer to HAPI context.

Return: None.

6.1.10.28 hapi_get_git_id

Gets the git ID.

```
char *

      hapi_get_git_id()
```

Arguments: None

Return: Git ID string.

6.1.10.29 is_hapi_hio_scrambling_enabled

Used to check whether HIO scrambling is enabled or not.

```
int

is_hapi_hio_scrambling_enabled(struct hapi *hapi)
```

Arguments:

1. hapi: Pointer to HAPI context.

Return: HIO scrambling state. 1=enabled, 0=disabled.

6.1.10.30 hapi_set_scrambling_enabled

Enables HIO scrambling.

```
int

hapi_set_scrambling_enabled(struct hapi *hapi,int val)
```

Arguments:

1. hapi: Pointer to HAPI context.
2. val: set '1' to enable and '0' to disable.

Return: None.

6.1.10.31 hapi_pkt_msg_alloc

Used for allocating a packet and sending a message.

```
struct hapi_packet *

hapi_pkt_msg_alloc(struct  hapi *hapi,

                  uint8_t msg_group,

                  uint8_t msg_id,

                  size_t  msg_hdr_size,

                  size_t  msg_payload_size)
```

Arguments:

1. hapi: Pointer to HAPI context.
2. msg_group: Message group ID.

3. msg_id: Message ID.
4. msg_hdr_size: Size of header.
5. msg_payload_size: Payload size of packet.

Return: Allocated packet.

6.1.10.32 hapi_send_recv_validate

Sends the packet and validates the reply packet.

```
struct hapi_packet *

hapi_send_recv_validate(struct hapi *hapi, struct hapi_packet

*pkt,

                          uint8_t rsp_group_id,

                          uint8_t rsp_msg_id)
```

Arguments:

1. hapi: Pointer to HAPI context.
2. hapi_packet *pkt: Packet to be sent.
3. rsp_group_id: Expected group ID of reply packet.
4. rsp_msg_id: Expected msg ID of reply packet.

Return: Packets received from Talaria TWO device.

6.1.10.33 hapi_send_recv_no_validate

Send the packet, and does not validate the reply packet.

```
struct hapi_packet *

hapi_send_recv_validate(struct hapi *hapi, struct hapi_packet

*pkt, uint8_t rsp_group_id, uint8_t rsp_msg_id)
```

Arguments:

1. hapi: Pointer to HAPI context.
2. hapi_packet *pkt: Packet to be sent.
3. rsp_group_id: Expected group ID of reply packet.

4. rsp_msg_id: Expected msg ID of reply packet.

Return: Packets received from Talaria TWO device.

6.1.10.34 hapi_pkt_validate

Used for packet validation.

```
bool

hapi_pkt_validate(struct hapi *hapi, struct hapi_packet *pkt,

uint8_t msg_group, uint8_t msg_id, bool check_trxid)
```

Arguments:

1. hapi: Pointer t HAPI context.
2. hapi_packet *pkt: Packet to be sent.
3. msg_group: Expected group ID.
4. msg_id: Expected message ID.
5. check_trxid: Specifies whether to check trxid of the received packet.

Return: Returns packet validate status. True=expected packet received, False otherwise.

6.1.10.35 hapi_get_max_msg_size

Used to get maximum size of the message.

```
unsigned int hapi_get_max_msg_size(struct hapi *hapi)
```

Arguments:

1. hapi: Pointer to HAPI context.

Return:

1. msg_max_size: Maximum message size of communication.

6.2 SPI Interface APIs

6.2.1 hapi_spi_init

Registers the SPI.

```
struct hapi* hapi_spi_init(void* hapi_spi_ptr, CS_HIGH_FN cs_hi,

CS_LOW_FN cs_low, IF_TX_FN tx_fn, IF_RX_FN rx_fn)
```

Arguments:

1. hapi_spi_ptr: pointer to the HAPI SPI instance.
2. CS_HIGH_FN cs_hi: sets the CS to high.
3. CS_LOW_FN cs_low: resets the CS to low
4. IF_TX_FN tx_fn: transmission function.
5. IF_RX_FN rx_fn: Receiving function.

Return: True(1) on Success. False(0) on Error

6.2.2 hapi_spi_cs_high

Sets the CS to high before calling `hapi_spi_init()`.

```
void hapi_spi_cs_high()
```

Arguments: None.

Return: None.

6.2.3 hapi_spi_cs_low

Resets the CS to low before calling `hapi_spi_init()`.

```
void hapi_spi_cs_low()
```

Arguments: None.

Return: None.

6.2.4 hapi_spi_tx

Used for transmitting an amount of data in blocking mode.

```
int  hapi_spi_tx(void *ptr, char *buff, int len)
```

Arguments:

1. buff: pointer to character buffer.
2. len: length of the data.

Return: True on Success, False on Failure.

6.2.5 hapi_spi_rx

Used for receiving an amount of data in blocking mode.

```
int  hapi_spi_rx(void *ptr, char *buff, int len)
```

Arguments:

1. buff: pointer to character buffer.
2. len: length of the data.

Return: True on Success, False on Failure.

6.2.6 hapi_spi_data_waiting

This function is used to inform HAPI that Talaria TWO wants to send data to host. Talaria TWO will raise interrupt when data is to be sent to host, and from host IRQ handler this function needs to be called.

```
void hapi_spi_data_waiting()
```

Arguments: None.

Return: None.

6.2.7 hapi_spi_write

Used to write data to SPI interface.

```
ssize_t hapi_spi_write(void *dev, const void *data, size_t length)
```

Arguments:

1. hapi Pointer to HAPI context.
2. data: Pointer to data.
3. length: Length of data.

Return:

1. length: Length of data written.

6.2.8 hapi_spi_read

Used to read data from SPI interface.

```
ssize_t hapi_spi_read(void *dev, void *data, size_t length
```

Arguments:

1. hapi: Pointer to HAPI context.
2. data: Pointer to data.
3. length: Length of data.

Return:

1. length: Length of data read.

6.3 UART Interface APIs

6.3.1 hapi_uart_init

Initializes the UART interface.

```
struct hapi*  hapi_uart_init(void* hapi_uart_ptr, IF_TX_FN tx_fn,

IF_RX_FN rx_fn, IF_ERR_FN err_fn, IF_UART_INIT uart_init)
```

Arguments:

1. hapi_uart_ptr: pointer to the HAPI UART instance.
2. IF_TX_FN tx_fn: transmitter function.
3. TF_RX_FN rx_fn: receiver function.
4. IF_ERR_FN err_fn: Error function.
5. IF_UART_INIT uart_init: UART initialization.

Return: hapi instance on Success and NULL on Failure.

6.3.2 hapi_uart_tx

Used for transmitting an amount of data in blocking mode in the UART interface.

```
int hapi_uart_tx(void *ptr, char *buff, int len)
```

Arguments:

1. buff: pointer to character buff.
2. len: length of the data to be transmitted.

Return: on Success returns the number of bytes transmitted and -1 on failure.

6.3.3 hapi_uart_rx

Used for Receiving an amount of data in blocking mode.

```
int hapi_uart_tx(void *ptr, char *buff, int len)
```

Arguments:

3. buff: pointer to character buff.
4. len: length of the data to be received.

Return: on Success returns number of bytes received else -1 on failure.

6.3.4 hapi_uart_read

Used to read data from UART interface.

```
ssize_t hapi_uart_read(void *dev, void *data, size_t length)
```

Arguments:

1. hapi Pointer to HAPI context.
2. data: Pointer to data.
3. length: Length of data to be read in bytes.

Return:

1. length: Number of bytes read.

6.3.5 hapi_uart_write

Used to write data to UART interface.

```
ssize_t hapi_uart_write(void *dev, void *data, size_t length)
```

Arguments:

1. hapi: Pointer to HAPI context.
2. data: Pointer to data.
3. length: Length of data to be written in bytes.

Return:

1. length: Number of bytes of data written.

6.4 Power Save APIs

6.4.1 hapi_t2_wakeup_config

Used for configuring the Talaria TWO pins.

```
void hapi_t2_wakeup_config(void* hapi, uint8_t type)
```

Arguments:

1. hapi: pointer to HAPI.
2. type: wake-up type.

Return: None.

6.4.2 hapi_spi_t2_wakeup_fn

Used to wake-up the SPI function in Talaria TWO.

```
void hapi_spi_t2_wakeup_fn(void* hapi, void*  wakeup_t2);
```

Arguments:

1. hapi: pointer to HAPI.
2. wakeup_t2: pointer to wakeup_t2 through spi

Return: None.

6.4.3 hapi_uart_t2_wakeup_fn

Used to wake-up the UART function in Talaria TWO.

```
void hapi_uart_t2_wakeup_fn(void* hapi, void*  wakeup_t2);
```

Arguments:

1. hapi: pointer to HAPI.
2. wakeup_t2: pointer to wakeup_t2 through uart

Return: None.

6.5 Unassoc APIs

6.5.1 hapi_unassoc_create

Creates the unassociation.

```
bool hapi_unassoc_create(struct hapi *hapi, uint8_t *addr);
```

Arguments:

1. hapi: instance of pointer.
2. addr: pointer to address

Return: True(1) on Success. False(0) on Error

6.5.2 hapi_unassoc_config

For configuring the parameters of unassociation in HAPI.

```
bool hapi_unassoc_config(struct hapi *hapi,

        uint32_t num_probes, uint32_t interval_ms, uint32_t verbose,

        char *ssid, uint32_t rate, uint32_t suspend_en,

        uint8_t ie_len, uint8_t *ie);
```

Arguments:

1. hapi: instance of the pointer.
2. num_probes: number of probes used.
3. interval_ms: interval in ms.
4. verbose: number of verbose.
5. ssid: SSID used for configuration.
6. rate: rate used for the unassociation configuration.
7. suspend_en: suspend encryption.
8. ie_len: length of optional, additional information elements included in the probe request frames.
9. ie: length

Return: True(1) on Success. False(0) on Error

6.5.3 hapi_unassoc_start

To start un-association in HAPI.

```
bool hapi_unassoc_start(struct hapi *hapi)
```

Arguments:

1. hapi: Pointer to HAPI context.

Return: True(1) on Success. False(0) on Error

6.5.4 hapi_unassoc_stop

To stop un-association in HAPI.

```
bool hapi_unassoc_stop(struct hapi *hapi)
```

Arguments:

1. hapi: Pointer to HAPI context.

Return: True(1) on Success. False(0) on Error

7 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.

8 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.

All trademarks, trade names and registered trademarks mentioned in this document are property of their respective owners and are hereby acknowledged.