## Description

This application represents various host interface APIs in groups called HIO groups. The HIO groups are similar to the HAPI groups, except that the HIO groups are part of the SMP application running on Talaria TWO.

HAPI and HIO groups maintain the same GROUP ID and MSGID numbers. The list of required APIs can be enabled from the Makefile. Few APIs (like WCM) are enabled by default.

Following is the description of the types of APIs supported by each of these groups.

1. WCM: Provides APIs to:
   1. Connect to and disconnect from a Wi-Fi network.
   2. Set and get IPV4/IPV6 address.
   3. Set/Get Wi-Fi power management.
   4. Get RSSI and Tx power values.
   5. Set regulatory domains.
2. Socket: Provides APIs to:
   1. Create a server and client socket.
   2. Send and receive data.
   3. Get peer information.
3. MDNS: Provides APIs to:
   1. Set up, stop, and resolve MDNS.
   2. Add and remove an MDNS service.
4. MQTT: Provides APIs to:
   1. Connect to and disconnect from MQTT server.
   2. Publish and subscribe to and unsubscribe from a topic.
   3. Store and delete certificates required to perform MQTT over TLS.
5. TLS: Provide APIs to:
   1. Set up and close a TLS connection.
   2. Upload a certificate.
   3. Perform TLS handshake.
   4. Read and write.
6. BLE: Provides APIs to:
   1. Create a BT host.
   2. Perform GAP operations like setting GAP address, scan, discovery, initiate/terminate a connection, configure broadcast, discoverable modes and configure advertising interval.
   3. Start a GATT server, initiate/destroy a GATT service, add characteristics, configure the device to send notifications, indications.
7. HTTP: Provides APIs to:
   1. Set up and start a HTTP client.
   2. Send client data, set, and delete the HTTP client header.
   3. Store and delete the certificates required to perform a TLS connection.
8. GPIO: Provides APIs to:
   1. Select a GPIO.
   2. Set and reset the state of the GPIO.
9. File: Provides APIs to:
   1. Add a file to the file system.
   2. Delete a file from file system.
10. AWS: Provides APIs to:
    1. Connect to and disconnect from AWS server.
    2. Set configuration like AWS host URL, thing name, AWS port, the path of the certificates, client id.
    3. Send and receive data from AWS server.
11. FOTA: Provides APIs to:
    1. Start FOTA operation.
    2. Send FOTA configuration data.
12. FOS: Provides APIs to:
    1. Start firmware upgrade over serial port.
    2. Send FOTA configuration data.
    3. Commit FOS data to mark as end of image data received by Talaria TWO.
13. Un-assoc: Provides APIs to:
    1. Start and stop the unassoc mode.
    2. Configure the mode with parameters like number of probes to be transmitted, transmission interval, data rate and information element.

Talaria TWO device will service the functionality over UART/SPI/SDIO interface. The normal operation is to first issue a HIO query request message. The HIO query response message will include a table of groups and messages supported by Talaria TWO. It also includes the maximum request size the device can handle.

## Request-Response Message Exchange

The SMP application registers a particular HIO group by calling the <group>\_hio\_init() function which in turn calls the hio\_api\_init(const struct hio\_api \*api, void \*ctx) API to register the HIO message group.

The first argument struct hio\_api \*apihas the following prototype:

|  |
| --- |
| static const struct hio\_api hio\_api = {  .group = 0,  .num\_handlers = n,  .handler = {  Handler\_1,  .  .  Handler\_n,  }  }; |

The hio\_api struct specifies the following:

1. Group ID.
2. Number of handlers to be registered as a part of HIO group registration.
3. Pointers to the handlers that call a Talaria TWO networking API (or a custom user API which will be discussed in the subsequent sections) to perform the request from the host.

Each handler is associated with a unique ID called MSGID. When a request packet is sent from a HAPI-based host, The HIO interpreter invokes the corresponding handler based on the group\_id and msg\_id. The handler calls the relevant Talaria TWO’s networking API to execute the request received from the host and returns a response.

The communication between a HAPI-based application and the SMP application is illustrated in Figure 7.

Diagram

Description automatically generated

Figure 7: Exchange of request-response packets between Host and Talaria TWO

## Talaria TWO Reset Sequence

After Talaria TWO powers up, a reset needs to be issued by the host application. This can be done by pulling down the reset pin low for a period of about 100 ms and then high for a period of about 100 ms.

A diagram of a data flow

Description automatically generated

Figure 8: Talaria TWO reset sequence block diagram

## Code Walkthrough

The application first mounts the filesystem to access the certificates and network configuration files.

|  |
| --- |
| rval = utils\_mount\_rootfs();  if(0 != rval) {  os\_printf("Muonting rootfs failed.!\n");  while(1);  }  os\_printf("\n[APP]root fs mounted, rval = %d", rval); |

The application enables the API groups depending on the flag value set in the make file (i.e., HAVE\_<GROUP>\_HIO). The HIO transport driver initialization also occurs when the first message group is initialized.

The following block initializes the WCM, socket and MDNS groups if the HAVE\_<GROUP>\_HIO value of the corresponding group is set to 1. Since the WCM group is the first HIO group which is being initialized, the HIO transport driver initialization occurs and the corresponding interface i.e., UART/SPI/SDIO is configured on Talaria TWO.

|  |
| --- |
| #if HAVE\_WCM\_HIO==1  strcat(banner, ", wcm");  wcm\_hio\_init();  num\_groups\_registered ++;  #endif  #if HAVE\_SOCK\_HIO==1  strcat(banner, ", sock");  sock\_hio\_init();  num\_groups\_registered ++;  #endif  #if HAVE\_MDNS\_HIO==1  strcat(banner, ", mdns");  mdns\_hio\_init();  num\_groups\_registered ++;  #endif |
|  |

The following block initializes the MQTT, TLS, BT, HTTP, FOTA, FOS, unassoc, provisioning and GPIO groups:

|  |
| --- |
| #if HAVE\_MQTT\_HIO==1  strcat(banner, ", mqtt");  mqtt\_hio\_init();  num\_groups\_registered ++;  #endif  #if HAVE\_TLS\_HIO==1  strcat(banner, ", tls");  tls\_hio\_init();  num\_groups\_registered ++;  #endif  #if HAVE\_BT\_HIO==1  strcat(banner, ", bt");  register\_bt\_host\_hio();  num\_groups\_registered ++;  #endif  #if HAVE\_HTTP\_HIO==1  strcat(banner, ", http");  hio\_http\_client\_init();  num\_groups\_registered ++;  #endif  #if HAVE\_HTTPS\_HIO==1  strcat(banner, ", https");  tls\_hio\_init();  num\_groups\_registered ++;  #endif  #if HAVE\_FOTA\_APP==1  strcat(banner, ", FOTA");  hio\_fota\_init();  num\_groups\_registered ++;  #endif  #if HAVE\_FOS\_APP==1  strcat(banner, ", FOS");  hio\_fos\_init();  num\_groups\_registered ++;  #endif  #if HAVE\_UNASSOC\_SUPPORT==1  strcat(banner, ", UNASSOC");  unassoc\_api\_init();  num\_groups\_registered ++;  #endif  #if HAVE\_PROV\_APP==1  strcat(banner, ", PROV");  hio\_prov\_init();  num\_groups\_registered ++;  #endif  #if HAVE\_FILE\_HIO==1  strcat(banner, ", file");  hio\_file\_init();  num\_groups\_registered ++;  #endif  #if HAVE\_AWS\_HIO==1  strcat(banner, ", aws");  aws\_app\_init();  num\_groups\_registered ++;  #endif |
|  |

register\_hio\_packet\_hook() function displays the group\_id and msg\_id of every packet being sent and received on Talaria TWO.

First, the hook functions to display the packet data being received by Talaria TWO (input hook) and the packet data being sent by Talaria TWO (output hook) are defined.

|  |
| --- |
| void hio\_input\_packet\_info(struct packet \*pkt)  {  os\_printf("input-hook\n");  show\_packet\_info(pkt);  }  void hio\_output\_packet\_info(struct packet \*pkt)  {  os\_printf("output-hook\n");  show\_packet\_info(pkt);  } |

The show\_packet\_info() function extracts the GROUP ID and the MSGID and displays it.

|  |
| --- |
| void show\_packet\_info(struct packet \*pkt)  {  const struct hio\_msghdr \*hdr;  hdr = packet\_data(pkt);  uint32\_t group = hdr->group, msgid = hdr->msgid;  os\_printf("hio: group=%d.msgid=%d\n", group, msgid);  } |

Now, the register\_hio\_packet\_hook() function registers the input packet hook and the output packet hook callback functions.

|  |
| --- |
| void register\_hio\_packet\_hook()  {  int ret;  ret = hio\_packet\_hook\_register(hio\_input\_packet\_info,  hio\_output\_packet\_info);  os\_printf("Packet hook register status = %d\n", ret);  } |

The application will register the HIO packet hook by calling the register\_hio\_packet\_hook () function when the bootarg displ\_pkt\_info=1 is issued.

|  |
| --- |
| if(os\_get\_boot\_arg\_int("disp\_pkt\_info", 0) != 0) {  /\* Register packet hook.  \* Hook will print the msg\_id and group\_id of every packets sent and received  \*/  register\_hio\_packet\_hook();  } |

Finally, if LWIP is running, the iPerf3 server is started and SNTP is initialized.

## Adding Custom Groups to SMP Application

When in hosted mode, there might be a requirement to add additional features to the existing SMP application depending on the end user’s requirements.

This section describes the procedure to add support for a custom group to the SMP application with an example. The example application demonstrates a message exchange between the host application and the SMP application. The host sends a request message to Talaria TWO and waits for a response message.

The enhancements need to be performed on both the HAPI-based host application and the SMP application.

Subsequent sections describe the procedure to add support to custom groups of the host application and then the procedure to add the support for custom group.

### Procedure to Add Custom Group to the Host Application

1. Define a group number greater than 150 as GROUP ID. The group numbers from 0 to 149 will be used by the existing groups.
2. Define a structure for the request message to be sent.
3. Define a structure for the response message to be received.
4. Create a HAPI packet to be sent.
5. Update/copy the data to be sent to the HAPI packet created.
6. Send the HAPI packet to Talaria TWO and wait for the response.
7. If the response is received, read the contents of the response packet.
8. Release the memory allocated for the packet.

From the above-described procedure, the following are defined in api/custom.h in the example application:

1. Custom GROUP ID -158
2. MSGIDs for the custom request message-0x03
3. Custom response message-0x83
4. Structures for the custom request message
5. Custom response message

|  |
| --- |
| #define HIO\_GROUP\_CUSTOM 158  #define HAPI\_CUSTOM\_MSG\_REQ 0x03  #define HAPI\_CUSTOM\_MSG\_RSP 0x83 |

Following is the structure defined for a custom request message:

|  |
| --- |
| struct hapi\_custom\_msg\_req {  char echo\_req[MAX\_MSG\_SIZE]; /\*Request message from host\*/  }; |

Corresponding structure for a custom response message also needs to be defined:

|  |
| --- |
| struct hapi\_custom\_msg\_rsp {  uint32\_t status; /\*\*< result status, zero is success \*/  char echo\_rsp[MAX\_MSG\_SIZE]; /\*\*< response from T2 \*/  }; |

A function to send and receive a packet to/from Talaria TWO (i.e., api\_send\_custom\_msg()) is defined in *hapi\lib\src\hapi\_custom.c.*

First, a packet has to be created by allocating the required amount of message buffer by calling hapi\_pkt\_msg\_alloc()API.This API will allocate the required memory and return a pointer of type struct hapi\_packet. The definitions of struct hapi\_packet, struct hapi\_msg\_frame and struct hapi\_msg\_hdr are as follows:

|  |
| --- |
| /\* Headers for all device communication ----- \*/  struct hapi\_msg\_hdr {  uint8\_t group;  uint8\_t msgid;  uint16\_t trxid;  };  struct hapi\_msg\_frame {  uint16\_t size; // sizeof data + msghdr  struct hapi\_msg\_hdr msg\_hdr;  };  /\* Packet definition ----------------------- \*/  #define HAPI\_PACKET\_MAGIC 0x600D  struct hapi\_packet {  uint16\_t magic;  struct hapi\_msg\_frame frame;  void \* msg;  struct hapi\_packet \* next;  }; |

The pkt\_out variable contains the pointer to the struct hapi\_packet type created. The hapi\_pkt\_msg(pkt\_out) API returns a pointer to the message buffer in the packet created.

|  |
| --- |
| struct hapi\_custom\_msg\_req \*req;  struct hapi\_packet \*pkt\_out =  hapi\_pkt\_msg\_alloc(hapi, HIO\_GROUP\_CUSTOM, HAPI\_CUSTOM\_MSG\_REQ, sizeof(struct hapi\_custom\_msg\_req), 0);  req = hapi\_pkt\_msg(pkt\_out);  /\* update req fields \*/  memcpy(req->echo\_req, msg, sizeof(req->echo\_req)); |

The data to be sent by the host (in the msg buffer in this example) is copied to the request message buffer. Now, the packet to be sent to Talaria TWO is ready. The packet contains the following fields:

A picture containing histogram

Description automatically generated

Figure 9: Packet contents

The host sends a message: “Hello from host”.

The length of the packet is: sum of size of Payload+TRX ID+MSG ID+GROUP ID.

This packet will be sent to Talaria TWO by the HIO transport layer. The application calls the hapi\_send\_recv\_validate() API to send the packet (i.e., pkt\_out) and blocks until a response is received from Talaria TWO.

|  |
| --- |
| struct hapi\_packet \*pkt\_in =  hapi\_send\_recv\_validate(hapi, pkt\_out, HIO\_GROUP\_CUSTOM, HAPI\_CUSTOM\_MSG\_RSP);  if (pkt\_in == NULL) {  /\* Unexpected behaviour \*/  printf("%s failed.\n", \_\_FUNCTION\_\_);  goto end;  } |

On receiving the response packet from Talaria TWO, hapi\_send\_recv\_validate()will return the pointer to the received response packet.

|  |
| --- |
| struct hapi\_custom\_msg\_rsp \*rsp = hapi\_pkt\_msg(pkt\_in);  if (rsp->status != 0) {  /\* Unexpected behaviour \*/  printf("%s status failed.\n", \_\_FUNCTION\_\_);  goto end;  }  ok = true;  printf("recieved :%s-%d\r\n",rsp->echo\_rsp,test);  memcpy(rsp->echo\_rsp,resp\_msg , sizeof(req->echo\_req)); |

The contents of the message buffer of the received packet are copied into a buffer for the application to process it.

### Procedure to Add Custom Group to Host Application.

The SMP application includes the same GROUP ID, MSG ID and the structures of request and response message used by the HAPI-based host application.

1. Create a header file – custom\_group.h and include the same custom GROUP ID -158, MSGIDs for the custom request message-0x03, custom response message-0x83 and the structures for the custom request message and custom response message.
2. Define a handler to receive a request packet from the host and send a response packet.
3. Define a HIO interface API of type struct hio\_api and provide the GROUP ID, number of the handlers to be registered and the pointer to the handler to be called when a request message with the corresponding GROUP ID and MSGID is received. Following is the definition of struct hio\_api:

|  |
| --- |
| struct hio\_api {  uint16\_t group;  uint16\_t num\_handlers;  struct packet \*(\*const handler[])(void \*ctx, struct packet \*pkt);  }; |

Refer document: Talaria\_TWO\_Host\_API\_Reference\_Guide.pdf (path: *sdk\_x.y\doc\reference\_guides\api\_reference\_guide*) for more information on struct hio\_api.

1. Register the custom group by calling the hio\_api\_init(const struct hio\_api \*api, void \*ctx) API.
2. Following the procedure described above, the custom\_group.h header file contains the GROUP ID, MSG ID, and the structure definitions of the request and response messages. These definitions are same as the ones defined in api/custom\_group.h file in HAPI-based host application.

|  |
| --- |
| #define HIO\_GROUP\_CUSTOM 158  #define HAPI\_CUSTOM\_MSG\_REQ 0x03  #define HAPI\_CUSTOM\_MSG\_RSP 0x83  struct hapi\_custom\_msg\_req {  char echo\_req[MAX\_MSG\_SIZE]; /\*Request message from host\*/  };  struct hapi\_custom\_msg\_rsp {  uint32\_t status; /\*\*< result status, zero is success \*/  char echo\_rsp[MAX\_MSG\_SIZE]; /\*\*< response from T2 \*/  }; |

1. The required functions to register the custom group and the handlers are defined in custom\_group.c.

custom\_data\_send\_recieve handler receives request packet, extracts the content of the packet, and sends a response packet.

|  |
| --- |
| static struct packet \*custom\_data\_send\_recieve(void \*ctx, struct packet \*msg)  {  struct hapi\_custom\_msg\_req \*req = packet\_data(msg);  os\_printf("host sent:%s\r\n",req->echo\_req);  return custom\_send\_resp();  } |

1. custom\_send resp() function creates a response packet, copies the payload to be included in the response packet and returns a pointer of type struct packet.

|  |
| --- |
| static struct packet\* custom\_send\_resp(void)  {  char t2\_rsp[64] = "Resp from T2";  struct packet \*pkt;  struct hapi\_custom\_msg\_rsp \*rsp;  pkt = OS\_ERROR\_ON\_NULL(alloc\_custom\_data\_rsp(&rsp));  rsp->status = 0;  memcpy(rsp->echo\_rsp, t2\_rsp,sizeof(t2\_rsp));  return pkt;  } |

1. alloc\_custom\_data\_rsp() function creates a packet by allocating memory for the response packet to be sent. The payload – “Resp from Talaria TWO” is copied to the message buffer of the packet created and the pointer to the created packet is returned by this function.

|  |
| --- |
| static inline struct packet \* alloc\_custom\_data\_rsp(struct hapi\_custom\_msg\_rsp \*\*rsp)  {  struct packet \*pkt = packet\_alloc(sizeof(struct hio\_msghdr) + sizeof \*\*rsp);  if (pkt) {  pfrag\_reserve(packet\_first\_frag(pkt), sizeof(struct hio\_msghdr));  \*rsp = packet\_insert\_tail(pkt, sizeof \*\*rsp);  }  return pkt;  } |

1. pfrag\_reserve() API returns the data and the address of head of the linked list in which the packet is included. packet\_insert\_tail returns the data and address of the tail node in the list. The address of the packet created is returned. custom\_send\_resp() function copies the response message payload and returns the packet.

This packet is sent to the host by the custom\_data\_send\_recieve() handler. The host now receives the response packet from Talaria TWO.