ATBTLC1000

ATBTLC1000 BluSDK BLE API SW Development Guide User's Guide

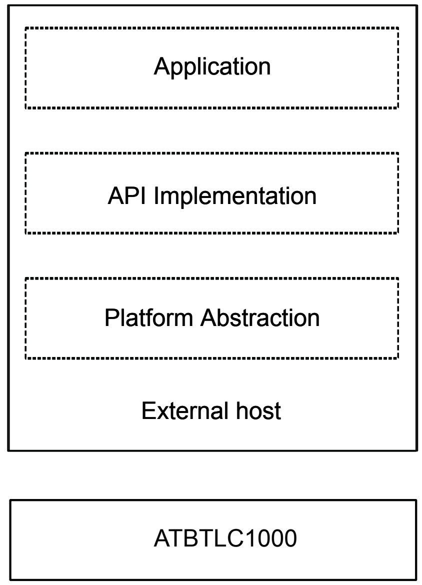
Introduction

This user guide details the functional description of Bluetooth Low Energy (BLE) Application Peripheral Interface (API) programming model. This also provides the example code to configure an API for Generic Access Profile (GAP), Generic Attribute (GATT) Profile, and other services using the ATBTLC1000.

Table of Contents

1. Overview

Figure -1. Overview of ATBTLC1000



The ATBTLC1000 provides Bluetooth Smart Link Controller in a single System on a Chip (SoC) that includes:

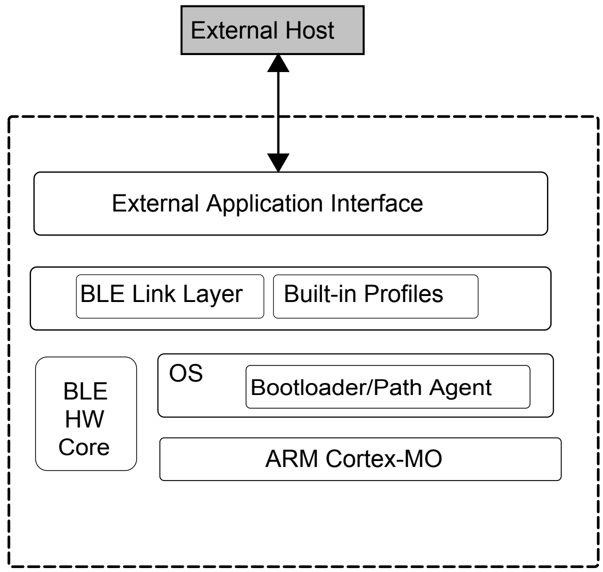
- Radio Frequency (RF)
- Link Layer
- Generic Access Profile (GAP)
- Generic Attribute (GATT) Profile
- Security Manager Protocol (SMP)

It provides the host microcontroller with methods to perform the following:

- Standard Bluetooth Smart
- GAP
- GATT server
- Client operations
- Security management with peer devices

The ATBTLC1000 runs firmware on chip which provides BLE 4.1 functionality. On top of the Link Layer Firmware, is an embedded L2CAP, GAP, SMP, and GATT layer that complies with Special Interest Group (SIG) standard 4.1.

Figure -2. External Host



1.1 ATSAMB11 Solution Architecture

The ATSAMB11 solution is mainly composed of three sub-systems running concurrently.

- User Internal Application running in an OS thread, communicating with BLE Stack via a messaging interface.
- Link Controller that implements up to GATT and GAP layers running in an OS thread.
- Application Atmel Adaptation API layer that maps the GAP/GATT functionalities into their respective messages, that is sent to the BLE stack over the messaging interface.

2. API Programming Model

This chapter describes the programming model of the app for ATBTLC1000 using APIs. The app perform following operations:

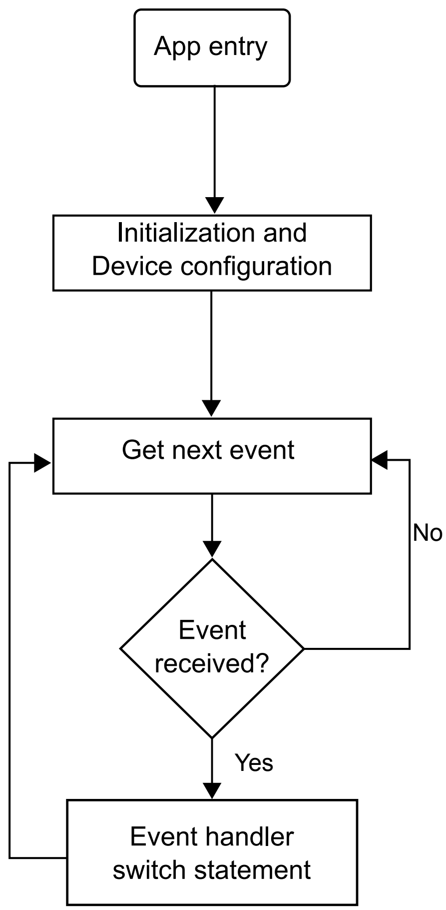
- Platform initialization/Link controller initialization
- Device Configuration
- Event Monitoring and handling

2.1 General Application Flow

The general app flow initializes the link controller and bus. The initialization is done by `at_ble_init();` call function.

The device configuration includes setting up the device address, device name, and device advertising data. API call functions has no event messages associated with device configuration, API call functions are called at the start of the app and return error code to validate an operation.

Figure 1-1. General Application Flow



2.2 Request Response Flow

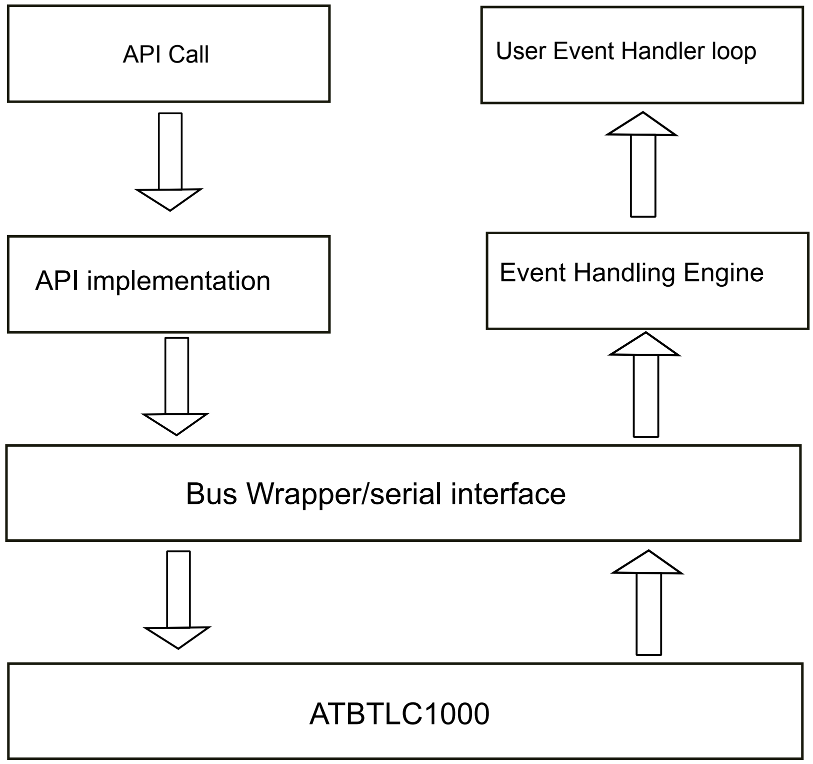
API operation relies on a request – response mechanism. The request is sent via the dedicated API. Calling an API triggers and returns one or more event message to the app. These messages are handled by the event handler loop of the user app. For example, if the user call `at_ble_scan_start()`, user expects the controller must return an event with `AT_BLE_SCAN_INFO` for each device scanned by the ATBTLC1000.

This code snippet below shows an example of the event loop within a valid complete.

```
at_ble_addr_t addr = {AT_BLE_ADDRESS_PUBLIC,
        {0x25, 0x75, 0x11, 0x6a, 0x7f, 0x7f} };
uint16_t handle;
    // init device
    at_ble_init(NULL);

    at_ble_addr_set(&addr);

    // start advertising
    at_ble_adv_data_set(adv_data, sizeof(adv_data), scan_rsp_data, sizeof(scan_rsp_data));
    at_ble_adv_start(AT_BLE_ADV_TYPE_UNDIRECTED, AT_BLE_ADV_GEN_DISCOVERABLE, NULL,
AT_BLE_ADV_FP_ANY, 100, 1000, 0);

while(at_ble_event_get(&event, params, -1) == AT_BLE_SUCCESS)
    {
        switch(event)
        {
            case AT_BLE_CONNECTED:
            {
                at_ble_connected_t* conn_params = (at_ble_connected_t*)params;
                printf("Device connected \n");
                handle = conn_params->handle;
            }
            break;

            case AT_BLE_DISCONNECTED:
            {
                printf("Device disconnected\n");
                at_ble_adv_start(AT_BLE_ADV_TYPE_UNDIRECTED,
                    AT_BLE_ADV_GEN_DISCOVERABLE,
                    NULL, AT_BLE_ADV_FP_ANY, 100, 1000, 0);

            }
            break;
```


2.3 Event Posting and Handling

Figure 1-2. Event Posting and Handling



Each event message returned by the controller is retrieved by calling the `API at_ble_event_get()` function. This is a blocking call and never returns the event message unless a new event is received from the controller, or event time out is reached the `at_ble_event_user_defined_post()` API is called. The purpose of the user defined event posting provides the flexibility to skip an iteration of the event handling loop, by sending a user defined event. This makes the blocking call to `at_ble_event_get` return with user event message ID. It is used when the user want to execute some code inside the event loop after handling a specific message from the controller, without the need to wait for a controller event that may occur at any time.

3. API Usage Examples

3.1 GAP Advertising

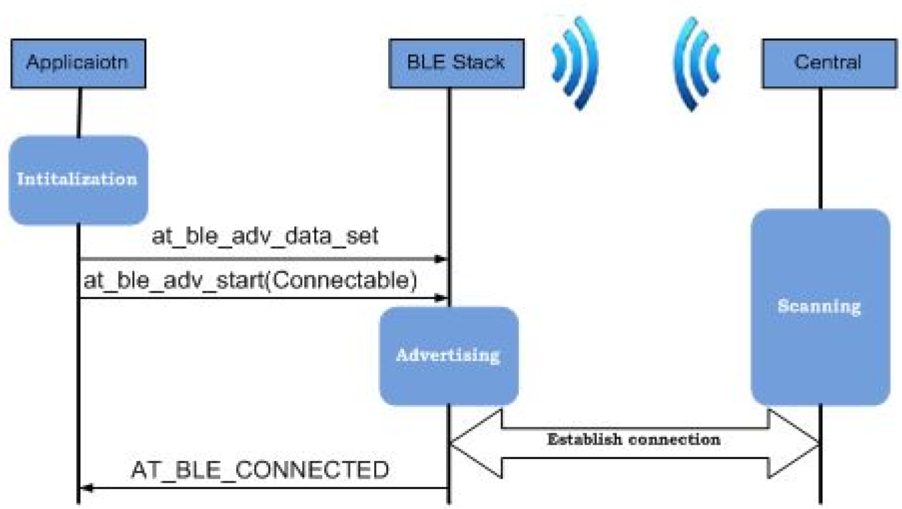
After initialization and setting address, to run device in peripheral role, it is required to advertise and in this case the device is called **Advertiser** or **Peripheral**.

Advertising data means that the peripheral sends unidirectional broadcast data on air to be discovered by other devices and react according to device capabilities such as advertising type, mode, and so on.

If it is needed to response to connection request from scanner devices, it is required to advertise in connectable mode. In addition to advertising capabilities, the advertising data can also include any custom information to broadcast to other devices.

Before advertising, it is required to set advertising data first using `at_ble_adv_data_set()`, also user can set additional user data called response data using the same function if needed, this data is sent to the active scanning device and request for more information.

Settings of advertising data must be done before start advertising. If the advertising is running, it must be stopped using `at_ble_adv_stop()` and apply settings of advertising data then start advertising again.



Example:

Device Address : `0x7f7f6a117525`

Advertising data length : `0x11`

AD type : Complete list of 128-bit UUIDs available `(0x07)`

Service UUID : `0x5730CD00DC2A11E3AA370002A5D5C51B`

```
#define DEVICE_NAME "BLE Device"

uint8_t adv_data[] = {  0x11, 0x07, 0x1b, 0xc5, 0xd5, 0xa5, 0x02, 0x00, 0x37,
0xaa, 0xe3, 0x11, 0x2a, 0xdc, 0x00, 0xcd, 0x30, 0x57};

static at_ble_handle_t service;
static at_ble_uuid_t service_uuid = {AT_BLE_UUID_128,
    {0x1b, 0xc5, 0xd5, 0xa5, 0x02, 0x00, 0x37, 0xaa,
    0xe3, 0x11, 0x2a, 0xdc, 0x00, 0xcd, 0x30, 0x57}};
```

```
at_ble_status_t init_peripheral_role(void)
{
    at_ble_status_t status;
    at_ble_addr_t addr = {AT_BLE_ADDRESS_PUBLIC,
            {0x25, 0x75, 0x11, 0x6a, 0x7f, 0x7f}};

    do
    {
        //Initializations of device
        status = at_ble_init(NULL);
        if(AT_BLE_SUCCESS == status)
        {
            break;
        }
        //Set device address
        if(AT_BLE_SUCCESS != at_ble_addr_set(&addr))
        {
            break;
        }
        //Set device name
        if(AT_BLE_SUCCESS != at_ble_device_name_set((uint8_t
*)DEVICE_NAME,                     sizeof(DEVICE_NAME)))
        {
            break;
        }
        //Establish peripheral database
        if(AT_BLE_SUCCESS != at_ble_primary_service_define(&service_uuid,
&service,             NULL, 0, chars, 2))
        {
            break;
        }
        //Set advertising data, instead of NULL set scan response data if needed
        if(AT_BLE_SUCCESS != at_ble_adv_data_set(adv_data, sizeof(adv_data), NULL, 0))
        {
            break;
        }
        //Start advertising
        if(AT_BLE_SUCCESS != at_ble_adv_start(AT_BLE_ADV_TYPE_UNDIRECTED,
AT_BLE_ADV_GEN_DISCOVERABLE, NULL, AT_BLE_ADV_FP_ANY, 100, 0, false))
        {
            break;
        }
    }while(1);

    return status;
}
```

3.2 GAP Scanning and Connection Creation

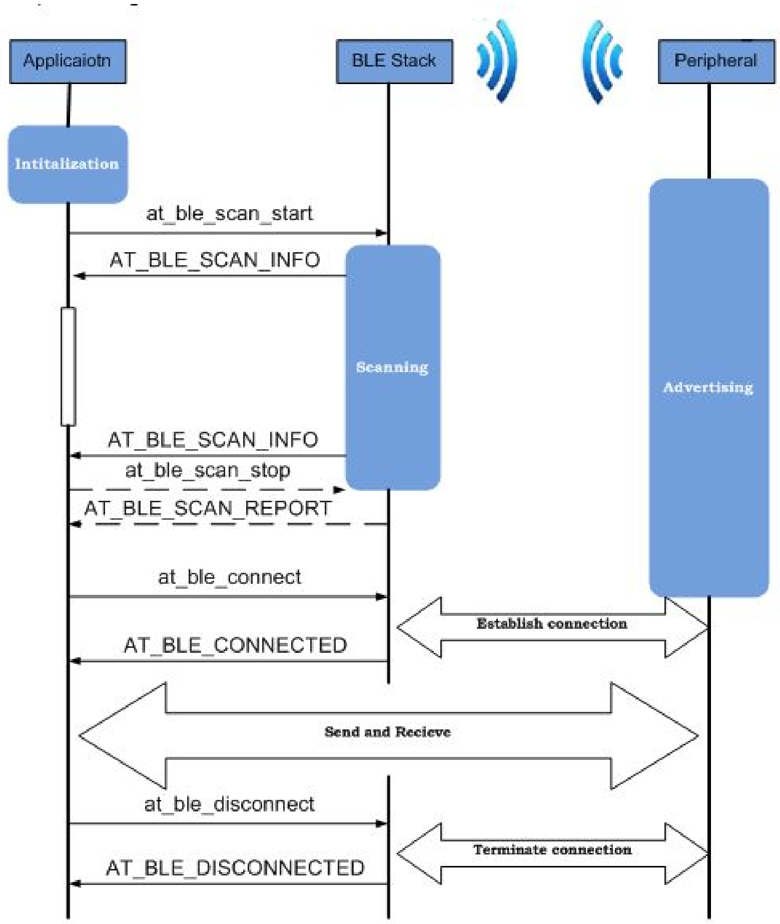
Device that scans for unidirectional broadcast advertising data is called as **Scanner** or **Central** and it uses `at_ble_scan_start()` to start scan with different configurations.

Central device request for more additional user data from the advertiser.

Application is triggered when receiving `AT_BLE_SCAN_INFO` event with each scan result. Also `AT_BLE_SCAN_REPORT` event is received in case of using `AT_BLE_SCAN_GEN_DISCOVERY` or `AT_BLE_SCAN_LIM_DISCOVERY`.

In `AT_BLE_SCAN_OBSERVER_MODE`, it is developer's responsibility to stop scan the operation using `at_ble_scan_stop()` In this mode, scanning is performed without timeout. Once a peer device is identified, stop the scanning process and initiates the connection request.

Figure 2-1. GAP Scanning and Connection Creation



Example: Code snippet of scanning

Device Address : `0x7f7f6a117525`

Peer Address : `0x001bdc060545`

```
#define DEVICE_NAME "BLE Device"

at_ble_addr_t addr = {AT_BLE_ADDRESS_PUBLIC,
    {0x24, 0x75, 0x11, 0x6a, 0x7f, 0x7f}};

at_ble_addr_t peer_addr = {AT_BLE_ADDRESS_PUBLIC,
    {0x45, 0x05, 0x06, 0xdc, 0x1b, 0x00}};

at_ble_status_t init_central_role(void)
{
    at_ble_status_t status = AT_BLE_SUCCESS;

    do
    {
        //Initiate device
        status = at_ble_init(NULL);
        if(AT_BLE_SUCCESS != status)
        {
            break;
        }
        //Set device name
        if(AT_BLE_SUCCESS != at_ble_device_name_set((uint8_t
*)DEVICE_NAME,                     sizeof(DEVICE_NAME)))
        {
            break;
        }
```

```
        //Set address
        at_ble_addr_set(&addr);
        if(AT_BLE_SUCCESS != status)
        {
            break;
        }
        //Start scan
        if(AT_BLE_SUCCESS != at_ble_scan_start(GAP_INQ_SCAN_INTV, GAP_INQ_SCAN_WIND,
0,              AT_BLE_SCAN_ACTIVE, AT_BLE_SCAN_GEN_DISCOVERY, FALSE, 1))
        {
            break;
        }

    } while (1);

    return status;
}

void main(void)
{
    at_ble_handle_t handle = -1;
    at_ble_scan_info_t* scan_params;
    at_ble_events_t at_event;
    uint8_t params[512];

    do
    {
        if(AT_BLE_SUCCESS != init_central_role())
        {
            printf("Unable to initialize\r\n");
            break;
        }
        printf("Scanning ...\r\n");

        while(AT_BLE_SUCCESS == at_ble_event_get(&at_event, params, -1))
        {
            switch(at_event)
            {
                case AT_BLE_SCAN_INFO:
                {
                    scan_params = (at_ble_scan_info_t*)params;
                    printf("Device Found 0x%02x%02x%02x%02x%02x%02x \n",
                    scan_params->dev_addr.addr[5],
                    scan_params->dev_addr.addr[4],
                    scan_params->dev_addr.addr[3],
                    scan_params->dev_addr.addr[2],
                    scan_params->dev_addr.addr[1],
                    scan_params->dev_addr.addr[0]
                    );
                    if((scan_params->type != AT_BLE_ADV_TYPE_SCAN_RESPONSE)&&
            !memcmp(scan_params->dev_addr.addr,peer_addr.addr,AT_BLE_ADDR_LEN))
                    {
                        at_ble_connection_params_t conn_params;
                        /* Stop Scan operation*/
                        at_ble_status_t status = at_ble_scan_stop();

                        if(status == AT_BLE_SUCCESS)
                        {
                            conn_params.ce_len_max = 0x0140;
                            conn_params.ce_len_min = 0x0000;
                            conn_params.con_intv_max = 0x00a0;
                            conn_params.con_intv_min = 0x00a0;
                            conn_params.con_latency = 0x0000;
                            conn_params.superv_to = 0xC80; //0x01f4;

                            /* Connect to peer device */
                            if(AT_BLE_SUCCESS != at_ble_connect(&peer_addr, 1,
GAP_INQ_SCAN_INTV, GAP_INQ_SCAN_WIND, &conn_params))
                            {
                                printf("Unable to connect\r\n");
                            }
                        }
                    }
                }
                break;
                case AT_BLE_CONNECTED:
                {
                    at_ble_connected_t* conn_params =
```

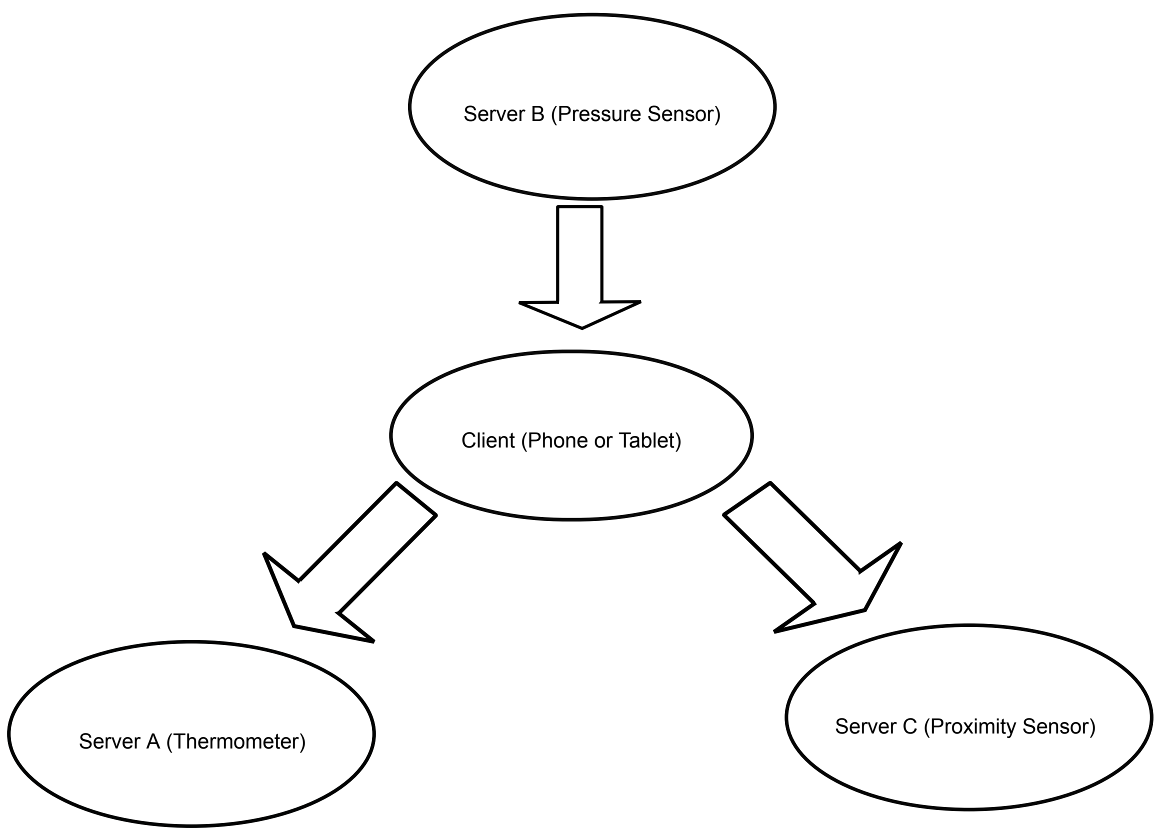
```
(at_ble_connected_t*)params;
                    handle = conn_params->handle;
                    printf("Device connected\r\n");
                    at_ble_disconnect(handle, AT_BLE_TERMINATED_BY_USER);
                }
                break;
                case AT_BLE_DISCONNECTED:
                {
                    printf("Device disconnected \n");
                }
                break;
            }
        }

    }while(1);
    while(1);
}
```

3.3 GATT Server – Service Definition

3.3.1 Introduction

Generic Attribute (GATT) Profile is an upper layer of the Bluetooth stack that defines how two connected Bluetooth devices can exchange information. It is based on the Attribute (ATT) Protocol, which "allows a device1 (server) to expose a set of attributes and their associated values to a device2 (peer device or client). These attributes are exposed by the server and it is discovered, read, and written by a client and is indicated and notified by the server" as per the standard.

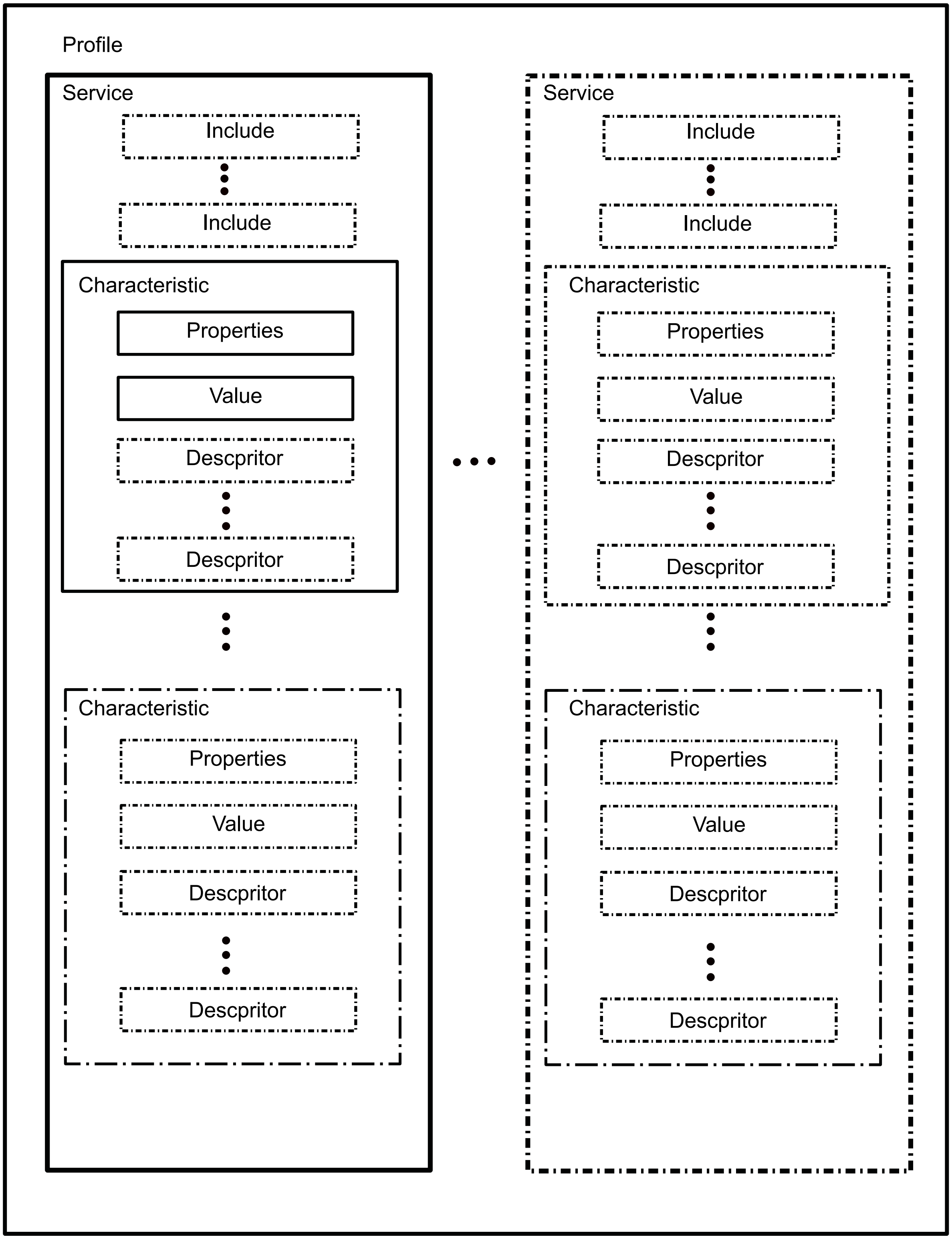
Figure 2-2. GATT Server Introduction



3.3.2 Services and Characteristics

The GATT profile defines a basic structure for data. Attributes are arranged in a hierarchal manner and profiles are available on top of the hierarchy. A profile is composed of a service and each service is composed of a set of characteristics. A service includes (link to) another services to encourage reusability. A characteristic has a value and contain extra descriptors that explain the characteristic format to the user.

Figure 2-3. Basic GATT hierarchy



3.3.3 Defining a Service

If a peer has defined a service with a set of characteristics, it implicitly gain the server role for any peer discovering these services.

To define a service:

- Service UUID `at_ble_uuid_t* uuid` and characteristics `at_ble_characteristic_t* charactristic_list` structures are properly filled.
- `at_ble_status_t` **`at_ble_primary_service_define`**`(at_ble_uuid_t* uuid, at_ble_handle_t* service_handle, at_ble_included_service_t* included_service_list, uint16_t included_service_count, at_ble_characteristic_t* charactristic_list, uint16_t charactristic_count)` are called with proper arguments which returns a handle to the service in `service_handle` and handle of its characteristics in the first field of the `charactristic_list` structure `charactristic_list[i].char_val_handle` returns handle of the first characteristic in the service, also handles to the client configuration, user descriptor, and server configuration is returned in `charactristic_list[i].client_config_handle, charactristic_list [i].user_desc_handle, charactristic_list[i].server_config_handle`, respectively.

Example code to define a service is given below.

```
            static at_ble_uuid_t service_uuid = {
              AT_BLE_UUID_128 ,
     { 0x1b, 0xc5, 0xd5, 0xa5, 0x02, 0x00, 0x37, 0xaa,
     0xe3, 0x11, 0x2a, 0xdc, 0x00, 0xcd, 0x30, 0x57}
};


      static at_ble_characteristic_t chars[] = {

       0, /* handle stored here */
       { AT_BLE_UUID_128, {0x1b, 0xc5, 0xd5, 0xa5, 0x02, 0x00, 0x3b, 0x8e,
       0xe3, 0x11, 0x2a, 0xdc, 0xa0, 0xd3, 0x20, 0x8e}}, /* UUID */
       AT_BLE_CHAR_READ | AT_BLE_CHAR_WRITE | AT_BLE_CHAR_NOTIFY, /* Properties */
       "char1", sizeof("char1"), 100, /* value */
             /*permissions */
       AT_BLE_ATTR_READABLE_NO_AUTHN_NO_AUTHR | AT_BLE_ATTR_WRITABLE_NO_AUTHN_NO_AUTHR,
             NULL, 0, 0, /* user friendly description */
             NULL, /*presentation format*/
       AT_BLE_ATTR_NO_PERMISSIONS, /*user description permissions*/
       AT_BLE_ATTR_READABLE_REQ_AUTHN_REQ_AUTHR, /*client config permissions*/
       AT_BLE_ATTR_NO_PERMISSIONS, /*server config permissions*/
       0,0,0    /*user desc, client config, and server config handles*/
    };

      static at_ble_handle_t service;
    // establish peripheral database
    at_ble_primary_service_define(&service_uuid, &service,
        NULL, 0, chars, 1);
```

Example code to define a service is given below.

```
      static at_ble_uuid_t service_uuid = {
              AT_BLE_UUID_128 ,
     { 0x1b, 0xc5, 0xd5, 0xa5, 0x02, 0x00, 0x37, 0xaa,
     0xe3, 0x11, 0x2a, 0xdc, 0x00, 0xcd, 0x30, 0x57}
};


      static at_ble_characteristic_t chars[] = {

       0, /* handle stored here */
       { AT_BLE_UUID_128, {0x1b, 0xc5, 0xd5, 0xa5, 0x02, 0x00, 0x3b, 0x8e,
       0xe3, 0x11, 0x2a, 0xdc, 0xa0, 0xd3, 0x20, 0x8e}}, /* UUID */
       AT_BLE_CHAR_READ | AT_BLE_CHAR_WRITE | AT_BLE_CHAR_NOTIFY, /* Properties */
       "char1", sizeof("char1"), 100, /* value */
             /*permissions */
       AT_BLE_ATTR_READABLE_NO_AUTHN_NO_AUTHR | AT_BLE_ATTR_WRITABLE_NO_AUTHN_NO_AUTHR,
```

```
            NULL, 0, 0, /* user friendly description */
            NULL, /*presentation format*/
        AT_BLE_ATTR_NO_PERMISSIONS, /*user description permissions*/
        AT_BLE_ATTR_READABLE_REQ_AUTHN_REQ_AUTHR, /*client config permissions*/
        AT_BLE_ATTR_NO_PERMISSIONS, /*server config permissions*/
        0,0,0    /*user desc, client config, and server config handles*/
    };

       static at_ble_handle_t service;
    // establish peripheral database
    at_ble_primary_service_define(&service_uuid, &service,
        NULL, 0, chars, 1);
```

3.3.4 Writing/Reading Characteristic Value

To write the value of a characteristic from the server:

```
at_ble_status_t at_ble_characteristic_value_set(at_ble_handle_t handle, uint8_t* value,
uint16_t offset, uint16_t len);
```

To read the value of a characteristic from the server:

```
at_ble_status_t at_ble_characteristic_value_get(at_ble_handle_t handle, uint8_t* value,
uint16_t offset, uint16_t len, uint16_t actual_read_len);
```

3.3.5 Sending Notifications/Indications to Client

If a client enables notifications/indications for a server, the server receives an `AT_BLE_CHARACTERISTIC_CHANGED` event , the handle returned in the characteristic changed event is compared with the client_config_handle `charactristic_list[i].client_config_handle`, If it matches then the new value returned in the characteristic changed event is checked for non zero value, then the server starts notifying/indicating the client using `at_ble_status_t at_ble_notification_send(at_ble_handle_t conn_handle, at_ble_handle_t attr_handle); or at_ble_status_t at_ble_indication_send(at_ble_handle_t conn_handle, at_ble_handle_t attr_handle);`

Example code of sending notifications/indications to client is given below.

```
case AT_BLE_CHARACTERISTIC_CHANGED:
{
    at_ble_characteristic_changed_t* change_params
            = (at_ble_characteristic_changed_t*) params;
    uint32_t i = 0;

    if (change_params->char_handle == client_config_handle)
    {

        switch (change_params->char_new_value)
        {
            case 1:
                at_ble_notification_send(handle, chars[0].char_val_handle);
                break;
            case 2:
                at_ble_indication_send(handle, chars[0].char_val_handle);
                break;
        }

    }

}
break;
```

3.4 GATT Client – Service Discovery

3.4.1 Discovering a Service

To discover services in a GATT server, any one of the following methods can be used:

- discover all services from a start handle to an end handle with the following functions:

```
at_ble_status_t at_ble_descriptor_discover_all(at_ble_handle_t conn_handle,
at_ble_handle_t start_handle, at_ble_handle_t end_handle);
```

- discover a specific service using its UUID with the following functions:

```
at_ble_status_t at_ble_characteristic_discover_by_uuid(at_ble_handle_t conn_handle,
at_ble_handle_t start_handle, at_ble_handle_t end_handle, at_ble_uuid_t* uuid);
```

In both cases two events returned and handled by the developer. `AT_BLE_DISCOVERY_COMPLETE` returns the status of the operation and `AT_BLE_PRIMARY_SERVICE_FOUND` is sent to the application whenever a service is found.

```
case AT_BLE_PRIMARY_SERVICE_FOUND:
        {
            at_ble_primary_service_found_t * primary_service =
                            (at_ble_primary_ser vice_found_t *) params;

            printf("Primary Service UUID: Type:%02x Value:%04x \t Start Handle:%04x \t End
Handle:%04x\n", primary_service->service_uuid.type,
            (uint16_t)((uint16_t)primary_service->service_uuid.uuid[0]
                     |    ((uint16_t)primary_service->service_uuid.uuid[1]<<8)),
            primary_service->start_handle, primary_service->end_handle);
        }
        break;
```

Once a primary service is found, based on its start and end handle, all characteristics of such primary service is found by calling function as explained below.

```
at_ble_status_t at_ble_characteristic_discover_all(at_ble_handle_t conn_handle,
        at_ble_handle_t start_handle, at_ble_handle_t end_handle);
Event AT_BLE_CHARACTERISTIC_FOUND will return the characteristics found.

        case AT_BLE_CHARACTERISTIC_FOUND:
        {
            at_ble_characteristic_found_t * characteristic =
                            (at_ble_ characteristic_found_t *) params;

            printf("Characteristic UUID: Type:%02x Value:%04x \t Char Handle:%04x \t Value
Handle:%04x, Properties:%02x\n", characteristic->char_uuid.type,
            (uint16_t)((uint16_t) characteristic->char_uuid.uuid[0]
                     |   ((uint16_t) characteristic->char_uuid.uuid[1]<<8)),
            characteristic->char_handle, characteristic->value_handle,
               characteristic->properties);
        }
    break;
```

3.4.2 Writing/Reading Characteristic Value

To write the value of a characteristic from the client:

```
at_ble_status_t at_ble_characteristic_write(at_ble_handle_t conn_handle, at_ble_handle_t
char_handle, uint16_t offset, uint16_t length, uint8_t* data, bool signed_write, bool
with_response );
```

Then an event `AT_BLE_CHARACTERISTIC_WRITE_RESPONSE` is sent to client that indicates the write status.

To read the value of a characteristic from the client:

```
at_ble_status_t at_ble_characteristic_read(at_ble_handle_t conn_handle, at_ble_handle_t
char_handle, uint16_t offset, uint16_t len);
```

The read data is sent to the client through an `AT_BLE_CHARACTERISTIC_READ_RESPONSE` event.

```
case AT_BLE_CHARACTERISTIC_READ_RESPONSE:
{
    at_ble_characteristic_read_response_t *read_resp =
                    (at_ble_characteristic_read_response_t *)params;
    uint32_t i=0;

    printf("READ RESPONSE: Characteristic Handle:%04x \t Length:%04x Offset:%04x\n",
                read_resp->char_handle,
                read_resp->char_len,
                read_resp->char_offset);
    printf("DATA:\t");

    for(i=0;i<read_resp->char_len;i++)
    {
        printf("%02x ", read_resp->char_value[i]);
    }
    printf("\n");

}
break;
```

3.5 Security Example

The purpose of the bonding procedure is to create a relation between two Bluetooth devices based on a common link key (a bond), the link key is created and exchanged during pairing procedure and is expected to be stored by both Bluetooth device that is used during another connection to avoid repeating pairing procedure.

Security is initiated by the device in the master role. The device in the slave role accepts the request and act as a responding device.The slave device request the master device to initiate pairing or other security procedures.

3.5.1 Pairing Procedure

Pairing is a three-phase process. The first two phases are used and followed by an optional transport specific key distribution phase, to share the keys which is used to encrypt a link in future reconnections, verify signed data, and perform random address resolution.

Phase 1: Pairing Feature Exchange

The devices first exchange IO capabilities, OOB “Out of Band” authentication data availability, authentication requirements, key size requirements and which transport specific keys to distribute in the pairing feature exchange.

IO Capabilities

- `AT_BLE_IO_CAP_DISPLAY_ONLY` – display only
- `AT_BLE_IO_CAP_DISPLAY_YES_NO` – can display and get a Yes/No input from user
- `AT_BLE_IO_CAP_KB_ONLY` – has only a keyboard
- `AT_BLE_IO_CAP_NO_INPUT_NO_OUTPUT` – has no input and no output
- `AT_BLE_IO_CAP_KB_DISPLAY` – has both a display and a keyboard

Authentication Requirements

The authentication requirements include the type of bonding and man-in-the-middle protection (MITM) requirements:

- Bonding – if no key is exchanged during the pairing, the bonding flag is set to zero.
- Man in the Middle protection (MITM) flag – according to the IO capabilities or Out Of Band (OOB) property, MITM flag is set to zero, if pairing is done using PIN code or OOB data.

Note: The link is considered authenticated by using the passkey entry pairing method (MITM) or by using the out of band pairing method.

Security Modes

Security requirement is used to force a certain level of authentication and presence of key exchange.

- LE Security mode 1 has three security levels:

1. `AT_BLE_NO_SEC` (no authentication and no encryption).
2. `AT_BLE_MODE1_L1_NOAUTH_PAIR_ENC` (unauthenticated pairing with encryption)
   Man in the middle protection is set to zero and Long Term Key (LTK) is exchanged
3. `AT_BLE_MODE1_L2_AUTH_PAIR_ENC` (authenticated pairing with encryption)
   Man in the middle protection shall be set to 1, a LTK is exchanged

- LE Security mode 2

1. `AT_BLE_MODE2_L1_NOAUTH_DATA_SGN` (unauthenticated pairing with data signing)
   Man in the middle protection is set to zero, a CSRK is exchanged.
2. `AT_BLE_MODE2_L2_AUTH_DATA_SGN` (authenticated pairing with data signing)
   Man in the middle protection is set to 1, a CSRK is exchanged.

Key Distribution

The initiating device indicates that the specific keys are transporting into the responding device and vice versa.

- `AT_BLE_KEY_DIST_ENC` – distribute Long Term Key (LTK), Encrypted Diversifier (EDIV), and random number
- `AT_BLE_KEY_DIST_SIGN` – distribute Connection signature key (CSRK)
- `AT_BLE_KEY_DIST_ID` – distribute Identity Resolving Key (IRK) and identity address
- `AT_BLE_KEY_DIS_ALL` – distribute all keys

The IO capabilities, OOB authentication data availability, and authentication requirements are used to determine the pairing method and it is used in Short Term Key (STK) generation in phase 2. Supported pairing methods are as follows::

- Just Works
- Passkey Entry
- Out Of Band (OOB)

All these pairing methods use and generate 2 keys:

- Temporary Key (TK) – a 128-bit temporary key is used in the pairing process, it is a key exchanged by out of band system such as NFC, or the pin code entered by user during Just Works pairing; this key is set to zero.
- Short Term Key (STK) – a 128-bit temporary key is used to encrypt a connection followed by pairing.

Phase 2: Short Term Key (STK) Generation

Calculated according to pairing information and provided TK, it is used to encrypt the link during pairing to exchange the following keys:

- Long Term Key (LTK) – is a 128-bit key used to encrypt the Link. In order to retrieve link key, a random number and key diversifier has to be stored with this key.
- Encrypted Diversifier (EDIV) – a 16-bit stored value used to identify the LTK. A new EDIV is generated each time a unique LTK is distributed.
- Random Number (Rand) – a 64-bit stored value that is used for identifying the LTK, A new Rand is generated each time a unique LTK is distributed.
- Identity Resolving Key (IRK) – a 128-bit key is used to generate and random address.
- Connection signature key (CSRK) – when link is not encrypted, the CSRK is used by GAP to sign and verify signature of an attribute write sign.

Phase 3: Transport Specific Key Distribution

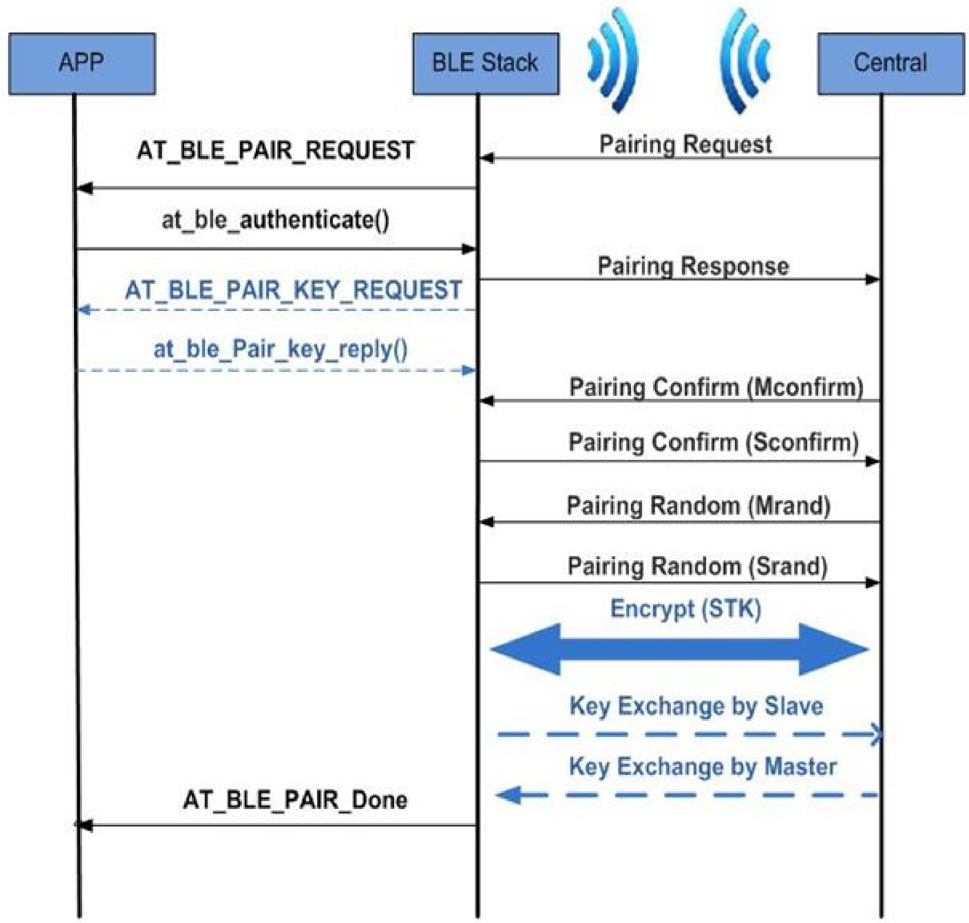
Application APIs Interface

- `at_ble_authenticate`

  `at_ble_send_slave_sec_request` APIs are used for initiating bonding and responding to pairing request from remote device.
- `AT_BLE_PAIR_KEY_REQUEST`

  `AT_BLE_SLAVE_SEC_REQUEST` events are triggered to indicate that bonding is required.
- `AT_BLE_PAIR_DONE` event is triggered to indicate bonding status.

Figure 2-4. Pairing Sequence Flow



3.5.2 Encryption Procedure

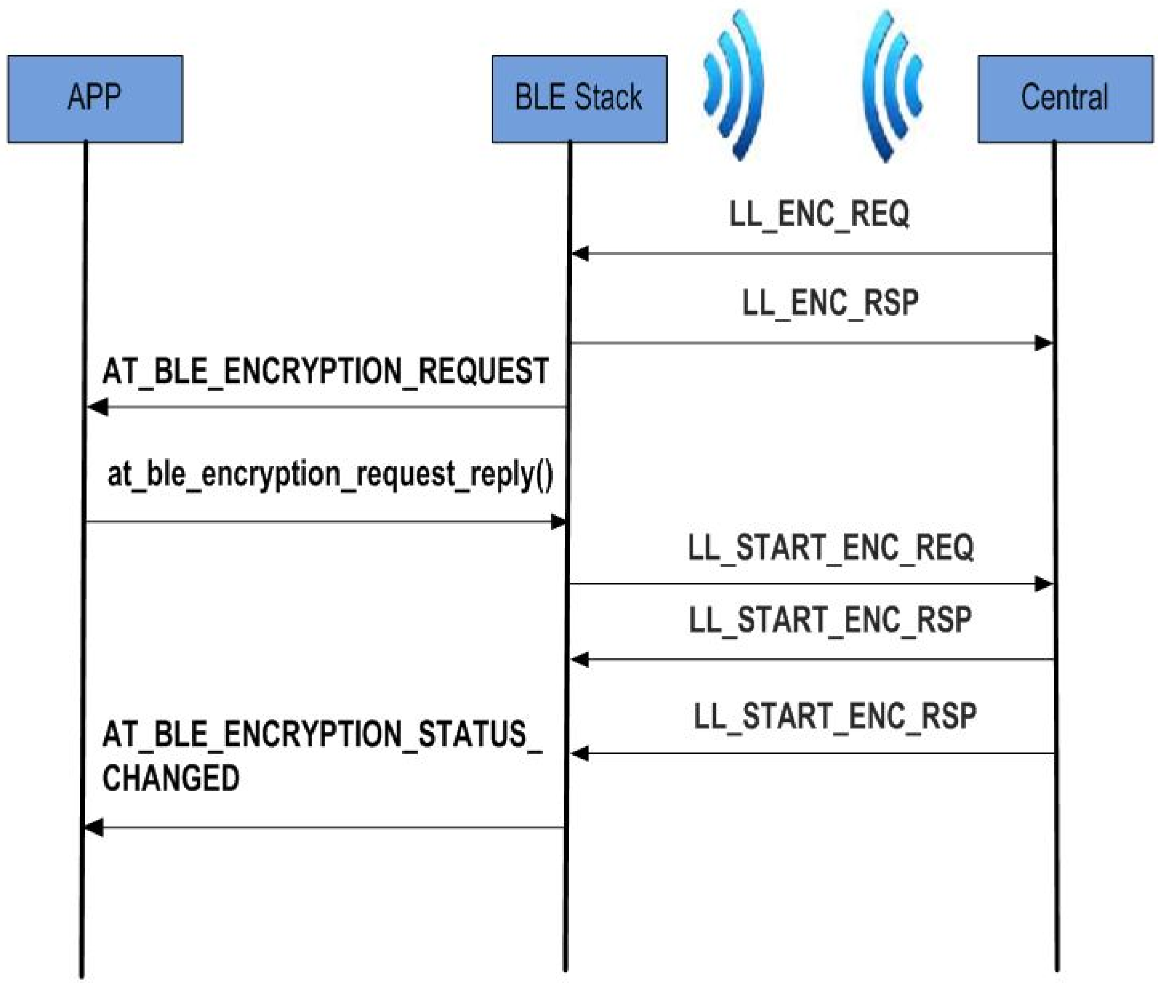
The encryption procedure is used to encrypt the link using a previously bonded Long Term Key (LTK). This procedure is initiated only by the master device.

During the encryption session setup, the master device sends a 16-bit EDIV, and a 64-bit Rand, distributed by the slave device during pairing, to the slave device.

The master's host provides the link layer with the LTK to use when setting up the encrypted session.

The slave's host receives the EDIV and Rand values and provides a LTK to the slave's link layer to use when setting up the encrypted link.

Figure 2-5. Encryption Sequence Flow



Example of the encryption procedure code is given below.

```
#define PRINT(...)              printf(__VA_ARGS__)
#define PRINT_LOG(...)          printf("[APP]"/**/__VA_ARGS__)

at_ble_LTK_t app_bond_info;
at_ble_auth_t auth_info;

void main(void)
{
    ...
    //Init
    ...

    while(at_ble_event_get(&event, params, -1) == AT_BLE_SUCCESS)
    {
        switch(event)
        {
            case AT_BLE_PAIR_REQUEST:
            {
                at_ble_pair_features_t features;
                uint8_t loopCntr;

                PRINT_LOG("Remote device request pairing \n");
                /* Authentication requirement is bond and MITM*/
                features.desired_auth =  AT_BLE_MODE1_L2_AUTH_PAIR_ENC;
                features.bond = TRUE;
                features.mitm_protection = TRUE;
```

```
                features.oob_avaiable = FALSE;
                /* Device cababilities is display only , key will be generated
                and displayed */
                features.io_cababilities = AT_BLE_IO_CAP_DISPLAY_ONLY;
                /* Distribution of LTK is required */
                features.initiator_keys =   AT_BLE_KEY_DIS_ALL;
                features.responder_keys =   AT_BLE_KEY_DIS_ALL;
                features.max_key_size = 16;
                features.min_key_size = 16;

                /* Generate LTK */
                for(loopCntr=0; loopCntr<8; loopCntr++)
                {
                    app_bond_info.key[loopCntr] = rand()&0x0f;
                    app_bond_info.nb[loopCntr] = rand()&0x0f;

                    for(loopCntr=8; loopCntr<16; loopCntr++)
                    {
                        app_bond_info.key[i] = rand()&0x0f;
                    }

                    app_bond_info.ediv = rand()&0xffff;
                    app_bond_info.key_size = 16;
                    /* Send pairing response */
                    if(AT_BLE_SUCCESS != at_ble_authenticate(handle,
&features,                              &app_bond_info, NULL))
                    {
                        PRINT("Unable to authenticate\r\n");
                    }
                }
            }
            break;
            case AT_BLE_PAIR_KEY_REQUEST:
            {
                /* Passkey has fixed ASCII value in this example MSB */
                uint8_t passkey[6]={'0','0','0','0','0','0'};
                uint8_t passkey_ascii[6];
                uint8_t loopCntr = 0;

                at_ble_pair_key_request_t* pair_key_request
=                                    (at_ble_pair_key_request_t*)params;

                /* Passkey is required to be generated by application and displayed to
user               */
                if(pair_key_request->passkey_type == AT_BLE_PAIR_PASSKEY_DISPLAY)
                {
                    PRINT_LOG("Enter the following code on the other device: ");
                    for(loopCntr=0; loopCntr<AT_BLE_PASSKEY_LEN; loopCntr++)
                    {
                        PRINT("%c",passkey_ascii[loopCntr]);
                    }
                    PRINT("\n");
                    if(AT_BLE_SUCCESS !=
at_ble_pair_key_reply(                                    pair_key_request->handle,
                        pair_key_request->type, passkey_ascii))
                    {
                        PRINT("Unable to pair reply\r\n");
                    }
                }
                else
                {
                    PRINT_LOG("AT_BLE_PAIR_PASSKEY_ENTRY\r\n");
                }
            }
            break;
            case AT_BLE_PAIR_DONE:
            {
                at_ble_pair_done_t* pair_params = (at_ble_pair_done_t*) params;
                if(pair_params->status == AT_BLE_SUCCESS)
                {
                    PRINT_LOG("Pairing procedure completed successfully\r\n");
                    auth_info = pair_params->auth;
                }
                else
                {
                    PRINT_LOG("Pairing failed\r\n");
                }
            }
```

```
            break;
            case AT_BLE_ENCRYPTION_REQUEST:
            {
                bool key_found = FALSE;
                at_ble_encryption_request_t *enc_req = (at_ble_encryption_request_t* )params;
                PRINT_LOG("Encrypting the connection...\r\n");
                /* Check if bond information is stored */
                if((enc_req-> ediv == app_bond_info.ediv)
                    && !memcmp(&enc_req->nb[0],&app_bond_info.nb[0],8))
                {
                    key_found = TRUE;
                }
                if(AT_BLE_SUCCESS != at_ble_encryption_request_reply(handle,
auth_info,                         key_found, app_bond_info))
                {
                    PRINT("Unable to send Encryption request\r\n");
                }
            }
            break;
            case AT_BLE_ENCRYPTION_STATUS_CHANGED:
            {
                at_ble_encryption_status_changed_t *enc_status
=                                         (at_ble_encryption_status_changed_t *)params;
                if(enc_status->status == AT_BLE_SUCCESS)
                {
                    PRINT_LOG("Encryption completed successfully\r\n");
                }
                else
                {
                    PRINT_LOG("Encryption failed\r\n");
                }
            }
            break;
        }
    }
}
```

4. RTC XO 32.768kHz Clock Output

This section guides the user to enable the clock output of the RTC XO 32.768 kHz.

Add the below code snippet of function definition to import the API to write to registers of ATBTLC1000. This function definition is added to the file where the clock output is intended to be enabled.

```
void write_32_to_BTLC1000(uint32_t u32address, uint32_t u32value);
```

After a successful initialization of ATBTLC1000 through `at_ble_init()`, the following code snippet is introduced to enable the clock output to pin LP_GPIO_10,

```
       uint32_t val;
       read_32_from_BTLC1000(0X4000F404, &val);
       val |= (0b0<<20); // Bits 20-23 control the value of internal tuning capacitors. Valid
value – 0b0000 to 0b1111
       write_32_to_BTLC1000(0X4000F404,val);

       //32.768kHz RTC XO clock output = 14
       write_32_to_BTLC1000(0x40020250, 14);

       read_32_from_BTLC1000(0x4000b048, &val);
       //MUX7(Test out 10) configured for LP_GPIO_10
       val |=0x7<<8;
       write_32_to_BTLC1000(0x4000b048, val);

       //Enable test MUX output
       write_32_to_BTLC1000(0x400201a0, 1);

       //Block BTLC1000 from entering ULP
       platform_gpio_set(AT_BLE_EXTERNAL_WAKEUP, AT_BLE_HIGH);
```

Based on the frequency of the clock output, either external load capacitor value is tuned or internal tuning capacitor is tuned to achieve the 32.768kHz clock. The internal tuning capacitor value is adjusted by writing to bits 20-23 of register with address `0X4000F404`. The valid values that is written to these bits vary from `0b0000` to `0b1111`. The above code snippet writes `0b0000` by default and this must be changed when the user needs to write a different value to these bits.

4.1 Internal tuning capacitor configuration

Internal tuning capacitor is tuned in the design, the value that is written to bits 20-23 of register with address `0X4000F404` must be finalized. This finalized value is stored in NVM of the host MCU and this value must be loaded to ATBTLC1000 during the application startup by the host MCU.

The below code snippet is reused for writing to bits 20-23 of register with address `0X4000F404` from the host MCU.

```
uint32_t val;

       read_32_from_BTLC1000(0X4000F404, &val);
       val |= (0bXXXX<<20); // Bits 20-23 control the value of internal tuning capacitors.
Valid value – 0b0000 to 0b1111
       write_32_to_BTLC1000(0X4000F404,val);
```

The Microchip Web Site

Microchip provides online support via our web site at http://www.microchip.com/. This web site is used as a means to make files and information easily available to customers. Accessible by using your favorite Internet browser, the web site contains the following information:

- **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- **General Technical Support** – Frequently Asked Questions (FAQ), technical support requests, online discussion groups, Microchip consultant program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

Customer Change Notification Service

Microchip's customer notification service helps keep customers current on Microchip products. Subscribers will receive e-mail notification whenever there are changes, updates, revisions or errata related to a specified product family or development tool of interest.

To register, access the Microchip web site at http://www.microchip.com/. Under "Support", click on "Customer Change Notification" and follow the registration instructions.

Customer Support

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Field Application Engineer (FAE)
- Technical Support

Customers should contact their distributor, representative or Field Application Engineer (FAE) for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in the back of this document.

Technical support is available through the web site at: http://www.microchip.com/support

Microchip Devices Code Protection Feature

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.

- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as “unbreakable.”

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip’s code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Legal Notice

Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION, INCLUDING BUT NOT LIMITED TO ITS CONDITION, QUALITY, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR PURPOSE. Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer’s risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights unless otherwise stated.

Trademarks

The Microchip name and logo, the Microchip logo, AnyRate, AVR, AVR logo, AVR Freaks, BeaconThings, BitCloud, CryptoMemory, CryptoRF, dsPIC, FlashFlex, flexPWR, Heldo, JukeBlox, KeeLoq, KeeLoq logo, Kleer, LANCheck, LINK MD, maXStylus, maXTouch, MediaLB, megaAVR, MOST, MOST logo, MPLAB, OptoLyzer, PIC, picoPower, PICSTART, PIC32 logo, Prochip Designer, QTouch, RightTouch, SAM-BA, SpyNIC, SST, SST Logo, SuperFlash, tinyAVR, UNI/O, and XMEGA are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

ClockWorks, The Embedded Control Solutions Company, EtherSynch, Hyper Speed Control, HyperLight Load, IntelliMOS, mTouch, Precision Edge, and Quiet-Wire are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Adjacent Key Suppression, AKS, Analog-for-the-Digital Age, Any Capacitor, AnyIn, AnyOut, BodyCom, chipKIT, chipKIT logo, CodeGuard, CryptoAuthentication, CryptoCompanion, CryptoController, dsPICDEM, dsPICDEM.net, Dynamic Average Matching, DAM, ECAN, EtherGREEN, In-Circuit Serial Programming, ICSP, Inter-Chip Connectivity, JitterBlocker, KleerNet, KleerNet logo, Mindi, MiWi, motorBench, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, MultiTRAK, NetDetach, Omniscient Code Generation, PICDEM, PICDEM.net, PICkit, PICtail, PureSilicon, QMatrix, RightTouch logo, REAL ICE, Ripple Blocker, SAM-ICE, Serial Quad I/O, SMART-I.S., SQI, SuperSwitcher, SuperSwitcher II, Total Endurance, TSHARC, USBCheck, VariSense, ViewSpan, WiperLock, Wireless DNA, and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

Silicon Storage Technology is a registered trademark of Microchip Technology Inc. in other countries.

GestIC is a registered trademark of Microchip Technology Germany II GmbH & Co. KG, a subsidiary of Microchip Technology Inc., in other countries.

All other trademarks mentioned herein are property of their respective companies.

ISBN: 978-1-5224-1889-4

Quality Management System Certified by DNV

ISO/TS 16949

Microchip received ISO/TS-16949:2009 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

Worldwide Sales and Service

AMERICAS	ASIA/PACIFIC	ASIA/PACIFIC	EUROPE
Corporate Office 2355 West Chandler Blvd. Chandler, AZ 85224-6199 Tel: 480-792-7200 Fax: 480-792-7277 Technical Support: http://www.microchip.com/support Web Address: www.microchip.com **Atlanta** Duluth, GA Tel: 678-957-9614 Fax: 678-957-1455 **Austin, TX** Tel: 512-257-3370 **Boston** Westborough, MA Tel: 774-760-0087 Fax: 774-760-0088 **Chicago** Itasca, IL Tel: 630-285-0071 Fax: 630-285-0075 **Dallas** Addison, TX Tel: 972-818-7423 Fax: 972-818-2924 **Detroit** Novi, MI Tel: 248-848-4000 **Houston, TX** Tel: 281-894-5983 **Indianapolis** Noblesville, IN Tel: 317-773-8323 Fax: 317-773-5453 Tel: 317-536-2380 **Los Angeles** Mission Viejo, CA Tel: 949-462-9523 Fax: 949-462-9608 Tel: 951-273-7800 **Raleigh, NC** Tel: 919-844-7510 **New York, NY** Tel: 631-435-6000 **San Jose, CA** Tel: 408-735-9110 Tel: 408-436-4270 **Canada - Toronto** Tel: 905-695-1980 Fax: 905-695-2078	**Asia Pacific Office** Suites 3707-14, 37th Floor Tower 6, The Gateway Harbour City, Kowloon **Hong Kong** Tel: 852-2943-5100 Fax: 852-2401-3431 **Australia - Sydney** Tel: 61-2-9868-6733 Fax: 61-2-9868-6755 **China - Beijing** Tel: 86-10-8569-7000 Fax: 86-10-8528-2104 **China - Chengdu** Tel: 86-28-8665-5511 Fax: 86-28-8665-7889 **China - Chongqing** Tel: 86-23-8980-9588 Fax: 86-23-8980-9500 **China - Dongguan** Tel: 86-769-8702-9880 **China - Guangzhou** Tel: 86-20-8755-8029 **China - Hangzhou** Tel: 86-571-8792-8115 Fax: 86-571-8792-8116 **China - Hong Kong SAR** Tel: 852-2943-5100 Fax: 852-2401-3431 **China - Nanjing** Tel: 86-25-8473-2460 Fax: 86-25-8473-2470 **China - Qingdao** Tel: 86-532-8502-7355 Fax: 86-532-8502-7205 **China - Shanghai** Tel: 86-21-3326-8000 Fax: 86-21-3326-8021 **China - Shenyang** Tel: 86-24-2334-2829 Fax: 86-24-2334-2393 **China - Shenzhen** Tel: 86-755-8864-2200 Fax: 86-755-8203-1760 **China - Wuhan** Tel: 86-27-5980-5300 Fax: 86-27-5980-5118 **China - Xian** Tel: 86-29-8833-7252 Fax: 86-29-8833-7256	**China - Xiamen** Tel: 86-592-2388138 Fax: 86-592-2388130 **China - Zhuhai** Tel: 86-756-3210040 Fax: 86-756-3210049 **India - Bangalore** Tel: 91-80-3090-4444 Fax: 91-80-3090-4123 **India - New Delhi** Tel: 91-11-4160-8631 Fax: 91-11-4160-8632 **India - Pune** Tel: 91-20-3019-1500 **Japan - Osaka** Tel: 81-6-6152-7160 Fax: 81-6-6152-9310 **Japan - Tokyo** Tel: 81-3-6880- 3770 Fax: 81-3-6880-3771 **Korea - Daegu** Tel: 82-53-744-4301 Fax: 82-53-744-4302 **Korea - Seoul** Tel: 82-2-554-7200 Fax: 82-2-558-5932 or 82-2-558-5934 **Malaysia - Kuala Lumpur** Tel: 60-3-6201-9857 Fax: 60-3-6201-9859 **Malaysia - Penang** Tel: 60-4-227-8870 Fax: 60-4-227-4068 **Philippines - Manila** Tel: 63-2-634-9065 Fax: 63-2-634-9069 **Singapore** Tel: 65-6334-8870 Fax: 65-6334-8850 **Taiwan - Hsin Chu** Tel: 886-3-5778-366 Fax: 886-3-5770-955 **Taiwan - Kaohsiung** Tel: 886-7-213-7830 **Taiwan - Taipei** Tel: 886-2-2508-8600 Fax: 886-2-2508-0102 **Thailand - Bangkok** Tel: 66-2-694-1351 Fax: 66-2-694-1350	**Austria - Wels** Tel: 43-7242-2244-39 Fax: 43-7242-2244-393 **Denmark - Copenhagen** Tel: 45-4450-2828 Fax: 45-4485-2829 **Finland - Espoo** Tel: 358-9-4520-820 **France - Paris** Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79 **France - Saint Cloud** Tel: 33-1-30-60-70-00 **Germany - Garching** Tel: 49-8931-9700 **Germany - Haan** Tel: 49-2129-3766400 **Germany - Heilbronn** Tel: 49-7131-67-3636 **Germany - Karlsruhe** Tel: 49-721-625370 **Germany - Munich** Tel: 49-89-627-144-0 Fax: 49-89-627-144-44 **Germany - Rosenheim** Tel: 49-8031-354-560 **Israel - Ra'anana** Tel: 972-9-744-7705 **Italy - Milan** Tel: 39-0331-742611 Fax: 39-0331-466781 **Italy - Padova** Tel: 39-049-7625286 **Netherlands - Drunen** Tel: 31-416-690399 Fax: 31-416-690340 **Norway - Trondheim** Tel: 47-7289-7561 **Poland - Warsaw** Tel: 48-22-3325737 **Romania - Bucharest** Tel: 40-21-407-87-50 **Spain - Madrid** Tel: 34-91-708-08-90 Fax: 34-91-708-08-91 **Sweden - Gothenberg** Tel: 46-31-704-60-40 **Sweden - Stockholm** Tel: 46-8-5090-4654 **UK - Wokingham** Tel: 44-118-921-5800 Fax: 44-118-921-5820