ATBTLC1000 BluSDK

Bluetooth Low Energy API: Software Development

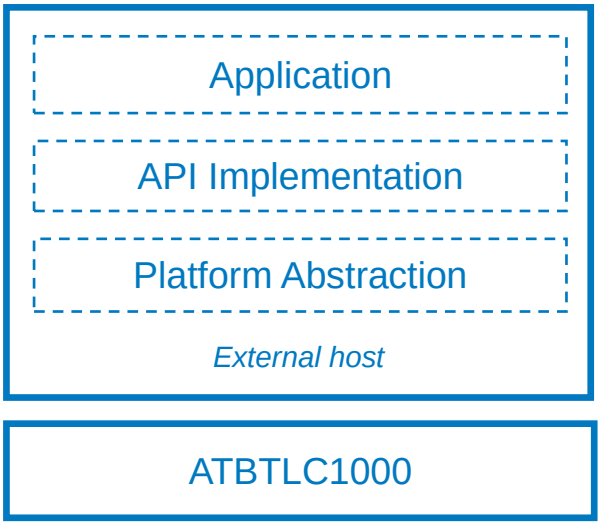
USER GUIDE

Description

This document describes the functional description of Atmel® Adapter API programming model and use cases for ATBTLC1000.

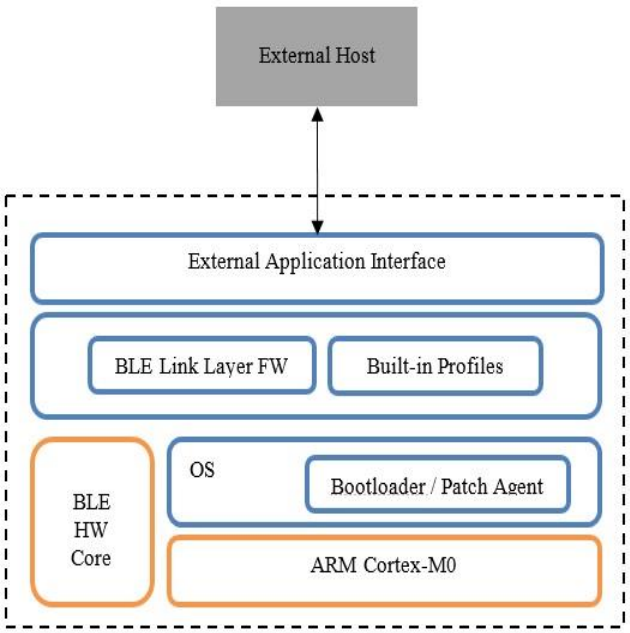
Table of Contents

1 Overview



The BTLC1000 provides Bluetooth Smart Link Controller that includes RF, Link Layer, GAP, GATT, and SMP in a single SOC. It provides the host microcontroller with methods to perform standard Bluetooth Smart GAP, GATT server and client operations as well as security management with peer devices.

The BTLC1000 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 SIG standard 4.1.



1.1 BTLC1000 Solution Architecture

The BTLC1000 solution is mainly composed of two sub-systems running concurrently.

- Link Controller that implements up to GATT and GAP layers
- A Host controller running Atmel Adaptation API layer that maps the GAP/GATT functionalities into their respective messages that need to be fed into the link controller via the serial interface.

This Document describes in details how to get started with the Atmel Adapter APIs to drive the BTLC1000 and make best use of its capabilities.

2 API Programming Model

This section describes what the typical app for the BTLC1000 using the APIs should look like. Typically the app should contain three groups of operations:

- Platform initialization/Link Controller Initialization
- Device Configuration
- Event Monitoring and handling

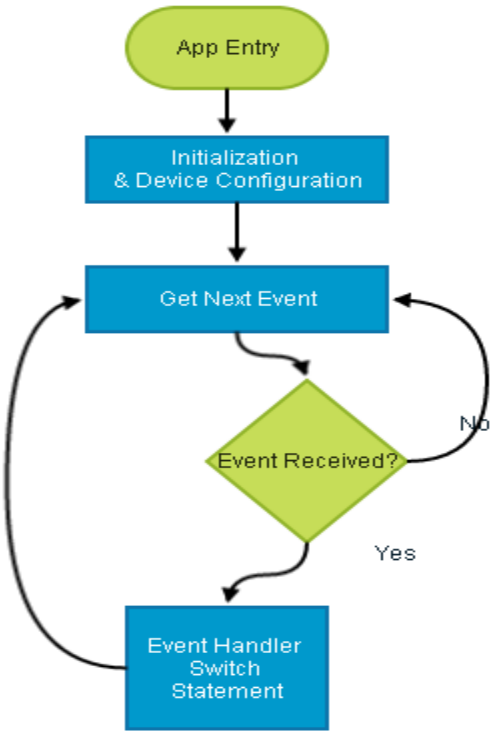
In the following section, we shall give more in depth info about each of these groups

2.1 General Application Flow

The General app flow would include initialization part. It is responsible for initializing the link controller and bus initialization. It should include a call to the function `at_ble_init`();

The device configuration includes setting up the device address, device name, and device advertising data. This is particularly important as most of the device configuration API calls has no Event messages associated with them, so, they have to be called at the start of the app and return error code has to be checked for successful operation

This diagram shows a simple flow chart of what the app should look like:



2.2 Request Response Flow

The API operation relies on a request – response mechanism. The request is sent via the dedicated API. Each API call may trigger one or more Event messages to be returned to the app. These event messages are handled in what is called the event handler loop, a major part of the user app. Each time the developer makes a call, it needs to handle the resulting messages coming back from the controller side. For example, if the user call at_ble_scan_start(), user should expect the controller to return an event with AT_BLE_SCAN_INFO for each device scanned by BTLC1000.

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

```
#include "platform.h"
#include "at_ble_api.h"

#define CHECK_ERROR(VAR,LABEL)     if(AT_BLE_SUCCESS != VAR) \
                                   { \
                                          goto LABEL; \
                                   }

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

static uint8_t adv_data[] = {0x1a, 0xff, 0x4c, 0x00, 0x02, 0x15, 0x21, 0x8A,
                             0xF6, 0x52, 0x73, 0xE3, 0x40, 0xB3, 0xB4, 0x1C,
                             0x19, 0x53, 0x24, 0x2C, 0x72, 0xf4, 0x00, 0xbb,
                             0x00, 0x44, 0xc5};
static uint8_t scan_rsp_data[] = {0x11, 0x07, 0x1b, 0xc5, 0xd5, 0xa5, 0x02, 0x00,
                                  0x37, 0xaa, 0xe3, 0x11, 0x2a, 0xdc, 0x00, 0xcd,
                                  0x30, 0x57};

int main (void)
{
       at_ble_events_t event;
       uint8_t params[512];
       at_ble_status_t enuStatus;
       at_ble_addr_t addr = {AT_BLE_ADDRESS_PUBLIC, {0xAB, 0xCD, 0xEF, 0xAB, 0xCD, 0xEF}};
       at_ble_init_config_t pf_cfg;
       platform_config busConfig;

       at_ble_handle_t gHandle;

       /*Memory allocation required by GATT Server DB*/
       pf_cfg.memPool.memSize = 0;
       pf_cfg.memPool.memStartAdd = NULL;
       /*Bus configuration*/
       busConfig.bus_type = UART;
       pf_cfg.plf_config = &busConfig;

       PRINT_LOG("\r\nInitialization ... ");
       enuStatus = at_ble_init(&pf_cfg);
       CHECK_ERROR(enuStatus, __EXIT);
       PRINT("Done");

       PRINT_LOG("\r\nAddress set ...");
       enuStatus = at_ble_addr_set(&addr);
```

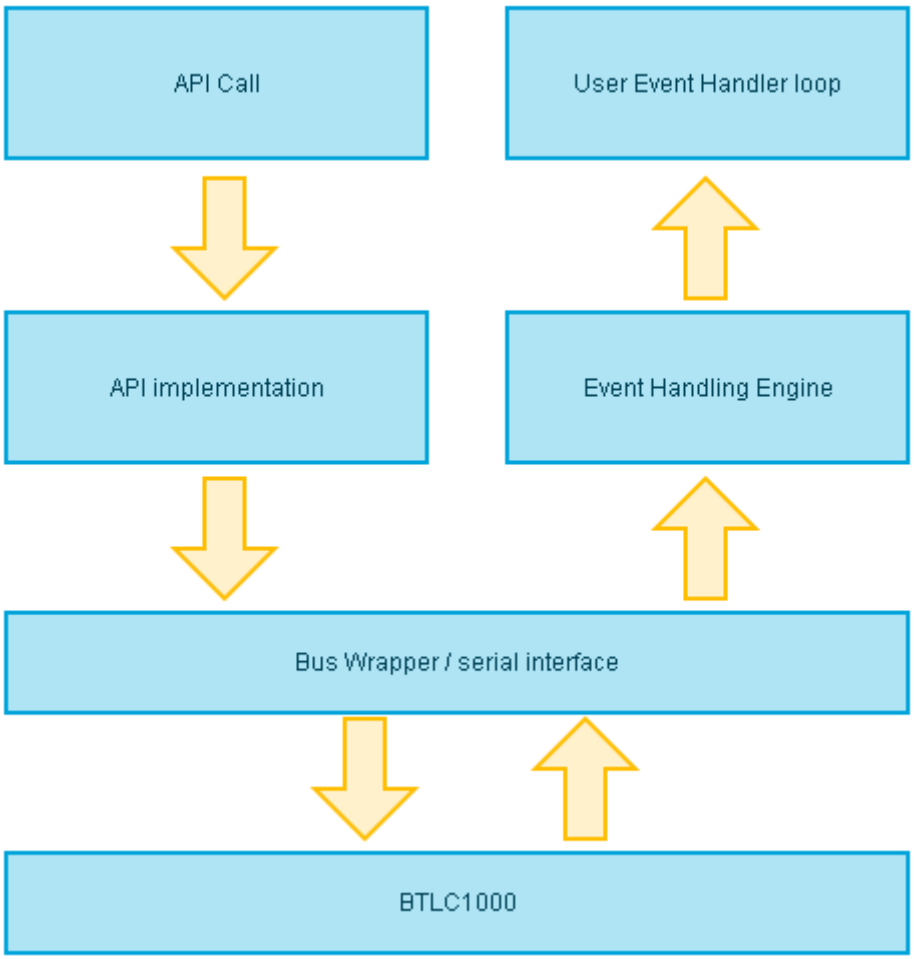
```
	CHECK_ERROR(enuStatus, __EXIT);
	PRINT("Done");

	PRINT_LOG("\r\nAdv data set ...");
	enuStatus = at_ble_adv_data_set(adv_data, sizeof(adv_data), scan_rsp_data,
					sizeof(scan_rsp_data));
	CHECK_ERROR(enuStatus, __EXIT);
	PRINT("Done");

	PRINT_LOG("\r\nAdv. start ...");
	enuStatus = at_ble_adv_start(AT_BLE_ADV_TYPE_UNDIRECTED, AT_BLE_ADV_GEN_DISCOVERABLE,
					NULL, AT_BLE_ADV_FP_ANY, 100, 1000, 0);
	CHECK_ERROR(enuStatus, __EXIT);
	PRINT("Done");

	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;
				PRINT_LOG("Device connected\r\n");
				PRINT("Address :0x%02x%02x%02x%02x%02x%02x\r\n"
				      "handle  :0x%04x\r\n",
					conn_params->peer_addr.addr[5],
					conn_params->peer_addr.addr[4],
					conn_params->peer_addr.addr[3],
					conn_params->peer_addr.addr[2],
					conn_params->peer_addr.addr[1],
					conn_params->peer_addr.addr[0],
					conn_params->handle);
				gHandle = conn_params->handle;
			}
			break;
			case AT_BLE_DISCONNECTED:
			{
				at_ble_disconnected_t *discon_params = (at_ble_disconnected_t *)params;
				PRINT_LOG("Device disconnected\n");
				PRINT("Reason : %d\r\n"
				      "Handle : 0x%04x\r\n", discon_params->reason,
				       discon_params->handle);
				gHandle = 0xFFFF;
				enuStatus = at_ble_adv_start(AT_BLE_ADV_TYPE_UNDIRECTED,
				       AT_BLE_ADV_GEN_DISCOVERABLE,
				       NULL, AT_BLE_ADV_FP_ANY, 100, 1000, 0);
				CHECK_ERROR(enuStatus, __EXIT);
			}
			break;
		}
	}
__EXIT:
	PRINT("Failed\r\n");
	PRINT_LOG("\r\nError while starting");
	while(1);
	return 0;
}
```

2.3 Event Posting and Handling



Each event message returned by the controller is retrieved by a call to the API `at_ble_event_get`(). This is a blocking call and will never return unless a new event is received from the controller, or a call to the API `at_ble_event_user_defined_post`() is made. The use of the user defined event posting gives the user flexibility to skip an iteration of the event handling loop, by sending a user defined event that makes the blocking call to at_ble_event_get return with user event message ID. This is particularly handy in situation where you want to execute some code inside the event loop right after handling a specific message from the controller, without the need to wait for a controller event that may come now or later.

3 API Usage Examples

3.1 GAP Advertising

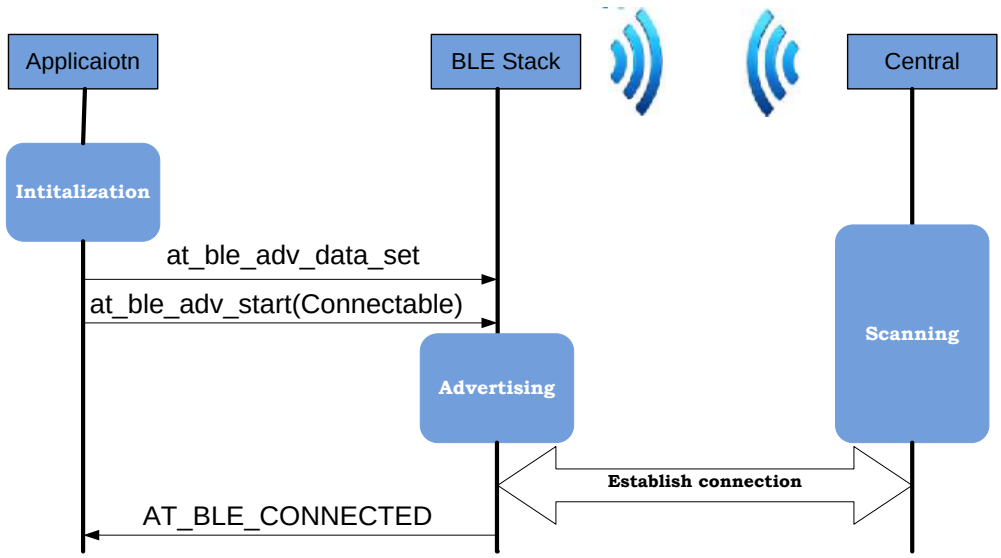
After initialization and setting address has been done, to run device in peripheral role it is required to advertise and in this case the device is called ***Advertiser*** or ***Peripheral***.

Advertising data means that peripheral will send unidirectional broadcast data on air to be discovered by other devices and behave according to device capabilities such as advertising type, mode …etc.

Say if it is needed to response to connection request from scanner devices, it is required to advertise in connectable mode.

In addition of 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 you can set additional user data called response data using the same function if needed, these data will be sent to the device making scan and request more information.
Settings advertising data must be done before start advertising. If the advertising is running, it must be stopped using `at_ble_adv_stop()` and apply advertising data then start advertising again.
Now, Advertiser peripheral can start advertise using `at_ble_adv_start()`.



Example:
Device Address : 0x7f7f6a117525

Advertising data length : 0x11

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

Service UUID : 0x5730CD00DC2A11E3AA370002A5D5C51B

```
#include "at_ble_api.h"

#define CHECK_ERROR(VAR,LABEL)     if(AT_BLE_SUCCESS != VAR) \
                                   { \
                                         goto LABEL; \
                                   }

#define DEVICE_NAME "Atmel BLE Device"
```

```
uint8_t adv_data[] = {  0x11, 0x07, 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_init_config_t pf_cfg;
	platform_config busConfig;
	at_ble_addr_t addr = {AT_BLE_ADDRESS_PUBLIC,
	                      {0x25, 0x75, 0x11, 0x6a, 0x7f, 0x7f}
	                     };

	//Set device name
	status = at_ble_device_name_set((uint8_t *)DEVICE_NAME, sizeof(DEVICE_NAME));
	CHECK_ERROR(status, __EXIT);

	/*Memory allocation required by GATT Server DB*/
	pf_cfg.memPool.memSize = 0;
	pf_cfg.memPool.memStartAdd = NULL;
	/*Bus configuration*/
	busConfig.bus_type = UART;
	pf_cfg.plf_config = &busConfig;

	//Initializations of device
	status = at_ble_init(&pf_cfg);
	CHECK_ERROR(status, __EXIT);

	//Set device address
	status = at_ble_addr_set(&addr);
	CHECK_ERROR(status, __EXIT);

	//Set advertising data, instead of NULL set scan response data if needed
	status = at_ble_adv_data_set(adv_data, sizeof(adv_data), NULL, 0))
	CHECK_ERROR(status, __EXIT);

	//Start advertising
	status = at_ble_adv_start(AT_BLE_ADV_TYPE_UNDIRECTED, AT_BLE_ADV_GEN_DISCOVERABLE,
	                                        NULL, AT_BLE_ADV_FP_ANY, 100, 0, false))

__EXIT:
	return status;
}
```

Refer to *AT_BLE_API_USER_MANUAL* for more information about APIs and their parameters.

3.2 GAP Scanning and Connection Creation

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

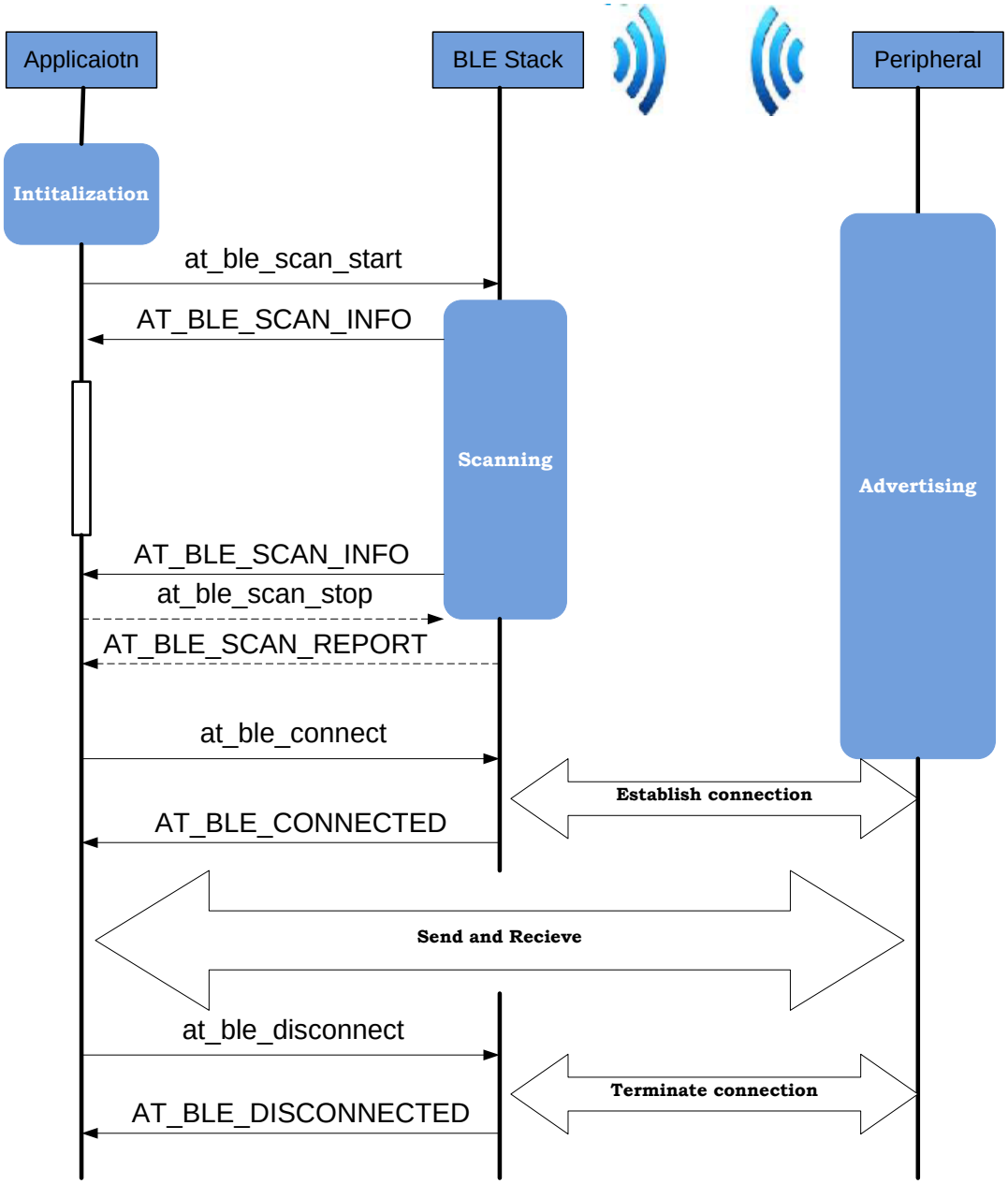
Central device may request for more additional user data from the advertiser.

Application will be triggered when receiving `AT_BLE_SCAN _INFO` event with each scan result, also there is another event `AT_BLE_SCAN_REPORT` will be 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 operation using `at_ble_scan_stop()` while in this mode scan is going with endless time.

When the desired peer device is found, stop scan and initiate connection request to it.

Sequence Diagram:



Example:

Device Address : 0x7f7f6a117525
Peer Address : 0x001bdc060545

```
#include "platform.h"
#include "at_ble_api.h"

#define CHECK_ERROR(VAR,LABEL)     if(AT_BLE_SUCCESS != VAR) \
                                   { \
                                          goto LABEL; \
                                   }

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

#define DEVICE_NAME "Atmel 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;
       at_ble_init_config_t pf_cfg;
       platform_config busConfig;
       /*Memory allocation required by GATT Server DB*/
       pf_cfg.memPool.memSize = 0;
       pf_cfg.memPool.memStartAdd = NULL;
       /*Bus configuration*/
       busConfig.bus_type = UART;
       pf_cfg.plf_config = &busConfig;

       //Set device name
       status = at_ble_device_name_set((uint8_t *)DEVICE_NAME, sizeof(DEVICE_NAME));
       CHECK_ERROR(status, __EXIT);

       //Initializations of device
       status = at_ble_init(&pf_cfg);
       CHECK_ERROR(status, __EXIT);

       //Set device address
       status = at_ble_addr_set(&addr);
       CHECK_ERROR(status, __EXIT);

       //Start scan
       status = at_ble_scan_start(GAP_INQ_SCAN_INTV, GAP_INQ_SCAN_WIND, 0,
       AT_BLE_SCAN_ACTIVE, AT_BLE_SCAN_OBSERVER_MODE, FALSE, 1);
       CHECK_ERROR(status, __EXIT);

__EXIT:
       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];
	at_ble_status_t status;

	status = init_central_role();
	CHECK_ERROR(status, __EXIT);

	PRINT_LOG("Scanning ...\r\n");
	while(AT_BLE_SUCCESS == at_ble_event_get(&at_event, params, -1))
	{
		switch(at_event)
		{
			case AT_BLE_SCAN_INFO:
			{
				at_ble_scan_info_t *scan_params = (at_ble_scan_info_t *)params;
				PRINT_LOG ("Device info:\r\n");
				PRINT("Address   : 0x%02x%02x%02x%02x%02x%02x\r\n"
					"Addr. Type: %d\r\n"
					"RSSI      : %d\r\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],
					scan_params->dev_addr.type,
					scan_params->rssi
				);
				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 = 0x01f4;

						/* Connect to peer device */
						PRINT_LOG ("Connecting..\r\n");
						status = at_ble_connect(&peer_addr, 1,
GAP_INQ_SCAN_INTV, GAP_INQ_SCAN_WIND, &conn_params);
						PRINT("Connection status %d\r\n", status);
					}
				}
			}
			break;
			case AT_BLE_CONNECTED:
			{
```

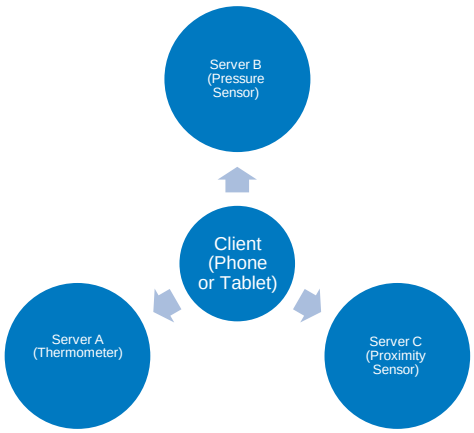
```
                        at_ble_connected_t* conn_params = (at_ble_connected_t*)params;
                        PRINT_LOG("Device connected\r\n");
                        PRINT("Address :0x%02x%02x%02x%02x%02x%02x\r\n"
                              "handle  :0x%04x\r\n",
                              conn_params->peer_addr.addr[5],
                              conn_params->peer_addr.addr[4],
                              conn_params->peer_addr.addr[3],
                              conn_params->peer_addr.addr[2],
                              conn_params->peer_addr.addr[1],
                              conn_params->peer_addr.addr[0],
                              conn_params->handle);
                }
                break;
                case AT_BLE_DISCONNECTED:
                {
                        at_ble_disconnected_t *discon_params = (at_ble_disconnected_t *)params;
                        PRINT_LOG("Device disconnected\n");
                        PRINT("Reason : %d\r\n"
                                "Handle : 0x%04x\r\n", discon_params->reason,
                              discon_params->handle);
                        status = at_ble_scan_start(GAP_INQ_SCAN_INTV, GAP_INQ_SCAN_WIND, 0,
                              AT_BLE_SCAN_ACTIVE, AT_BLE_SCAN_OBSERVER_MODE, FALSE, 1);
                        PRINT("Scan again status %d\r\n", status);
                }
                break;
            }
        }
__EXIT:
        PRINT("Failed\r\n");
        PRINT_LOG("\r\nExit\r\n");
        while(1);
        return 0;
}
```

3.3 GATT Server – Service Definition

3.3.1 Introduction

Generic Attribute Profile (GATT) is an upper layer of the Bluetooth stack that defines how two connected Bluetooth devices can exchange information. It is based on the Attribute Protocol (ATT) which according to the standard:

"Allows a device referred to as the server to expose a set of attributes and their associated values to a peer device referred to as the client. These attributes exposed by the server can be discovered, read, and written by a client, and can be indicated and notified by the server."



3.3.2 Services and Characteristics

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

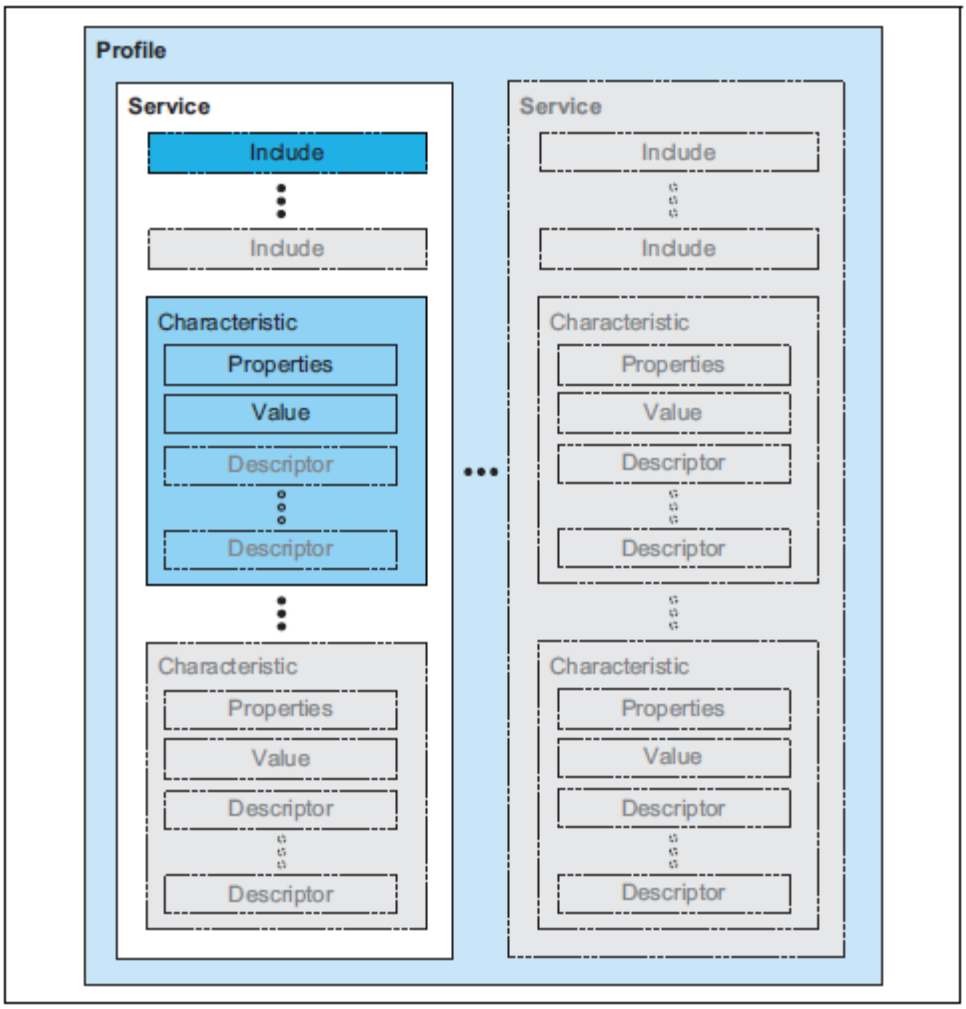


Figure 3-1. Basic GATT hierarchy.

3.3.3 Defining a Service

If a peer defined a service with a set of characteristics it will 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 should be 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 characteristic_count)` should be called with proper arguments which will return 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` will return handle of the first characteristic in the service , also handles to the client configuration, user descriptor, and server configuration will be returned in `charactristic_list[i].client_config_handle, charactristic_list [i].user_desc_handle, charactristic_list[i].server_config_handle` respectively.

```
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 defined name */
       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*/
       NULL /* presentation format */
};

static at_ble_handle_t service;
// establish peripheral database
at_ble_primary_service_define(&service_uuid, &service,
       NULL, 0, chars, 2);
```

Table 3-1. Service Definition Code snippet

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 will receive an `AT_BLE_CHARACTERISTIC_CHANGED` event with, the developer should compare the handle returned in the parameters to the client config handle `charactristic_list[i].client_config_handle` returned from the previous step and check if its new value is not 0, then the server can start 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);
```

```
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;
```

Table 3-2. Sending Notifications Code Snippet

3.4 GATT Client – Service Discovery

3.4.1 Discovering a Service

There are two ways to discover services in a GATT server:

- either to discover all services from a start handle to an end handle with

  ```
  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);
  ```

- or to discover a specific service using its UUID with

  ```
  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 will be returned and should be handled by the developer, `AT_BLE_DISCOVERY_COMPLETE` will return the status of the operation and `AT_BLE_PRIMARY_SERVICE_FOUND` will be 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;
```

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` will be 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 will be 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 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, to be used during another connection to avoid repeating pairing procedure.

Security shall be initiated by the device in the master role. The device in the slave role shall be the responding device. The slave device may 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 always used and may be followed by an optional transport specific key distribution phase to share the keys which can be used to encrypt a link in future reconnections verify signed data and perform random address resolution.

Phase 1: Pairing Feature Exchange

The devices shall 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 can be exchanged during the pairing, the bonding flag is set to zero.
- Man in the Middle protection (MITM) Flag: According to IO capabilities or Out Of Band (OOB) property, if it is not possible to perform a pairing using PIN code or OOB data, this flag shall be set to zero.

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 can be used to force a certain level of authentication and presence of key exchange.

- LE Security Mode 1, which 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 shall be set to zero and LTK shall be exchanged
    3. `AT_BLE_MODE1_L2_AUTH_PAIR_ENC` (Authenticated pairing with encryption)

       Authenticated pairing with encryption, Man in the middle protection shall be set to 1, a LTK shall be exchanged

- LE Security Mode 2
    1. AT_BLE_MODE2_L1_NOAUTH_DATA_SGN (Unauthenticated pairing with data signing) **Unauthenticated pairing with data signing, Man in the middle protection shall be set to zero, a CSRK shall be exchanged.**
    2. AT_BLE_MODE2_L2_AUTH_DATA_SGN (Authenticated pairing with data signing) **Authentication pairing with data signing, Man in the middle protection shall be set to 1, a CSRK shall be exchanged.**

Key distribution
The initiating device indicates to the responding device which transport specific keys it would like to send to the responding device and which keys it would like the responding device to send to the initiator. The responding device replies with the keys that the initiating device shall send and the keys that the responding device shall send.

- AT_BLE_KEY_DIST_ENC : Distribute LTK , EDIV and random number
- AT_BLE_KEY_DIST_SIGN : Distribute CSRK
- AT_BLE_KEY_DIST_ID : Distribute 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 which of the following pairing method shall be used in STK Generation in Phase 2

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

All the pairing methods use and generate 2 keys:

- Temporary Key (TK): a 128-bit temporary key used in the pairing process , it can be 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 used to encrypt a connection following pairing.

Phase 2: Short Term Key (STK) Generation
Calculated according to pairing information and provided TK, it's used to encrypt the link during pairing in order 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 (EDIV) has to be stored with this key.
- **Encrypted Diversifier (EDIV):** is 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):** is a 64-bit stored valued used to identify the LTK, A new Rand is generated each time a unique LTK is distributed
- **Identity Resolving Key (IRK):** is a 128-bit key used to generate and random address
- **Connection signature key (CSRK):** when link is not encrypted, the CSRK should be used by GAP to sign and verify signature of an attribute write sign.

Phase 3: Transport Specific Key Distribution

Application APIs interface

- **APIs for initiating bonding and responding to pairing request from remote device**
  `at_ble_authenticate`
  `at_ble_send_slave_sec_request`

- **Events triggered to indicate that bonding is required**

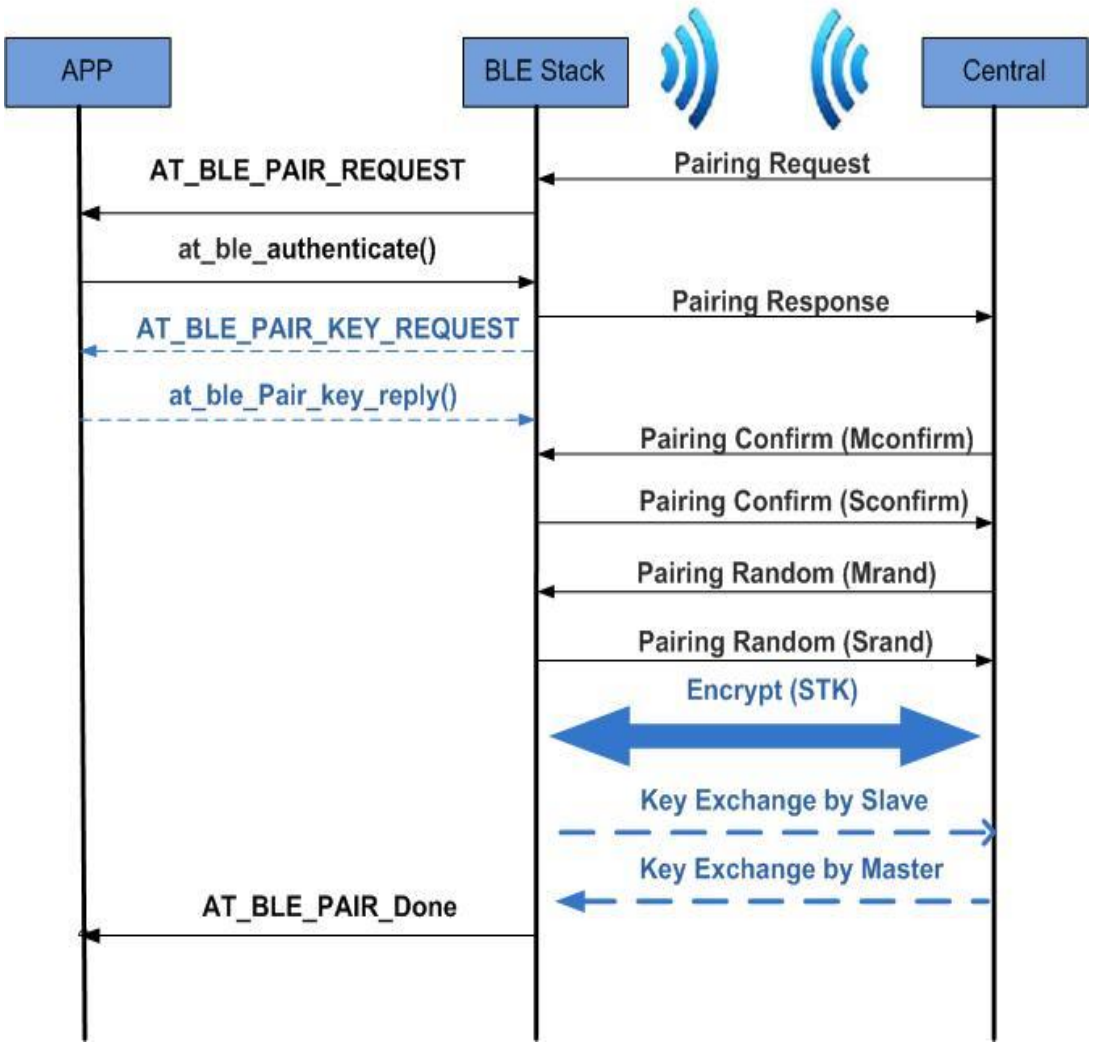
  `AT_BLE_PAIR_KEY_REQUEST`

  `AT_BLE_SLAVE_SEC_REQUEST`

- **Event triggered to indicate bonding status**

  `AT_BLE_PAIR_DONE`

Sequence Diagram:



3.5.2 Encryption procedure

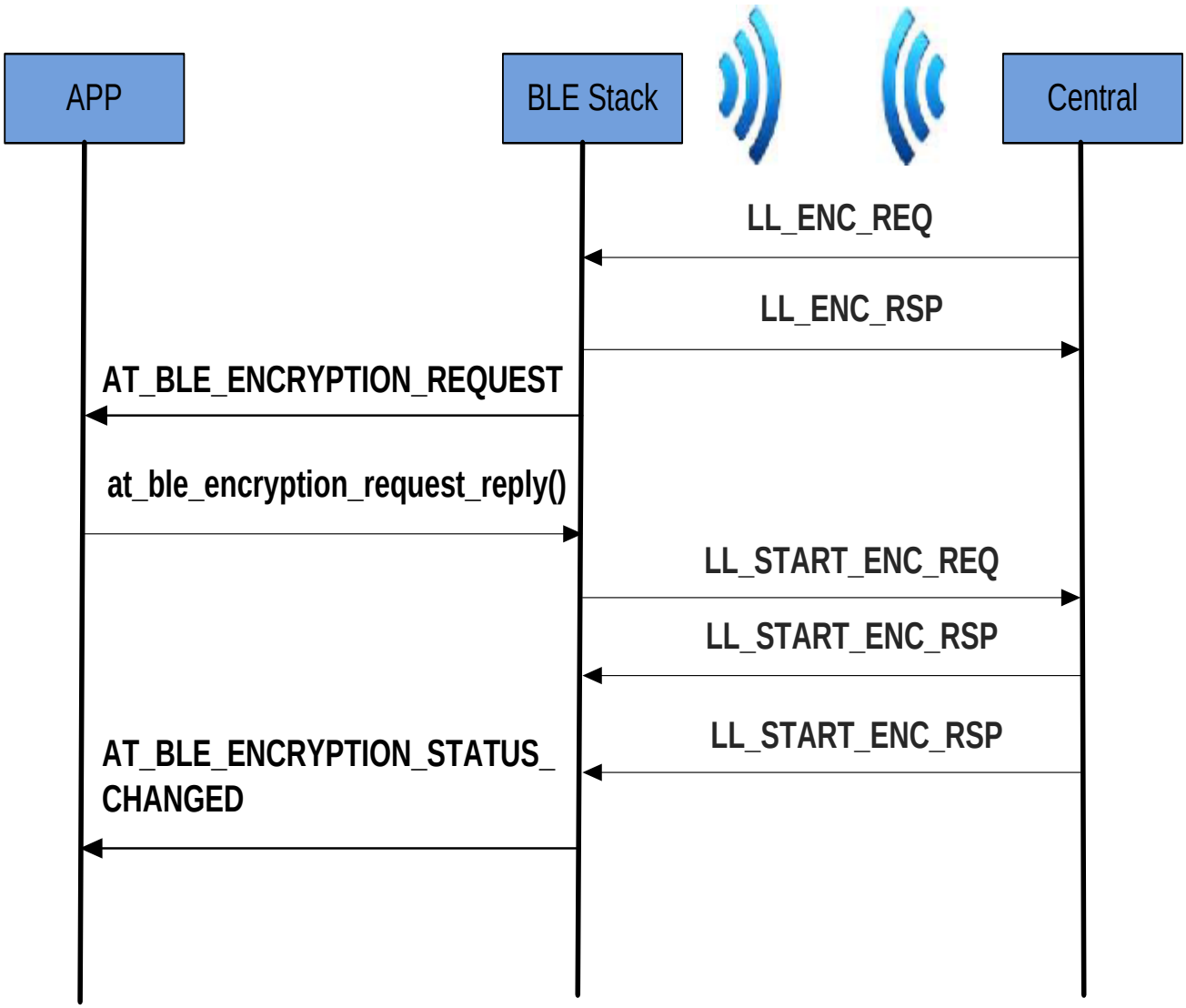
The encryption procedure is used to encrypt the link using **a previously bonded Long term Key (LTK)**.This procedure can be initiated only by master of the connection.

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

The master's Host provides the Link Layer with the Long Term Key to use when setting up the encrypted session.

The slave's Host receives the EDIV and Rand values and provides a Long Term Key to the slaves Link Layer to use when setting up the encrypted link.

Sequence Diagram:



Example:

```
#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;
        }
    }
}
```

Refer to *AT_BLE_API_USER_MANUAL* for more information about APIs and their parameters.

4 Revision History

Doc Rev.	Date	Comments
42521A	09/2015	Initial document release.

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**