BlueLitE

Bluetooth Low Energy Protocol Stack & Profiles
Application Developer's Guide

30 September 2014
Mindtree Limited,
Global Village, Mysore Road
Bangalore - 560 059
www.mindtree.com

ABSTRACT

This document is the Application Developer's Guide for BlueLitE Bluetooth Low Energy Protocol Stack & Profiles.

REVISION HISTORY

Owner contact: ethermind_support@mindtree.com

Date	Version	Description	Author
01 November 2011	0.1	Initial Draft	BlueLitE Team
17 March 2012	0.2	Added sample code snippet for ATT/GATT	BlueLitE Team
19 March 2012	1.0	Baselined	BlueLitE Team
12 May 2012	1.1	Updated usage of ATT/GATT and SMP APIs. Added MSCs.	BlueLitE Team
7 August 2012	1.2	Added recommendation to send HCI Reset command, before calling BT_bluetooth_off()	BlueLitE Team
30 September 2014	1.3	Updated MSC for Just Works pairing. Added MSCs for Passkey pairing, re-authentication and data signing procedures.	BlueLitE Team

Table of Contents

Introduction

The BlueLitE Bluetooth Low Energy Protocol Stack & Profiles provide highly flexible and feature rich APIs for application developer to use and develop portable application for a variety of Bluetooth enabled devices and accessories.

This is the generic version of the BlueLitE Developer's Guide document. Content in this document is written generically to address application development in any platform, or, operating systems on which BlueLitE Protocol Stack & Profiles are supported.

Certain platform, and/or, operating system, may require special/additional handling of certain APIs, and also may have additional guidance for certain aspect of application development. Refer to the BlueLitE Developer's Guide for your platform for such details.

Document Purpose

The purpose of this document is to provide an application developer, with existing knowledge of the Bluetooth wireless technology, with guidelines for developing application using BlueLitE Protocol Stack & Profiles APIs. This document serves as a central point of information and as a starting guide for the application developers, and, complements the BlueLitE API (Protocol Stack & Profiles) Documents ([1]).

Document Organization

This document is organized in a manner most suitable for the application developer to read and understand. The key concepts are described in step-by-step manner, separated by chapters, as described below:

- Step 0: Describes important notes to consider prior to writing application using BlueLitE APIs.
- Step 1: Describes how to initialize the Stack, and switch on/off Bluetooth functionalities
- Step 2: Describes how to use HCI and SM (Security Manager) APIs & Callbacks
- Step 3: Describes how to use ATT & GATT Database APIs & Callbacks

Finally, a section on most frequently asked questions is included to answer certain common questions that the application developers may have.

Document Symbols & Conventions

In this document the following symbols and conventions are used.

An important "note" is stated as follows:

> ✍ **Note**
>
> ✍ This is an important note concerning …

Additional references & reading are documented as follows:

See Also

- An example of application calling `BT_xyz()` API.
- Refer to XYZ Document for further details …

Sample code segment and examples are documented as follows:

Example: Sample Code for …

```
/*
 *  This is an example/sample code segment
 */
```

Following arrow conventions are used in signaling/message sequence charts:

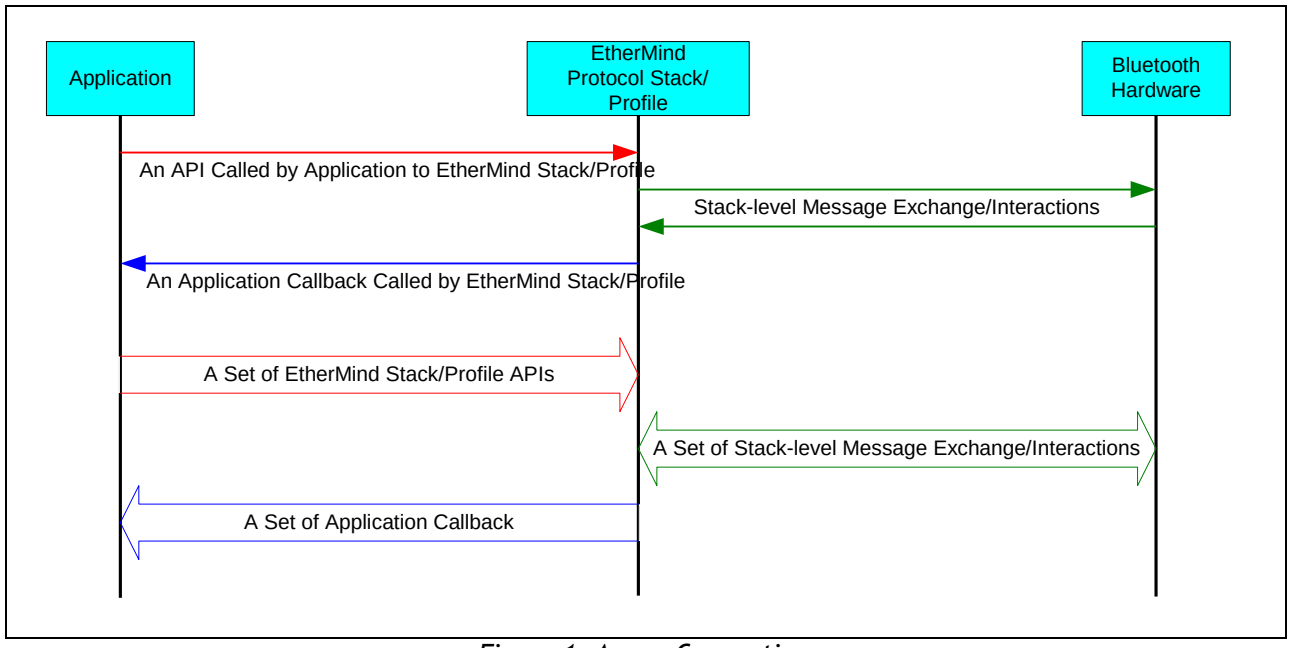


Figure 1: Arrow Convention

Step 0: Setting Up the Development Environment

Depending on the target platform, an application developer can make use of any available IDE of choice to build application using BlueLitE Bluetooth Protocol Stack & Profile APIs. However there are certain important considerations that need to be understood prior to starting to write an application.

Configuring BlueLitE Stack & Profile Modules

The BlueLitE Bluetooth Host Protocol stack is a highly portable and configurable stack. Two most important of the many configuration options provided by the BlueLitE Stack are as stated below:

- Tunable Constants
- Compilation Flags

BlueLitE Tunable Constants

Various BlueLitE Stack modules use Tunable Constant definitions to control limits of Queues, Buffer Size etc., which can be configured during the time of compilation and build of the Stack & Profile modules. These tunable constants are specified in the BlueLitE Header File `BT_limits.h`.

See Also

Refer to `BT_limits.h` for details on the available tunable constants for a module and understand their scope and purpose. Certain tunable constants are inter-dependent, and the care should be taken while configuring values for the same.

In general, each tunable constant is preceded with a code comment block containing the following information:

- Name, purpose, description & background of the tunable constant
- Description of dependency of this tunable constant on other tunable constant
- Recommended minimum & maximum values for the tunable constant

An example of an BlueLitE Tunable Constant from the L2CAP module is provided for reference below:

Example: BlueLitE Tunable Constants

```
/**
 * ATT_MAX_CONNECTION_INSTANCES
 *
 * Maximum Instances supported by ATT. This is the maximum number of
 * simultaneous Transactions that the module can have.
 *
 * Minimum Value:   1.
 * Maximum Value: 255.
 */
```

Example: BlueLitE Tunable Constants

```
#define ATT_MAX_CONNECTION_INSTANCES                    1
```

Note

- Configuration of Tunable Constants controls BlueLitE Stack & Profile module's data size (RAM) requirement.
- Configuration of Tunable Constants is required only when the application developer has access to the BlueLitE Stack & Profile Source Codes, and/or, the same needs to be built/compiled before application could be linked.
- Reconfiguration of Tunable Constants should not be done when the application developer does not have access to the BlueLitE Stack & Profile Source Codes, and/or, the same has been supplied in pre-built object code format (DLL or Library or Object module).

BlueLitE Compilation Flags

The purposes of the BlueLitE Compilation Flags, or Switches, are as follows:

- To enable/disable inclusion of a stack module
- To enable/disable/control the inclusion of features provided by various stack modules
- To choose the platform, operating system etc for which the stack to be compiled
- To choose the transport (reading & writing from/to Bluetooth hardware) option – UART, USB etc.
- To choose the compliance to specific architectural design, or, Bluetooth specification version

The available compilation flags/switches are specified and described in detail, including dependencies, in the BlueLitE Compilation Flags Document [5].

Note

- Inclusion or non-inclusion of BlueLitE Compilation Flags controls BlueLitE Stack & Profile module's code size (ROM) requirement. Hence, it is recommended that correct set of required compilation flags should be decided to limit features included in the final executable.
- Decision on correct set of BlueLitE Compilation Flags is required only when the application developer has access to the BlueLitE Stack & Profile Source Codes, and/or, the same needs to be built/compiled before application could be linked.
- When the application developer does not have access to the BlueLitE Stack & Profile Source Codes, and/or, the same has been supplied in pre-built object code format (DLL or Library or Object module), the application should be built and linked to supplied BlueLitE object code using the same set of BlueLitE Compilation Flags that have been used to build the BlueLitE object codes. Refer to the Release Notes, and/or, Developer's Guide for your Platform/Release for details.
- Latest information about BlueLitE Compilation Flags is available in BT_features.h, in certain platforms. The BT_features.h is used for defining the compilation flags, and thereby choose features for compilation. BT_os.h includes BT_features.h after the standard header files are included.

Inclusion of BlueLitE Header Files

The table below describes various BlueLitE Header Files, which may be included in application source files, with their purposes.

BlueLitE Header File	Comments/Purpose/Description
BT_common.h	This is the most important BlueLitE header file, and, must be included by every application source file before including any other BlueLitE header files. This header file, in turn, includes several other important and required BlueLitE header files, some of which are described below: • BT_os.h - BlueLitE OS Abstraction • BT_common_pl.h - BlueLitE Platform Abstraction • BT_limits.h - BlueLitE Tunable Constants • BT_assigned_numbers.h - Bluetooth SIG defined Assigned Numbers • BT_error.h - BlueLitE API/Callback Result/Error Code Definitions
BT_debug.h	BlueLitE Debug Library header file. Application may include this header file in its source codes, if it chooses to use BlueLitE Debug Library for its debug prints.
BT_timer.h	BlueLitE Timer Library header file. Application may include this header file in its source codes, if it chooses to use BlueLitE Timer Library for its timer requirements.
Protocol API Header Files: BT_hci_api.h BT_smp_api.h BT_att_api.h BT_gatt_db_api.h etc.	Depending on application's need some or all of these API header files may be needed to be included in application source code files.

BlueLitE Header Files Include Directories

The development tool, which is being used to compile/build the application, should be configured to add the following directories/paths for the additional (header file) include directories. This facilitates the development tool to search for the BlueLitE header files during compilation of the application source codes.

- BlueLitE Common Header Files Directory:

  bluetooth/export/include

- BlueLitE Platform/Operating System specific Header Files Directory:

  bluetooth/private/platforms/<your-platform>

All paths mentioned above are relative to the directory where BlueLitE source codes are installed by the application developer.

✍ **Note**

✍ **Note**

✍ All paths mentioned above are relative to the directory where BlueLitE source codes are installed by the application developer.

Step 1: Initializing BlueLitE Stack

There are three most important APIs, exported by the BlueLitE Protocol Stack to initialize the stack and switch on & off the Bluetooth functionality:

- BT_ethermind_init()
- BT_bluetooth_on()
- BT_bluetooth_off()

For power sensitive application (e.g. cell phone), the "Stack Initialization" and the "Bluetooth (Baseband Controller) Initialization" are split into two phases. The "Stack Initialization", using the BT_ethermind_init() API, takes care of starting the required BlueLitE tasks [1] and operating system specific initializations.

When the Bluetooth functionality is required, the Bluetooth part of the initialization can be performed, using the BT_bluetooth_on() API, and also shut down dynamically at any point of time, using the BT_bluetooth_off() API.

Application, which do not require this distinction, can call both the functions, BT_ethermind_init() & BT_bluetooth_on(), one after the other.

See Also

A detailed description of these APIs can be found in BlueLitE Stack API[2].

BlueLitE Initialization

The `BT_ethermind_init()` is the first API that the application must call to initialize the entire BlueLitE Protocol Stack, including all its modules. During this initialization, various modules create and initialize their respective synchronization and conditional variables and allocate any static memory (if required). All modules perform platform level initialization during this process. The Bluetooth level initialization is not performed here.

Internally, `BT_ethermind_init()` calls the initialization routine of each module, one after the other. It follows a bottom-up approach, i.e., the lower layers are initialized before the higher layers. The Debug and Timer Libraries, and Transport modules are initialized before others. The BlueLitE tasks [1] are created during this time – and they are moved to a dormant state.

Application is required to include the "`BT_common.h`" to be able to invoke `BT_ethermind_init()`. An example of application calling `BT_ethermind_init()` is shown below:

Example: BlueLitE Initialization

```
#include "BT_common.h"

int main (int argc, char **argv)
{
    /* Initialize EtherMind */
```

Example: BlueLitE Initialization

```
    BT_ethermind_init();

    ...

    return 0;
}
```

See Also

- An example of application calling `BT_ethermind_init()` API.

Note

- The BlueLitE Stack modules may behave in unpredictable manner if `BT_ethermind_init()` is not invoked to initialize the BlueLitE Protocol Stack prior to calling any other BlueLitE Stack API(s).
- The `BT_ethermind_init()` initializes the platform specific components of the BlueLitE Protocol Stack, which includes creation of BlueLitE Tasks (the Read, Write and/or Timer task). The method of creating tasks/threads is platform-specific, and it is assumed that these are handled during porting the stack to the platform. For example, some platform may require static task creation, as opposed to dynamic task creation – in that case, BlueLitE Tasks are already created and available when the system (platform and/or operating system) starts up, and hence, the `BT_ethermind_init()` may not need to create any tasks. Refer to the platform specific BlueLitE Developer's Guide for more details.

Switching ON Bluetooth Functionality

The `BT_bluetooth_on()` API brings the BlueLitE Protocol Stack alive and performs the Bluetooth level initialization, whereby each BlueLitE Stack module is initialized for their internal data structures and variables. Once this initialization is complete (successfully), the BlueLitE Protocol Stack is ready for use for Bluetooth related functionalities.

The BlueLitE tasks [1], which may have been created during `BT_ethermind_init()` API, are signaled to wake up from their dormant state to start servicing their respective queues.

The `BT_bluetooth_on()` is particularly important for the HCI module. This is the time when the HCI-Transport interface (UART/USB/BCSP/SDIO etc.) is opened and HCI sends a number of commands, one after the other, to the Bluetooth Host Controller (Baseband & LMP) hardware – such as, HCI Reset, HCI Read BD_ADDR, HCI Read Buffer Size etc. – which are fundamental for the operation of the stack. The "HCI Reset" Command, specifically, resets the Host Controller hardware to bring it up in initial/default state.

The `BT_bluetooth_on()` API is also very important for the Security Manager (SM) module. During this time, the SM attempts to open the "Persistent Storage Media" (such as, EEPROM etc.), as configured and enabled for the platform/operating system, and reads its configuration information.

Function Prototype of `BT_bluetooth_on()` API is given below. The application is required to include the "`BT_common.h`" to be able to invoke `BT_bluetooth_on()`.

```
#include "BT_common.h"

API_RESULT  BT_bluetooth_on
            (
                API_RESULT (* appl_hci_event_ind_cb) (UCHAR, UCHAR *, UCHAR),
                API_RESULT (* appl_bluetooth_on_complete_cb) (void),
                CHAR * local_name
            );
```

The "appl_hci_event_ind_cb" parameter is an application provided callback function pointer to handle HCI Application Event Indication Callback. Details on this parameter can be found in How to Use HCI APIs section.

The "appl_bluetooth_on_complete_cb" parameter is an application provided callback function pointer that is called on completion of the Bluetooth-ON procedures [see message sequence charts for BT_bluetooth_on()]. This callback function is mandatory and must be provided by application at the time of calling BT_bluetooth_on().

The BT_bluetooth_on() API returns with API_SUCCESS immediately, which only means that Bluetooth-ON procedures are started, but not completed. Completion of the same will be indicated to the application by invoking the registered "appl_bluetooth_on_complete_cb" callback function.

See Also

- An example of application calling BT_bluetooth_on() API.
- Expected Message Sequence Chart for BT_bluetooth_on()

Note

- The BlueLitE Stack modules may behave in unpredictable manner if BT_bluetooth_on() is not invoked to switch on Bluetooth functionality in the BlueLitE Protocol Stack prior to calling any other BlueLitE Stack API(s) that involves Bluetooth procedures (such as, creating a ACL connection, Bluetooth Inquiry etc.). Hence, the BT_bluetooth_on() is the second API that the application must call after initializing the stack using BT_ethermind_init() API.
- The BT_bluetooth_on() only enables the Bluetooth functionality in BlueLitE Protocol modules. BlueLitE Profile Modules are not initialized during the BT_bluetooth_on() API. Initialization of required BlueLitE Profile modules needs to be done by application when Bluetooth-ON operation completes successfully, and, the application provided "appl_bluetooth_on_complete_cb" callback is invoked.

Switching OFF Bluetooth Functionality

The BT_bluetooth_off() API shuts down the Bluetooth functionality of the BlueLitE Protocol Stack, and re-initializes data structures, states and variables of various stack modules. The BlueLitE Read, Write and (optional) Timer tasks return to their dormant state. The HCI-Transport interface is closed, and it will not be possible for the BlueLitE Protocol Stack to continue communication with the Bluetooth Host Controller (Baseband & LMP) hardware.

Before calling BT_bluetooth_off() application should call BT_hci_reset() to reset the controller and wait for the corresponding HCI callback event. This will stop any ongoing link layer procedure, like Scanning or Advertising etc.

For all practical application's purposes, on return of the `BT_bluetooth_off()` API, the BlueLitE Protocol Stack returns to pre Bluetooth-ON state. At this point, Bluetooth functionality can again be switched ON, using the `BT_bluetooth_on()` API.

The diagram below explains this concept pictorially.

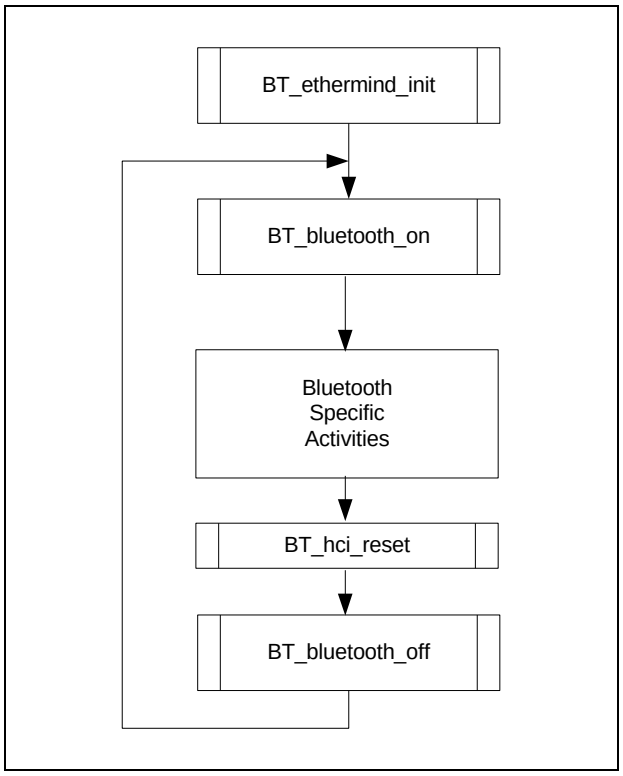


Figure 2: Cyclic Bluetooth ON & OFF

The `BT_bluetooth_off()` API is particularly important for the Security Manager (SM) module. During this time, the SM attempts to open the "Persistent Storage" (such as, EEPROM etc.), as configured and enabled for the platform/operating system, and writes its configuration information.

See Also

An example of application calling `BT_bluetooth_off()` API.

Note

Note

- The `BT_bluetooth_off()` only disables the Bluetooth functionality in BlueLitE Protocol modules. BlueLitE Profile Modules are not shut down during the `BT_bluetooth_off()` API. The shutdown of BlueLitE Profile modules needs to be done by application prior to calling the `BT_bluetooth_off()` API.
- If there exist connections in BlueLitE Protocol modules (such as, HCI BLE Link etc.), these are closed during `BT_bluetooth_off()`, and appropriate registered callbacks are called to inform the application that those connections are now dropped.
- Successful invocation of `BT_bluetooth_off()`, returns the BlueLitE Protocol Stack modules to pre Bluetooth-ON state. Hence, The BlueLitE Stack modules may behave in unpredictable manner if BlueLitE Stack API(s), that involves Bluetooth procedures (such as, creating a ACL connection, Bluetooth Inquiry etc.), are called after calling `BT_bluetooth_off()` and before next `BT_bluetooth_on()`.

BlueLitE Shutdown

The BlueLitE Protocol Stack does not provide an API to shutdown the Stack modules completely. To be precise, there is no API to kill or exit the BlueLitE Tasks created during the `BT_ethermind_init()` API. The reasons for this kind of implementation are as follows:

- Typical Bluetooth application (such as, cell phones etc.) needs to have ability to switch on & off Bluetooth functionality on demand (e.g., the user "turns on" Bluetooth). This functionality can be implemented using the `BT_bluetooth_on()` and the `BT_bluetooth_off()` APIs.
- The Bluetooth Stack itself can be initialized when the platform/operating system starts up. This is provided by the `BT_ethermind_init()` API. Among other initialization and configuration, `BT_ethermind_init()` is responsible for creating and/or instantiating the required BlueLitE tasks.
- The BlueLitE tasks created during `BT_ethermind_init()` API do not die/exit, and continue to be alive as long as the platform/operating system is working. Upon creation/instantiation, the BlueLitE tasks go into dormant/hibernating inactive state. The `BT_bluetooth_on()` API triggers the tasks to be active and service their respective queues. On the other hand, `BT_bluetooth_off()` API signals the tasks to go back to their dormant/hibernating inactive state.

Initialization, Bluetooth ON & OFF: An Example

The example below shows how to call `BT_ethermind_init()`, `BT_bluetooth_on()` and `BT_bluetooth_off()` APIs. The example below assumes a hypothetical UI front-end from where user triggers various Bluetooth actions.

Example: Initialization, Bluetooth ON & OFF

```
#include "BT_common.h"

/* Function Prototype for BlueLitE HCI Application Event IND Handler */
API_RESULT UI_hci_appl_event_ind_cb
           (
```

Example: Initialization, Bluetooth ON & OFF

```
                UCHAR   event_type,
                UCHAR * event_data,
                UCHAR   event_datalen
            );

/* Function Prototype for BlueLitE Bluetooth ON Complete Handler */
API_RESULT UI_bluetooth_on_complete_cb (void);

/* Local Bluetooth Device Name */
CHAR ui_bt_name[] = "EtherMind";

/* Local Bluetooth Device Address (BD_ADDR) */
UCHAR ui_bt_bd_addr [6];


/* Main UI Function */
int UI_Main (params ...)
{
    /* UI Specific Initialization */
    ...

    /* Initialize EtherMind */
    BT_ethermind_init();

    ...

    /* UI Main Loop */
    while (1)
    {
        /* Handle UI Events/Actions */
        ...
    }

    return 0;
}


/* UI Bluetooth ON Select/Button Press Handler */
int UI_Bluetooth_ON (params ...)
{
    API_RESULT retval;

    /* Switch ON Bluetooth */
    retval = BT_bluetooth_on
             (
                 UI_hci_appl_event_ind_cb,
                 UI_bluetooth_on_complete_cb,
                 ui_bt_name
```

Example: Initialization, Bluetooth ON & OFF

```
            );

    if (API_SUCCESS != retval)
    {
        /* Failed to Turn On Bluetooth. Do Error Handling here */
        ...
        ...

        return error_code;
    }

    /* Bluetooth ON Started. Wait for Bluetooth-ON Complete */
    ...

    return 0;
}


/* BlueLitE Bluetooth ON Complete Handler */
API_RESULT UI_bluetooth_on_complete_cb (void)
{
    UCHAR bd_addr[6];

    /* Bluetooth-ON Complete. Get Local BD_ADDR */
    BT_hci_get_local_bd_addr (ui_bt_bd_addr);

    /* Initialize BlueLitE Profiles here */
    ...
    ...

    /* Signal Complete for Bluetooth-ON */
    ...
}

/* UI Bluetooth OFF Select/Button Press Handler */
int UI_Bluetooth_OFF (params ...)
{
    /* Reset/Stop application modules using BlueLitE Protocols/Profiles */
    ...
    ...

    /* Shutdown BlueLitE Profiles here */
    ...
    ...

    /* Switch OFF Bluetooth */
    BT_bluetooth_off ();
```

Example: Initialization, Bluetooth ON & OFF

```
    return 0;
}
```

Initialization, Bluetooth ON & OFF: Message Sequence Chart

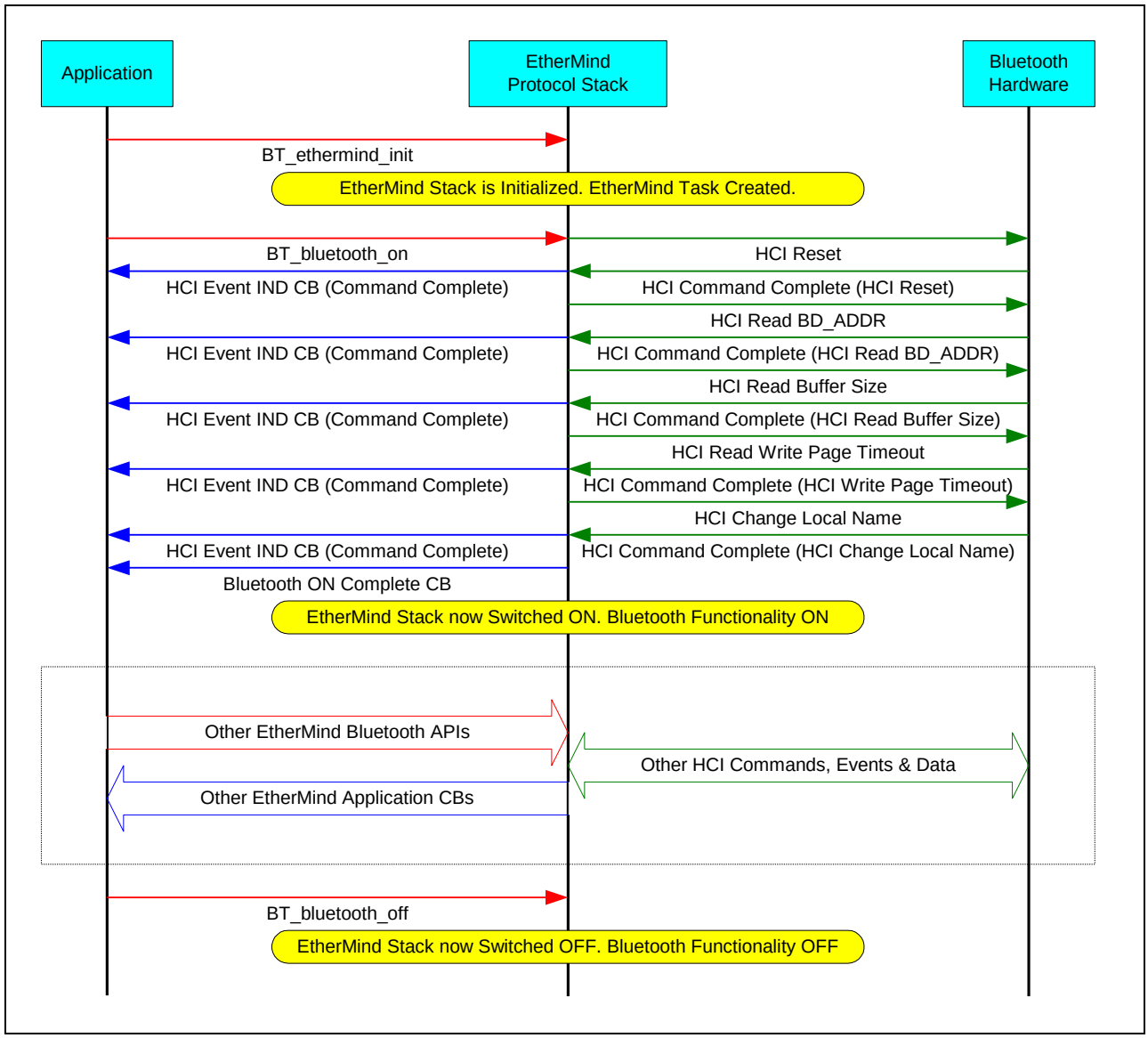


Figure 3: Message Sequence Chart: BlueLitE Init, Bluetooth ON & OFF

Step 2: How to Use BlueLitE HCI APIs

After the Stack Initialization and Bluetooth ON, interaction with BlueLitE HCI module is the most important aspect for application.

The HCI, or, the Bluetooth Host Controller Interface, provides a uniform interface method of accessing the capabilities of the local Bluetooth device, or, the Host Controller Hardware. The HCI layer interacts with the Host Controller by means of HCI Commands and Events. It also provides easy and portable interface to the application to access the hardware capabilities and perform various Bluetooth specific operations hiding Host Controller Interface details.

Even though SM module is maintained separately with respect to HCI, as far as application is concerned these modules cannot be considered to work separately. In other words, it is not possible to eliminate SM functionality and have HCI working stand-alone.

The architecture of BlueLitE HCI APIs can be described in the figure below.

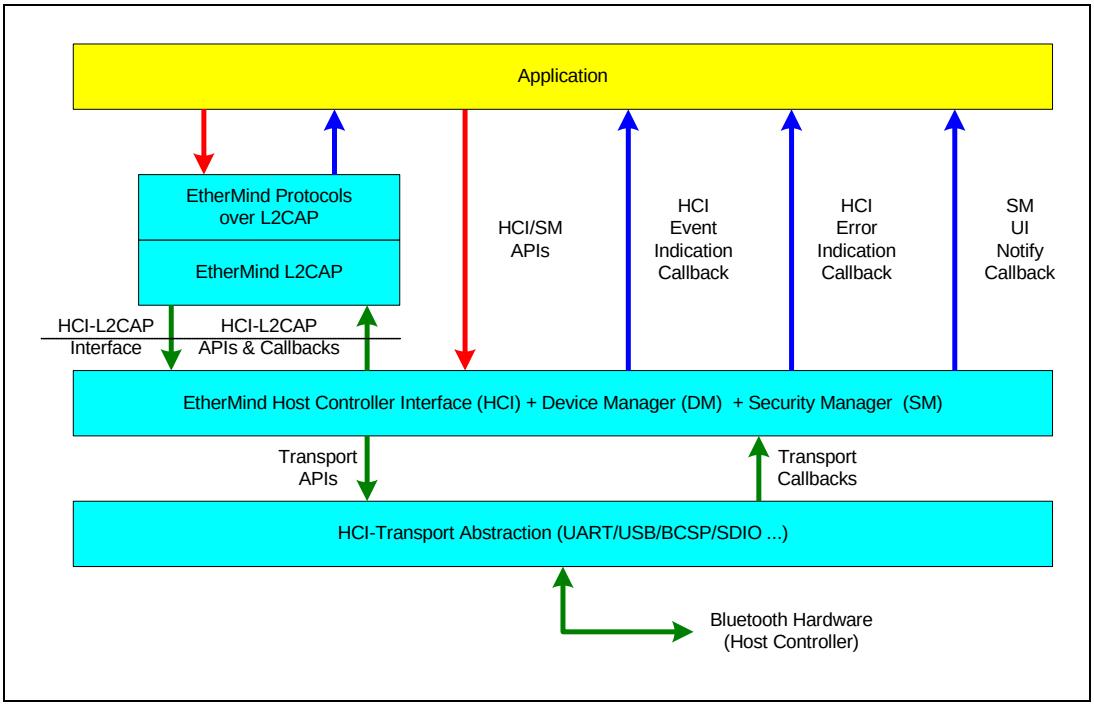


Figure 4: HCI/SM API & Callback Overview

HCI & SM APIs are briefly introduced in HCI & SM APIs section. The Callbacks and their usage are described later in this document.

See Also

A detailed description of the HCI & Security Manager APIs and Callbacks can be found in BlueLitE Stack API Document [2].

Introduction to HCI APIs

The BlueLitE HCI APIs can be broadly categorized as shown in the table below:

HCI API Category	Description/Remarks/Usage/Example
Callback Registration APIs	These APIs enable application to register & deregister various HCI Callbacks with the HCI module. Example: `BT_hci_register_event_indication_callback()`

Advertising APIs	These APIs allows the application to advertise and control the visibility and connectability of the device.
	
	Example:
	`BT_hci_le_set_advertising_parameters()`
	`BT_hci_le_set_advertising_data()`
	`BT_hci_le_set_advertising_enable()`

Scanning APIs	These APIs allows the application to scan for the peer devices.
	
	Example:
	`BT_hci_le_set_scan_parameters()`
	`BT_hci_le_set_scan_enable()`

Connection Management APIs	These APIs allows the application to control establishment of Bluetooth Connection and configuring White List.
	
	Example:
	`BT_hci_le_create_connection()`
	`BT_hci_le_create_connection_cancel()`
	`BT_hci_disconnect()`
	`BT_hci_le_add_device_to_white_list()`
	`BT_hci_le_clear_white_list()`
	`BT_hci_le_remove_device_from_white_list()`

Encryption and Security related APIs	These APIs allows the application and SM to authenticate a peer device and to establish an encrypted link.
	
	Example:
	`BT_hci_le_long_term_key_requested_reply()`
	`BT_hci_le_rand()`
	`BT_hci_le_start_encryption()`
	`BT_hci_le_encrypt()`

HCI API Category	Description/Remarks/Usage/Example

Remote Feature Request APIs	These APIs allows the application query various features of a remote Bluetooth device.
	
	Example:
	`BT_hci_le_read_remote_used_features()`
	`BT_hci_read_remote_version_information()`

Local Device Configuration & Control APIs	These APIs allows the application to configure, read or set values for various parameter on the local Bluetooth device
	
	Example:
	`BT_hci_le_set_event_mask()`
	`BT_hci_le_set_random_address()`
	`BT_hci_le_read_advertising_channel_tx_power()`

Test APIs	These APIs provides mechanisms to test Bluetooth Link Layer/Radio.
	
	Example:
	`BT_hci_le_transmitter_test_command()`
	`BT_hci_le_receiver_test_command()`
	`BT_hci_le_test_end()`

Local Device Information APIs	These APIs retrieve the values for informational parameters that are set by the manufacturers of the Baseband Controller. These informational parameters provide information about the local Bluetooth device and the capabilities of the Link Manager Protocol and the Baseband Controller. In general, the BlueLitE Protocol Stack cannot modify any of these parameters
	
	Example:
	`BT_hci_read_local_version_information()`
	`BT_hci_read_local_supported_features()`
	`BT_hci_read_local_supported_commands()`
	`BT_hci_le_read_local_supported_features()`
	`BT_hci_le_read_supported_states()`

Link Parameter APIs	These APIs retrieve the values for various ACL Link related parameters.
	
	Example:
	`BT_hci_read_transmit_power_level()`
	`BT_hci_le_connection_update()`
	`BT_hci_read_rssi()`
	`BT_hci_le_read_channel_map()`
	`BT_hci_le_set_host_channel_classification()`

HCI API Category	Description/Remarks/Usage/Example
Vendor Specific APIs	These APIs enable application to send Bluetooth Hardware vendor specific commands to control various/available features of the Bluetooth Hardware. Example: `BT_hci_vendor_specific_command()`
Utility APIs & Macros	These APIs & Macros help application to perform various jobs such as decoding a HCI Command Opcode, extracting various HCI Event parameters from Event Byte Stream etc. Example: `BT_hci_decode_opcode()`

For all HCI APIs, application must include the "`BT_hci_api.h`" header file.

Introduction to SMP APIs

The BlueLitE Security Manager provides the following important and useful functionalities, through its SMP APIs:

- Support and Management for Bluetooth Security Modes – 1 (Level 1, 2 & 3) & 2 (Level 1 & 2)
- In-built Device Database, which allows application to
    - Get current security configuration of a peer Bluetooth device
    - Get peer Bluetooth device attributes like Connection and Authentication states

The SM Device Database functionality allows application to support and implement the Bluetooth "Pairing" functionality, without the need of creating a device database and complex logic in the application.

The SM/GATT Service Database functionality allows application to implement desired Security Mode and Level using the BlueLitE Protocol Stack.

In brief, the BlueLitE SM APIs can be broadly categorized as shown in the table below:

SM API Category	Description/Remarks/Usage/Example

SM API Category	Description/Remarks/Usage/Example

Configuration APIs	These SMP platform APIs allow application to configure BlueLitE Security Manager for default configurations of IO capability, OOB data availability, Maximum Encryption Key size, Key Distribution information and Local Key values.
	
	Example:
	`BT_smp_set_io_cap_pl()`
	`BT_smp_set_key_distribution_flag_pl()`
	`BT_smp_set_max_encryption_key_size_pl()`
	`BT_smp_set_oob_data_pl()`
	`BT_smp_set_key_distribution_flag_pl()`

Device Database APIs	These APIs allow application to get device security configuration and attributes. Example: `BT_smp_get_device_security_info()`
Callback Registration APIs	These API allow application to register Callbacks for UI Notification Callback. Example: `BT_smp_register_user_interface()`

UI Notification Reply APIs	These APIs allows application to provide replies for the UI Notification Callback Requests. These APIs are applicable only when the application has registered its UI Notification Callback.
	
	Example:
	`BT_smp_authentication_request_reply()`
	`BT_smp_passkey_entry_request_reply()`
	`BT_smp_long_term_key_request_reply()`
	`BT_smp_key_exchange_info_request_reply()`

Security Procedure APIs	These set of Security Manager APIs enable application to make explicit request for link-level Authentication, Encryption etc. Example: `BT_smp_authenticate()`

For all Security Manager APIs, application must include the "`BT_smp_api.h`" header file. For all Security manager Platform APIs, application must include "smp_pl.h"

See Also

- For a detailed description of various Security Manager concepts (such as, "Authorization", "Authentication", "Security Modes" etc.), refer to the Security Manager API section in BlueLitE Stack API Documentation, Part-I [2].

HCI Application Event Indication Callback

This callback mechanism is provided to notify the application of incoming HCI "Event" packets, as they are received from the local Bluetooth device, or, the Host Controller hardware.

This HCI callback must be registered by the application, preferably using `BT_bluetooth_on()` API.

Not all received HCI Events are sent to application using the callback. The HCI Events, which are more appropriate for Security related functionalities, can be made available to application via the SM UI Notification Callback.

See Also

- An example of application calling registering the HCI Application Event Indication Callback.
- For complete description of these HCI Events packets, and corresponding packet formats, please refer to the Specification of the Bluetooth System, v4.0, Host Controller Interface Functional SpecificatioN.

SMP UI Notification Callback

This Callback mechanism is provided by the Security Manager Module to notify the application that a certain security related event has been received, which may need application or user intervention – such as providing Bluetooth Passkey, Long Term Key, accepting an Authentication Requestetc.

The HCI Events, which are provided using the SMP UI Notification Callback, are:

- LE Long Term Key Request Event

The application must register its UI Notification Callback with SM, using the `BT_smp_register_user_interface()` API [2].

See Also

- An example of application calling registering the SM UI Notification Callback.
- For a detailed description of the SMP UI Notification Callback, and how to use it, refer to the Security Manager API section in BlueLitE Stack API Documentation [2].

Examples for HCI & SMP API Usage

All the examples shown in this section assume a hypothetical UI front end from where user triggers various Bluetooth actions.

Bluetooth Device Discovery

Example: Advertising

```
#include "BT_common.h"
```

Example: Advertising

```
#include "BT_hci_api.h"


/*
    Other application codes here
*/

...
...
...

/* UI Device Initiate Advertising/Button Press Handler */
int UI_Device_Start_Advertising (params ...)
{
    API_RESULT retval;

    /**
     * Adv Data
     * Format: Tuple of (Length, AD Type, AD Data)
     * Some of the important AD Types
     * 0x01 : Flags (one octet Bit-Mask)
     *        bit 0: LE Limited Discoverable Mode
     *        bit 1: LE General Discoverable Mode
     *        bit 2: BR/EDR Not Supported
     * 0x02: 16-bit Service UUIDs
     *       More 16-bit UUIDs available
     * 0x03: 16-bit Service UUIDs
     *       Complete list of 16-bit UUIDs available
     * 0x08: Shortened local name
     * 0x09: Complete local name
     *
     * Flags (General discoverable, BR/EDR Not Supported)
     * Service (DIS and HTP Added) - Change as required.
     * Possible Values listed below
     *     #define GATT_DEVICE_INFO_SERVICE           0x180A
     *     #define GATT_HEALTH_THERMOMETER_SERVICE    0x1809
     *     #define GATT_PHONE_ALERT_STATUS_SERVICE    0x180E
     *
     *     #define GATT_LINK_LOSS_SERVICE             0x1803
     *     #define GATT_IMMEDIATE_ALERT_SERVICE       0x1802
     *     #define GATT_TX_POWER_SERVICE              0x1804
     *     #define GATT_FIND_ME_SERVICE               0x18A3
     *     #define GATT_BATTERY_SERVICE               0x1808
     *     #define GATT_CURRENT_TIME_SERVICE          0x1805
     *
     *     # Name: EtherMind
     */
```

Example: Advertising

```
UCHAR advertising_data[] = {
                                0x02, 0x01, 0x06, 0x05, 0x03, 0x0A, 0x18, 0x09,
                                0x18, 0x0A, 0x09, 0x45, 0x74, 0x68, 0x65, 0x72,
                                0x4D, 0x69, 0x6E, 0x64
                            };

/* Advertising Parameters */
UINT16   advertising_interval_min;
UINT16   advertising_interval_max;
UCHAR    advertising_type;
UCHAR    own_addr_type;
UCHAR    direct_addr_type;
UCHAR    direct_addr[BT_BD_ADDR_SIZE] = {/* Peer BD Address */};
UCHAR    advertising_channel_map;
UCHAR    advertising_filter_policy;

...
...

/* Set Advertising Data */
retval = BT_hci_le_set_advertising_data
         (
             sizeof (advertising_data),  /* Length of Advertising Data */
             advertising_data            /* Advertising Data */
         );

/*
    Do Error Handling here
*/

/* Set Advertising Parameters */
advertising_interval_min = ...;
advertising_interval_max = ...;
advertising_type = ...;
own_addr_type = ...;
direct_addr_type = ...;
advertising_channel_map = ...;
advertising_filter_policy = ...;

retval = BT_hci_le_set_advertising_parameters
         (
             advertising_interval_min,
             advertising_interval_max,
             advertising_type,
             own_addr_type,
             direct_addr_type,
             direct_addr,
             advertising_channel_map,
```

Example: Advertising

```
                advertising_filter_policy
            );

    /*
        Do Error Handling here
    */
    ...
    ...
}


/* HCI Application Event Indication Handler */
API_RESULT UI_hci_appl_event_ind_cb
           (
               UCHAR   event_type,
               UCHAR * event_data,
               UCHAR   event_datalen
           )
{
    UCHAR status;
    UCHAR value_1;
    UINT16 value_2;

    /* Handle Event Type, based on HCI Events */
    switch (event_type)
    {
    case HCI_COMMAND_COMPLETE_EVENT:
        /* Number of Command Packets */
        hci_unpack_1_byte_param(&value_1, event_data);
        event_data += 1;

        /* Command Opcode */
        hci_unpack_2_byte_param(&value_2, event_data);
        event_data += 2;

        /* Command Status */
        hci_unpack_1_byte_param(&status, event_data);
        event_data += 1;

        /* Check for HCI_LE_SET_ADVERTISING_PARAMETERS_OPCODE */
        if (HCI_LE_SET_ADVERTISING_PARAMETERS_OPCODE == value_2)
        {
            if (API_SUCCESS == status)
            {
                retval = BT_hci_le_set_advertising_enable
                         (
                             /* advertising_enable */
                             0x01
```

Example: Advertising

```
                            );
                }
            }

            break;

    /* Handling for other HCI Events */
    ...
    ...
    }

    return API_SUCCESS;
}
```

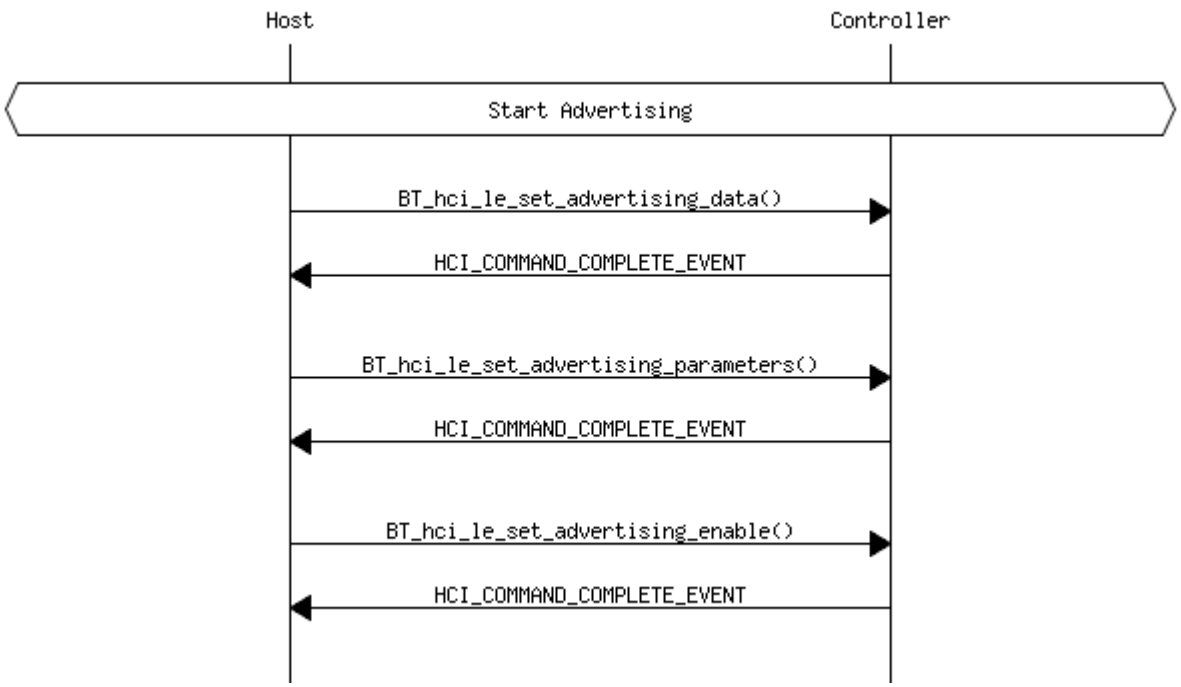


Figure 5: Advertising

Example: Scanning

```
#include "BT_common.h"
#include "BT_hci_api.h"


/*
    Other application codes here
```

Example: Scanning

```
*/

...
...
...

/* UI Device Start Scanning/Button Press Handler */
int UI_Device_Start_Scanning (params ...)
{
    API_RESULT retval;

    /* Scan Parameters */
    UCHAR    le_scan_type;
    UINT16   le_scan_interval;
    UINT16   le_scan_window;
    UCHAR    own_addr_type;
    UCHAR    scan_filter_policy;

    ...
    ...

    le_scan_type = ...;
    le_scan_interval = ...;
    le_scan_window = ...;
    own_addr_type = ...;
    scan_filter_policy = ...;

    /* Set Scan Parameters */
    retval = BT_hci_le_set_scan_parameters
             (
                 le_scan_type,
                 le_scan_interval,
                 le_scan_window,
                 own_addr_type,
                 scan_filter_policy
             );

    /*
        Do Error Handling here
    */

    ...
    ...
}


/* HCI Application Event Indication Handler */
API_RESULT UI_hci_appl_event_ind_cb
```

Example: Scanning

```
            (
                UCHAR   event_type,
                UCHAR * event_data,
                UCHAR   event_datalen
            )
{
    UCHAR status;
    UCHAR value_1;
    UINT16 value_2;

    /* Handle Event Type, based on HCI Events */
    switch (event_type)
    {
    case HCI_COMMAND_COMPLETE_EVENT:
        /* Number of Command Packets */
        hci_unpack_1_byte_param(&value_1, event_data);
        event_data += 1;

        /* Command Opcode */
        hci_unpack_2_byte_param(&value_2, event_data);
        event_data += 2;

        /* Command Status */
        hci_unpack_1_byte_param(&status, event_data);
        event_data += 1;

        /* Check for HCI_LE_SET_SCAN_PARAMETERS_OPCODE */
        if (HCI_LE_SET_SCAN_PARAMETERS_OPCODE == value_2)
        {
            if (API_SUCCESS == status)
            {
                retval = BT_hci_le_set_scan_enable
                         (
                             0x01 /* le_scan_enable */,
                             0x01 /* filter_dups */
                         );
            }
        }

        break;

    case HCI_LE_META_EVENT:
        hci_unpack_1_byte_param(&sub_event_code, event_data);
        event_data = event_data + 1;
        switch(sub_event_code)
        {
        case HCI_LE_ADVERTISING_REPORT_SUBEVENT:
            /* Number of Responses */
```

Example: Scanning

```
            hci_unpack_1_byte_param(&num_reports, event_data);
            event_data += 1;

            /* For each Response, Print the Inquiry Result */
            for (count = 0; count < num_reports; count ++)
            {
                hci_unpack_1_byte_param(&event_type, event_data);
                event_data += 1;
                hci_unpack_1_byte_param(&address_type, event_data);
                event_data += 1;
                address = event_data;
                event_data += 6;
                hci_unpack_1_byte_param(&length_data, event_data);
                event_data += 1;
                data = event_data;
                event_data += length_data;
                hci_unpack_1_byte_param(&rssi, event_data);
                event_data += 1;
            }

            break;

        /* Handling for other HCI BLE Sub-Events */
        ...
        ...
        }

    /* Handling for other HCI Events */
    ...
    ...
    }

    return API_SUCCESS;
}
```

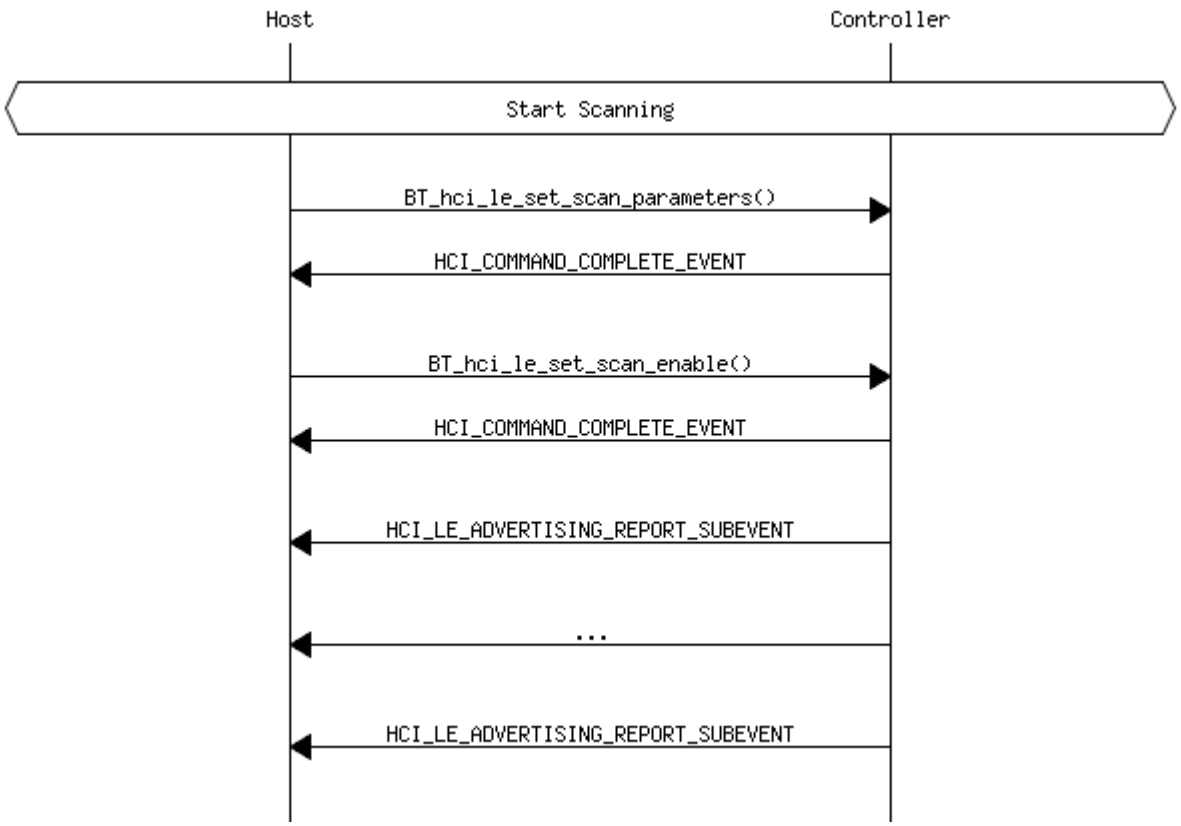


Figure 6: Scanning

Creating BLE Link

Example: Creating BLE Link

```
#include "BT_common.h"
#include "BT_hci_api.h"


/*
    Other application codes here
*/

...
...
...

/* UI Device BLE Connect Select/Button Press Handler */
int UI_Device_BLE_Connect (params ...)
{
    UINT16   le_scan_interval;
    UINT16   le_scan_window;
    UCHAR    initiator_filter_policy;
    UCHAR    peer_address_type;
```

Example: Creating BLE Link

```
UCHAR    peer_address[BT_BD_ADDR_SIZE];
UCHAR    own_address_type;
UINT16   conn_interval_min;
UINT16   conn_interval_max;
UINT16   conn_latency;
UINT16   supervision_timeout;
UINT16   maximum_ce_length;
UINT16   minimum_ce_length;
...
...

/* Set Connection Parameters */
le_scan_interval = ...;
le_scan_window = ...;
initiator_filter_policy = ...;
peer_address_type = ...;

/* Set Peer Device Address */
peer_address[0] = ...;
peer_address[1] = ...;
...
...

own_address_type = ...;
conn_interval_min = ...;
conn_interval_max = ...;
conn_latency = ...;
supervision_timeout = ...;
maximum_ce_length = ...;
minimum_ce_length = ...;

/* Initiate HCI LE Create Connection */
retval = BT_hci_le_create_connection
         (
             le_scan_interval,
             le_scan_window,
             initiator_filter_policy,
             peer_address_type,
             peer_address,
             own_address_type,
             conn_interval_min,
             conn_interval_max,
             conn_latency,
             supervision_timeout,
             maximum_ce_length,
             minimum_ce_length
         );
/*
```

Example: Creating BLE Link

```
        Do Error Handling here
    */
    ...
    ...
}


/* HCI Application Event Indication Handler */
API_RESULT UI_hci_appl_event_ind_cb
           (
               UCHAR   event_type,
               UCHAR * event_data,
               UCHAR   event_datalen
           )
{
    UCHAR  * peer_addr;
    UINT16   conn_handle;
    UINT16   conn_interval;
    UINT16   conn_latency;
    UINT16   super_vision_timeout;
    UCHAR    status;
    UCHAR    clock_accuracy;
    UCHAR    role;
    UCHAR    peer_addr_type;

    /* Handle Event Type, based on HCI Events */
    switch (event_type)
    {
    case HCI_LE_META_EVENT:
        hci_unpack_1_byte_param(&sub_event_code, event_data);
        event_data = event_data + 1;
        switch(sub_event_code)
        {
        case HCI_LE_CONNECTION_COMPLETE_SUBEVENT:
            hci_unpack_1_byte_param(&status, event_data + 0);
            hci_unpack_2_byte_param(&conn_handle, event_data+1);
            hci_unpack_1_byte_param(&role, event_data +3);
            hci_unpack_1_byte_param(&peer_addr_type, event_data + 4);
            peer_addr = 5 + event_data;
            hci_unpack_2_byte_param(&conn_interval, event_data+11);
            hci_unpack_2_byte_param(&conn_latency, event_data+13);
            hci_unpack_2_byte_param(&super_vision_timeout, event_data+15);
            hci_unpack_1_byte_param(&clock_accuracy, event_data + 17);

        break;

        /* Handling for other HCI BLE Sub-Events */
        ...
```

Example: Creating BLE Link

```
        ...
        }

    /* Handling for other HCI Events */
    ...
    ...
    }

    return API_SUCCESS;
}
```

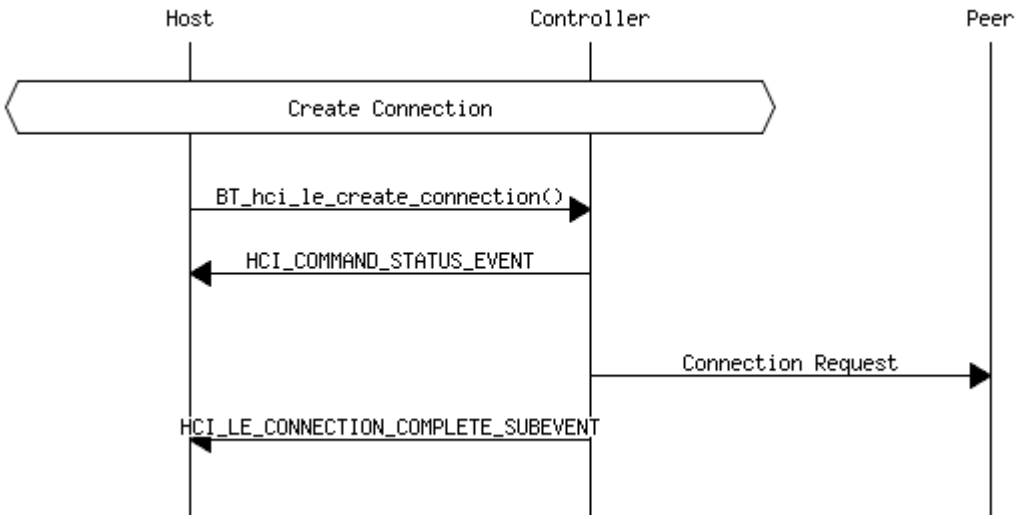


Figure 7: Creating BLE Link

Disconnecting BLE Link

Example: Disconnecting BLE Link

```
#include "BT_common.h"
#include "BT_hci_api.h"


/*
    Other application codes here
*/

...
...
...

/* UI Device BLE Disconnect Select/Button Press Handler */
int UI_Device_BLE_Disconnect (params ...)
```

Example: Disconnecting BLE Link

```
{
    UINT16 connection _handle;
    API_RESULT retval;
    UCHAR reason;

    ...
    ...

    /* Get Device Connection Handle for disconnection */
    acl_handle = ...;

    /* Set Reason for Disconnection */
    reason = 0x13;    /* Connection Terminated by Local Host */

    /* Initiate HCI Disconnection */
    retval = BT_hci_disconnect
             (
                 connection _handle,         /* ACL Connection Handle */
                 reason                      /* Reason for Disconnection */
             );

    /*
        Do Error Handling here
    */
    ...
    ...
}


/* HCI Application Event Indication Handler */
API_RESULT UI_hci_appl_event_ind_cb
           (
               UCHAR   event_type,
               UCHAR * event_data,
               UCHAR   event_datalen
           )
{
    UCHAR  status, reason;
    UINT16 connection_handle;

    /* Handle Event Type, based on HCI Events */
    switch (event_type)
    {
    case HCI_DISCONNECTION_COMPLETE_EVENT:
        /* Handle Disconnection Complete Event */

        /* Octet 0: Status of Disconnect */
        hci_unpack_1_byte_param(&status, event_data);
```

Example: Disconnecting BLE Link

```
        /* Octet 1-2: Connection Handle Disconnected */
        hci_unpack_2_byte_param(&connection_handle, event_data+1);

        /* Octet 3: Reason for Disconnection */
        hci_unpack_1_byte_param(&reason, event_data+3);

        /* Handle Disconnection Complete Event */
        ...
        ...

        break;

    /* Handling for other HCI Events */
    ...
    ...
    }

    return API_SUCCESS;
}
```

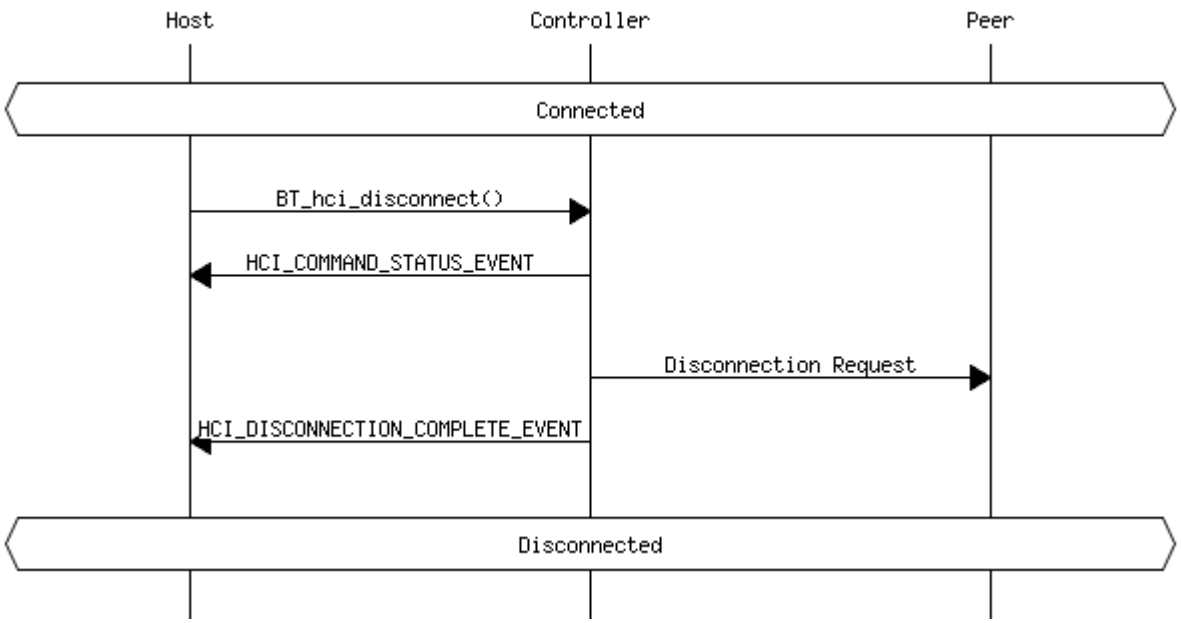


Figure 8: Disconnecting BLE Link

Updating Connection Parameters (from Master)

This example shows how to use BlueLitE HCI APIs to update Connection Parameter from the application on Master device.

Example: Updating Connection Parameters (from Master)

```
#include "BT_common.h"
#include "BT_hci_api.h"


/*
    Other application codes here
*/

...
...
...

/* UI Device BLE Connection Parameter Update/Button Press Handler */
int UI_Device_BLE_Connection_Parameter_Update (params ...)
{
    UINT16   connection_handle;
    UINT16   conn_interval_min;
    UINT16   conn_interval_max;
    UINT16   conn_latency;
    UINT16   supervision_timeout;
    UINT16   maximum_ce_length;
    UINT16   minimum_ce_length;
    ...
    ...

    /* Set Connection Update Parameters */
    connection_handle = ...;
    conn_interval_min = ...;
    conn_interval_max = ...;
    conn_latency = ...;
    supervision_timeout = ...;
    maximum_ce_length = ...;
    minimum_ce_length = ...;

    /* Initiate HCI LE Connection Update */
    retval = BT_hci_le_connection_update
             (
                 connection_handle,
                 conn_interval_min,
                 conn_interval_max,
                 conn_latency,
                 supervision_timeout,
                 maximum_ce_length,
                 minimum_ce_length
             );
    /*
        Do Error Handling here
    */
```

Example: Updating Connection Parameters (from Master)

```
    ...
    ...
}


/* HCI Application Event Indication Handler */
API_RESULT UI_hci_appl_event_ind_cb
           (
               UCHAR   event_type,
               UCHAR * event_data,
               UCHAR   event_datalen
           )
{
    UINT16   conn_handle;
    UINT16   conn_interval;
    UINT16   conn_latency;
    UINT16   super_vision_timeout;
    UCHAR    status;

    /* Handle Event Type, based on HCI Events */
    switch (event_type)
    {
    case HCI_LE_META_EVENT:
        hci_unpack_1_byte_param(&sub_event_code, event_data);
        event_data = event_data + 1;
        switch(sub_event_code)
        {
        case HCI_LE_CONNECTION_UPDATE_COMPLETE_SUBEVENT:
            hci_unpack_1_byte_param(&status, event_data + 0);
            hci_unpack_2_byte_param(&conn_handle, event_data+1);
            hci_unpack_2_byte_param(&conn_interval, event_data+3);
            hci_unpack_2_byte_param(&conn_latency, event_data+5);
            hci_unpack_2_byte_param(&super_vision_timeout, event_data+7);

            break;

        /* Handling for other HCI BLE Sub-Events */
        ...
        ...
        }

    /* Handling for other HCI Events */
    ...
    ...
    }

    return API_SUCCESS;
}
```

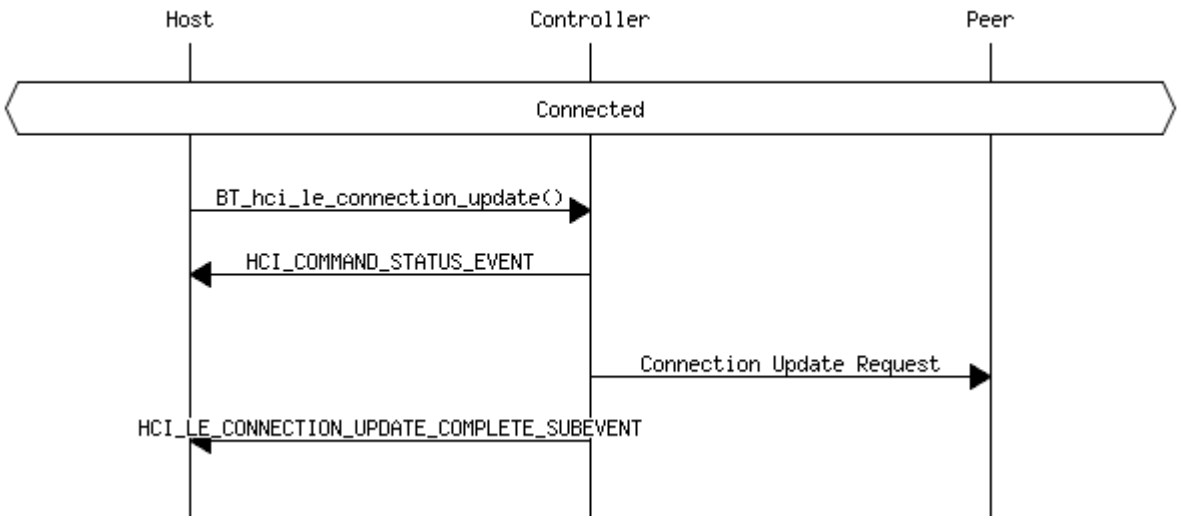


Figure 9: Updating Connection Parameters (from Master)

Updating Connection Parameters (from Slave)

This example shows how to use BlueLitE L2CAP APIs to update Connection Parameter from the application on Slave device.

Example: Updating Connection Parameters (from Slave)

```
#include "BT_common.h"
#include "BT_hci_api.h"


/*
    Other application codes here
*/

...
...
...

/* UI Device BLE Connection Parameter Update/Button Press Handler */
int UI_Device_BLE_Connection_Parameter_Update (params ...)
{
    DEVICE_QUEUE_HANDLE remote_bd_handle;
    UINT16   conn_interval_min;
    UINT16   conn_interval_max;
    UINT16   conn_latency;
    UINT16   supervision_timeout;
    ...
    ...

    /* Set Connection Update Parameters */
    remote_bd_handle = ...;
    conn_interval_min = ...;
```

Example: Updating Connection Parameters (from Slave)

```
    conn_interval_max = ...;
    conn_latency = ...;
    supervision_timeout = ...;

    /* Initiate HCI LE Connection Update */
    retval = BT_l2cap_le_connection_param_update_request
             (
                 &remote_bd_handle,
                 conn_interval_min,
                 conn_interval_max,
                 conn_latency,
                 supervision_timeout
             );
    /*
        Do Error Handling here
    */
    ...
    ...
}


/* L2CAP Application Event Indication Handler */
void UI_l2cap_appl_event_ind_cb
     (
         DEVICE_QUEUE_HANDLE * bd_handle,
         UCHAR     event,
         UCHAR *   data,
         UINT16    length
     )
{
    UINT16 reason, result;
    API_RESULT retval;

    /* Handle Event Type, based on L2CAP Events */
    if (L2CAP_CONNECTION_UPDATE_RESPONSE_EVENT == event)
    {
        appl_unpack_2_byte_param(result, &data[2]);

        UI_PRINT("Received : L2CAP_CONNECTION_UPDATE Response\n");
        UI_PRINT("\tResult       : %02X\n", result);
    }
    else if (L2CAP_COMMAND_REJECTED_EVENT  == event)
    {
        appl_unpack_2_byte_param(reason, &data[2]);

        UI_PRINT("Received : L2CAP_COMMAND_REJ\n");
        UI_PRINT("\tReason       : %02X\n", reason);
    }
```

Example: Updating Connection Parameters (from Slave)

```
    else
    {
        UI_PRINT("Received Invalid Event. Event = 0x%02X\n", event);
    }
}
```

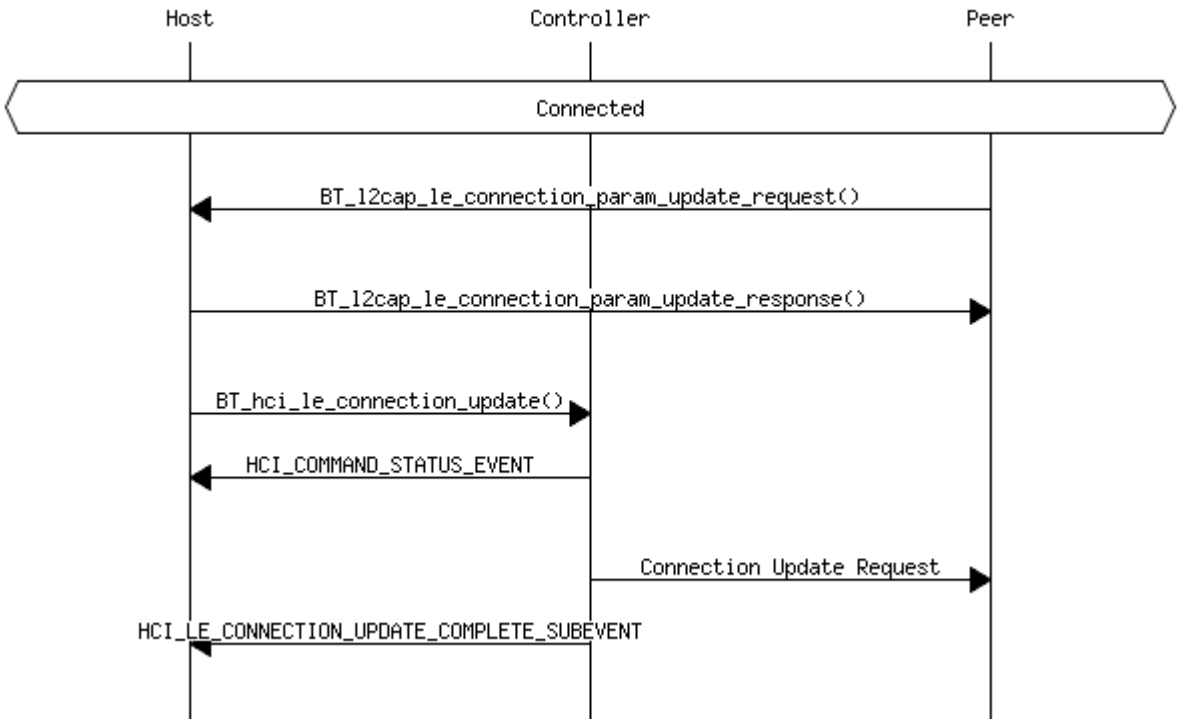


Figure 10: Updating Connection Parameters (from Slave)

Initiating Pairing

This example shows how to use BlueLitE SMP APIs to initiate pairing from the application.

Example: Initiating Pairing

```
#include "BT_common.h"
#include "BT_hci_api.h"


/*
    Other application codes here
*/

...
...
...

/* UI Initiate Pairing/Button Press Handler */
int UI_Pair_Remote_Device (params ...)
```

Example: Initiating Pairing

```
{
    API_RESULT    retval;

    SMP_KEY_DIST  * key_info;
    SMP_BD_ADDR   bd_addr;
    SMP_BD_HANDLE bd_handle;
    SMP_AUTH_INFO auth;

    unsigned int  read;

    /* Used only for SMP Slave Role */
    UINT16   ediv;
    UCHAR  * peer_rand;
    UCHAR    ltk[SMP_LTK_SIZE];

    ...
    ...

    UI_PRINT("Enter BD_ADDR : ");
    UI_get_bd_addr(bd_addr.addr);

    UI_PRINT("\n0 - Public\n1 - Random\n");
    UI_PRINT("Enter BD_ADDR type : ");
    UI_INPUT("%u", &read);
    bd_addr.type = (UCHAR) read;

    UI_PRINT(" 0 - Encryption Only/Unauthenticated (Without MITM)\n");
    UI_PRINT(" 1 - Authenticated (With MITM)\n");
    UI_PRINT("Enter Security level required : ");
    UI_INPUT("%u", &read);
    auth.security = (UCHAR) ((read)?SMP_SEC_LEVEL_2: SMP_SEC_LEVEL_1);

    /* If MITM is required */
    if (read)
    {
        UI_PRINT("Enter Platform IO Capability\n");
        UI_PRINT("(0-DisplayOnly, 1-DisplayYesNo, 2-KeybdOnly, 3-NoIO, 4-KeybdDisplay,
5-Default): ");
        UI_INPUT("%u", &read);

        /* Update platform IO Capability */
        BT_smp_set_io_cap_pl (read);
    }

    UI_PRINT("\n0 - non-Bonding\n1 - Bonding\n");
    UI_PRINT("Enter Bonding type : ");
    UI_INPUT("%u", &read);
    auth.bonding = (UCHAR) read;
```

Example: Initiating Pairing

```
    UI_PRINT("Enter Encryption Key size required : ");
    UI_INPUT("%u", &read);
    auth.ekey_size = (UCHAR)read;

    /* Get the BD handle */
    BT_smp_get_bd_handle (&bd_addr, &bd_handle);

    retval = BT_smp_authenticate
             (
                 &bd_handle,
                 &auth
             );

    /*
        Do Error Handling here
    */
    ...
    ...
}


/* SMP Application Event Indication Handler */
API_RESULT UI_smp_cb
           (
               IN SMP_BD_HANDLE   * bd_handle,
               IN UCHAR      event,
               IN API_RESULT status,
               IN UCHAR    * event_data,
               IN UINT16     data_len
           )
{
    API_RESULT retval;

    SMP_KEY_DIST * key_info;
    SMP_AUTH_INFO * auth;
    SMP_BD_ADDR bdaddr;

    UCHAR * bd_addr;
    UCHAR   bd_addr_type;

    /* Get the BD Address from handle */
    BT_smp_get_bd_addr (bd_handle, &bdaddr);

    bd_addr = bdaddr.addr;
    bd_addr_type = bdaddr.type;

    switch(event)
```

Example: Initiating Pairing

```
{
case SMP_AUTHENTICATION_COMPLETE:
    UI_PRINT("\nRecvd SMP_AUTHENTICATION_COMPLETE\n");
    UI_PRINT("BD Address : %02X %02X %02X %02X %02X %02X\n",
    bd_addr[0],bd_addr[1],bd_addr[2],bd_addr[3],bd_addr[4],bd_addr[5]);
    UI_PRINT("BD addr type : %s\n",
    (bd_addr_type == 0)? "Public Address": "Random Address");
    UI_PRINT("Status : %04X\n", status);

    if(status == API_SUCCESS)
    {
        auth = (SMP_AUTH_INFO *)event_data;

        UI_PRINT("Authentication type : %s\n",
        (SMP_SEC_LEVEL_2 == (auth->security & 0x0F))?  "With MITM":
        "Encryption Only (without MITM)");

        UI_PRINT("Bonding type : %s\n",
        (auth->bonding)? "Bonding": "Non-Bonding");

        UI_PRINT("Encryption Key size : %d\n", auth->ekey_size);

    }
    break;

case SMP_AUTHENTICATION_REQUEST:
    UI_PRINT("\nRecvd SMP_AUTHENTICATION_REQUEST\n");
    UI_PRINT("BD Address : %02X %02X %02X %02X %02X %02X\n",
    bd_addr[0],bd_addr[1],bd_addr[2],bd_addr[3],bd_addr[4],bd_addr[5]);
    UI_PRINT("BD addr type : %s\n",
    (bd_addr_type == 0)? "Public Address": "Random Address");

    auth = (SMP_AUTH_INFO *)event_data;

    UI_PRINT("Authentication type : %s\n",
    (SMP_SEC_LEVEL_2 == (auth->security & 0x0F))?  "With MITM":
    "Encryption Only (without MITM)");

    UI_PRINT("Bonding type : %s\n",
    (auth->bonding)? "Bonding": "Non-Bonding");

    UI_PRINT("Encryption Key size : %d\n", auth->ekey_size);

    /* To accept authentication request */
    auth->accept = SMP_ERROR_NONE;

    UI_PRINT("\n\nSending +ve Authentication request reply.\n");
    retval = BT_smp_authentication_request_reply
```

Example: Initiating Pairing

```
                (
                    bd_handle,
                    auth
                );
        break;

    case SMP_PASSKEY_ENTRY_REQUEST:
        UI_PRINT("\nEvent    : SMP_PASSKEY_ENTRY_REQUEST\n");
        UI_PRINT("BD Address : %02X %02X %02X %02X %02X %02X\n",
        bd_addr[0],bd_addr[1],bd_addr[2],bd_addr[3],bd_addr[4],bd_addr[5]);
        UI_PRINT("BD addr type : %s\n",
        (bd_addr_type == 0)? "Public Address": "Random Address");
        break;

    case SMP_PASSKEY_DISPLAY_REQUEST:
        UI_PRINT("\nEvent    : SMP_PASSKEY_DISPLAY_REQUEST\n");
        UI_PRINT("BD Address : %02X %02X %02X %02X %02X %02X\n",
        bd_addr[0],bd_addr[1],bd_addr[2],bd_addr[3],bd_addr[4],bd_addr[5]);
        UI_PRINT("BD addr type : %s\n",
        (bd_addr_type == 0)? "Public Address": "Random Address");

        UI_PRINT("Passkey : %06u", *((UINT32 *)event_data));

        break;

    case SMP_KEY_EXCHANGE_INFO_REQUEST:
        UI_PRINT("\nEvent    : SMP_KEY_EXCHANGE_INFO_REQUEST\n");
        UI_PRINT("BD Address : %02X %02X %02X %02X %02X %02X\n",
        bd_addr[0],bd_addr[1],bd_addr[2],bd_addr[3],bd_addr[4],bd_addr[5]);
        UI_PRINT("BD addr type : %s\n",
        (bd_addr_type == 0)? "Public Address": "Random Address");

        /* Get platform data of key informations */
        BT_smp_get_key_exchange_info_pl (&key_info);
        BT_smp_key_exchange_info_request_reply (bd_handle, key_info);
        break;

    case SMP_KEY_EXCHANGE_INFO:
        UI_PRINT ("\nRecvd SMP_KEY_EXCHANGE_INFO\n");
        UI_PRINT ("Status - 0x%04X\n", status);

        key_info = (SMP_KEY_DIST *)event_data;

        UI_PRINT ("Encryption Info:\n");
        appl_dump_bytes(key_info->enc_info, sizeof (key_info->enc_info));
        UI_PRINT ("Master Identification Info:\n");
        appl_dump_bytes(key_info->mid_info, sizeof (key_info->mid_info));
        UI_PRINT ("Identity Info:\n");
```

Example: Initiating Pairing

```
        appl_dump_bytes(key_info->id_info, sizeof (key_info->id_info));
        UI_PRINT ("Identity Address Info:\n");
        appl_dump_bytes(key_info->id_addr_info, sizeof (key_info->id_addr_info));
        UI_PRINT ("Signature Info:\n");
        appl_dump_bytes(key_info->sign_info, sizeof (key_info->sign_info));

        break;

    /* For SMP Slave Role */
    case SMP_LONG_TERM_KEY_REQUEST:
        /* Copy parameters to local variables */
        smp_unpack_2_byte_param(&ediv, &event_data[8]);
        peer_rand = event_data;

        UI_PRINT("\nEvent    : SMP_LONG_TERM_KEY_REQUEST\n");
        UI_PRINT("BD Address : %02X %02X %02X %02X %02X %02X\n",
        bd_addr[0],bd_addr[1],bd_addr[2],bd_addr[3],bd_addr[4],bd_addr[5]);
        UI_PRINT("BD addr type : %s\n",
        (bd_addr_type == 0)? "Public Address": "Random Address");
        UI_PRINT("Div  : 0x%04X\n", ediv);
        UI_PRINT("Rand : %02X %02X %02X %02X %02X %02X %02X %02X\n",
        peer_rand[0], peer_rand[1], peer_rand[2], peer_rand[3],
        peer_rand[4], peer_rand[5], peer_rand[6], peer_rand[7]);

        /* Get LTK for remote device */
        retval = BT_smp_get_long_term_key_pl
                 (
                     peer_rand,
                     ediv,
                     ltk
                 );

        if(retval == API_SUCCESS)
        {
            UI_PRINT("\n\nSending +ve LTK request reply.\n");
            retval = BT_smp_long_term_key_request_reply
                     (
                         bd_handle,
                         ltk,
                         SMP_TRUE
                     );
        }
        else
        {
            UI_PRINT("\n\nSending -ve LTK request reply.\n");
            retval = BT_smp_long_term_key_request_reply
                     (
                         bd_handle,
```

Example: Initiating Pairing

```
                        NULL,
                        SMP_FALSE
                    );
        }
        break;

    /* Handling for other SMP Events */
    ...
    ...
    }

    return API_SUCCESS;
}
```

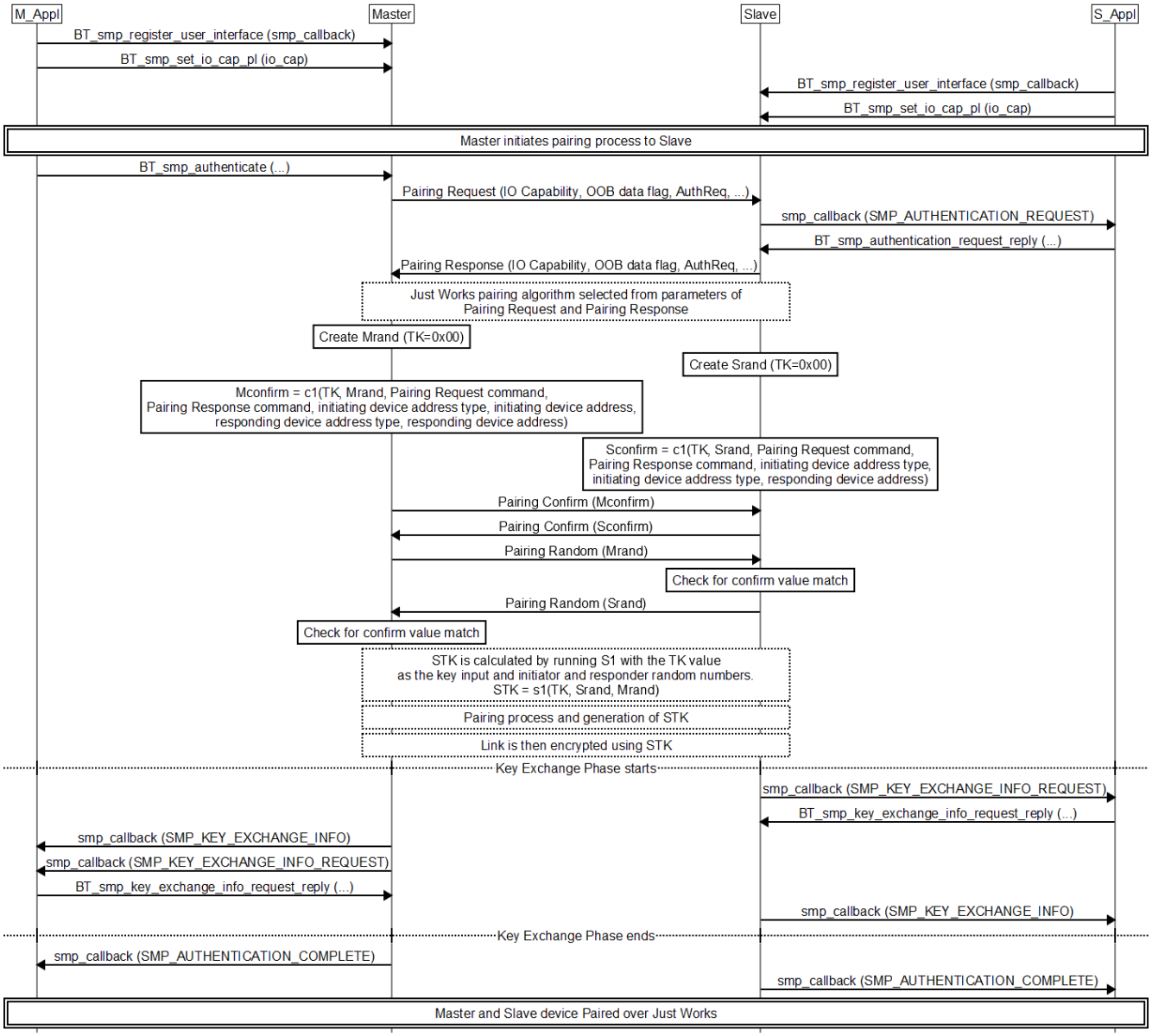


Figure 11: Pairing (Just Works)

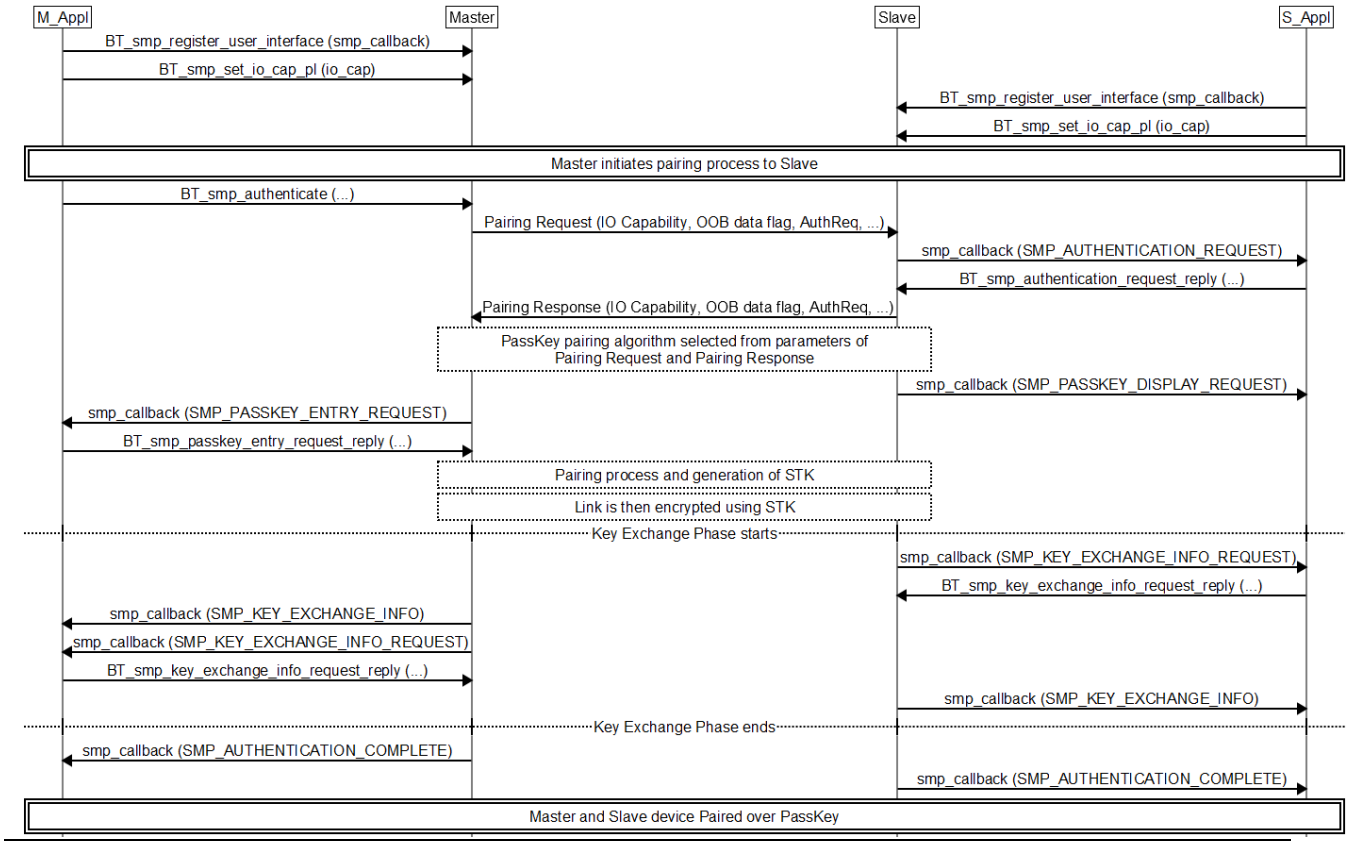


Figure 12: Pairing (Passkey)

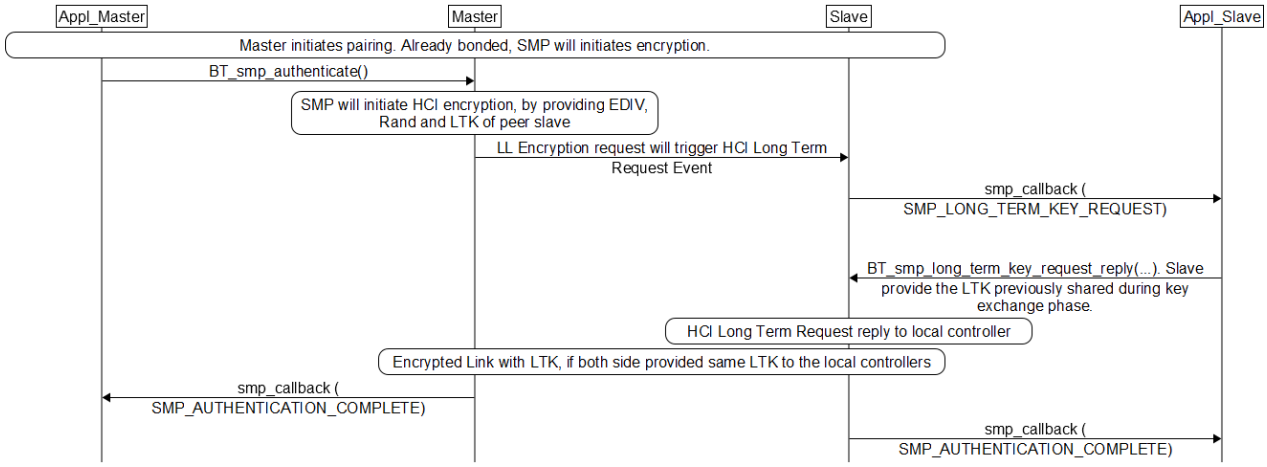


Figure 13: Re-Authentication

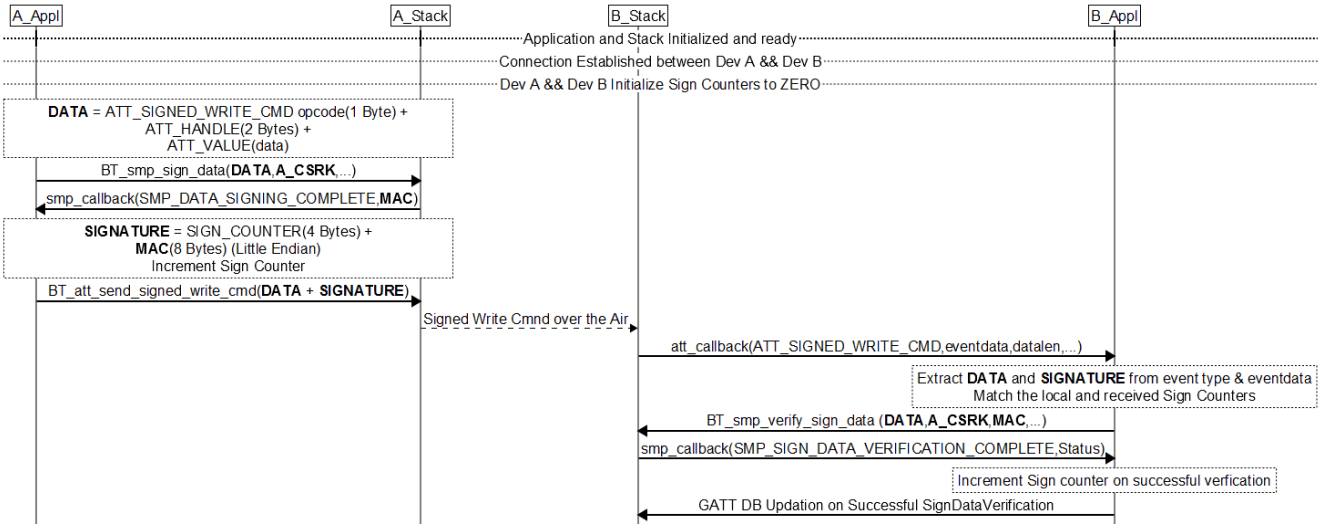


Figure 14: Data Signing

Step 3: How to Use BlueLitE ATT/GATT APIs

Once the BLE connection has been created for a remote Bluetooth device, ATT is also considered connected. The ATT Protocol provides a means for application to discover which services are available on the remote Bluetooth device, and characteristics & capabilities of those services.

The BlueLitE ATT provides two different kinds of APIs for application:

- APIs for Configuration of Services (Locally or requested from remote device)

  This set of ATT APIs enables application to configure services based on the request from remote device, and characteristics of those services. The BlueLitE GATT Database APIs are also included in this category.

  The BlueLitE GATT Database APIs can also be used to configure services and its characteristics supported by local dervice.

- APIs for Discovery of Remote Services

  This set of ATT APIs enable application to discover remotely offered services on a remote Bluetooth device. The application is required to register ATT Callback with ATT to receive confirmation of initiated ATT Client operations.

APIs for Configuration of Local Services

The APIs in this category are primary from the BlueLitE GATT Database module.

All the information about locally supported Bluetooth service (for example, Bluetooth Profile) is stored in a GATT Service Record. The GATT Database acts as a prime repository for the GATT server to store information about the different kinds of services/profiles it supports. Thus, the GATT Database is the main interface for various application, and services/profiles, to store information about their respective services.

The BlueLitE Protocol Stack has its own optimized GATT Database implementation, keeping memory restriction in the embedded application environment in mind. The BlueLitE GATT Database is configured statically before compilation of the stack based on the various kinds of services the current system is required to provide.

During runtime, the BlueLitE GATT Database being a Static Database, it is not possible to:

- To add a new GATT Service Record
- To add a new Characteristic to an existing Service Record

To add a new Service Record, or, a new attribute to an existing Service Record, the Static GATT Database has to be re-configured again with the appropriate configuration.

However, during runtime, the BlueLitE GATT Database module provides APIs for the following features:

- To change/modify the values of an existing Service Attribute for an existing Service Record

The following APIs are provided for application, and services/profiles, to interact with the GATT Static Database module, to make use of the above features:

GATT Database API	Description/Remarks/Usage/Example
BT_gatt_db_access_handle ()	To Read/Write Attribute Value identified by Attribute Handle
BT_gatt_db_set_char_val ()	To update characteristic value

For all GATT Database APIs, application must include the "BT_gatt_db_api.h" header file.

See Also

For a detailed description of the above APIs, refer to the BlueLitE Stack API Document [2].

Example for GATT Database API Usage

Changing local characteristic value

The following example explains the use of GATT Database APIs:

Example: Changing local characteristic value

```
#include "BT_common.h"
#include "BT_gatt_db_api.h"


/*
    Other application codes here
*/

...
...
...

/* UI GATT Database Update Characteristic/Button Press Handler */
int UI_GATT_Database_Update_Characteristic_Value (params ...)
{
    API_RESULT        retval;
    GATT_DB_HANDLE    handle;
    ATT_VALUE         attr_value;

    ...
    ...

    /* Set GATT Database Handle */
    handle = ...;

    /* Set Attribute Value */
    attr_value = ...;
```

Example: Changing local characteristic value

```
    retval = BT_gatt_db_set_char_val
             (
                &handle,
                &attr_value
             );

    if (API_SUCCESS != retval)
    {
        /*
            Do Error Handling here
        */
        ...
    }

    ...
    ...
}
```

APIs for Discovery and Access of Remote Services/Characteristics

The BlueLitE ATT Protocol Client APIs can be used to discover and access various Bluetooth services offered by a remote Bluetooth device.

For all ATT APIs, application must include the "`BT_att_api.h`" header file.

ATT Request APIs

The client devices use the ATT Request APIs to discover and access the services provided by the server. The ATT Callback function registered with ATT is called to indicate the completion of the operation initiated by ATT Request API.

The BlueLitE ATT Request APIs are briefly described in the table below:

ATT Request API Category	Description/Remarks/Usage/Example

Primary Service Discovery	These APIs enable application to discover all primary services or a specific primary service based on Service UUID..
	
	Example:
	`BT_att_send_read_by_group_req ()`
	`BT_att_send_find_by_type_val_req ()`

ATT Request API Category	Description/Remarks/Usage/Example
Characteristic Discovery	These APIs enable application to discover all characteristics or a specific characteristic based on its UUID. Example: `BT_att_send_read_by_type_req ()`
Characteristic Descriptor Discovery	These APIs enable application to discover all characteristic descriptors. Example: `BT_att_send_find_info_req ()`

Access/Change Characteristic Values	These APIs enable application to read and update characteristic values.
	
	Example:
	`BT_att_send_read_req ()`
	`BT_att_send_read_blob_req ()`
	`BT_att_send_write_req ()`
	`BT_att_send_prepare_write_req ()`
	`BT_att_send_execute_write_req ()`

See Also

For a detailed description of the ATT Request APIs, refer to the BlueLitE Stack API Document [2].

ATT Callback for Discovery and Access of Remote Services

For ATT Request APIs, ATT invokes an application registered callback function to inform the application of the result and response of certain ATT requests. The application must register the callback function with ATT before calling any ATT APIs.

See Also

For a detailed description of the ATT Request APIs, refer to the BlueLitE Stack API Document [2].

Examples of Using GATT APIs for Discovery and Access of Remote Services

All the examples shown in this section assume a hypothetical UI front end from where user triggers various Bluetooth actions.

GATT Service Discovery

Example: GATT Service Discovery

```
#include "BT_common.h"
#include "BT_att_api.h"
#include "BT_gatt_db_api.h"


/*
    Other application codes here
*/

...
...
...

/* UI GATT Service Discovery/Button Press Handler */
void UI_GATT_Service_Discovery(params ...)
{
    ATT_HANDLE_RANGE   range;
    UINT16             uuid;
    ATT_READ_BY_GROUP_TYPE_REQ_PARAM    find_info_param;
    API_RESULT   retval;

    ...
    ...

    range.start_handle = ATT_ATTR_HANDLE_START_RANGE;
    range.end_handle = ATT_ATTR_HANDLE_END_RANGE;


    /* Set the required UUID */
    uuid = ...;

    find_info_param.range = range;
    find_info_param.group_type = uuid;
    find_info_param.uuid_format = ATT_16_BIT_UUID_FORMAT;

    retval = BT_att_send_read_by_group_req
             (
                 &appl_gatt_client_handle,
                 &find_info_param
             );

    UI_PRINT(
    "[ATT]: Read by Group Type Request Initiated with result 0x%04X",retval);


    if (API_SUCCESS != retval)
    {
        /*
            Do Error Handling here
        */
        ...
    }
```

Example: GATT Service Discovery

```
    ...
    ...
}


/* ATT Callback Handler */
API_RESULT UI_att_callback
           (
               ATT_HANDLE    * handle,
               UCHAR         att_event,
               API_RESULT    event_result,
               UCHAR         * event_data,
               UINT16        event_datalen
           )
{
    UINT16 attribute_handle;

    /* Handle ATT Events */
    switch (att_event)
    {
    case ATT_ERROR_RSP:
        BT_UNPACK_LE_1_BYTE(&op_code,event_data);
        BT_UNPACK_LE_2_BYTE(&attr_handle, event_data+1);
        BT_UNPACK_LE_1_BYTE(&rsp_code,event_data+3);
        UI_PRINT (
        "Received Error Response, for Op-Code 0x%02X for Handle 0x%04X, Rsp "
        "Code 0x%02X!\n", op_code, attr_handle,rsp_code);

        /* Handle Error Response */
        ...
        ...

        break;

    case ATT_READ_BY_GROUP_RSP:
        if (NULL == event_data)
        {
            break;
        }

        /* Handle Read By Group Type Response */

        /*
         * After Discovering all the primary and secondary services,
         * initiate Characteristic Discovery.
         */
        UI_GATT_Characteristic_Discovery(...);
```

Example: GATT Service Discovery

```
        break;

    case ATT_READ_BY_TYPE_RSP:
        if (NULL == event_data)
        {
            break;
        }

       /*
        * After Discovering all the Characteristics, read a characteristic value.
        */

      /* Select Attribute Handle to read */
      attribute_handle = ...;

      retval = BT_att_send_read_req
               (
                   &appl_gatt_client_handle,
                   &attribute_handle
               );

         if (API_SUCCESS != retval)
         {
            /*
                Do Error Handling here
             */
            ...
          }

        break;

    case ATT_READ_RSP:
        UI_PRINT ("Received Read Response Opcode!\n");

        UI_dump_bytes(event_data, event_datalen);
        break;

    /* Handling for other ATT Event Types */
    ...
    ...
    }

    return;
}

void UI_GATT_Characteristic_Discovery(params ...)
{
    ATT_HANDLE_RANGE   range;
    UINT16             uuid;
    ATT_READ_BY_TYPE_REQ_PARAM    find_info_param;
    API_RESULT   retval;

    ...
```

Example: GATT Service Discovery

```
    ...

    /* Set Range of Attribute Handle for the service */
    range.start_handle = ...;
    range.end_handle = ...;


    /* Set the required UUID */
    uuid = ...;

    find_info_param.range = range;
    find_info_param.group_type = uuid;
    find_info_param.uuid_format = ATT_16_BIT_UUID_FORMAT;

    retval = BT_att_send_read_by_type_req
             (
                 &appl_gatt_client_handle,
                 &find_info_param
             );

    UI_PRINT(
    "[ATT]: Read by Type Request Initiated with result 0x%04X",retval);


    if (API_SUCCESS != retval)
    {
        /*
            Do Error Handling here
        */
        ...
    }


    ...
    ...
}
```

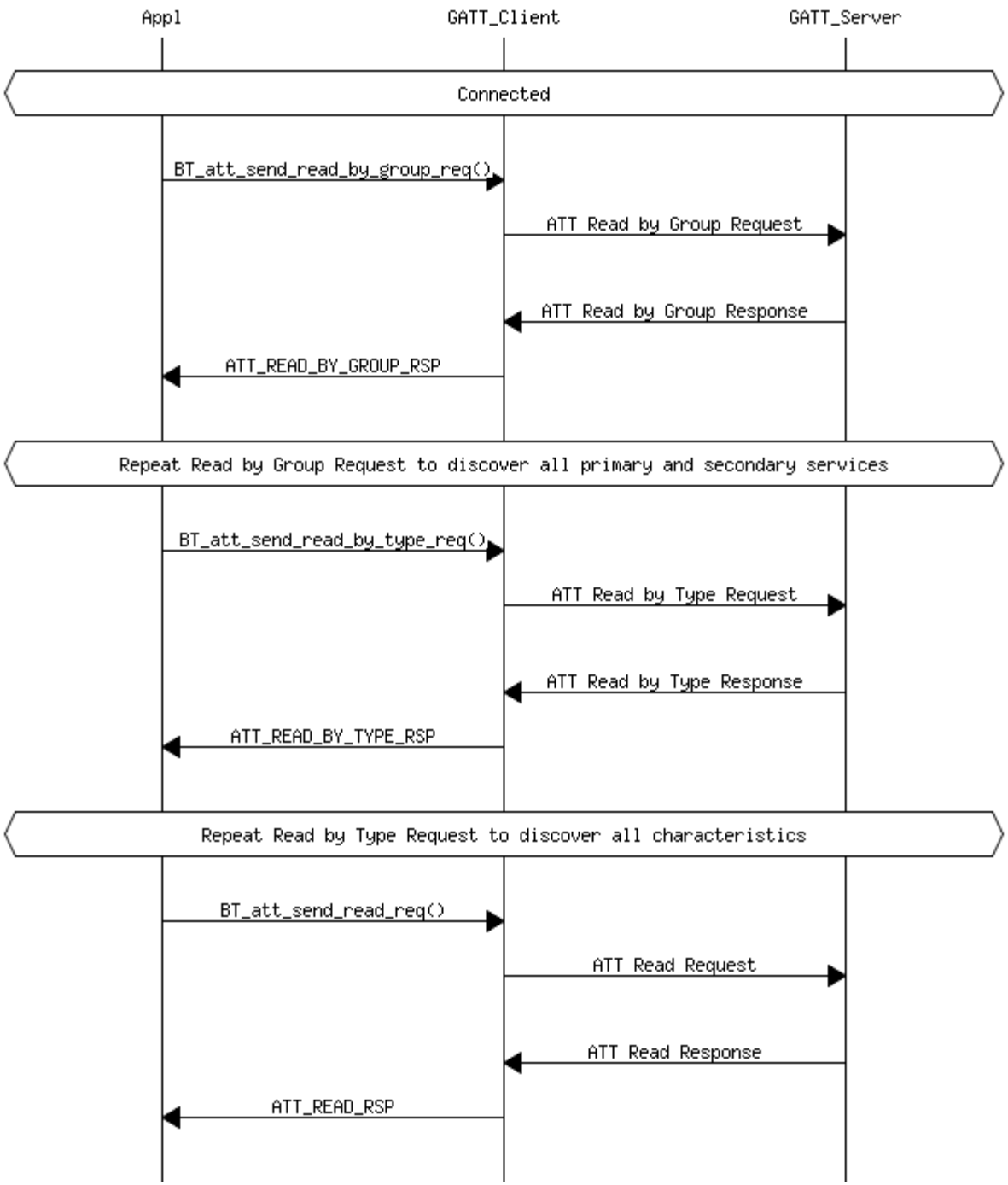


Figure 15: GATT Service Discovery

ATT Write Request

Example: ATT Write Request

```
/* ATT Callback Handler */
API_RESULT UI_att_callback
           (
```

Example: ATT Write Request

```
                ATT_HANDLE    * handle,
                UCHAR         att_event,
                API_RESULT    event_result,
                UCHAR         * event_data,
                UINT16        event_datalen
            )
{
    API_RESULT retval;
    UINT16 attribute_handle;
    UINT16 attr_val_len;

    /* Handle ATT Events */
    switch (att_event)
    {
    case ATT_WRITE_REQ:
        BT_UNPACK_LE_2_BYTE(&attribute_handle, event_data);
        /* 2 octets of attribute handle, everything else is attribute value */
        attr_val_len = event_datalen - 2;
        UI_PRINT (
        "Received Write Request for handle 0x%04X, Attribute Value Length 0x%04X\n",
        attribute_handle, attr_val_len);

        /* Handle Write Request */
        ...
        ...
        retval = BT_att_write_rsp
                 (
                      handle
                 );

        UI_PRINT (
        "[ATT]: Write Response sent with result 0x%04X\n",retval);
        break;

    default:
        break;
    }
}
```

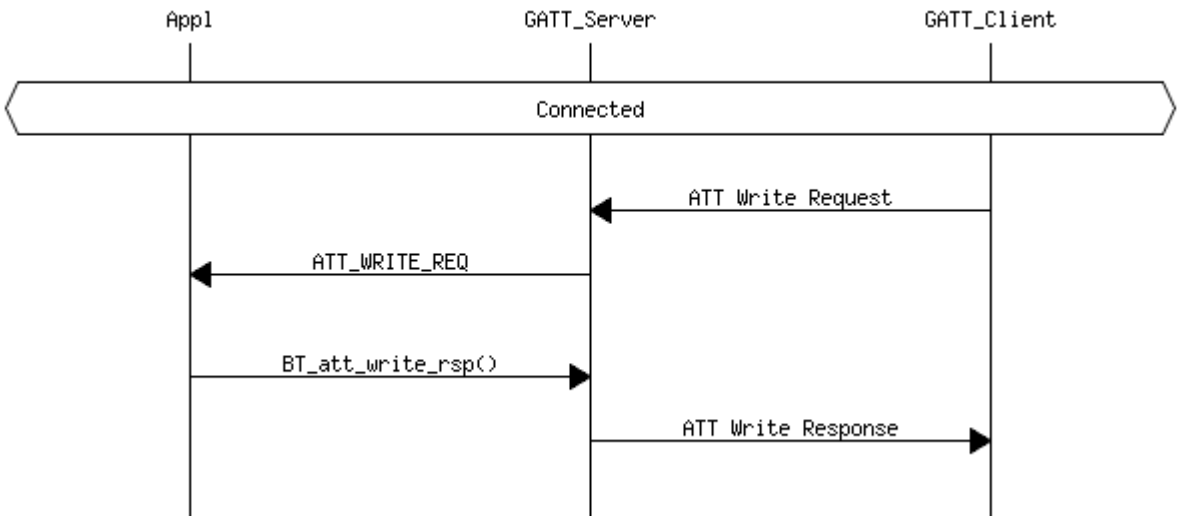


Figure 16: ATT Write

ATT Read Request

Example: ATT Read Request

```
/* ATT Callback Handler */
API_RESULT UI_att_callback
           (
               ATT_HANDLE    * handle,
               UCHAR         att_event,
               API_RESULT    event_result,
               UCHAR         * event_data,
               UINT16        event_datalen
           )
{
    API_RESULT retval;
    ATT_READ_RSP_PARAM  rsp_param;
    UINT16 attribute_handle;

    /* Handle ATT Events */
    switch (att_event)
    {
    case ATT_READ_REQ:
        BT_UNPACK_LE_2_BYTE(&attribute_handle, event_data);
        UI_PRINT (
        "Received Read Request for handle 0x%04X\n",attribute_handle);

        /**
         *  Handle Read Request, fill appropriate attribute value and length in
         *  rsp_param
         */
        ...
        ...
        retval = BT_att_read_rsp
```

Example: ATT Read Request

```
                    (
                        handle,
                        &rsp_param
                    );

            UI_PRINT (
            "[ATT]: Read Response sent with result 0x%04X\n",retval);
            break;

        default:
            break;
        }
    }
```

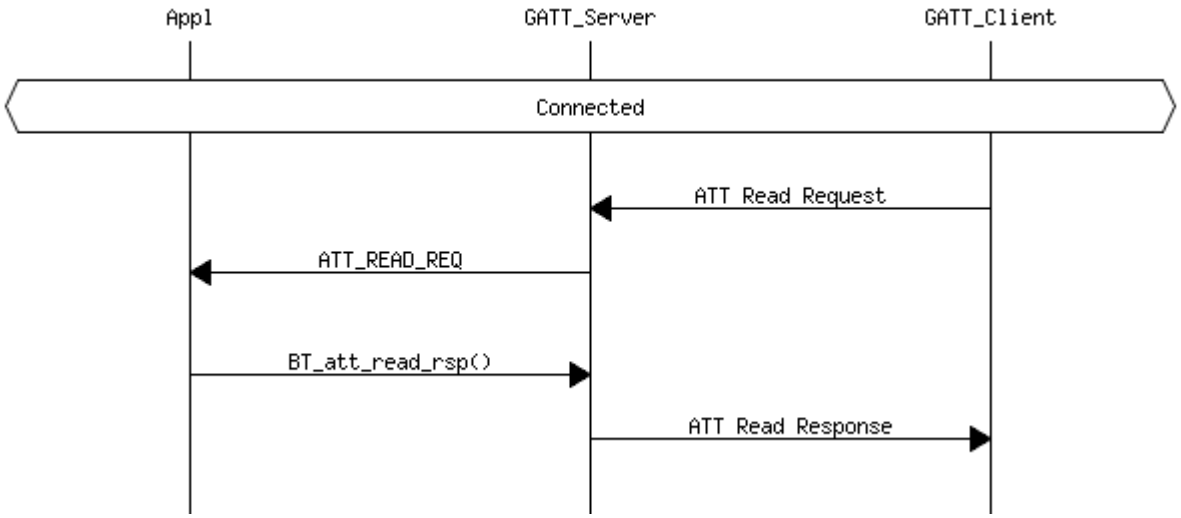


Figure 17: ATT Read

ATT Handle Value Indication

Example: ATT Handle Value Indication

```
API_RESULT UI_send_measurement_ind (void)
{
    ATT_HNDL_VAL_IND_PARAM    hndl_val_param;
    API_RESULT                retval;

    /**
     *  Initialize hndl_val_param to Characteristic Value Handle and Value
     *  appropriately
     */
    retval = BT_att_send_hndl_val_ind
             (
                 &ui_att_server_handle,
                 &hndl_val_param
             );
```

Example: ATT Handle Value Indication

```
     UI_PRINT (
     "[ATT]: Handle Value Indication Sent with result 0x%04X", retval);

     return retval;
}

/* ATT Callback Handler */
API_RESULT UI_att_callback
           (
               ATT_HANDLE    * handle,
               UCHAR         att_event,
               API_RESULT    event_result,
               UCHAR         * event_data,
               UINT16        event_datalen
           )
{
    /* Handle ATT Events */
    switch (att_event)
    {
    case ATT_HANDLE_VALUE_CNF:

        UI_PRINT (
        "[ATT]: Handle Value Confirmation Received);
        break;

    default:
        break;
    }
}
```

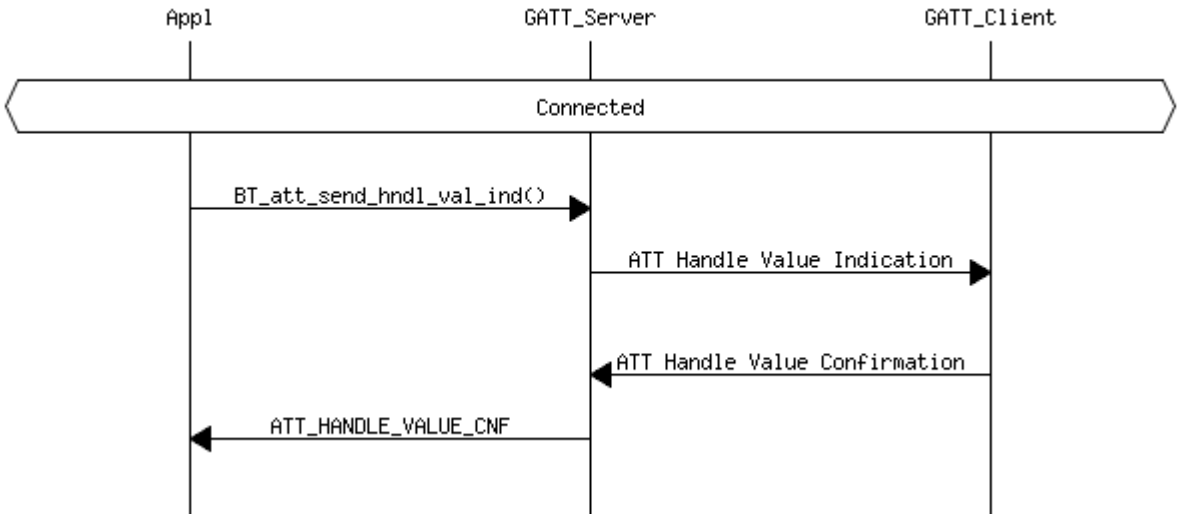


Figure 18: ATT Indication

ATT Handle Value Notification

Example: ATT Handle Value Notification

```
API_RESULT UI_send_measurement_ntf (void)
{
     ATT_HNDL_VAL_NTF_PARAM    hndl_val_param;
     API_RESULT                retval;

     /**
      *  Initialize hndl_val_param to Characteristic Value Handle and Value
      *  appropriately
      */
     retval = BT_att_send_hndl_val_ntf
              (
                  &ui_att_server_handle,
                  &hndl_val_param
              );

     UI_PRINT (
     "[ATT]: Handle Value Confirmation Sent with result 0x%04X", retval);

     return retval;
}
```

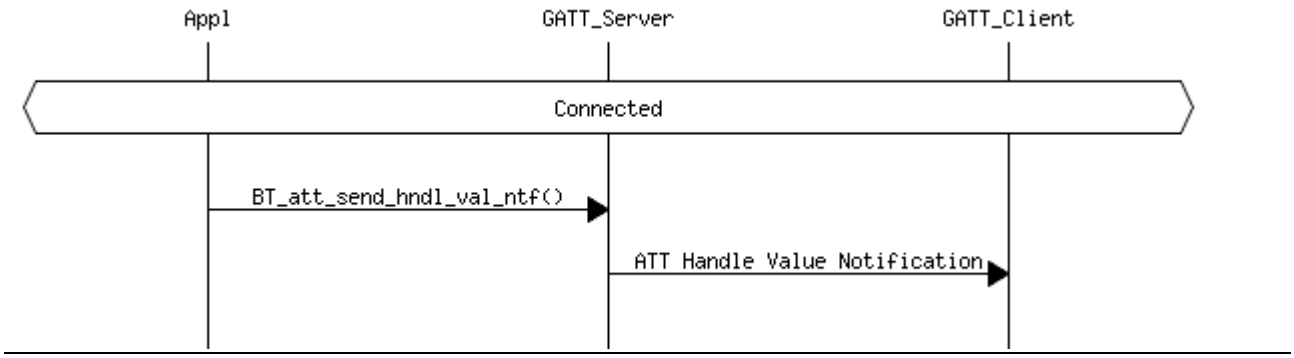


Figure 19: ATT Notification

Frequently Asked Questions

Does one need to be Bluetooth aware to be able to write application Using the BlueLitE Protocol Stack APIs?

Yes, to some extent. However, an application developer is not required to know everything about Bluetooth wireless technology to be able to write application using BlueLitE Protocol Stack & Profile APIs. In general, knowledge on the following Bluetooth wireless technology topics should be sufficient:

- Architecture of Bluetooth Protocol System & Profiles
- Common Bluetooth Nomenclatures:
    - Bluetooth Device Address (BD_ADDR)
    - Universally Unique Identifier
    - GATT Database
    - Profiles, Services and Characteristics
    - Security Manager

Following web links are recommended for online study:

- Bluetooth Specifications: www.bluetooth.org/spec
- Assigned Numbers: www.bluetooth.org/foundry/assignnumb/document/assigned_numbers
- Bluetooth Developers Site: https://developer.bluetooth.org/

Abbreviations

Abbreviation	Reference
ACL	Asynchronous Connectionless
API	Application Programmer's Interface
ATT	Attribute Protocol
BT	Bluetooth
GATT	Generic Attribute Profile
HCI	Host Controller Interface
SMP	Security Manager Protocol

References

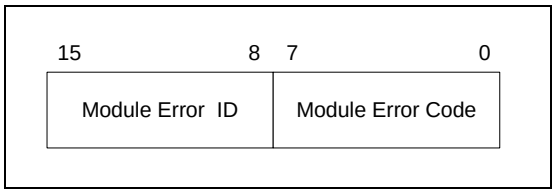
Serial No.	Reference
[1]	BlueLitE Stack API Documents

Appendix [A]: Using BlueLitE Error Codes

This section provides an informative description of BlueLitE Error Codes and the design philosophies behind it.

Overview of BlueLitE Error Codes

Each Error Code, from `BT_error.h`, is a 16-bit, 2-octet, `UINT16` value, as specified below:



The Module Error ID identifies the BlueLitE module (HCI, ATT, SMP, Write Task etc.) that is responsible for generating the error. Each module under BlueLitE Bluetooth Protocol Stack & Profile is given a unique Module Error ID. For BlueLitE Protocol Modules (for example, HCI, ATT, SMP etc.), the Module Error IDs are assigned from the range `0x10` to `0x4F`. For BlueLitE Profiles, the range is from `0x50` to `0x7F`.

The Module Error Code identifies the specific error generated by the BlueLitE module (as identified by the Module Error ID field).

The Module Error ID `0x00` is of special significance as it refers to the Error Codes received from the Bluetooth device (Baseband controller) over Host Controller Interface (HCI), and specified in detail in the "Specification of the Bluetooth System, v1.2, Vol. 2, Part D – Error Codes".

The definition of `API_SUCCESS` is `0x0000` - which is the "Success'" return value for an API returning `API_RESULT` data type. All other values for should be treated as errors, unless otherwise specified.

The definition of `API_FAILURE` is `0xFFFF` - which stands for "Unknown Error Situation".

How BlueLitE Error Codes Used Internally

Internally, the BlueLitE Protocol Stack & Profile modules propagate error return value received from the lower layer/module (when their API was called, or, callback called) to the higher layer.

To explain further, following hypothetical example is provided:

Example: Internal Use of BlueLitE Error Codes

```
/* An API from ATT */
API_RESULT BT_att_xyz_api (params ...)
{
    API_RESULT retval;

    /* Handle Error Situations ... */
    if (error ...)
    {
```

Example: Internal Use of BlueLitE Error Codes

```
        return ATT_ERROR_XYZ;
    }

    /* Call L2CAP API */
    retval = l2ca_xyz_api (params ...);
    if (retval != API_SUCCESS)
    {
         /* Propagate Error Code to API Caller */
        return retval;
    }

    return API_SUCCESS;
}


/* An API from L2CAP */
API_RESULT l2ca_xyz_api (params ...)
{
    API_RESULT retval;

    /* Handle Error Situations ... */
    if (error ...)
    {
        return L2CAP_ERROR_XYZ;
    }

    /* Call HCI API */
    retval = hci_xyz_api (params ...);
    if (retval != API_SUCCESS)
    {
         /* Propagate Error Code to API Caller */
        return retval;
    }

    return API_SUCCESS;
}


/* An API from HCI */
API_RESULT hci_xyz_api (params ...)
{
    API_RESULT retval;

    /* Handle Error Situations ... */
    if (error ...)
    {
        return HCI_ERROR_XYZ;
    }
```

Example: Internal Use of BlueLitE Error Codes

```
    return API_SUCCESS;
}
```

In the above example, when an application calls `BT_att_xyz()`, and, an error occurs in HCI, the application may receive the `HCI_ERROR_XYZ` error code, instead of an error code from ATT module.

This method enables the application (and developers) to know which module is actually malfunctioning, and why.

Why BlueLitE APIs Do Not Mention Expected Error Codes

BlueLitE API Documentations usually do not mention which specific Error Code is expected, if an error situation occurs. Instead, API documentations ask the developer to refer to `BT_error.h` for error code details. The reasons are as follows:

- It is not possible to guess which module would ultimately be responsible for an error condition. As described in the example above, an ATT API may return an Error Code from HCI module.
- BlueLitE Error Codes are subject to change. Some error codes may get deleted or modified in later revision of the stack modules. Also, new Error Codes may be added.

How Application Should Call BlueLitE APIs

Below is the suggested mechanism.

Example: Method of Calling BlueLitE APIs

```
retval = BT_some_api (params ...);
if (retval != API_SUCCESS)
{
    /* Error Handling */
}
else
{
    /* Proceed with next action */
}
```

Making Best Use of BlueLitE Error Codes

When an application/product is being developed using the BlueLitE Protocol Stack & Profile, an BlueLitE API may return any of the following:

(A) Success (API_SUCCESS)

(B) Recoverable Failures (Error Codes). For example,

- Incorrect Parameters

  Actions for Developer: Correct application to pass appropriate parameters

- Incorrect State for API

  Actions for Developer: Correct application design & state machine

- Memory Allocation Failure

  Actions for Developer: Increase available memory, or, free unused memory

- Queue Full Condition

  Actions for Developer: Revisit BT_limits.h and configure appropriate values.

(C) Irrecoverable Failures (Error Codes). For example,

- Incorrect Response from remote device or peer protocol/profile/application

  Actions for Developer: Attempt to re-initiate the procedure, or, abort

- Negative Response from Peer Protocol/Profile

  Actions for Developer: Attempt to re-initiate the procedure, or, abort

Over time as the application & product is developed and tested, the (B) error conditions should get less and less likely to happen, and ultimately Zero, and the system is fine-tuned.

On the other hand, for case (C) errors, application/product (depending on its design) has to take drastic decision as to whether to continue with the operation or abort. Most likely, it is going to be "abort".

To summarize, the application does not need perform explicit handling each and every error that an API can possibly return. In the final application/product, it can be viewed as either an API returns "Success", or, "Irrecoverable Error" requiring abort/system reset.

Contact: Bluetooth@mindtree.com
www.mindtree.com

United States	Japan	United Kingdom	Singapore	India
MindTree Consulting Suite #105 #2855 Kifer Road, Santa Clara CA 95051. USA. Tel: +1 408 986 1000 Fax: +1 408 986 0005	Yurakucho Building 11th Floor 1-10-1, Yurakucho, Chiyoda-Ku Tokyo, Japan 100-0006 Tel: +81 (3) 5219 2094 Fax: +81 (3) 5219 2021	Regus House Windmill Hill Business Park Whitehill Way Swindon Wiltshire SN5 6QR UK. Tel: +44 (0) 1793 441418 Fax: +44 (0) 1793 441618	Suite #12 Level 15, Prudential Tower 30 Cecil Street Singapore 049712. Tel: +65 232 2751, 52, 53 Fax: +65 232 2888	#42 27th Cross Banashankari II Stage Bangalore - 560 070 Karnataka. India. Tel: +91 80 671 1777 Fax: +91 80 671 4000