Getting started with the X-CUBE-BLE1 Bluetooth Low Energy software expansion for STM32Cube

Introduction

The X-CUBE-BLE1 expansion software package for STM32Cube runs on the STM32 and includes drivers for BlueNRG-MS/ BlueNRG-M0 Bluetooth low energy devices.

The expansion is built on STM32Cube software technology to ease portability across different STM32 microcontrollers.

The software comes with sample implementations of the drivers running on the X-NUCLEO-IDB05A2 expansion board, when connected to a NUCLEO-L476RG development board.

Related links

Visit the STM32Cube ecosystem web page on www.st.com for further information

1 Acronyms and abbreviations

Table 1. Acronyms and abbreviations

Acronym	Description
ACI	Application controller interface
ATT	Attribute protocol
BLE	Bluetooth Low Energy
BSP	Board support package
BT	Bluetooth
GAP	Generic access profile
GATT	Generic attribute profile
GUI	Graphical user interface
HAL	Hardware abstraction layer
HCI	Host controller interface
HRS	Heart rate sensor
IDE	Integrated development environment
L2CAP	Logical link control and adaptation protocol
LED	Light emitting diode
LL	Link layer
LPM	Low power manager
MCU	Micro controller unit
PCI	Profile command interface
PHY	Physical layer
SIG	Special interest group
SM	Security manager
SPI	Serial peripheral interface
UUID	Universally unique identifier

2 X-CUBE-BLE1 software expansion for STM32Cube

2.1 Overview

The X-CUBE-BLE1 software package expands STM32Cube functionality and provides the Bluetooth Low Energy connectivity.

The key features are:

- Complete middleware to build Bluetooth low energy applications using BlueNRG-MS/BlueNRG-M0 devices
- Easy portability across different MCU families, thanks to STM32Cube
- Package compatible with STM32CubeMX, can be downloaded from and installed directly into STM32CubeMX
- Free, user-friendly license terms

Important: The X-CUBE-BLE1 software package also supports the X-NUCLEO-IDB05A1 expansion board.

2.1.1 Bluetooth Low Energy

Bluetooth Low Energy is a wireless personal area network technology designed and marketed by Bluetooth SIG. It can be used to develop new innovative applications (fitness, security, healthcare, etc.) using devices which run on coin cell batteries, and can remain indefinetely operative without draining the battery.

2.1.1.1 Operating modes

According to the Bluetooth standard specification (version 4.0 and above), Bluetooth Classic and Bluetooth Low Energy can be supported on the same device (dual-mode, also called Bluetooth smart ready).

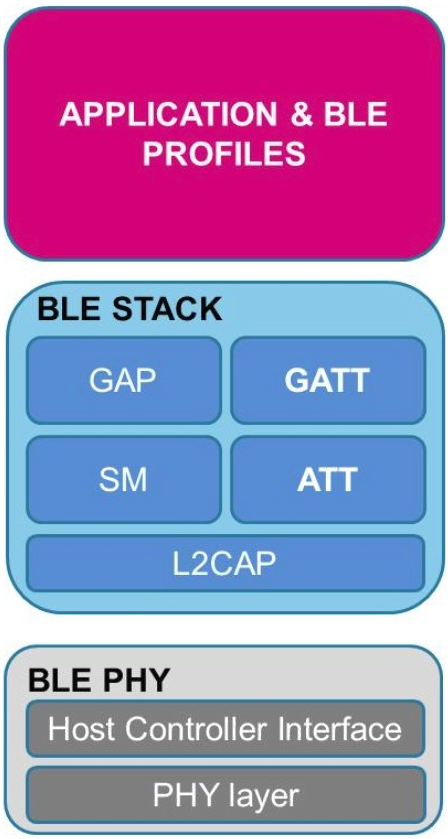
Instead, a single-mode (Bluetooth smart) device supports the BLE protocol only.

2.1.1.2 Software partitioning

A typical BLE system consists of :

- an LE controller, containing a physical layer (PHY) including the radio, a link layer (LL) and a standard host controller interface (HCI)
- a host, containing an HCI and other higher protocol layers (e.g. L2CAP, SM, ATT/GATT and GAP)

Figure 1. **BLE protocol stack**



The host can send HCI commands to control the LE controller. The HCI interface and the HCI commands are standardized by the Bluetooth core specification (refer to the Bluetooth standard for further information).

The PHY layer ensures communication with the stack and data (bits) transmission over-the-air. BLE operates in the 2.4 GHz Industrial Scientific Medical (ISM) band and defines 40 radio frequency (RF) channels with 2 MHz channel spacing.

In BLE, when a device only needs to broadcast data, it transmits the data in advertising packets through the advertising channels. Any device that transmits advertising packets is called advertiser. Devices that aim only at receiving data through the advertising channels are called scanners. Bidirectional data communication between two devices requires them to connect to each other.

BLE defines two device roles at the link layer (LL) for a created connection: the master and the slave. These are the devices that act as initiator and advertiser during the connection creation, respectively.

The host controller interface (HCI) layer provides a standardized interface to enable communication between the host and controller. In BlueNRG, this layer is implemented through the SPI hardware interface.

In BLE, the main goal of L2CAP is to multiplex the data of three higher layer protocols, ATT, SMP and link layer control signaling, on top of a link layer connection.

The SM layer is responsible for pairing and key distribution, and enables secure connection and data exchange with another device.

At the highest level of the core BLE stack, the GAP specifies device roles, modes and procedures for the discovery of devices and services, the management of connection establishment and security. In addition, GAP handles the initiation of security features. The BLE GAP defines four roles with specific requirements on the underlying controller: Broadcaster, Observer, Peripheral and Central.

The ATT protocol allows a device to expose certain pieces of data, known as attributes, to another device. The ATT defines the communication between two devices playing the roles of server and client, respectively, on top of a dedicated L2CAP channel. The server maintains a set of attributes. An attribute is a data structure that stores the information managed by the GATT, the protocol that operates on top of the ATT. The client or server role is determined by the GATT, and is independent of the slave or master role.

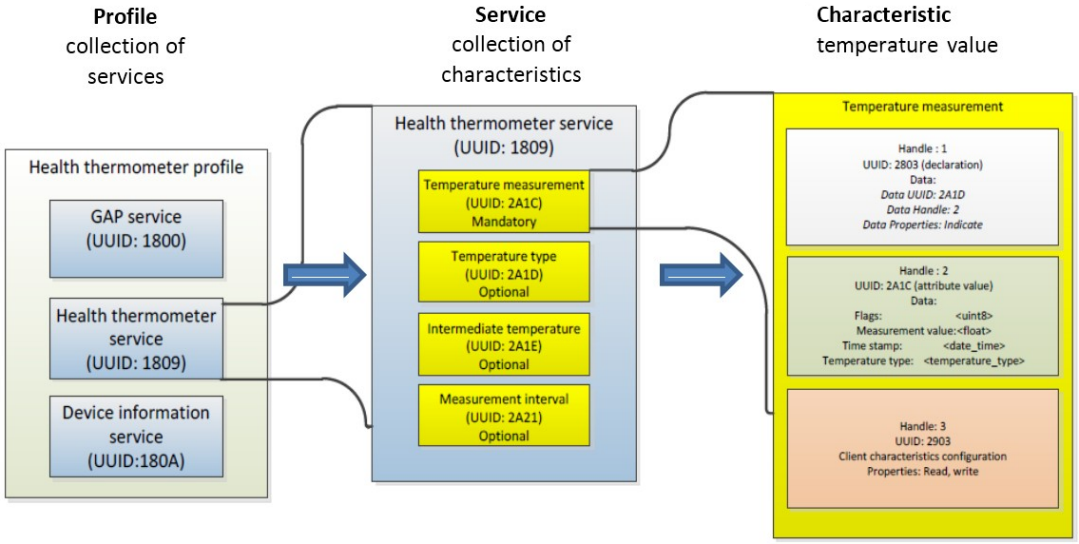
The GATT defines a framework that uses the ATT for the discovery of services, and the exchange of characteristics from one device to another. GATT specifies the structure of profiles. In BLE, all pieces of data that are being used by a profile or service are called characteristics. A characteristic is a set of data which includes a value and properties.

2.1.1.3 *Profiles and services*

The BLE protocol stack is used by the applications through its GAP and GATT profiles. The GAP profile is used to initialize the stack and setup the connection with other devices. The GATT profile is a way of specifying the transmission - sending and receiving - of short pieces of data known as attributes over a Bluetooth smart link.

All current Low Energy application profiles are based on GATT. The GATT profile allows the creation of profiles and services within these application profiles.

Figure 2. Structure of a GATT-based profile



In this example, the profile is created with the following services:

- GAP service, which has to be always set up
- health thermometer service
- device information service

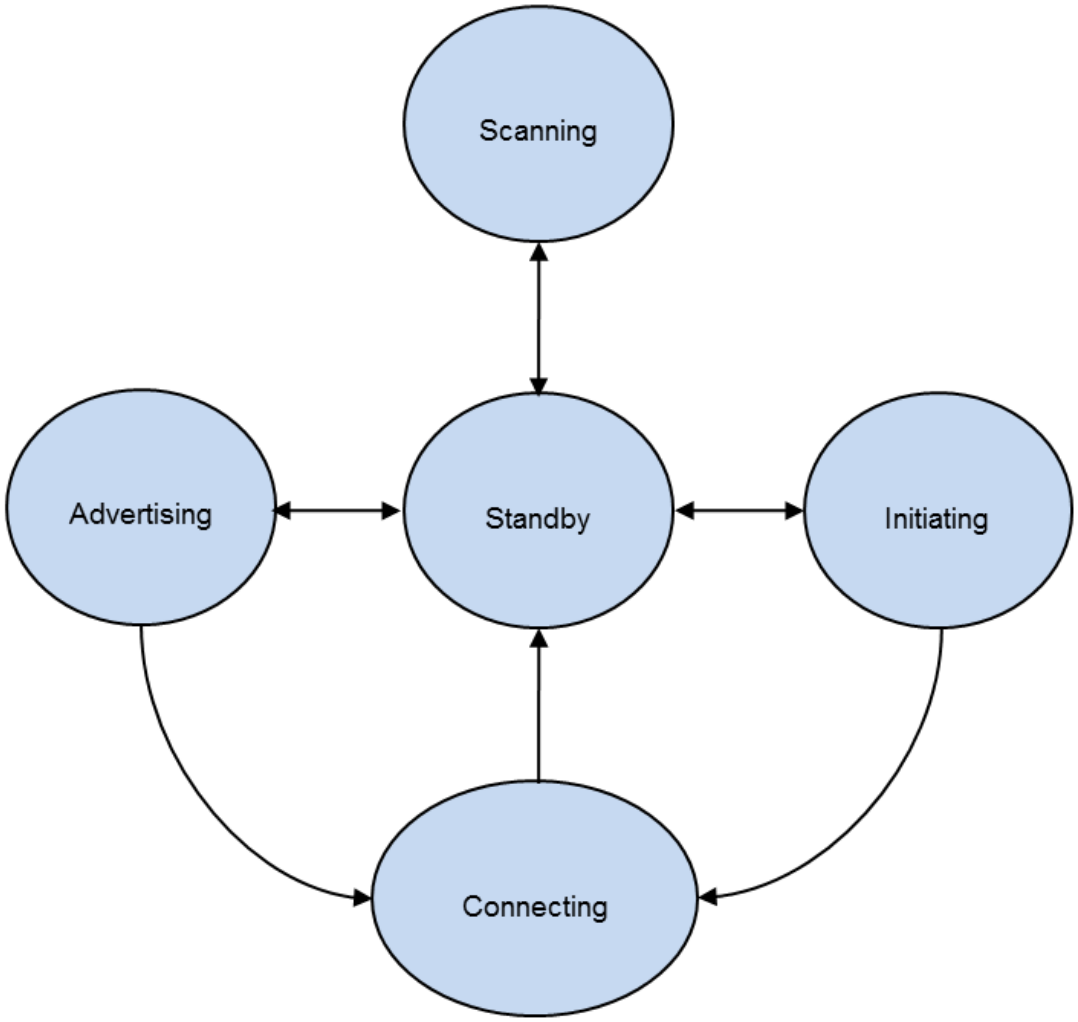
Each service consists of a set of characteristics defining the service and the type of data it provides as part of the service. In the above example, the health thermometer service contains the following characteristics:

- temperature measurement
- temperature type
- intermediate temperature
- measurement interval

Each characteristic details the data type and value, and is defined by attributes (at least, two for characteristic): the main attribute (0x2803) and a value attribute that actually contains the data. The main attribute defines the value attribute handle and UUID.

2.1.1.4 *State machine*

Figure 3. State machine during BLE operations



During normal operation, a BLE device can be in the following states:

- Standby: does not transmit or receive packets.
- Advertising: broadcasts advertisements in advertising channels. The device transmits advertising channel packets and possibly listens and answers to, triggered by the advertising channel packets.
- Scanning: looks for advertisers. The device is listening for advertising channel packets from advertising devices.
- Initiating: the device initiates connection to the advertiser and is listens to advertising channel packets from specific device(s) and responds to these packets to initiate a connection with another device.
- Connection: connection has been established and the device is transmitting or receiving:
  - the initiator device plays the master role, that is, it communicates with the device in the slave role and defines timings of transmission;
  - the advertiser device plays the slave role, that is, it communicates with a single device in the master role.

2.2 Architecture

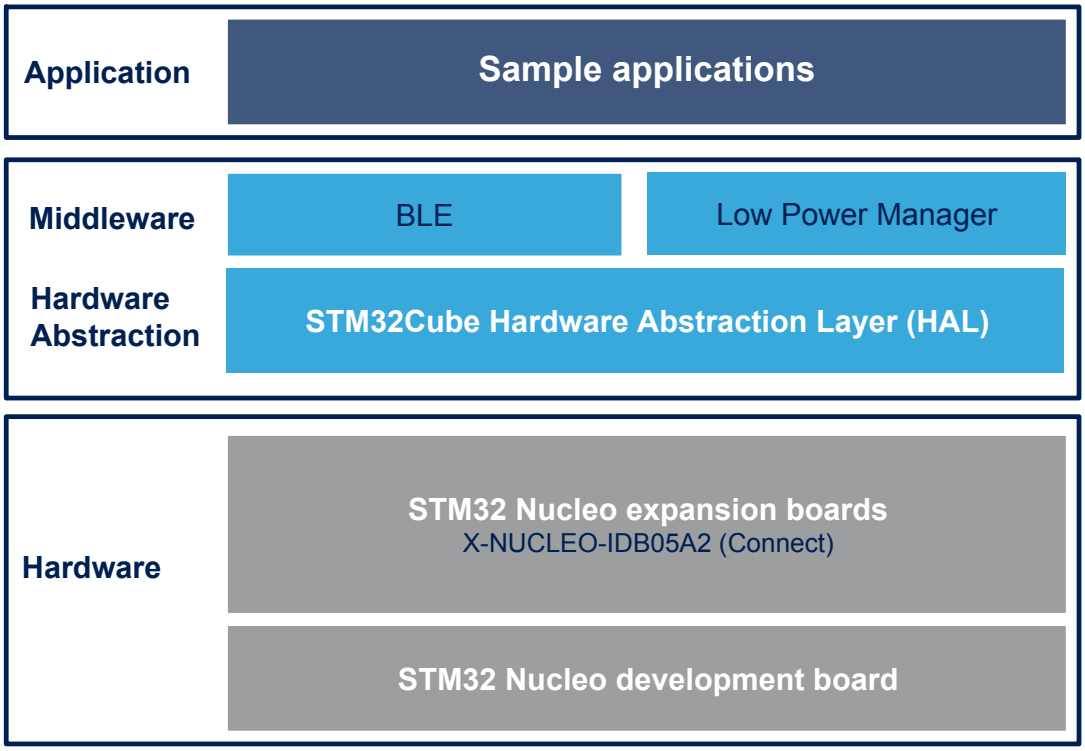
This software is a fully compliant expansion for STM32Cube enabling development of applications using the Bluetooth Low Energy connectivity.

The software is based on the hardware abstraction layer for the STM32 microcontroller, STM32CubeHAL. The package extends STM32Cube by providing complete middleware for the Bluetooth Low Energy expansion board and some middleware components for serial communication with a PC.

The software layers used by the application software to access and use the BlueNRG-MS/BlueNRG-M0 expansion board are:

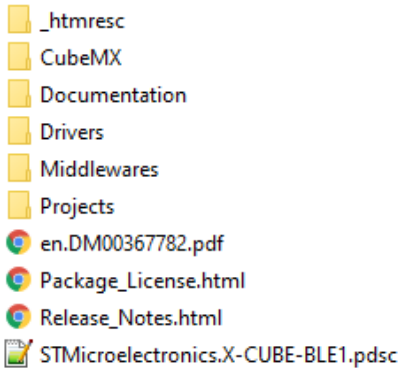
- The STM32Cube HAL driver layer provides a simple, generic and multi-instance set of APIs (application programming interfaces) to interact with the upper layers (application, libraries and stacks). It includes generic and extension APIs and is based on a generic architecture which allows the layers built on it (such as the middleware layer) to implement their functionalities without dependence on the specific hardware configuration of a given Microcontroller Unit (MCU). This structure improves library code reusability and guarantees high portability across other devices.
- The Board Support Package (BSP) layer provides supporting software for the peripherals on the STM32 Nucleo board, except for the MCU. It has a set of APIs to provide a programming interface for certain board-specific peripherals (e.g. the LED, the user button etc.) and allow identification of the specific board version. For the BlueNRG-MS/BlueNRG-M0 expansion board, it provides support for Bluetooth Low Energy connectivity.

Figure 4. X-CUBE-BLE1 software architecture



2.3 Folders

Figure 5. **X-CUBE-BLE1 package folder structure**



The following folders are included in the software package:

- **CubeMX**: contains the metadata files for the package support in the STM32CubeMX tool.
- **Documentation**: contains a compiled HTML file generated from the source code, detailing the software components and APIs.
- **Drivers**: contains the HAL drivers, the specific drivers for each supported board or hardware platform, including the on-board components and the CMSIS layer (vendor-independent hardware abstraction layer for the Cortex-M processor series).
- **Middlewares**: contains libraries and protocols related to host software and applications to interface the BlueNRG-MS/BlueNRG-M0 module.
- **Projects**: contains the BLE sample applications for the NUCLEO-L476RG platform with three development environments (IAR Embedded Workbench for ARM, RealView Microcontroller Development Kit (MDK-ARM) and STM32CubeIDE).

2.4 Guide for writing applications

To write a BLE application based on the STM32 Nucleo development board with the BlueNRG-MS/BlueNRG-M0 expansion board, and add GATT services and characteristics, follow the procedure described in the following sections.

2.4.1 APIs

The STM32 Nucleo development board acts as a GATT server (and as a peripheral device) and the PC as a GATT client (and as a central device).

Detailed technical information about the available APIs can be found in a compiled HTML file in the "Documentation" folder.

2.4.2 Initialization

To function correctly, every application has to perform the basic initialization steps to configure and set up the STM32 Nucleo development board with the BlueNRG-MS/BlueNRG-M0 expansion board hardware.

2.4.2.1 STM32 Cube HAL

To correctly configure the necessary hardware components, the STM32Cube HAL library must be initialized via the `HAL_Init();` function.

This API configures Flash prefetch, Flash preread, Buffer cache, time base source, vectored interrupt controller and low-level hardware.

2.4.2.2 *STM32 Nucleo board peripherals*

Some of the STM32 Nucleo on-board peripherals have to be configured before using them, via the following functions:

- `BSP_LED_Init(Led_TypeDef Led);` - to configure STM32 Nucleo LEDs
- `BSP_PB_Init(Button_TypeDef Button, ButtonMode_TypeDef Button_Mode);` - to configure the user button in GPIO mode or external interrupt (EXTI) mode
- `BSP_JOY_Init();` - to configure the board joystick (if any)

2.4.2.3 *BlueNRG HAL and HCI*

The BlueNRG HAL provides the API and the functionality to perform operations related to the BlueNRG-MS/ BlueNRG-M0 device via the following functions:

- `HCI_TL_SPI_Init();` - to initialize the SPI communication with the BlueNRG-MS/BlueNRG-M0 device
- `hci_init();` - to initialize the host controller interface (HCI)
- `HCI_TL_SPI_Reset();` - to reset the BlueNRG-MS/BlueNRG-M0 device

2.4.2.4 *Service characteristics*

The STM32 Nucleo plus the BLE expansion board stack must be correctly initialized before establishing a connection with another BLE device via the following APIs:

- `aci_gap_init(uint8_t role, uint16_t* service_handle, uint16_t* dev_name_char_handle, uint16_t* appearance_char_handle);` - to initialize the BLE device for a particular role (peripheral, broadcaster, central device, etc.)
- `aci_gatt_add_serv(UUID_TYPE_128, service_uuid, PRIMARY_SERVICE, 7, &servHandle);` - to add a service to the GATT server device.

2.4.3 Security requirements

The GATT client application can use an API to specify its security requirements. If a characteristic has security restrictions, the central device has to initiate a pairing procedure to access it. In the provided SensorDemo_BLESensor-App, the following API is used with a fixed pin (123456) for pairing:

- `aci_gap_set_auth_requirement(MITM_PROTECTION_REQUIRED, OOB_AUTH_DATA_ABSENT, NULL, 7, 16, USE_FIXED_PIN_FOR_PAIRING, 123456, BONDING);`

2.4.4 Connectable mode

On the GATT server device the following GAP API is used to enter general discoverable mode:

- `aci_gap_set_discoverable(ADV_IND, 0, 0, PUBLIC_ADDR, NO_WHITE_LIST_USE,8, local_name, 0, NULL, 0, 0);`

2.4.5 Connection with central device

Once the device used as GATT server is put in discoverable mode, the GATT client role device can see it and establish a BLE connection.

On the GATT client device, the following GAP API is used for connection with the GATT server device in advertising mode:

- `aci_gap_create_connection(0x4000, 0x4000, PUBLIC_ADDR, bdaddr, PUBLIC_ADDR, 9, 9, 0, 60, 1000 , 1000)` where `bdaddr` is the peer address of the GATT client role device

Once the two devices are connected, the BLE communication works as follows:

- on the GATT server role device, the following API should be invoked to update characteristic value: `aci_gatt_update_char_value(chatServHandle, TXCharHandle, 0, len, (tHalUint8 *)data)`, where `data` is the value that updates the attribute pointed to by the `TxCharHandle` characteristic handle in the `chatServHandle` service
- on the GATT client device, the following API should be invoked to write to a characteristic handle: `aci_gatt_write_without_response(connection_handle, RX_HANDLE+1, len, (tHalUint8 *)data)`, where `data` is the value of the attribute pointed to by the `RX_HANDLE` attribute handle in the `connection_handle` which is the handle returned at connection creation as `EVT_LE_CONN_COMPLETE` event parameter.

2.5 SensorDemo-BLESensor-App application description

The project files for the SensorDemo-BLESensor-App can be found in the X-CUBE-BLE1 software package in the *$BASE_DIR\Projects\NUCLEO-L476RG\Applications\SensorDemo* folder.

In the SensorDemo-BLESensor-App, the STM32 Nucleo development board creates two services:

- one for environmental (temperature, pressure, and humidity) and motion (accelerometer, gyroscope, and magnetometer) data
- one for the Sensor Fusion algorithm (which combines motion data in quaternions)

Note: The STM32 Nucleo development board does not embed an actual environmental or accelerometer sensor: output data are the result of a simulation.

The SensorDemo-BLESensor-App creates services and characteristics using APIs (see Section 2.4.1 APIs) and looks for a central device to connect to it. The application advertises its services and characteristics to the listening client devices while waiting for a connection to be established. After connection, data are periodically updated.

The following sections show how to modify an application adding new GATT services and characteristics.

2.5.1 Time Service

You can add Time Service with the following characteristics:

- *seconds characteristic*, a read-only characteristic that shows the number of seconds elapsed since system boot.
- *minutes characteristic* that shows the number of minutes elapsed since system boot.

Note: This characteristic can be read by the GATT server: a notification event is generated at intervals of one minute.

2.5.1.1 Adding Time Service and characteristics

As explained in Section 2.4.2.4 Service characteristics, the `aci_gatt_add_serv()` API is used to add a service to the application and the `aci_gatt_add_char()` is used to add the characteristics.

Note: The seconds characteristic can support the read operation by using the `CHAR_PROP_READ` argument.
The minutes characteristic can be read and notified by using the `CHAR_PROP_NOTIFY|CHAR_PROP_READ` argument.

The following code is an example of how to add the Time Service and its characteristics.

```
/**
 * @brief  Add a time service using a vendor specific profile
 * @param  None
 * @retval Status
 */
tBleStatus Add_Time_Service(void)
{
  tBleStatus ret;
  uint8_t uuid[16];

  /* copy "Timer service UUID" defined above to 'uuid' local variable */
  COPY_TIME_SERVICE_UUID(uuid);

  /*
   * now add "Time service" to GATT server, service handle is returned
   * via 'timeServHandle' parameter of aci_gatt_add_serv() API.
   * Please refer to 'BlueNRG Application Command Interface.pdf' for detailed
   * API description
  */
  ret = aci_gatt_add_serv(UUID_TYPE_128,  uuid, PRIMARY_SERVICE, 7,
                          &timeServHandle);
  if (ret != BLE_STATUS_SUCCESS) goto fail;

  /*
   * now add "Seconds characteristic" to Time service, characteristic handle
   * is returned via 'secondsCharHandle' parameter of aci_gatt_add_char() API.
   * This characteristic is read only, as specified by CHAR_PROP_READ parameter.
   * Please refer to 'BlueNRG Application Command Interface.pdf' for detailed
```

```
   * API description
  */
  COPY_TIME_UUID(uuid);
  ret =  aci_gatt_add_char(timeServHandle, UUID_TYPE_128, uuid, 4,
                           CHAR_PROP_READ, ATTR_PERMISSION_NONE, 0,
                           16, 0, &secondsCharHandle);
                           if (ret != BLE_STATUS_SUCCESS) goto fail;

  COPY_MINUTE_UUID(uuid);
  /*
   * Add "Minutes characteristic" to "Time service".
   * This characteristic is readable as well as notifiable only, as specified
   * by CHAR_PROP_NOTIFY|CHAR_PROP_READ parameter below.
   */
  ret =  aci_gatt_add_char(timeServHandle, UUID_TYPE_128, uuid, 4,
                           CHAR_PROP_NOTIFY|CHAR_PROP_READ, ATTR_PERMISSION_NONE, 0,
                           16, 0, &minuteCharHandle);
  if (ret != BLE_STATUS_SUCCESS) goto fail;

  PRINTF("Service TIME added. Handle 0x%04X, TIME Charac handle: 0x%04X\n",timeServHandle,
secondsCharHandle);
  return BLE_STATUS_SUCCESS;

  /* return BLE_STATUS_ERROR if we reach this tag */
fail:
  PRINTF("Error while adding Time service.\n");
  return BLE_STATUS_ERROR ;
}
```

The `Add_Time_Service()` function can be called by the following code, after the BLE device initialization and before the advertising operations.

```
    /* instantiate timer service with 2 characteristics:-
   * 1. seconds characteristic: Readable only
   * 2. Minutes characteristics: Readable and Notifiable
   */
  ret = Add_Time_Service();

  if(ret == BLE_STATUS_SUCCESS)
    PRINTF("Time service added successfully.\n");
  else
    PRINTF("Error while adding Time service.\n");
```

2.5.1.2 *Updating and notifying characteristic value*

The `Seconds_Update()` function updates Time Service *seconds characteristic*.

```
/**
 * @brief  Update seconds characteristic value of Time service
 * @param  AxesRaw_t structure containing acceleration value in mg
 * @retval Status
 */
tBleStatus Seconds_Update(void)
{
  tHalUint32 val;
  tBleStatus ret;

  /* Obtain system tick value in milliseconds, and convert it to seconds. */
  val = HAL_GetTick();
  val = val/1000;

  /* create a time[] array to pass as last argument of aci_gatt_update_char_value() API*/
  const tHalUint8 time[4] = {(val >> 24)&0xFF, (val >> 16)&0xFF, (val >> 8)&0xFF,
(val)&0xFF};
/*
   * Update value of "Seconds characteristic" using aci_gatt_update_char_value() API
```

```
   * Please refer to 'BlueNRG Application Command Interface.pdf' for detailed
   * API description
   */
  ret = aci_gatt_update_char_value(timeServHandle, secondsCharHandle, 0, 4,
                                   time);

  if (ret != BLE_STATUS_SUCCESS){
    PRINTF("Error while updating TIME characteristic.\n") ;
    return BLE_STATUS_ERROR ;
  }
  return BLE_STATUS_SUCCESS;
}
```

The `Minutes_Notify()` function updates the *minutes characteristic* value once a minute.

```
    /**
 * @brief  Send a notification for a minute characteristic of time service
 * @param  None
 * @retval Status
 */
tBleStatus Minutes_Notify(void)
{
  tHalUint32 val;
  tHalUint32 minuteValue;
  tBleStatus ret;

  /* Obtain system tick value in milliseconds */
  val = HAL_GetTick();
/* update "Minutes characteristic" value iff it has changed w.r.t. previous
   * "minute" value.
   */
  if((minuteValue=val/(60*1000))!=previousMinuteValue) {
    /* memmorize this "minute" value for future usage */
    previousMinuteValue = minuteValue;

    /* create a time[] array to pass as last argument of aci_gatt_update_char_value() API*/
    const tHalUint8 time[4] = {(minuteValue >> 24)&0xFF, (minuteValue >> 16)&0xFF,
(minuteValue >> 8)&0xFF, (minuteValue)&0xFF};

  /*
   * Update value of "Minutes characteristic" using aci_gatt_update_char_value() API
   * Please refer to 'BlueNRG Application Command Interface.pdf' for detailed
   * API description
   */
    ret = aci_gatt_update_char_value(timeServHandle, minuteCharHandle, 0, 4,
                                     time);
  if (ret != BLE_STATUS_SUCCESS){
      PRINTF("Error while updating TIME characteristic.\n") ;
      return BLE_STATUS_ERROR ;
    }
  }
  return BLE_STATUS_SUCCESS;
}
```

In the application user process, you can invoke `Update_Time_Characteristics()` which invokes in turn `Seconds_Update()` and `Minutes_Notify()`.

```
    /**
 * @brief  Updates "Seconds and Minutes characteristics" values
 * @param  None
 * @retval None
 */
void Update_Time_Characteristics() {
  /* update "seconds and minutes characteristics" of time service  */
  Seconds_Update();
  Minutes_Notify();
```

```
}
```

2.5.2 LED service

LED service controls the STM32 Nucleo LED2 status.

When the GATT client application modifies the value of the writable *LED button characteristic*, LED2 is toggled.

2.5.2.1 *Adding GATT service and characteristics*

The code below is an example of how to create the LED service and the related *LED button characteristic*.

```
    /*
 * @brief  Add LED button service using a vendor specific profile
 * @param  None
 * @retval Status
 */

tBleStatus Add_LED_Service(void)
{
  tBleStatus ret;
  uint8_t uuid[16];

  /* copy "LED service UUID" defined above to 'uuid' local variable */
  COPY_LED_SERVICE_UUID(uuid);
  /*
   * now add "LED service" to GATT server, service handle is returned
   * via 'ledServHandle' parameter of aci_gatt_add_serv() API.
   * Please refer to 'BlueNRG Application Command Interface.pdf' for detailed
   * API description
  */
  ret = aci_gatt_add_serv(UUID_TYPE_128,  uuid, PRIMARY_SERVICE, 7,
                          &ledServHandle);
  if (ret != BLE_STATUS_SUCCESS) goto fail;
  /* copy "LED button characteristic UUID" defined above to 'uuid' local variable */
  COPY_LED_UUID(uuid);
  /*
   * now add "LED button characteristic" to LED service, characteristic handle
   * is returned via 'ledButtonCharHandle' parameter of aci_gatt_add_char() API.
   * This characteristic is writable, as specified by 'CHAR_PROP_WRITE' parameter.
   * Please refer to 'BlueNRG Application Command Interface.pdf' for detailed
   * API description
  */
  ret =  aci_gatt_add_char(ledServHandle, UUID_TYPE_128, uuid, 4,
                           CHAR_PROP_WRITE | CHAR_PROP_WRITE_WITHOUT_RESP,
ATTR_PERMISSION_NONE, GATT_SERVER_ATTR_WRITE,
                           16, 1, &ledButtonCharHandle);

  if (ret != BLE_STATUS_SUCCESS) goto fail;
PRINTF("Service LED BUTTON added. Handle 0x%04X, LED button Charac handle:
0x%04X\n",ledServHandle, ledButtonCharHandle);
  return BLE_STATUS_SUCCESS;

fail:
  PRINTF("Error while adding LED service.\n");
  return BLE_STATUS_ERROR ;
}
```

2.5.2.2 *Obtaining characteristic value*

When the BlueNRG stack detects an ACI event, it invokes the `HCI_Event_CB()` function which analyzes the value of the received event packet.

Note: In some sample applications included in the X-CUBE-BLE1 package, the `HCI_Event_CB()` is also called `user_notify()`.

```
    /**
 * @brief  This function is called whenever there is an ACI event to be    processed.
 * @note   Inside this function each event must be identified and correctly
 *         parsed.
* @param  pckt  Pointer to the ACI packet
* @retval None
*/
void HCI_Event_CB(void *pckt)
{
  hci_uart_pckt *hci_pckt = pckt;
  /* obtain event packet */
  hci_event_pckt *event_pckt = (hci_event_pckt*)hci_pckt->data;

  if(hci_pckt->type != HCI_EVENT_PKT)
    return;
switch(event_pckt->evt){
    .
    .
    .
  case EVT_VENDOR:
    {
      evt_blue_aci *blue_evt = (void*)event_pckt->data;
      switch(blue_evt->ecode){

       case EVT_BLUE_GATT_ATTRIBUTE_MODIFIED:
        {
          /* this callback is invoked when a GATT attribute is modified
             extract callback data and pass to suitable handler function */
          evt_gatt_attr_modified *evt = (evt_gatt_attr_modified*)blue_evt->data;

          Attribute_Modified_CB(evt->attr_handle, evt->data_length, evt-
>att_data);
        }
        break;
    .
    .
      }
    }
    break;
  }
}
```

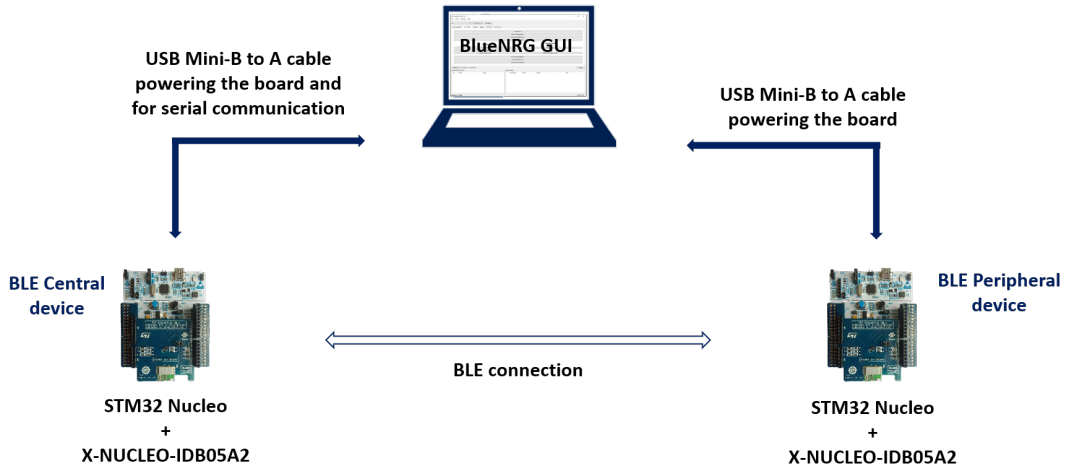
`Attribute_Modified_CB()` performs the event handling for the LED service. It toggles the STM32 Nucleo LED when the value of the *LED button characteristic* is modified by the GATT client.

```
    /**
 * @brief  This function is called attribute value corresponding to
 *         ledButtonCharHandle characteristic gets modified
 * @param  handle : handle of the attribute
 * @param  data_length : size of the modified attribute data
 * @param  att_data : pointer to the modified attribute data
 * @retval None
 */
void Attribute_Modified_CB(tHalUint16 handle, tHalUint8 data_length, tHalUint8 *att_data)
{
  /* If GATT client has modified 'LED button characteristic' value, toggle LED2 */
  if(handle == ledButtonCharHandle + 1){
      BSP_LED_Toggle(LED2);
  }
}
```

2.5.3 Testing the application

To test a sample application, you have to download the BlueNRG GUI installer included in the STSW-BNRGUI software package and set up the hardware components described in Section 2.6.3 Hardware setup.

Figure 6. Hardware setup



2.5.3.1 *Testing an application using BlueNRG GUI*

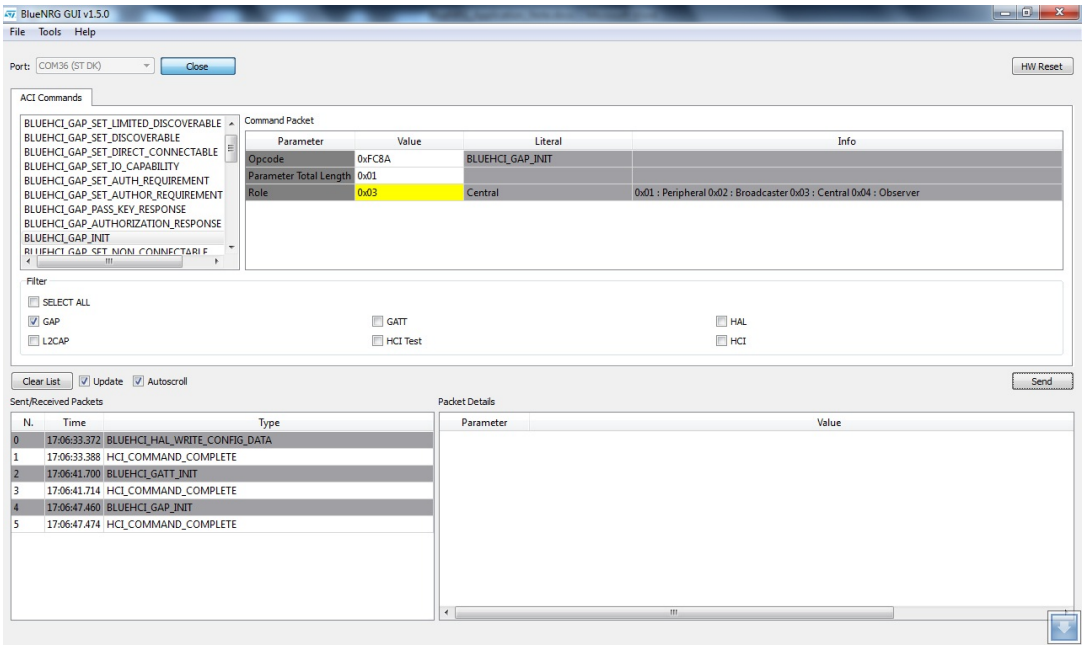
BlueNRG GUI can initialize and configure an STM32 Nucleo development board plus an X-NUCLEO-IDB05A2 expansion board stack as a BLE central device to test a BLE application, together with another board stack (STM32 Nucleo plus X-NUCLEO-IDB05A2) configured as BLE peripheral device .

2.5.3.2 *Initializing the BLE Central device*

The BLE central device is initialized by the following functions:

- `BLUEHCI_HAL_WRITE_CONFIG_DATA`
- `BLUEHCI_GATT_INIT`
- `BLUEHCI_GAP_INIT`

Figure 7. Initializing the BLE Central device



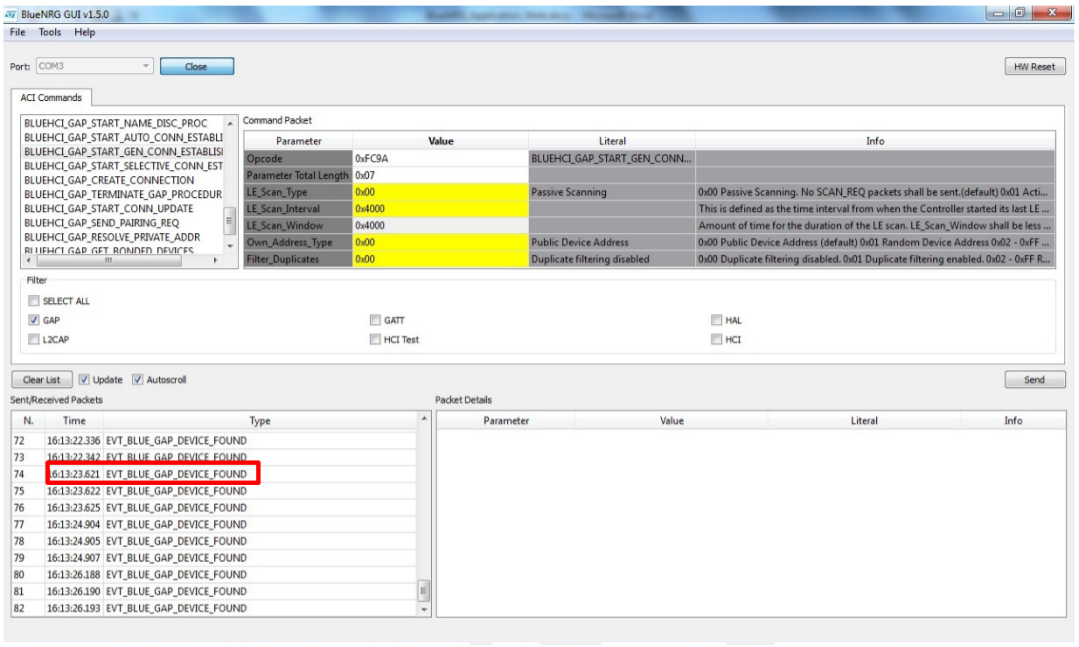
Note: The screenshot above may vary according to the BlueNRG GUI version used.

2.5.3.3 *Scanning for BLE peripheral device*

`BLUEHCI_GAP_START_GEN_DISC_PROC` function discovers the BLE peripheral device.

`EVT_BLUE_GAP_DEVICE_FOUND` confirms the device has been discovered by the BLE central device.

Figure 8. BlueNRG GUI scanning for devices



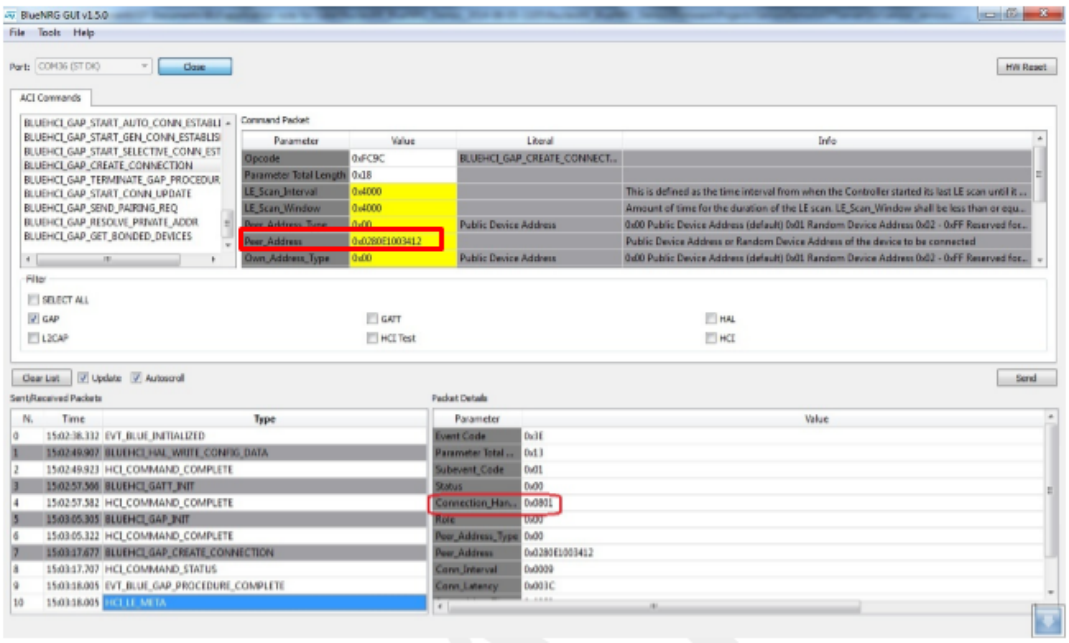
2.5.3.4 *Connecting to BLE peripheral device*

`BLUEHCI_GAP_CREATE_CONNECTION` function establishes the connection among the BLE central and peripheral devices, and returns the connection handle.

The peer address required for the connection is the server address: `tHalUint8 SERVER_BDADDR[] = {0x12, 0x34, 0x00, 0xE1, 0x80, 0x02};`

The connection handle can be determined by the `HCI_LE_META` response of the server and is required to retrieve services and characteristics.

Figure 9. **Establishing a connection**



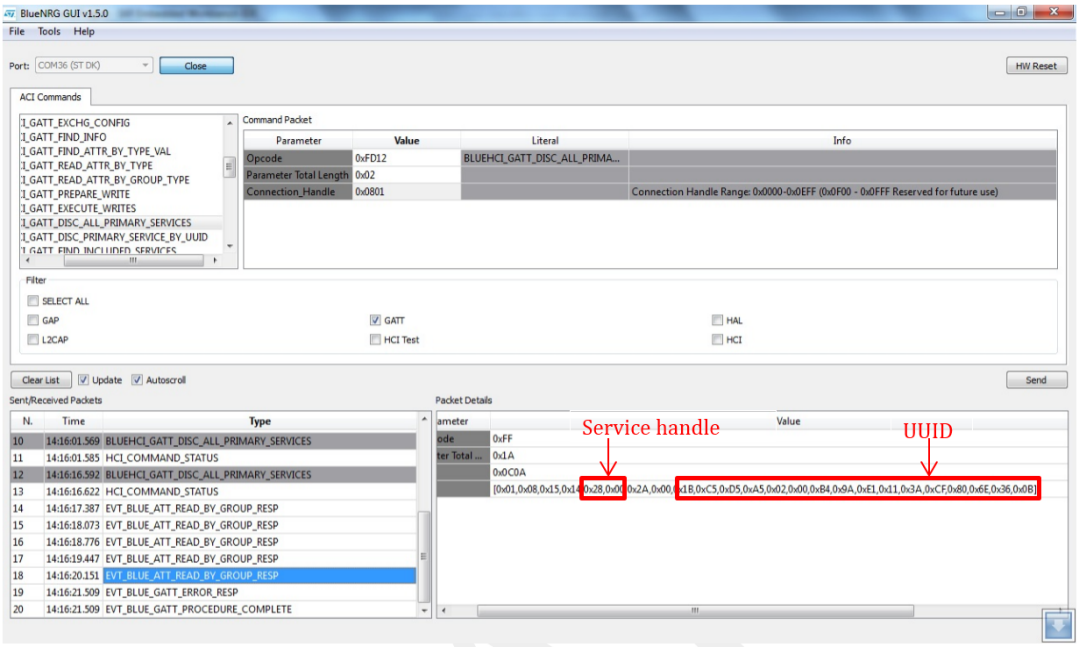
2.5.3.5 *Getting services supported by the BLE peripheral device*

The `BLUEHCI_GATT_DISC_ALL_PRIMARY_SERVICES` function is used to discover BLE peripheral device GATT services.

The server returns `EVT_BLUE_ATT_READ_BY_GROUP_RESP` for each service supported.

Each response includes the connection handle, the response length, the data length, the handle-value pair and the service UUID.

Figure 10. **Discovering all supported services**



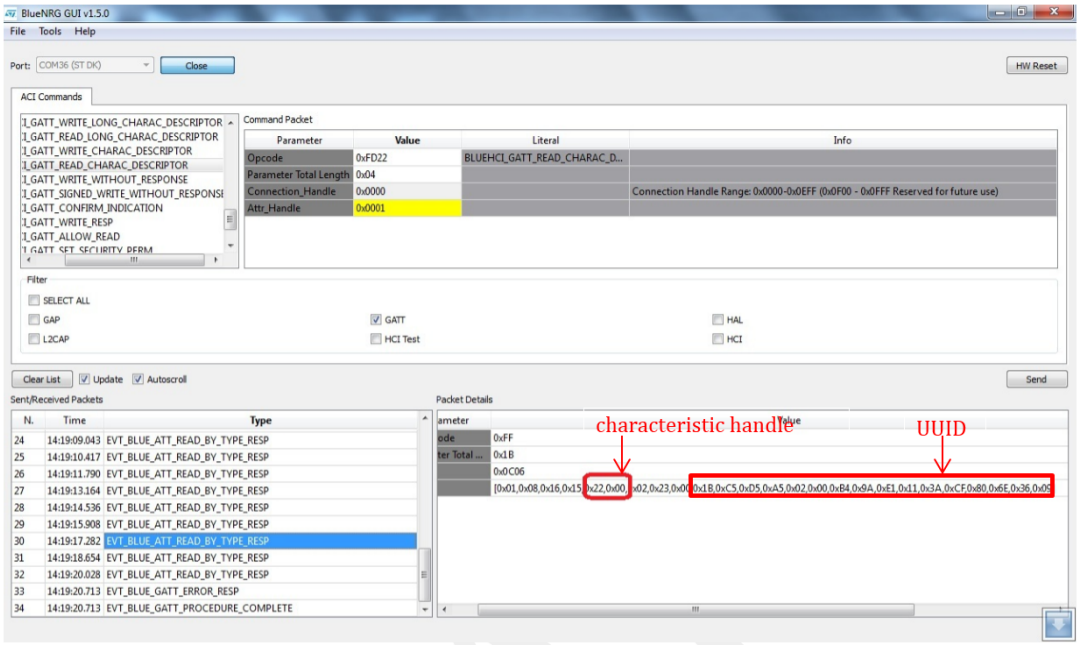
To find the handle of a service the UUID of that particular service (determined from the server code inserted) has to match with the response payload (`EVT_BLUE_ATT_READ_BY_GROUP_RESP`). The service UUID consists in the last 16 bytes of the payload.

2.5.3.6 *Getting characteristics supported by BLE peripheral device*

The `BLUEHCI_GATT_DISC_ALL_CHARAC_OF_A_SERVICE` function is used to discover the GATT characteristics supported. The server returns `EVT_BLUE_ATT_READ_BY_TYPE_RESP` for each characteristic supported.

Each response includes the connection handle, the response length, the data length, the handle-value pair and the characteristic UUID.

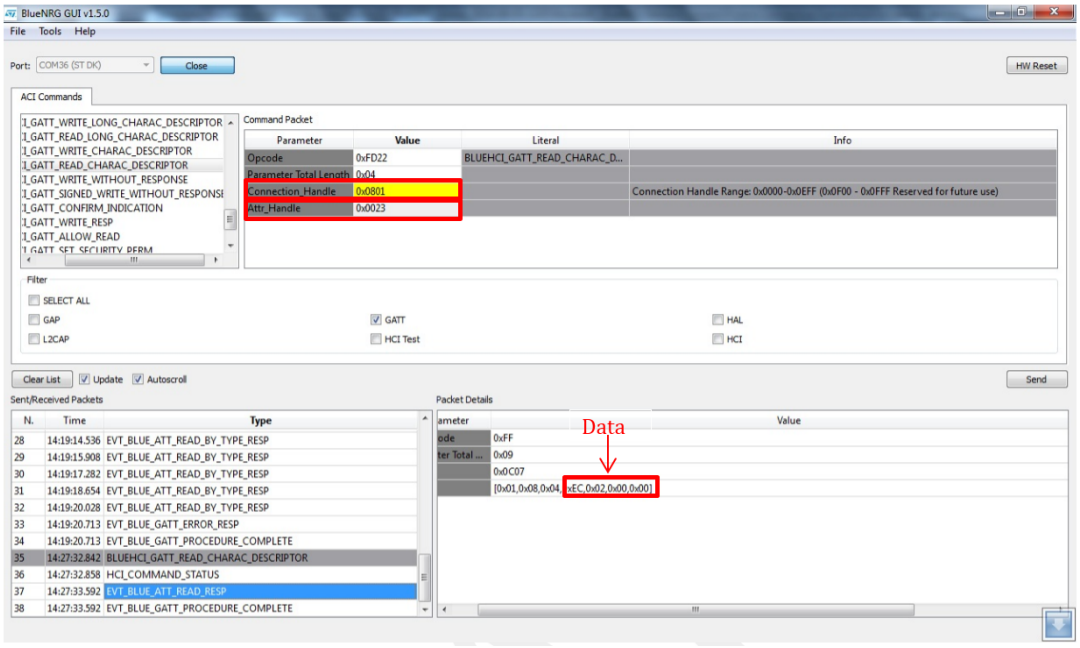
Figure 11. Discovering all supported characteristics



2.5.3.7 *Reading characteristic values*

The `BLUEHCI_GATT_READ_CHARACTERISTIC_VAL` function reads the server characteristics. Two parameters are required: the connection handle (0x0801) and the characteristic attribute handle.

Figure 12. Reading data from time characteristic



The value of the time characteristic is available in the response from the server (EVT_BLUE_ATT_READ_RESP) and consists in the payload last 4 bytes.

2.5.3.8 *Write characteristic value*

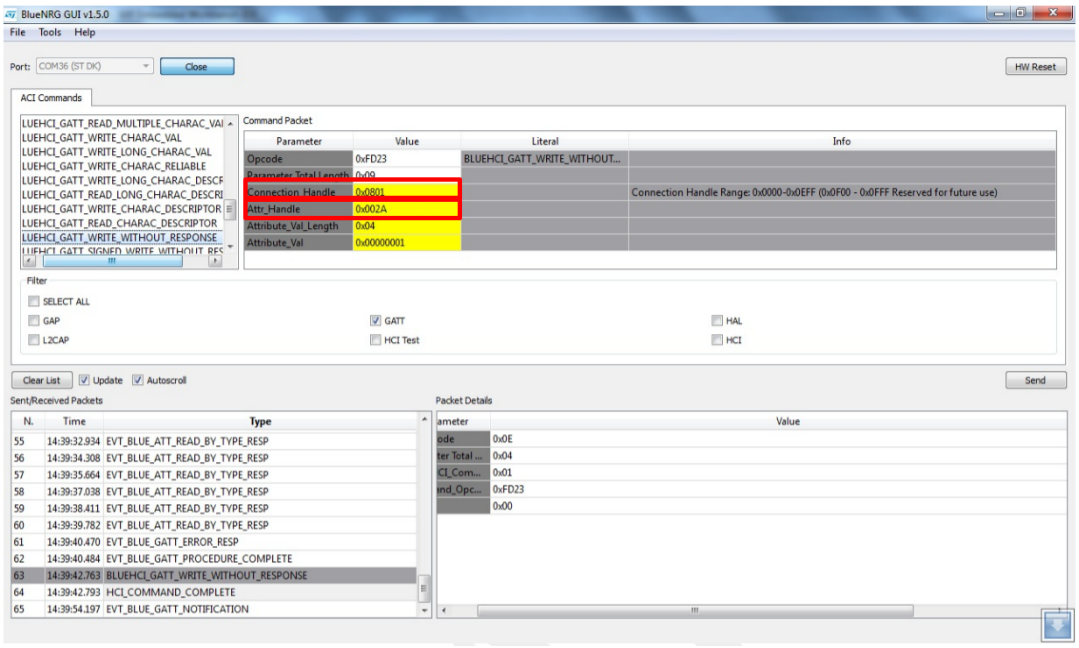
The BLUEHCI_GATT_WRITE_WITHOUT_RESPONSE function reads a particular characteristic on the server. Four parameters are required:

- characteristic attribute handle
- connection handle (0x0801)
- data length
- data value to write

For example, the *LED button characteristic* has only one writable attribute.

As explained in Section 2.5.1 Time Service, by writing data to this characteristic we can toggle the peripheral device LED2: the BLUEHCI_GATT_WRITE_WITHOUT_RESPONSE function can switch the LED2 on/off.

Figure 13. Writing data to LED button time characteristic

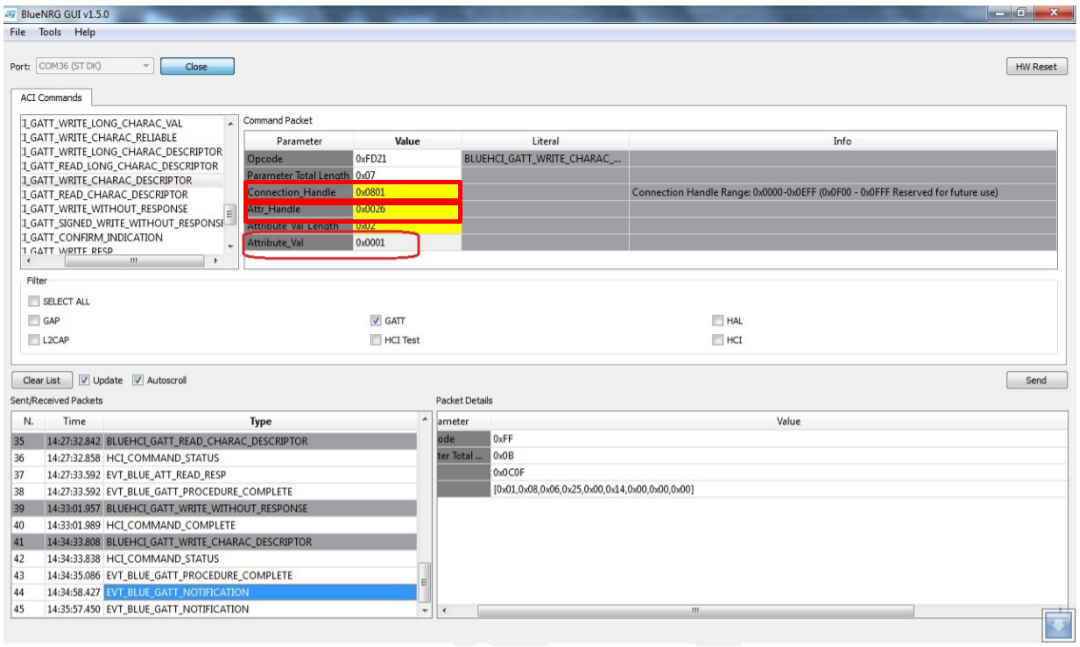


2.5.3.9 *Enabling notifications for a characteristic*

The BLUEHCI_GATT_WRITE_CHARAC_DESCRIPTOR function enables notifications for notifiable characteristics (i.e. characteristics which have CHAR_PROP_NOTIFY property).

This command can be used to write a descriptor to an attribute, by setting correct values for the attribute handle and configuration data ({0x00, 0x01}).

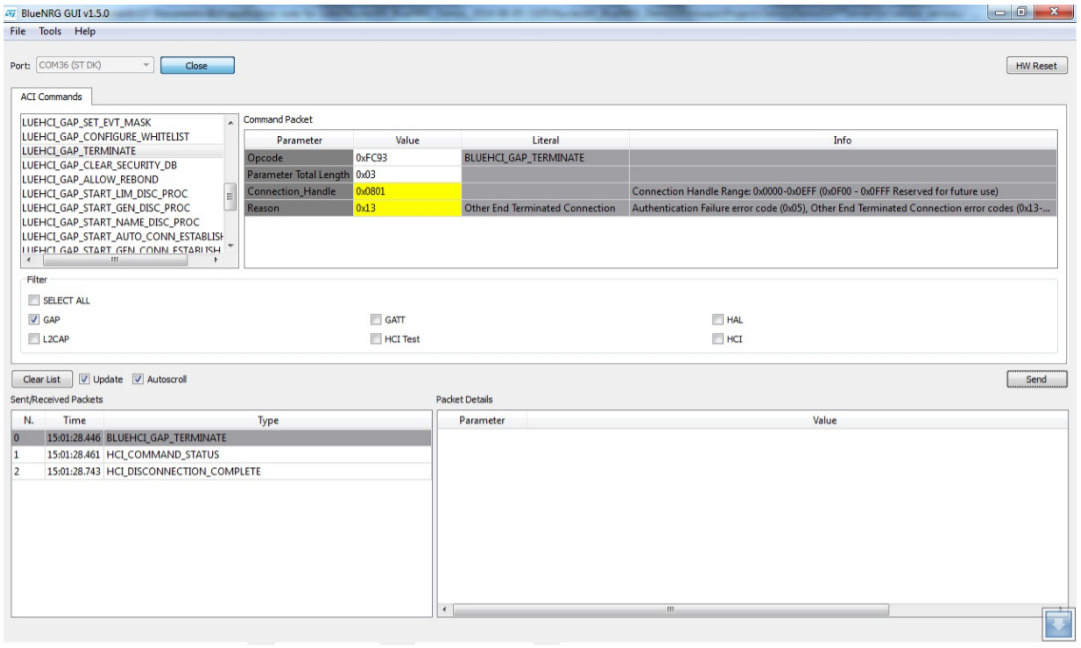
Figure 14. **Enabling notifications from server**



2.5.3.10 *Disconnecting from remote device*

The `BLUEHCI_GAP_TERMINATE` function disconnects the peripheral device from the central device.

Figure 15. **Disconnecting the peripheral device**



2.5.4 Adding security to sample application

The sample application code can be modified to protect a certain characteristic and perform the pairing from the BlueNRG GU to access it.

2.5.4.1 *Protecting the characteristic*

To add read protection to a characteristic, you have to modify the *secPermissions* flag of the `aci_gatt_add_char` function.

The following example shows how to set up the Time Service *seconds characteristic* with a protection requiring an authenticated pairing to ensure the data exchange encryption.

```
ret = aci_gatt_add_char(timeServHandle, UUID_TYPE_128, uuid, 4, CHAR_PROP_READ,
ATTR_PERMISSION_ENCRY_READ | ATTR_PERMISSION_AUTHEN_READ, 0, 16, 0,
&secondsCharHandle);
```

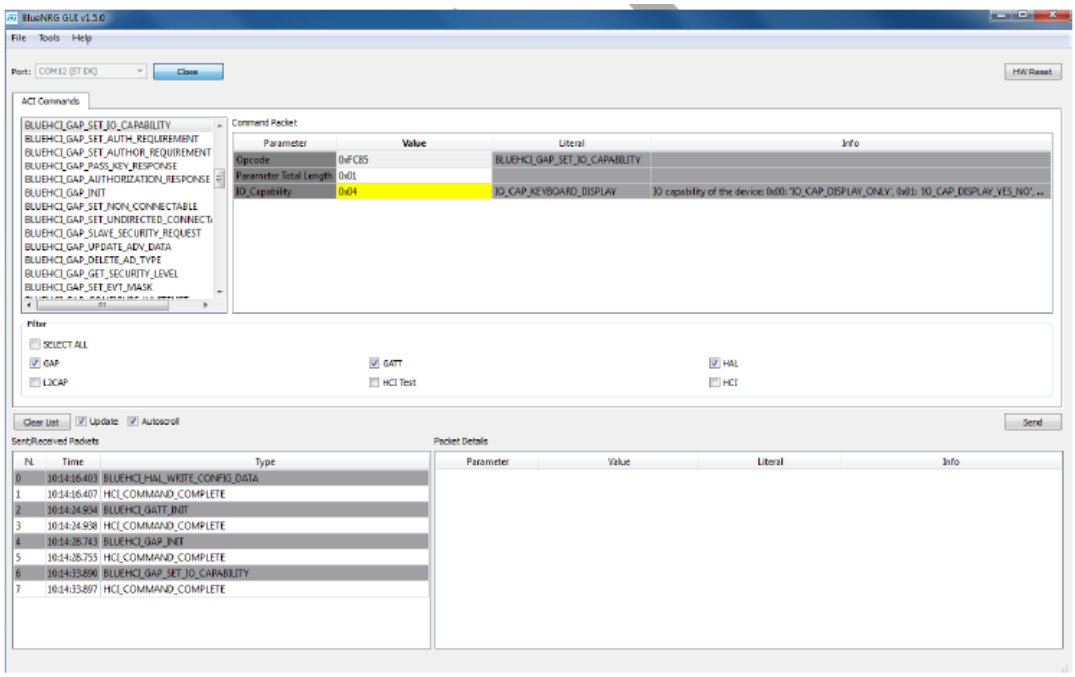
If the GATT client tries reading this characteristic without an authentication pairing, the reading returns an error.

2.5.4.2 *Performing the pairing*

To perform the pairing with the BlueNRG GUI, after having initialized the BLE central device (refer to Section 2.5.3.2 Initializing the BLE Central device), you have to declare the device I/O capabilities through the `BLUEHCI_GAP_SET_IO_CAPABILITY` function.

In the figure below, the selected capability is indicated by `IO_CAP_KEYBOARD_DISPLAY`, as on the PC running the BlueNRG GUI there are a keyboard and a display.

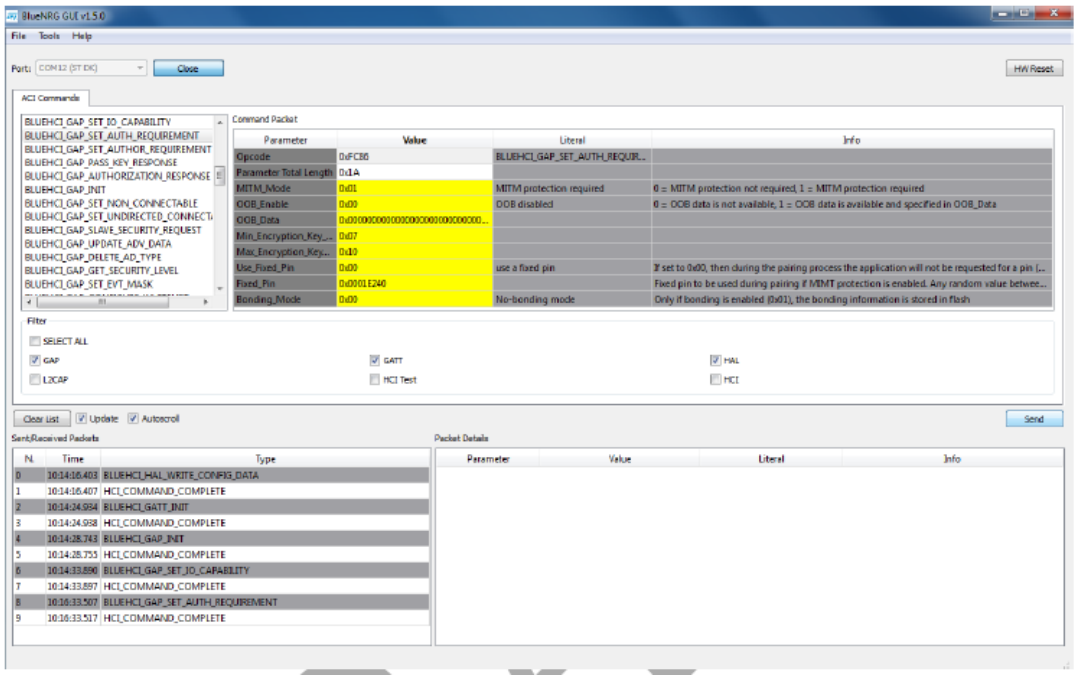
Figure 16. Setting the device I/O capabilities



You also have to configure the pairing parameters via the `BLUEHCI_GAP_SET_AUTH_REQUIREMENT` function.

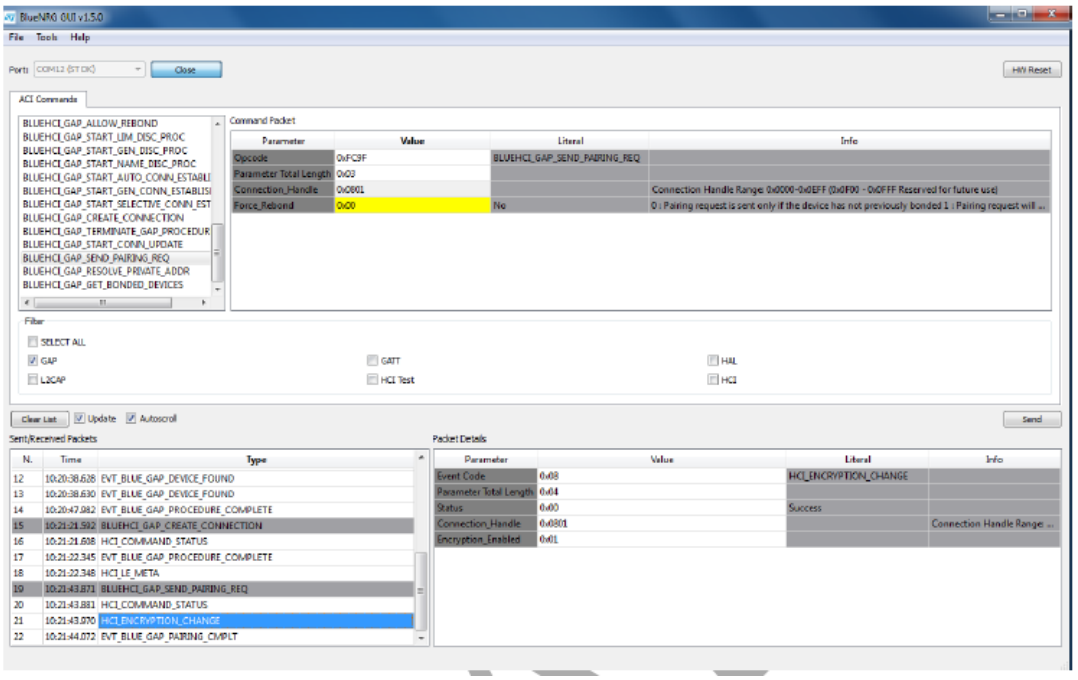
To match with the sample application code, the man-in-the-middle protection is set, OOB is disabled, and a fixed PIN (123456) is configured.

Figure 17. Setting the pairing parameters



After this operation, and after a connection has been set up as described in Section 2.5.3.4 Connecting to BLE peripheral device, you can start the pairing procedure via the `BLUEHCI_GAP_SEND_PAIRING_REQUEST` function, which will result in an `HCI_ENCRYPTION_CHANGE` and in an `EVT_BLUE_GAP_PAIRING_CMPL` event indicating the result of the pairing.

Figure 18. Pairing with the device



After successful pairing, you can read the protected characteristic via the procedure described in Section 2.5.3.7 Reading characteristic values.

2.6 System setup guide

2.6.1 Hardware description

2.6.1.1 STM32 Nucleo

STM32 Nucleo development boards provide an affordable and flexible way for users to test solutions and build prototypes with any STM32 microcontroller line.

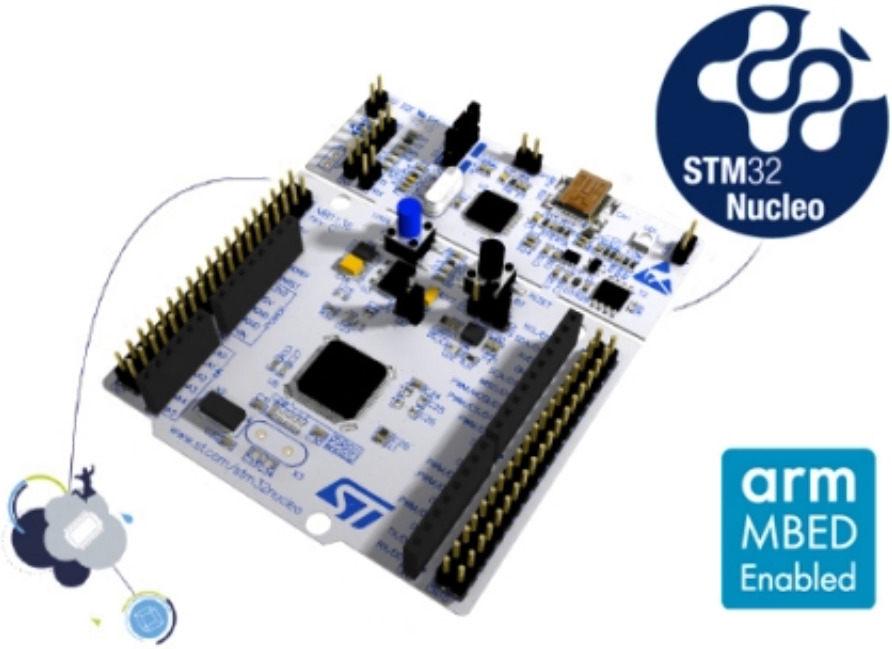
The Arduino connectivity support and ST morpho connectors make it easy to expand the functionality of the STM32 Nucleo open development platform with a wide range of specialized expansion boards to choose from.

The STM32 Nucleo board does not require separate probes as it integrates the ST-LINK/V2-1 debugger/ programmer.

The STM32 Nucleo board comes with the comprehensive STM32 software HAL library together with various packaged software examples for different IDEs (IAR EWARM, Keil MDK-ARM, STM32CubeIDE, mbed and GCC/ LLVM).

All STM32 Nucleo users have free access to the mbed online resources (compiler, C/C++ SDK and developer community) at www.mbed.org to easily build complete applications.

Figure 19. STM32 Nucleo board



2.6.1.2 X-NUCLEO-ID05A2 expansion board

The X-NUCLEO-IDB05A2 Bluetooth® Low Energy expansion board is based on the BlueNRG-M0 Bluetooth® Low Energy network processor module.

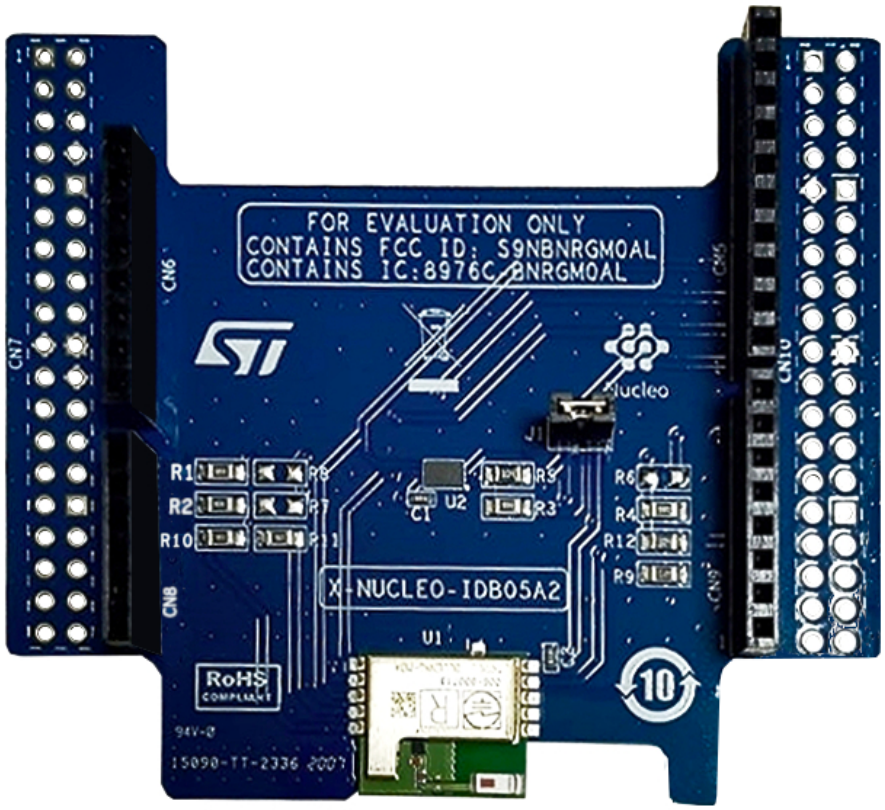
The BlueNRG-M0 is Bluetooth v4.2 compliant, FCC, and IC certified (FCC ID: S9NBNRGM0AL; IC: 8976C-BNRGM0AL). It supports simultaneous master/slave roles and can behave as a Bluetooth® Low Energy sensor and hub device at the same time.

The BlueNRG-M0 provides a complete RF platform in a tiny form factor, with integrated radio, antenna, high frequency, and LPO oscillators.

The X-NUCLEO-IDB05A2 is compatible with the ST morpho (not mounted) and Arduino UNO R3 connector layout.

The X-NUCLEO-IDB05A2 interfaces with the STM32 microcontroller via the SPI pin and allows changing the default SPI clock, SPI chip select, and SPI IRQ by replacing a resistor on the expansion board.

Figure 20. **X-NUCLEO-IDB05A2 expansion board**



2.6.2 Software description

The following software components are needed to setup a suitable development environment for creating applications for the STM32 Nucleo equipped with the BlueNRG-MS/BlueNRG-M0 expansion board:

- X-CUBE-BLE1: an STM32Cube expansion for Bluetooth Low Energy application development. The X-CUBE-BLE1 firmware and related documentation is available on www.st.com.
- Development toolchain and compiler. The STM32Cube expansion software supports the following environments:
  - IAR Embedded Workbench for ARM® (EWARM) toolchain + ST-LINK/V2
  - RealView Microcontroller Development Kit (MDK-ARM) toolchain + ST-LINK/V2
  - STM32CubeIDE + ST-LINK/V2
- STSW-BNRGUI graphical user interface

2.6.3 Hardware setup

The following hardware components are required:

1. One STM32 Nucleo development platform
2. One BlueNRG-MS/BlueNRG-M0 expansion board (order code: X-NUCLEO-IDB05A2)
3. One USB type A to Mini-B USB cable to connect the Nucleo to the PC

Note: To test a BLE application running on the STM32 Nucleo platform, you have to add another STM32 Nucleo platform and X-NUCLEO-IDB05A2 stack.

2.6.4 STM32 Nucleo and BlueNRG-MS/BlueNRG-M0 expansion board setup

The STM32 Nucleo development board allows the exploitation of the BLE capabilities provided by the BlueNRG-MS/BlueNRG-M0 module.

The STM32 Nucleo board integrates the ST-LINK/V2-1 debugger/programmer. Developers can download the relevant version of the ST-LINK/V2-1 USB driver by searching STSW-LINK009 on www.st.com (according to the Windows version).

The X-NUCLEO-IDB05A2 expansion board can be easily connected to the STM32 Nucleo through the Arduino UNO R3 extension connector, and can interface with the external STM32 microcontroller using the SPI transport layer.

Revision history

Table 2. **Document revision history**

Date	Revision	Changes
27-Aug-2015	1	Initial release.
30-Jan-2017	2	Updated Introduction, Section 2: "What is STM32Cube?", Figure 5: "STM32 Nucleo + BlueNRG/ BlueNRG-MS expansion board software architecture", Section 3.3: "Folder structure", Section 3.5: "SensorDemo application description", Section 3.5.3: "Testing the application", Section 4.2: "Software description", Section 4.3.3.3: "STM32 Nucleo and BlueNRG/BlueNRG-MS expansion board setup". Added BlueNRG-MS references throughout document. Added BlueNRG-MS references throughout document.
14-Feb-2017	3	Updated How does this software complement STM32Cube?, Figure 6: "X-CUBE-BLE1 package folder structure"and Section 3.5: "SensorDemo application description".
27-Apr-2020	4	Updated Introduction, Section 2.1 Overview, Figure 4. X-CUBE-BLE1 software architecture, Section 2.3 Folders, Section 2.5 SensorDemo-BLESensor-App application description, Section 2.5.1 Time Service, Section 2.5.1.1 Adding Time Service and characteristics, Section 2.5.2.2 Obtaining characteristic value, Section 2.5.3 Testing the application, Section 2.5.3.2 Initializing the BLE Central device, Section 2.5.3.3 Scanning for BLE peripheral device, Section 2.5.3.4 Connecting to BLE peripheral device, Section 2.5.3.5 Getting services supported by the BLE peripheral device and Section 2.6.2 Software description. Added BlueNRG-M0 module and X-NUCLEO-IDB05A2 expansion board compatibility information. Added Section 2.6.1.2 X-NUCLEO-IDB05A2 expansion board. Text changes throughout the document.
28-Oct-2021	5	Updated introduction, Section 2.3 Folders, and Section 2.5 SensorDemo-BLESensor-App application description.
20-Oct-2022	6	Updated Section 2.1 Overview.

Contents

List of tables

List of figures