Bluetooth User Manual

This document introduces how to use Bluetooth.

Table of Contents

1 Introduction

Ameba provides a set of Bluetooth functionality, such as BLE Beacon, BT Config, BLE GATT server and SPP. This document intends to provide a comprehensive guide to the Bluetooth examples and APIs.

NOTE: Set **CONFIG_BT** flag in **platform_opts.h** to enable Bluetooth.

2 AT command

2.1 AT command list

AT Command	Description
Common	
ATB?	Get BT Status
ATBA	Get BT Device address
ATBB	Set BT Undiscoverable/Discoverable
ATBC	Create a Bluetooth LE connection (GATT)
ATBD	Disconnect Bluetooth Connection
ATBG	Get Pairing Infomation
ATBH	Delete Pairing Infomation
ATBI	Get RSSI value of connected device
ATBK	Reply GAP passkey
ATBM	Set default GATT mtu
ATBN	Get BT device name (BR/EDR)
ATBP	Power on/off BT
ATBp	Get BT peer address
ATBS	Scan BT
ATBY	Reply GAP user confirm
Example GATT Client	
ATBR	Read handle
ATBW	Write handle
Example SPP	
ATBE	Create Bluetooth SPP connection
ATBF	Send data (last created SPP connection)

2.2 AT command list

2.2.1 COMMON

2.2.1.1 'ATB?' Get BT Status

Description:	
Command Format:	ATB?<CR>
Default Value:	None
Response:	INITIALIZING / ON / DEINITIALIZING / OFF / RESERVED

2.2.1.2 'ATBA' Get BT Device address

Description:	
Command Format:	ATBA<CR>
Default Value:	None

Response:	BR_BD_ADDR = br_addr
	BLE_BD_ADDR = ble_addr

2.2.1.3 'ATBB' Set BT Undiscoverable/Discoverable

Description:	

Command Format:	ATBB=BR,0/1<CR>
	ATBB=LE,0/1 <CR>

Default Value:	None
Response:	TBD

2.2.1.4 'ATBC' Create a Bluetooth LE connection (GATT)

Description:	P: public address, R: random address
Command Format:	ATBC=P/R,addr[,mtu]<CR>
Default Value:	mtu: default 256
Response:	TBD

2.2.1.5 'ATBD' Disconnect Bluetooth Connection

Description:	
Command Format:	ATBD=GATT/SPP<CR>
Default Value:	None
Response:	TBD

2.2.1.6 'ATBG' Get Pairing Infomation

Description:	
Command Format:	ATBG <CR>
Default Value:	None
Response:	[ATBG] [BT_GAP_EV_REMOTE_DEVICE_PAIRED_INFO] Number of paired device: [num] [BT_GAP_EV_REMOTE_DEVICE_PAIRED_INFO][idx] DeviceType= [LE/BR/EDR], AddrType = [public/random], BT_Addr= [addr]

2.2.1.7 'ATBH' Delete Pairing Infomation

Description:	
Command Format:	ATBH=BR/LE,P/R,BLE_BD_ADDR<CR> ATBH=ALL<CR>
Default Value:	None
Response:	TBD

2.2.1.8 'ATBI' Get RSSI value of connected device

Description:	
Command Format:	ATBI=GATT/SPP<CR>
Default Value:	None
Response:	RSSI of connected device: [rssi]

2.2.1.9 'ATBK' Reply GAP passkey

Description:	
Command Format:	ATBK=passkey<CR>
Default Value:	None
Response:	TBD

2.2.1.10 'ATBM' Set default GATT mtu

Description:	
Command Format:	ATBM=mtu<CR>
Default Value:	None
Response:	TBD

2.2.1.11 'ATBN' Get BT device name (BR/EDR)

Description:	
Command Format:	ATBN<CR>
Default Value:	None

Response:	TBD

2.2.1.12 ‘ATBP’ Power on/off BT

Description:	
Command Format:	ATBP=0/1<CR>
Default Value:	None
Response:	TBD

2.2.1.13 ‘ATBp’ Get BT peer address

Description:	
Command Format:	ATBp<CR>
Default Value:	None
Response:	BT peer_addr_type = [public/random], peer_addr = [addr]

2.2.1.14 ‘ATBS’ Scan BT

Description:

Command Format: ATBS=BR|LE|BOTH|0,[options]<CR>

```
-f    #   [LE] enable ble duplicate filtering
-r    #   [LE] enable ble scan request
-i    #   [LE] le scan interval [0x0004-0x4000]
      #        Time = N * 0.625 msec
-w    #   [LE] le scan window [0x0004-0x4000]
      #        Time = N * 0.625 msec
-t    #   scan time: [BR][0x01-0x30] (default 0x0A <12.8 sec>)
      #              [LE][0x00-0x30], 0 means BLE continuous scan
      #              Time = N * 1.28 sec
-n    #   [BR] Maximum number of responses, 0 means unlimited
```

Default Value:	-f	is 1 (ble duplicate filtering enabled)
	-r	is 1 (ble scan request enabled)
	-i	is 0x0048 (45 msec)
	-w	Is 0x0030 (30 ms)
	-t	is 0x0A (12.8 sec)
	-n	is 0 (unlimited number of responses)
Response:	TBD	

2.2.1.15 'ATBY' Reply GAP user confirm

Description:	
Command Format:	ATBY=0/1<CR>
Default Value:	None
Response:	TBD

2.2.2 Example SPP

2.2.2.1 'ATBE' Create Bluetooth SPP connection

Description:	
Command Format:	ATBE=addr[,mtu]<CR>
Default Value:	mtu: default 490
Response:	None

2.2.2.2 'ATBF' Send data (last created SPP connection)

Description:	
Command Format:	ATBF=data<CR>
Default Value:	None
Response:	None

2.2.3 Example GATT Client

2.2.3.1 'ATBR' Read handle

Description:	
Command Format:	ATBR=handle<CR>
Default Value:	None
Response:	None

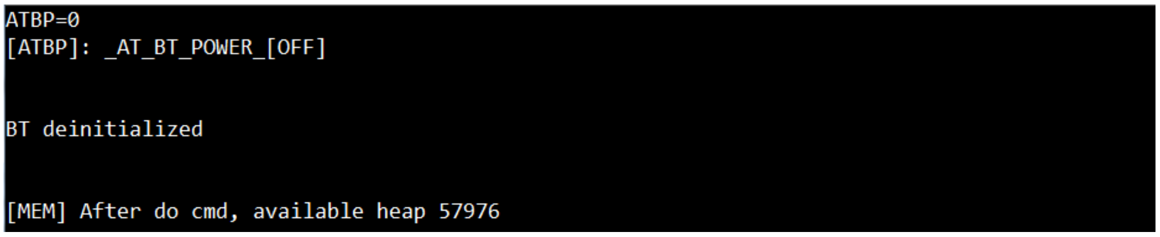
2.2.3.2 'ATBW' Write handle

Description:	
Command Format:	ATBW=handle,req/cmd,string<CR>
	ATBW=handle,req/cmd,0xdata<CR>
Default Value:	None
Response:	None

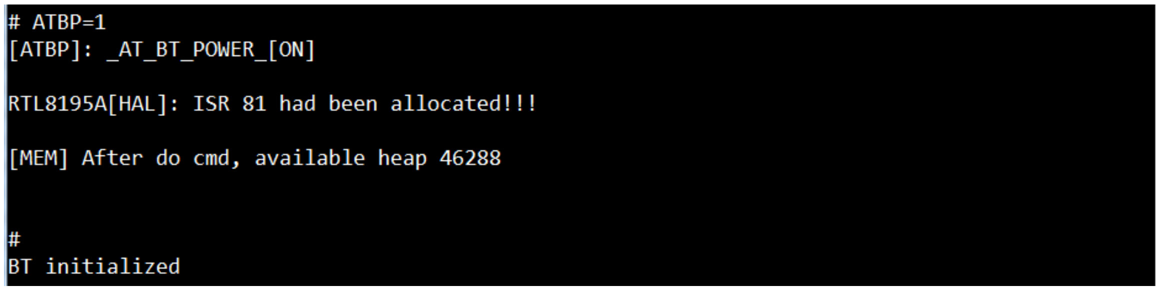
2.3 AT command Usage

2.3.1 Disable/Enable Bluetooth

The “ATBP=0/1” commands are used to initialize and de-initialize BT correspondingly. Before using BT related function, it needs to be initialized.



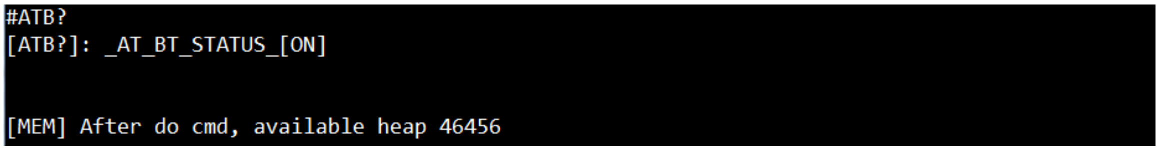
```
ATBP=0
[ATBP]: _AT_BT_POWER_[OFF]


BT deinitialized


[MEM] After do cmd, available heap 57976
```



```
# ATBP=1
[ATBP]: _AT_BT_POWER_[ON]

RTL8195A[HAL]: ISR 81 had been allocated!!!

[MEM] After do cmd, available heap 46288


#
BT initialized
```

2.3.2 Bluetooth Status

The “ATB?” command can be used to get Bluetooth status to see if BT is powered on or not.

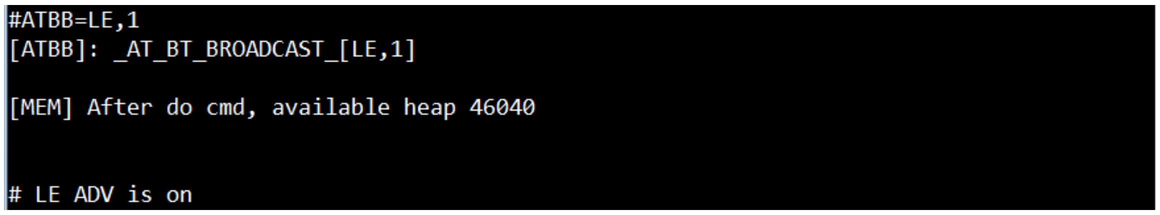


```
#ATB?
[ATB?]: _AT_BT_STATUS_[ON]


[MEM] After do cmd, available heap 46456
```

2.3.3 BT Discoverable/ Undiscoverable

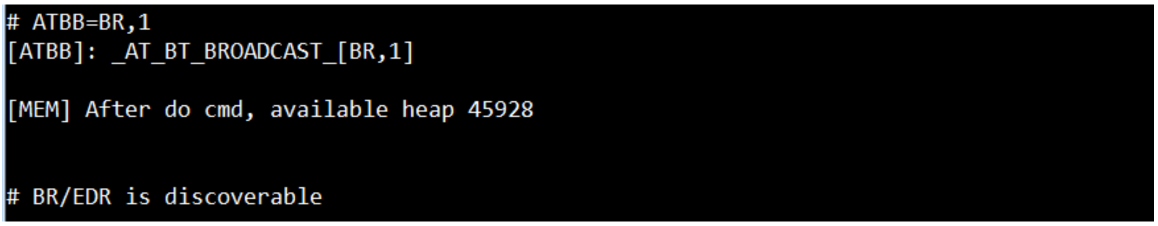
The “ATBB” command can be used to let BT become discoverable or undiscoverable.

By using **ATBB=LE,1** , Ameba starts broadcasting ADV packets. When running Beacon example, the Beacon can be detected. When running GATT Server example, GATT Server advertises its presence and therefore is connectable.



```
#ATBB=LE,1
[ATBB]: _AT_BT_BROADCAST_[LE,1]

[MEM] After do cmd, available heap 46040


# LE ADV is on
```

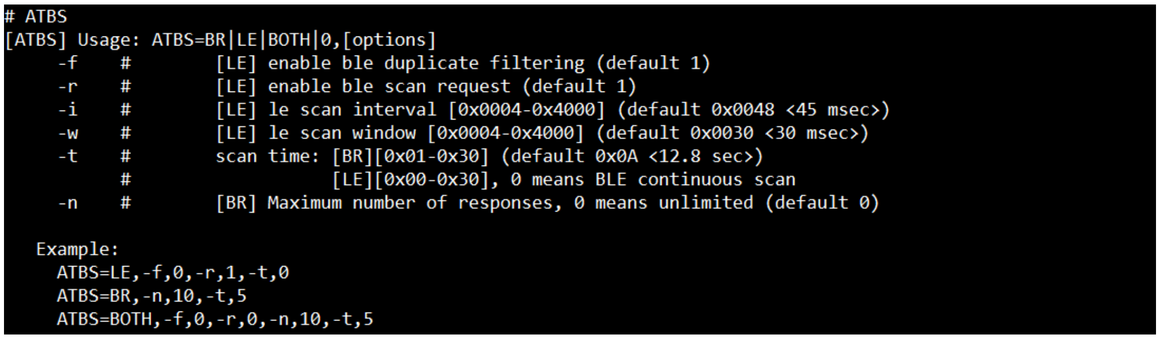
By using **ATBB=BR,1**, Ameba will respond to Inquiry and Page, so it becomes discoverable to other Classic BT devices.



```
# ATBB=BR,1
[ATBB]: _AT_BT_BROADCAST_[BR,1]

[MEM] After do cmd, available heap 45928


# BR/EDR is discoverable
```

2.3.4 Scan BT devices

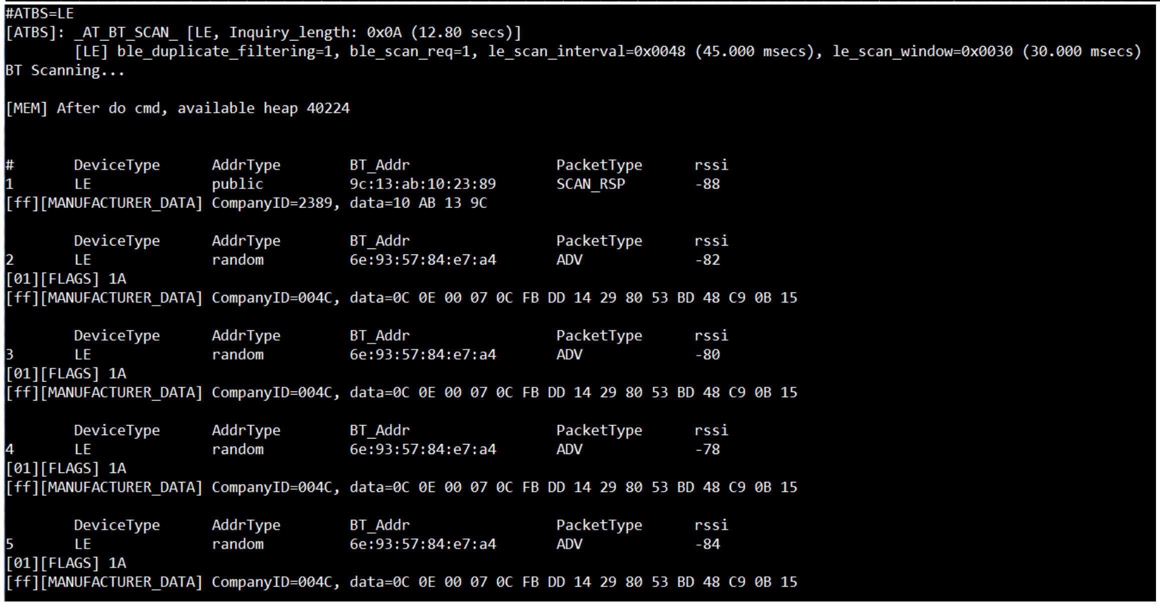
The “ATBS” command can be used to scan BT devices.



```
# ATBS
[ATBS] Usage: ATBS=BR|LE|BOTH|0,[options]
     -f    #        [LE] enable ble duplicate filtering (default 1)
     -r    #        [LE] enable ble scan request (default 1)
     -i    #        [LE] le scan interval [0x0004-0x4000] (default 0x0048 <45 msec>)
     -w    #        [LE] le scan window [0x0004-0x4000] (default 0x0030 <30 msec>)
     -t    #        scan time: [BR][0x01-0x30] (default 0x0A <12.8 sec>)
           #                   [LE][0x00-0x30], 0 means BLE continuous scan
     -n    #        [BR] Maximum number of responses, 0 means unlimited (default 0)

   Example:
     ATBS=LE,-f,0,-r,1,-t,0
     ATBS=BR,-n,10,-t,5
     ATBS=BOTH,-f,0,-r,0,-n,10,-t,5
```

User can choose to scan Classic BT device only (ATBS=BR), BLE devices only (ATBS=LE), both Classic BT and LE devices (ATBS=BOTH), or stop the scanning process (ATBS=0).

User can also specify more scan options. For example, user can use **ATBS=LE,-f,0,-r,1,-t,0** to do continuous scan with every ADV event, or use **ATBS=LE,-f,1,-r,0,-t,10** to scan only ADV packets for a short time and filter the ADV packets with the same content.



```
#ATBS=LE
[ATBS]: _AT_BT_SCAN_ [LE, Inquiry_length: 0x0A (12.80 secs)]
        [LE] ble_duplicate_filtering=1, ble_scan_req=1, le_scan_interval=0x0048 (45.000 msecs), le_scan_window=0x0030 (30.000 msecs)
BT Scanning...

[MEM] After do cmd, available heap 40224


#       DeviceType      AddrType        BT_Addr                 PacketType      rssi
1       LE              public          9c:13:ab:10:23:89       SCAN_RSP        -88
[ff][MANUFACTURER_DATA] CompanyID=2389, data=10 AB 13 9C

        DeviceType      AddrType        BT_Addr                 PacketType      rssi
2       LE              random          6e:93:57:84:e7:a4       ADV             -82
[01][FLAGS] 1A
[ff][MANUFACTURER_DATA] CompanyID=004C, data=0C 0E 00 07 0C FB DD 14 29 80 53 BD 48 C9 0B 15

        DeviceType      AddrType        BT_Addr                 PacketType      rssi
3       LE              random          6e:93:57:84:e7:a4       ADV             -80
[01][FLAGS] 1A
[ff][MANUFACTURER_DATA] CompanyID=004C, data=0C 0E 00 07 0C FB DD 14 29 80 53 BD 48 C9 0B 15

        DeviceType      AddrType        BT_Addr                 PacketType      rssi
4       LE              random          6e:93:57:84:e7:a4       ADV             -78
[01][FLAGS] 1A
[ff][MANUFACTURER_DATA] CompanyID=004C, data=0C 0E 00 07 0C FB DD 14 29 80 53 BD 48 C9 0B 15

        DeviceType      AddrType        BT_Addr                 PacketType      rssi
5       LE              random          6e:93:57:84:e7:a4       ADV             -84
[01][FLAGS] 1A
[ff][MANUFACTURER_DATA] CompanyID=004C, data=0C 0E 00 07 0C FB DD 14 29 80 53 BD 48 C9 0B 15
```

2.3.5 SPP Connection

The “ATBE” command can be used to connect to an indicated BT address by creating an SPP connection.

To disconnect, type “ATBD=SPP”.

lease refer to **SPP (Serial Port Profile)** in this document for more detail.

2.3.6 GATT Connection

The “ATBC” command can be used to connect to an indicated BT address by creating a GATT connection.

To disconnect, type “ATBD=GATT”.

Please refer to **GATT Client** in this document for more detail.

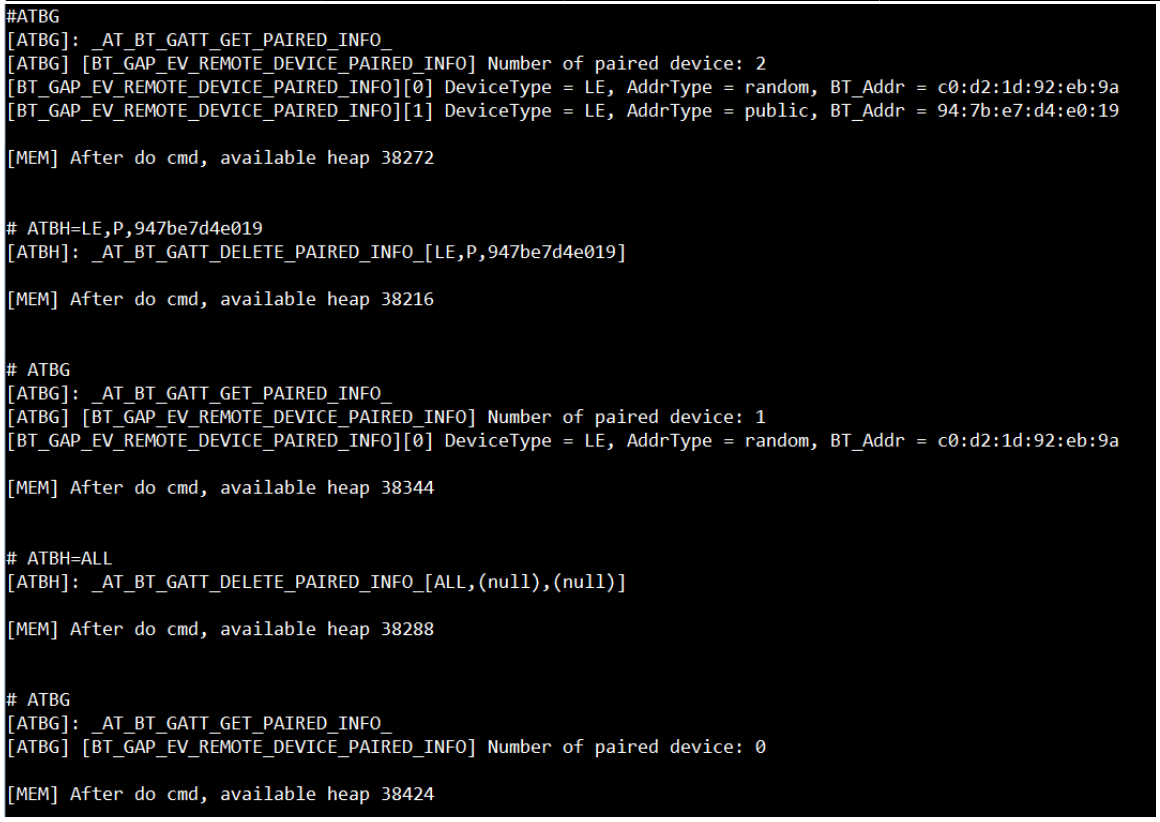
2.3.7 Get / Delete Pairing Information

The “ATBG” command can be used to get pairing information.

To get pairing information, type “ATBG”.

The “ATBH” command can be used to delete pairing information.

User can specify an entry to delete, or delete the whole list (ATBH=ALL).



```
#ATBG
[ATBG]: _AT_BT_GATT_GET_PAIRED_INFO_
[ATBG] [BT_GAP_EV_REMOTE_DEVICE_PAIRED_INFO] Number of paired device: 2
[BT_GAP_EV_REMOTE_DEVICE_PAIRED_INFO][0] DeviceType = LE, AddrType = random, BT_Addr = c0:d2:1d:92:eb:9a
[BT_GAP_EV_REMOTE_DEVICE_PAIRED_INFO][1] DeviceType = LE, AddrType = public, BT_Addr = 94:7b:e7:d4:e0:19

[MEM] After do cmd, available heap 38272


# ATBH=LE,P,947be7d4e019
[ATBH]: _AT_BT_GATT_DELETE_PAIRED_INFO_[LE,P,947be7d4e019]

[MEM] After do cmd, available heap 38216


# ATBG
[ATBG]: _AT_BT_GATT_GET_PAIRED_INFO_
[ATBG] [BT_GAP_EV_REMOTE_DEVICE_PAIRED_INFO] Number of paired device: 1
[BT_GAP_EV_REMOTE_DEVICE_PAIRED_INFO][0] DeviceType = LE, AddrType = random, BT_Addr = c0:d2:1d:92:eb:9a

[MEM] After do cmd, available heap 38344


# ATBH=ALL
[ATBH]: _AT_BT_GATT_DELETE_PAIRED_INFO_[ALL,(null),(null)]

[MEM] After do cmd, available heap 38288


# ATBG
[ATBG]: _AT_BT_GATT_GET_PAIRED_INFO_
[ATBG] [BT_GAP_EV_REMOTE_DEVICE_PAIRED_INFO] Number of paired device: 0

[MEM] After do cmd, available heap 38424
```

3 BT Address

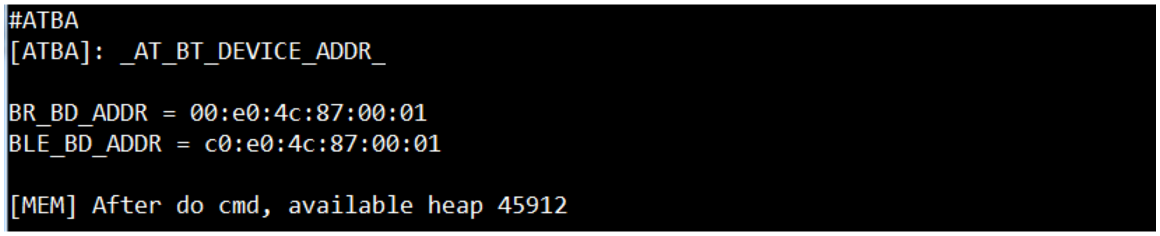
There are two types of address setting in tBT_GapLEAdvParamStru: **BT_GAP_BLE_OWN_ADDRESS_TYPE_PUBLIC** and **BT_GAP_BLE_OWN_ADDRESS_TYPE_RANDOM**.

If use **BT_GAP_BLE_OWN_ADDRESS_TYPE_PUBLIC**, classic BT and BLE will use same BT address. For example, in the picture below, both classic BT and BLE address are **00:e0:4c:87:00:01**

If use **BT_GAP_BLE_OWN_ADDRESS_TYPE_RANDOM**, classic BT and BLE will use different BT address. Classic BT uses static address get by AT command **ATBA**, and BLE uses different BT address by setting the two most significant bits of the static address to 1. For example, in the picture below, if use **BT_GAP_BLE_OWN_ADDRESS_TYPE_RANDOM**, classic BT address will be **00:e0:4c:87:00:01**, and BLE address will be **c0:e0:4c:87:00:01**

* IMPORTANT:

Due to **Android OS problem**, some smartphones will wrongly connect **BLE** device with public address type in **BR (Classic BT) way**, which causes connection fail. So, **BLE address is forced to be BT_GAP_BLE_OWN_ADDRESS_TYPE_RANDOM.** Classic BT address and BLE uses different BT address.



```
#ATBA
[ATBA]: _AT_BT_DEVICE_ADDR_

BR_BD_ADDR = 00:e0:4c:87:00:01
BLE_BD_ADDR = c0:e0:4c:87:00:01

[MEM] After do cmd, available heap 45912
```

4 Beacon

Ameba provides two types of Beacon: Apple iBeacon and Radius Networks AltBeacon.

4.1 How to Run the Example

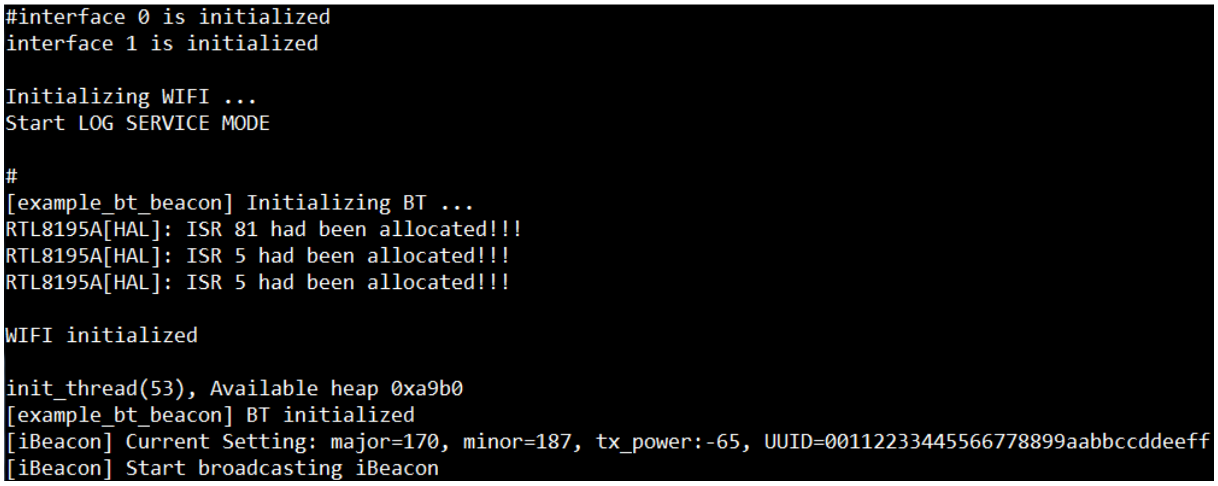
To run the example of bt_beacon, set CONFIG_BT and CONFIG_EXAMPLE_BT_BEACON flag in \project\realtek_ameba1_va0_example\inc\platform_opts.h

```
//Config in platform_opts.h
#define CONFIG_BT    1
…
#define CONFIG_EXAMPLE_BT_BEACON                1
```

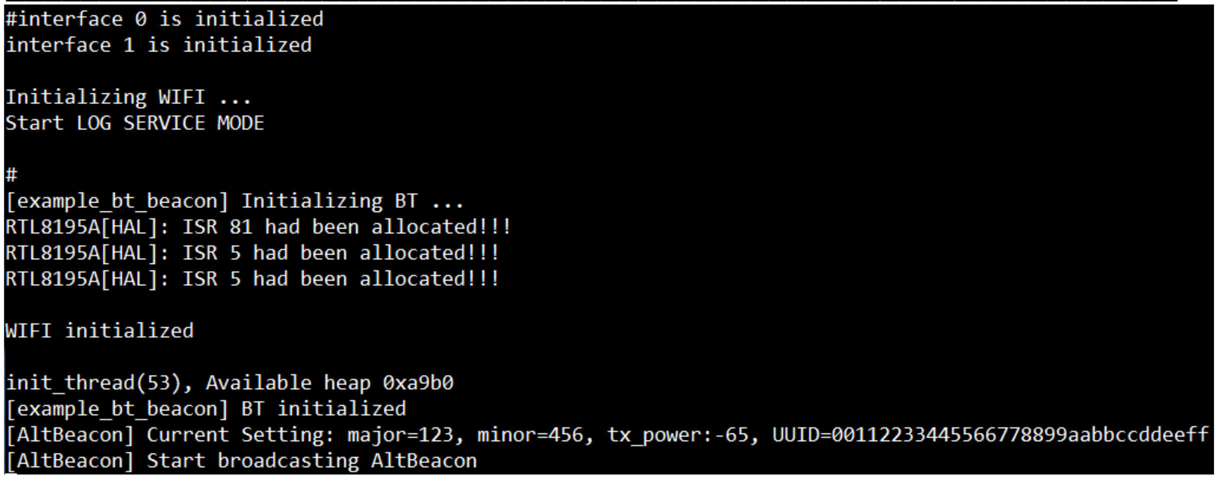
Then, open component\common\example\bt_beacon\bt_beacon.c. Set BEACON_TYPE to BEACON_TYPE_IBEACON or BEACON_TYPE_ALTBEACON.

```
// bt_beacon.c
#define BEACON_TYPE         BEACON_TYPE_IBEACON
```

Ameba will start broadcasting Beacons.



```
#interface 0 is initialized
interface 1 is initialized

Initializing WIFI ...
Start LOG SERVICE MODE

#
[example_bt_beacon] Initializing BT ...
RTL8195A[HAL]: ISR 81 had been allocated!!!
RTL8195A[HAL]: ISR 5 had been allocated!!!
RTL8195A[HAL]: ISR 5 had been allocated!!!

WIFI initialized

init_thread(53), Available heap 0xa9b0
[example_bt_beacon] BT initialized
[iBeacon] Current Setting: major=170, minor=187, tx_power:-65, UUID=00112233445566778899aabbccddeeff
[iBeacon] Start broadcasting iBeacon
```



```
#interface 0 is initialized
interface 1 is initialized

Initializing WIFI ...
Start LOG SERVICE MODE

#
[example_bt_beacon] Initializing BT ...
RTL8195A[HAL]: ISR 81 had been allocated!!!
RTL8195A[HAL]: ISR 5 had been allocated!!!
RTL8195A[HAL]: ISR 5 had been allocated!!!

WIFI initialized

init_thread(53), Available heap 0xa9b0
[example_bt_beacon] BT initialized
[AltBeacon] Current Setting: major=123, minor=456, tx_power:-65, UUID=00112233445566778899aabbccddeeff
[AltBeacon] Start broadcasting AltBeacon
```

Users can use “Locate Beacon” on iOS or “nRF Connect” on Android to scan Beacons.

5 BT Config

It provides a way for Ameba to associate to AP easily. User only has to enter passphrase (if needed) for the first time associating to a new AP. After that, whenever user wants Ameba to associate to that AP, user just needs to click a button on the BT Config app, and everything is done.

5.1 Install APP

BT Config has two version: BT Config for iOS and BT Config for Android. iOS BT Config file is under the folder of "tools tools\bluetooth\BT Config\iOS", and Android BT Config is under the folder of "tools tools\bluetooth\BT Config\Android".

To install iOS BT Config app, first connect your phone to computer. Next, drag the ipa file into iTunes library. Then, click on Sync and done. Finally, trust the app by entering Setteings > General > Device Management > Realtek Semiconductor Corp > BTConfig.

BT Config on app store:
https://itunes.apple.com/tw/app/btconfig/id1194919510?mt=8

To install Android BT Config app, first connect your phone to computer. Next, drag the apk file into your phone. Then install it on your phone.

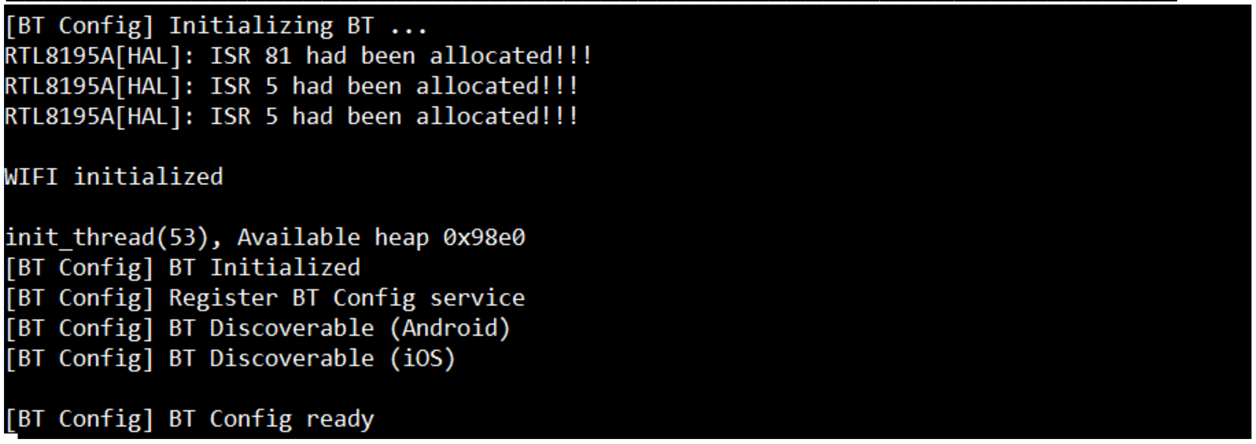
BT Config on Google Play:
https://play.google.com/store/apps/details?id=com.rtk.btconfig&hl=zh_TW

5.2 How to Run the Example

To run the example of BT Config, set CONFIG_BT and CONFIG_EXAMPLE_BT_CONFIG flag in \project\realtek_ameba1_va0_example\inc\platform_opts.h

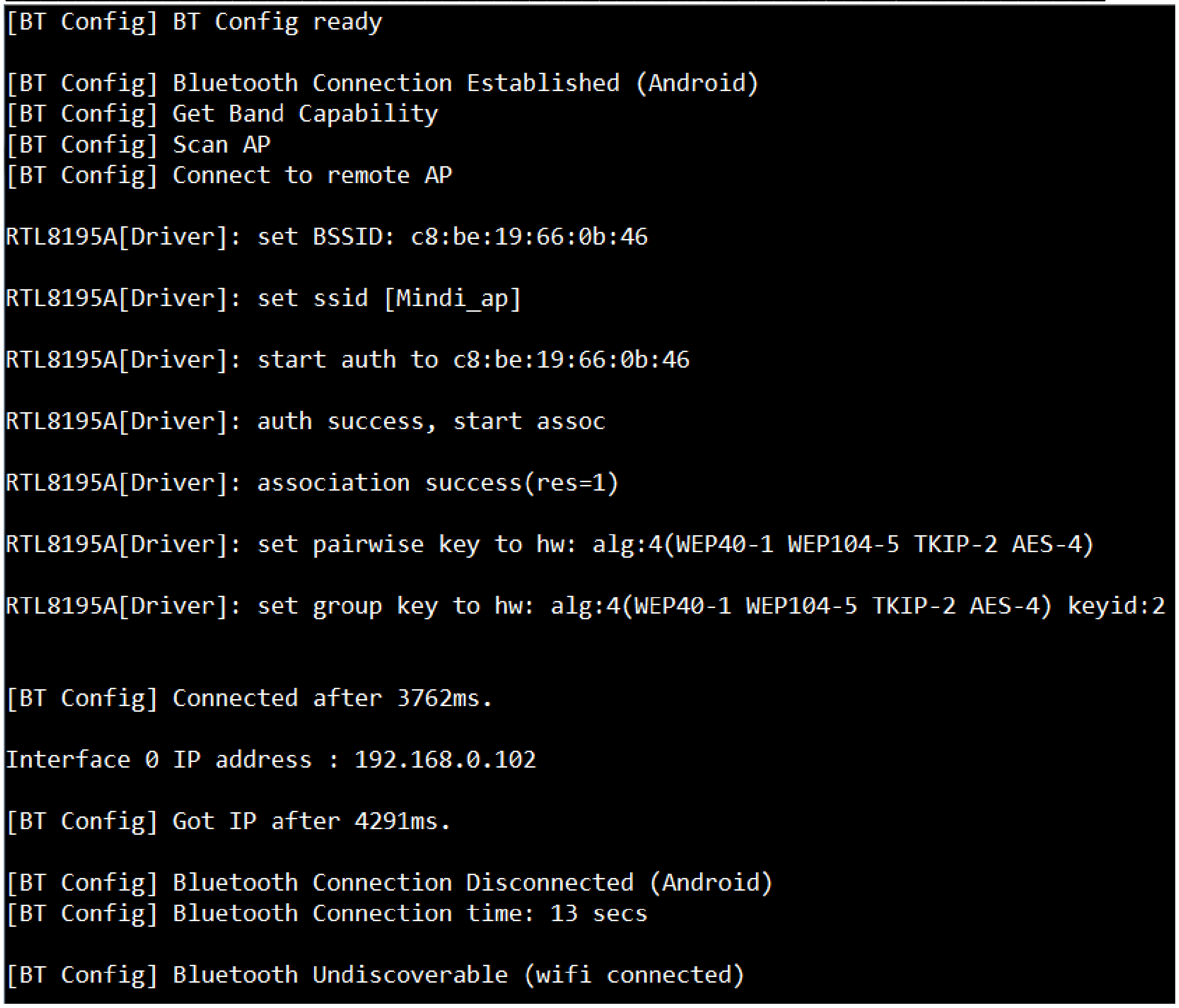
```
//Config in platform_opts.h
#define CONFIG_BT       1
…
#define CONFIG_EXAMPLE_BT_CONFIG                1
```

After seeing **[BT Config] BT Config ready**, user can start the app on the phone.



```
[BT Config] Initializing BT ...
RTL8195A[HAL]: ISR 81 had been allocated!!!
RTL8195A[HAL]: ISR 5 had been allocated!!!
RTL8195A[HAL]: ISR 5 had been allocated!!!

WIFI initialized

init_thread(53), Available heap 0x98e0
[BT Config] BT Initialized
[BT Config] Register BT Config service
[BT Config] BT Discoverable (Android)
[BT Config] BT Discoverable (iOS)

[BT Config] BT Config ready
```

After user starting the app on the phone and connecting to Ameba via Bluetooth, Ameba will automatically start scanning APs nearby, and report an AP list to the app. Then, user can select an AP on the list and Ameba will connect to the target AP specified by the app. After that, whenever user wants Ameba to associate to that AP, user just needs to click buttons on the BT Config app.



```
[BT Config] BT Config ready

[BT Config] Bluetooth Connection Established (Android)
[BT Config] Get Band Capability
[BT Config] Scan AP
[BT Config] Connect to remote AP

RTL8195A[Driver]: set BSSID: c8:be:19:66:0b:46

RTL8195A[Driver]: set ssid [Mindi_ap]

RTL8195A[Driver]: start auth to c8:be:19:66:0b:46

RTL8195A[Driver]: auth success, start assoc

RTL8195A[Driver]: association success(res=1)

RTL8195A[Driver]: set pairwise key to hw: alg:4(WEP40-1 WEP104-5 TKIP-2 AES-4)

RTL8195A[Driver]: set group key to hw: alg:4(WEP40-1 WEP104-5 TKIP-2 AES-4) keyid:2


[BT Config] Connected after 3762ms.

Interface 0 IP address : 192.168.0.102

[BT Config] Got IP after 4291ms.

[BT Config] Bluetooth Connection Disconnected (Android)
[BT Config] Bluetooth Connection time: 13 secs

[BT Config] Bluetooth Undiscoverable (wifi connected)
```

For details of BT Config app, please refer to the document in the folder of “tools\bluetooth\BT Config\iOS” and “tools\bluetooth\BT Config\Android”.

NOTE: When BT or wifi is connected, Ameba BT will be undiscoverable to BT Config app. Once both BT and wifi are disconnected, Ameba BT automatically resumes discoverable to BT Config app.

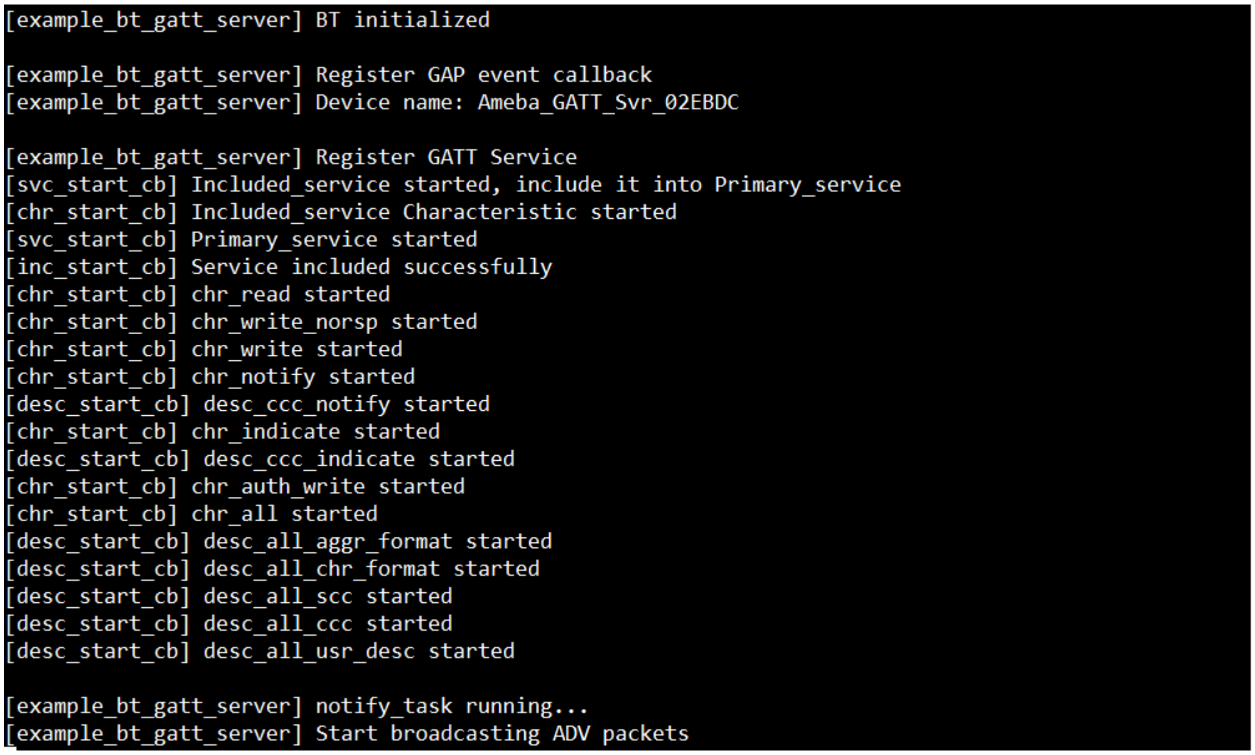
6 GATT Server

6.1 How to Run the Example

To run the example of bt_gatt_server, set CONFIG_BT and CONFIG_EXAMPLE_BT_GATT_SERVER flag in \project\realtek_ameba1_va0_example\inc\platform_opts.h

```
//Config in platform_opts.h
#define CONFIG_BT      1
…
#define CONFIG_EXAMPLE_BT_GATT_SERVER                1
```

Once see the message **[example_bt_gatt_server] BT initialized** and the following services/characteristics/descriptors started message, it means that the example is ready. Users can use GATT apps such as "LightBlue" on iOS or "nRF Connect" on Android to run GATT service defined in the example.



```
[example_bt_gatt_server] BT initialized

[example_bt_gatt_server] Register GAP event callback
[example_bt_gatt_server] Device name: Ameba_GATT_Svr_02EBDC

[example_bt_gatt_server] Register GATT Service
[svc_start_cb] Included_service started, include it into Primary_service
[chr_start_cb] Included_service Characteristic started
[svc_start_cb] Primary_service started
[inc_start_cb] Service included successfully
[chr_start_cb] chr_read started
[chr_start_cb] chr_write_norsp started
[chr_start_cb] chr_write started
[chr_start_cb] chr_notify started
[desc_start_cb] desc_ccc_notify started
[chr_start_cb] chr_indicate started
[desc_start_cb] desc_ccc_indicate started
[chr_start_cb] chr_auth_write started
[chr_start_cb] chr_all started
[desc_start_cb] desc_all_aggr_format started
[desc_start_cb] desc_all_chr_format started
[desc_start_cb] desc_all_scc started
[desc_start_cb] desc_all_ccc started
[desc_start_cb] desc_all_usr_desc started

[example_bt_gatt_server] notify_task running...
[example_bt_gatt_server] Start broadcasting ADV packets
```

6.2 Service

GATT defines two types of services: primary and secondary. A primary service is a service that provides the primary functionality of a device. A secondary service is used to be included only in other primary services as modifier, which has no real meaning on its own. This example shows how to include a secondary service into a primary service:

1. Create both primary and secondary service.
2. Start secondary service.
3. After secondary service is started (event BT_GATTS_EV_SERVICE_START), include it into primary service.
4. Start primary service.

Follow the steps above, the secondary service can be included into primary service successfully.

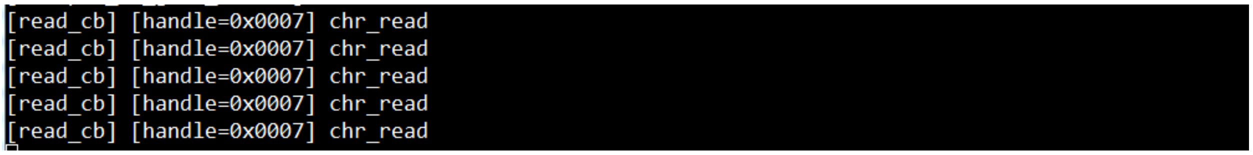
6.3 Characteristic

The following section will introduce each characteristic corresponding to characteristic property.

6.3.1 Read

In the GATT server example, characteristic UUID 0xFF02 is configured with characteristic property read. GATT client can read data from GATT server if the characteristic property BT_GATT_PROP_READ is configured. When client sends a Read Request to server, server reports an event BT_GATTS_EV_READ to the registered callback function. User can determine the value in the Read Response by using the API below:

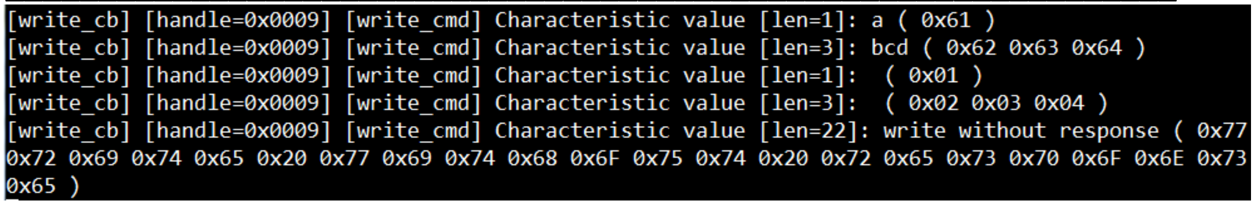
void BT_GATTS_ReadRsp(tBT_GattConnIF conn, tBT_GattResEnum res, tUint8 *val, tUint16 val_len);



```
[read_cb] [handle=0x0007] chr_read
[read_cb] [handle=0x0007] chr_read
[read_cb] [handle=0x0007] chr_read
[read_cb] [handle=0x0007] chr_read
[read_cb] [handle=0x0007] chr_read
```

6.3.2 Write Without Response

In the GATT server example, characteristic UUID 0xFF03 is configured with characteristic property write without response. GATT client can write data to GATT server without the need of server’s response if the characteristic property BT_GATT_PROP_WRITE_CMD is configured. When client writes data to server, server reports an event BT_GATTS_EV_WRITE to the registered callback function. User can get data from the callback function parameter.

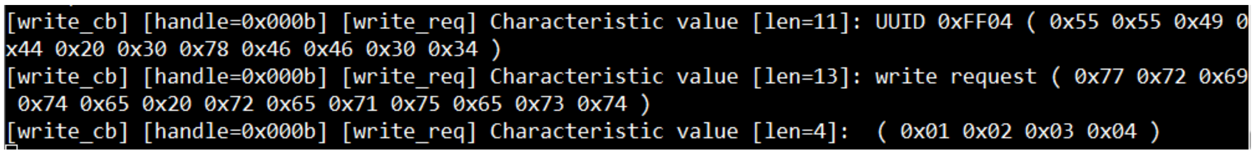


```
[write_cb] [handle=0x0009] [write_cmd] Characteristic value [len=1]: a ( 0x61 )
[write_cb] [handle=0x0009] [write_cmd] Characteristic value [len=3]: bcd ( 0x62 0x63 0x64 )
[write_cb] [handle=0x0009] [write_cmd] Characteristic value [len=1]:  ( 0x01 )
[write_cb] [handle=0x0009] [write_cmd] Characteristic value [len=3]:  ( 0x02 0x03 0x04 )
[write_cb] [handle=0x0009] [write_cmd] Characteristic value [len=22]: write without response ( 0x77
0x72 0x69 0x74 0x65 0x20 0x77 0x69 0x74 0x68 0x6F 0x75 0x74 0x20 0x72 0x65 0x73 0x70 0x6F 0x6E 0x73
0x65 )
```

6.3.3 Write

In the GATT server example, characteristic UUID 0xFF04 is configured with characteristic property write. GATT client can write data to GATT server if the characteristic property BT_GATT_PROP_WRITE_REQ is configured. GATT client sends Write Request with data to GATT server, and server sends back a write response to client. When client writes data to server, server reports an event BT_GATTS_EV_WRITE_REQ to the registered callback function. User can get data from the callback function parameter. User must use the API below to send write response to the client:

void BT_GATTS_WriteRsp(tBT_GattConnIF conn, tBT_GattResEnum res);

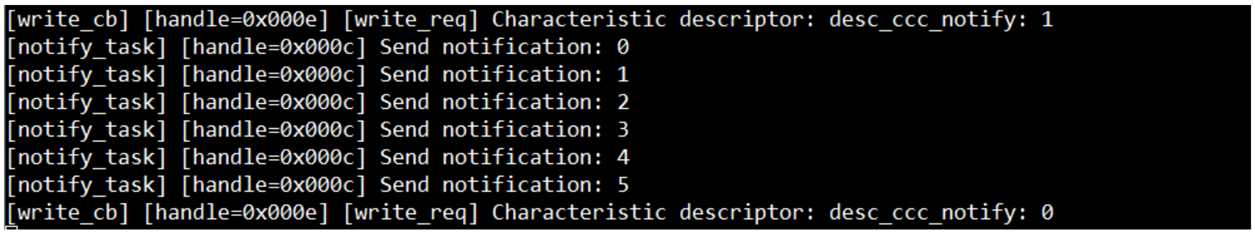


```
[write_cb] [handle=0x000b] [write_req] Characteristic value [len=11]: UUID 0xFF04 ( 0x55 0x55 0x49 0
x44 0x20 0x30 0x78 0x46 0x46 0x30 0x34 )
[write_cb] [handle=0x000b] [write_req] Characteristic value [len=13]: write request ( 0x77 0x72 0x69
 0x74 0x65 0x20 0x72 0x65 0x71 0x75 0x65 0x73 0x74 )
[write_cb] [handle=0x000b] [write_req] Characteristic value [len=4]:  ( 0x01 0x02 0x03 0x04 )
```

6.3.4 Notify

In the GATT server example, characteristic UUID 0xFF05 is configured with characteristic property notify. GATT server can notify data to GATT client if the characteristic property BT_GATT_PROP_NOTIFY is configured. GATT server can notify data to GATT client when the client characteristic configuration descriptor (CCCD) is on. In this example, **notify_task** keeps sending notifications to client with increasing number when the CCCD is set to 1. User can send notifications by the API below:

void BT_GATTS_ChrValueSend(tBT_GattConnIF conn, tBT_GattChrIF chr, tUint8 *val, tUint16 val_len);



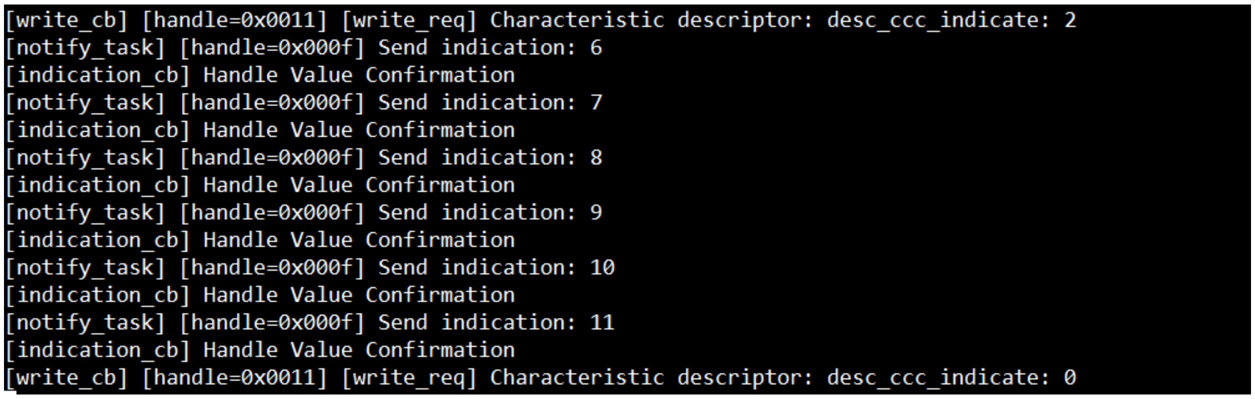
```
[write_cb] [handle=0x000e] [write_req] Characteristic descriptor: desc_ccc_notify: 1
[notify_task] [handle=0x000c] Send notification: 0
[notify_task] [handle=0x000c] Send notification: 1
[notify_task] [handle=0x000c] Send notification: 2
[notify_task] [handle=0x000c] Send notification: 3
[notify_task] [handle=0x000c] Send notification: 4
[notify_task] [handle=0x000c] Send notification: 5
[write_cb] [handle=0x000e] [write_req] Characteristic descriptor: desc_ccc_notify: 0
```

6.3.5 Indicate

In the GATT server example, characteristic UUID 0xFF06 is configured with characteristic property indicate. GATT server can send indication to GATT client if the characteristic property

BT_GATT_PROP_INDICATE is configured. GATT server can send indication to GATT client when the client characteristic configuration descriptor (CCCD) is set to 2, and client sends back a handle value confirmation to server. In this example, **notify_task** keeps sending indications to client with increasing number when the CCCD is on. When client receives an indication, it sends handle value confirmation to server, and then server reports an event BT_GATTS_EV_HANDLE_VALUE_INDICATION_CFM to the registered callback function. User can send indications by the API below:

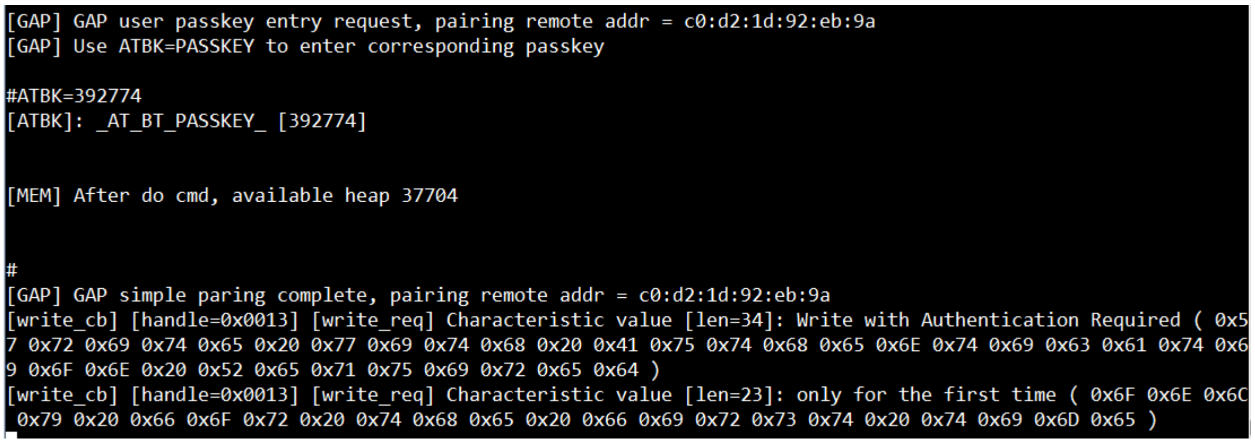
void BT_GATTS_ChrValueSend(tBT_GattConnIF conn, tBT_GattChrIF chr, tUint8 *val, tUint16 val_len);



```
[write_cb] [handle=0x0011] [write_req] Characteristic descriptor: desc_ccc_indicate: 2
[notify_task] [handle=0x000f] Send indication: 6
[indication_cb] Handle Value Confirmation
[notify_task] [handle=0x000f] Send indication: 7
[indication_cb] Handle Value Confirmation
[notify_task] [handle=0x000f] Send indication: 8
[indication_cb] Handle Value Confirmation
[notify_task] [handle=0x000f] Send indication: 9
[indication_cb] Handle Value Confirmation
[notify_task] [handle=0x000f] Send indication: 10
[indication_cb] Handle Value Confirmation
[notify_task] [handle=0x000f] Send indication: 11
[indication_cb] Handle Value Confirmation
[write_cb] [handle=0x0011] [write_req] Characteristic descriptor: desc_ccc_indicate: 0
```

6.3.6 Write with Authentication Required

In the GATT server example, characteristic UUID 0xFF07 is configured with characteristic property write. If ATT permission is set to BT_GATT_PERM_WRITE_AUTHEN_REQUEIRED, GATT client has to pass authentication process before sending data to GATT server for the first time. In this example, passkey is shown on client, and server needs to use **ATBK=PASSKEY** to enter the key which is shown on client. This authentication is required for the first time of sending write request to server.



```
[GAP] GAP user passkey entry request, pairing remote addr = c0:d2:1d:92:eb:9a
[GAP] Use ATBK=PASSKEY to enter corresponding passkey

#ATBK=392774
[ATBK]: _AT_BT_PASSKEY_ [392774]


[MEM] After do cmd, available heap 37704


#
[GAP] GAP simple paring complete, pairing remote addr = c0:d2:1d:92:eb:9a
[write_cb] [handle=0x0013] [write_req] Characteristic value [len=34]: Write with Authentication Required ( 0x5
7 0x72 0x69 0x74 0x65 0x20 0x77 0x69 0x74 0x68 0x20 0x41 0x75 0x74 0x68 0x65 0x6E 0x74 0x69 0x63 0x61 0x74 0x6
9 0x6F 0x6E 0x20 0x52 0x65 0x71 0x75 0x69 0x72 0x65 0x64 )
[write_cb] [handle=0x0013] [write_req] Characteristic value [len=23]: only for the first time ( 0x6F 0x6E 0x6C
 0x79 0x20 0x66 0x6F 0x72 0x20 0x74 0x68 0x65 0x20 0x66 0x69 0x72 0x73 0x74 0x20 0x74 0x69 0x6D 0x65 )
```

6.3.7 All Characteristic Properties ON

In the GATT server example, characteristic UUID 0xFF08 is configured with all characteristic properties on and descriptors defined by Bluetooth Specification. User can use all the functions mentioned above on this characteristic.

7 GATT Client

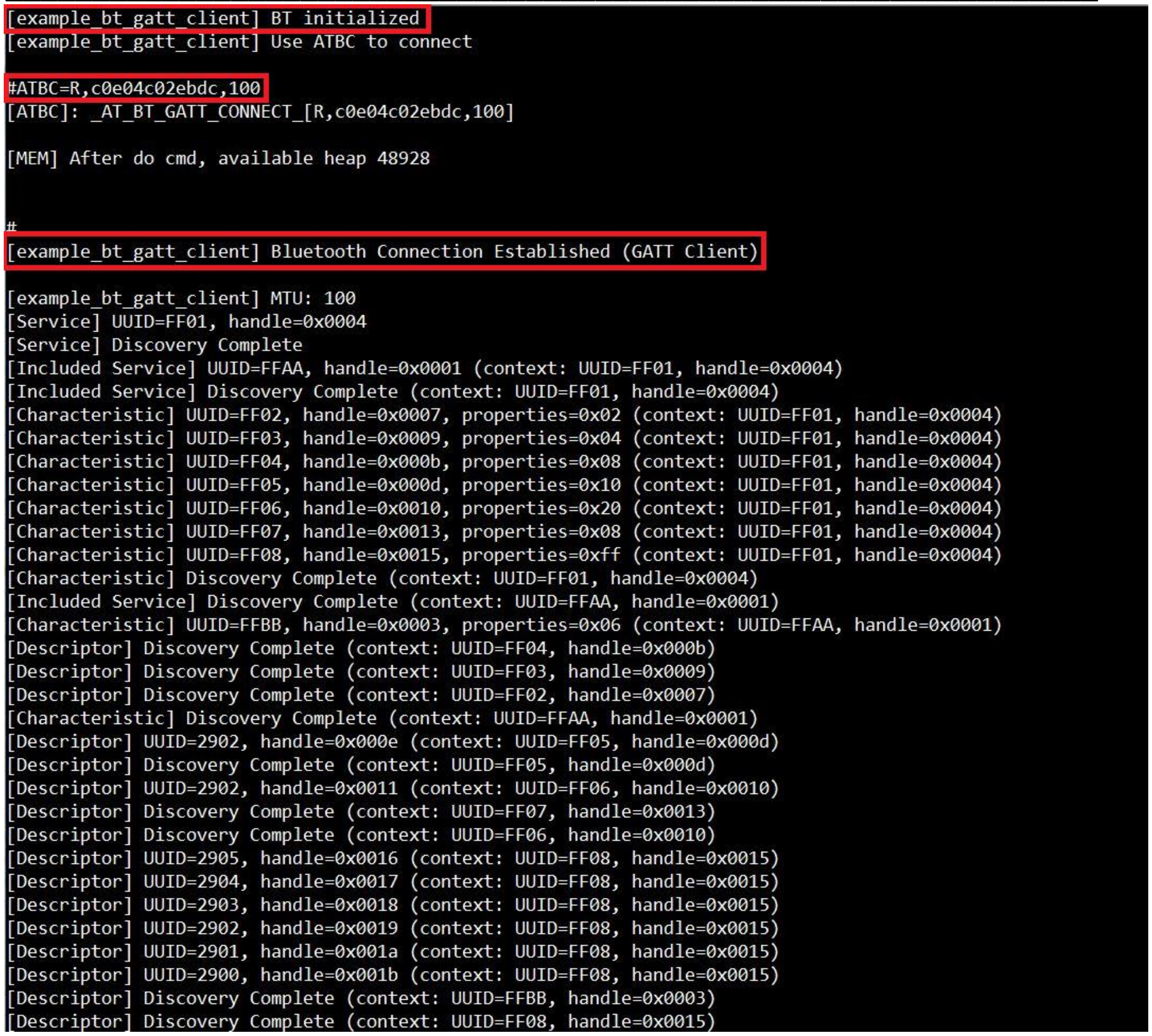
7.1 How to Run the Example

To run the example of bt_gatt_client, set CONFIG_BT and CONFIG_EXAMPLE_BT_GATT_CLIENT flag in \project\realtek_ameba1_va0_example\inc\platform_opts.h

```
//Config in platform_opts.h
#define CONFIG_BT      1
…
#define CONFIG_EXAMPLE_BT_GATT_CLIENT                    1
```

Once see the message **[example_bt_gatt_client] BT initialized**, it means that the example is ready. Users can use AT Command **ATBC** to connect to target device. In this example, it connects to another Ameba which running example bt_gatt_server (**GATT Server** in this document). After connecting to GATT Server, it will do Primary Service Discovery, Relationship Discovery, Characteristic Discovery and Characteristic Descriptor Discovery.

ATBC=P/R,addr[,mtu]   (P: public address; R: random address; default mtu: 256)



```
[example_bt_gatt_client] BT initialized
[example_bt_gatt_client] Use ATBC to connect

#ATBC=R,c0e04c02ebdc,100
[ATBC]: _AT_BT_GATT_CONNECT_[R,c0e04c02ebdc,100]

[MEM] After do cmd, available heap 48928


#
[example_bt_gatt_client] Bluetooth Connection Established (GATT Client)

[example_bt_gatt_client] MTU: 100
[Service] UUID=FF01, handle=0x0004
[Service] Discovery Complete
[Included Service] UUID=FFAA, handle=0x0001 (context: UUID=FF01, handle=0x0004)
[Included Service] Discovery Complete (context: UUID=FF01, handle=0x0004)
[Characteristic] UUID=FF02, handle=0x0007, properties=0x02 (context: UUID=FF01, handle=0x0004)
[Characteristic] UUID=FF03, handle=0x0009, properties=0x04 (context: UUID=FF01, handle=0x0004)
[Characteristic] UUID=FF04, handle=0x000b, properties=0x08 (context: UUID=FF01, handle=0x0004)
[Characteristic] UUID=FF05, handle=0x000d, properties=0x10 (context: UUID=FF01, handle=0x0004)
[Characteristic] UUID=FF06, handle=0x0010, properties=0x20 (context: UUID=FF01, handle=0x0004)
[Characteristic] UUID=FF07, handle=0x0013, properties=0x08 (context: UUID=FF01, handle=0x0004)
[Characteristic] UUID=FF08, handle=0x0015, properties=0xff (context: UUID=FF01, handle=0x0004)
[Characteristic] Discovery Complete (context: UUID=FF01, handle=0x0004)
[Included Service] Discovery Complete (context: UUID=FFAA, handle=0x0001)
[Characteristic] UUID=FFBB, handle=0x0003, properties=0x06 (context: UUID=FFAA, handle=0x0001)
[Descriptor] Discovery Complete (context: UUID=FF04, handle=0x000b)
[Descriptor] Discovery Complete (context: UUID=FF03, handle=0x0009)
[Descriptor] Discovery Complete (context: UUID=FF02, handle=0x0007)
[Characteristic] Discovery Complete (context: UUID=FFAA, handle=0x0001)
[Descriptor] UUID=2902, handle=0x000e (context: UUID=FF05, handle=0x000d)
[Descriptor] Discovery Complete (context: UUID=FF05, handle=0x000d)
[Descriptor] UUID=2902, handle=0x0011 (context: UUID=FF06, handle=0x0010)
[Descriptor] Discovery Complete (context: UUID=FF07, handle=0x0013)
[Descriptor] Discovery Complete (context: UUID=FF06, handle=0x0010)
[Descriptor] UUID=2905, handle=0x0016 (context: UUID=FF08, handle=0x0015)
[Descriptor] UUID=2904, handle=0x0017 (context: UUID=FF08, handle=0x0015)
[Descriptor] UUID=2903, handle=0x0018 (context: UUID=FF08, handle=0x0015)
[Descriptor] UUID=2902, handle=0x0019 (context: UUID=FF08, handle=0x0015)
[Descriptor] UUID=2901, handle=0x001a (context: UUID=FF08, handle=0x0015)
[Descriptor] UUID=2900, handle=0x001b (context: UUID=FF08, handle=0x0015)
[Descriptor] Discovery Complete (context: UUID=FFBB, handle=0x0003)
[Descriptor] Discovery Complete (context: UUID=FF08, handle=0x0015)
```

7.2 Characteristic Properties

The Characteristic Properties bit field determines how the Characteristic Value can be used, or how the characteristic descriptors can be accessed. Multiple Characteristic Properties can be set.

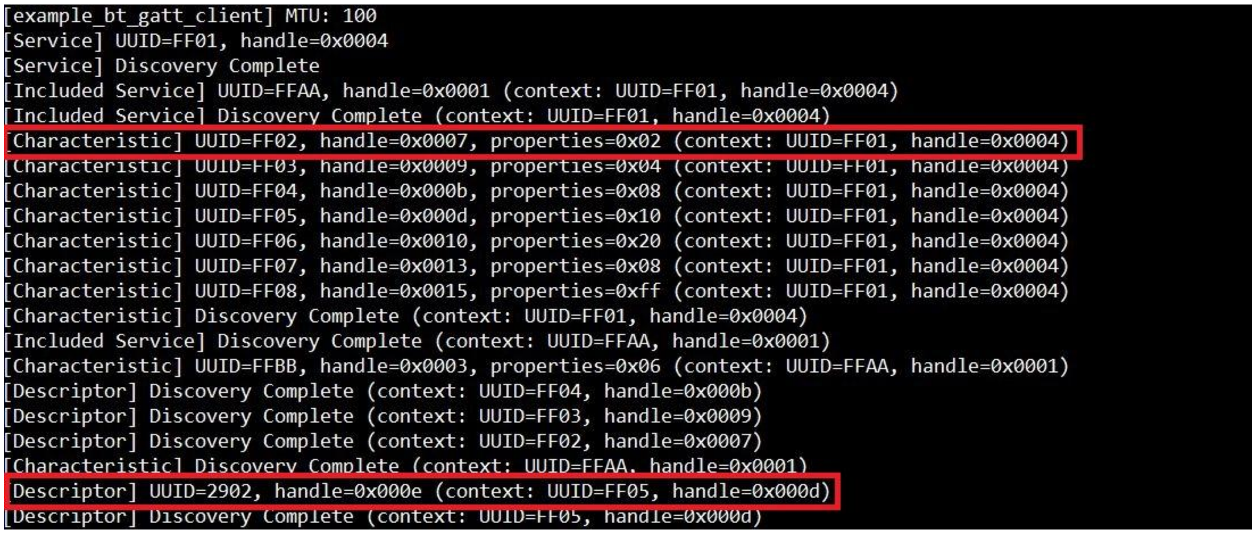
Property	Value	Description
Read	0x02	If set, allows clients to read this characteristic.
Write Without Response	0x04	If set, allows clients to write command to this characteristic.
Write	0x08	If set, allows clients to send write request to this characteristic (Write response is required).
Notify	0x10	If set, permits notifications of a Characteristic Value without acknowledgement. If set, the Client Characteristic Configuration

		Descriptor shall exist.
Indicate	0x20	If set, permits indications of a Characteristic Value with acknowledgement. If set, the Client Characteristic Configuration Descriptor shall exist.
Authenticated Signed Writes	0x40	If set, permits signed writes to the Characteristic Value.
Extended Properties	0x80	If set, additional characteristic properties are defined in the Characteristic Extended Properties Descriptor

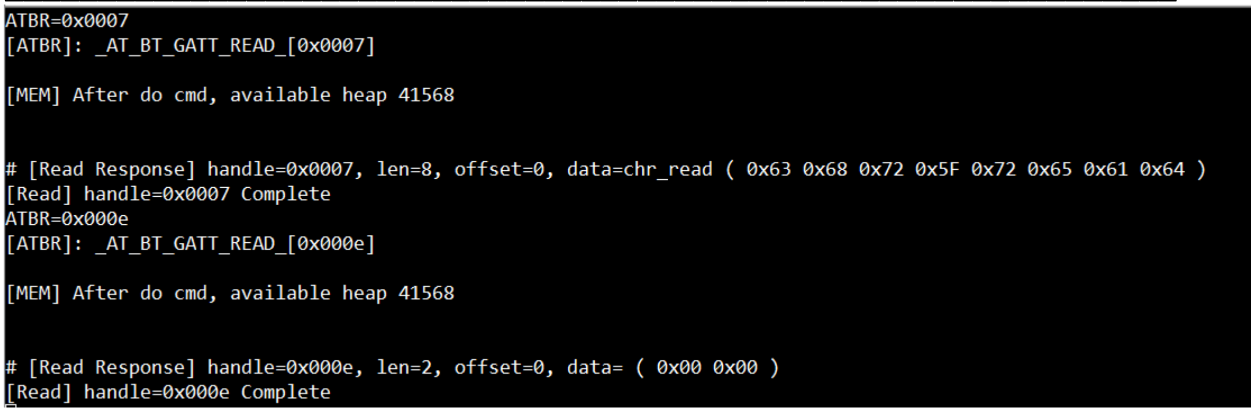
7.3 Read Characteristic/Descriptor

User can use AT command **ATBR=handle** to read characteristic value/descriptor.

In this example, from the service discovery result, we can see that the characteristic value handle of UUID FF02 is 0x0007, and the characteristic property bit Read (0x02) is set. The descriptor handle of UUID 2902 is 0x000e, and from the context, we can see that it is characteristic UUID FF05's CCCD (Client Characteristic Configuration Descriptor, UUID 2902). We can use **ATBR=handle** to read them.



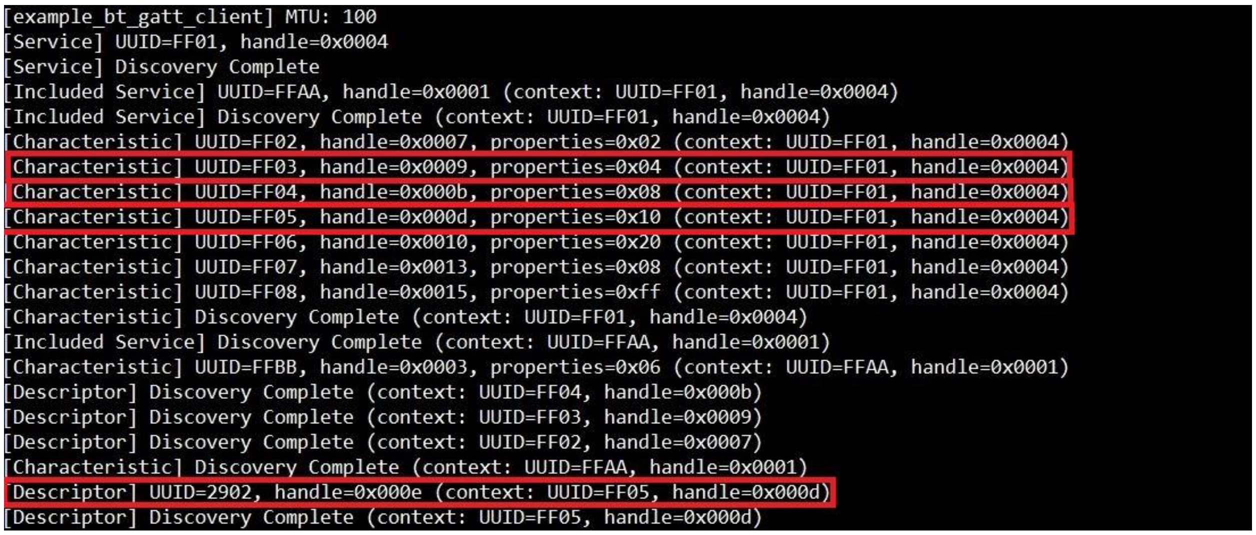
```
[example_bt_gatt_client] MTU: 100
[Service] UUID=FF01, handle=0x0004
[Service] Discovery Complete
[Included Service] UUID=FFAA, handle=0x0001 (context: UUID=FF01, handle=0x0004)
[Included Service] Discovery Complete (context: UUID=FF01, handle=0x0004)
[Characteristic] UUID=FF02, handle=0x0007, properties=0x02 (context: UUID=FF01, handle=0x0004)
[Characteristic] UUID=FF03, handle=0x0009, properties=0x04 (context: UUID=FF01, handle=0x0004)
[Characteristic] UUID=FF04, handle=0x000b, properties=0x08 (context: UUID=FF01, handle=0x0004)
[Characteristic] UUID=FF05, handle=0x000d, properties=0x10 (context: UUID=FF01, handle=0x0004)
[Characteristic] UUID=FF06, handle=0x0010, properties=0x20 (context: UUID=FF01, handle=0x0004)
[Characteristic] UUID=FF07, handle=0x0013, properties=0x08 (context: UUID=FF01, handle=0x0004)
[Characteristic] UUID=FF08, handle=0x0015, properties=0xff (context: UUID=FF01, handle=0x0004)
[Characteristic] Discovery Complete (context: UUID=FF01, handle=0x0004)
[Included Service] Discovery Complete (context: UUID=FFAA, handle=0x0001)
[Characteristic] UUID=FFBB, handle=0x0003, properties=0x06 (context: UUID=FFAA, handle=0x0001)
[Descriptor] Discovery Complete (context: UUID=FF04, handle=0x000b)
[Descriptor] Discovery Complete (context: UUID=FF03, handle=0x0009)
[Descriptor] Discovery Complete (context: UUID=FF02, handle=0x0007)
[Characteristic] Discovery Complete (context: UUID=FFAA, handle=0x0001)
[Descriptor] UUID=2902, handle=0x000e (context: UUID=FF05, handle=0x000d)
[Descriptor] Discovery Complete (context: UUID=FF05, handle=0x000d)
```



```
ATBR=0x0007
[ATBR]: _AT_BT_GATT_READ_[0x0007]

[MEM] After do cmd, available heap 41568


# [Read Response] handle=0x0007, len=8, offset=0, data=chr_read ( 0x63 0x68 0x72 0x5F 0x72 0x65 0x61 0x64 )
[Read] handle=0x0007 Complete
ATBR=0x000e
[ATBR]: _AT_BT_GATT_READ_[0x000e]

[MEM] After do cmd, available heap 41568


# [Read Response] handle=0x000e, len=2, offset=0, data= ( 0x00 0x00 )
[Read] handle=0x000e Complete
```

7.4 Write Characteristic/Descriptor

User can use AT command **ATBW=handle,req/cmd,data** to write characteristic value/descriptor.

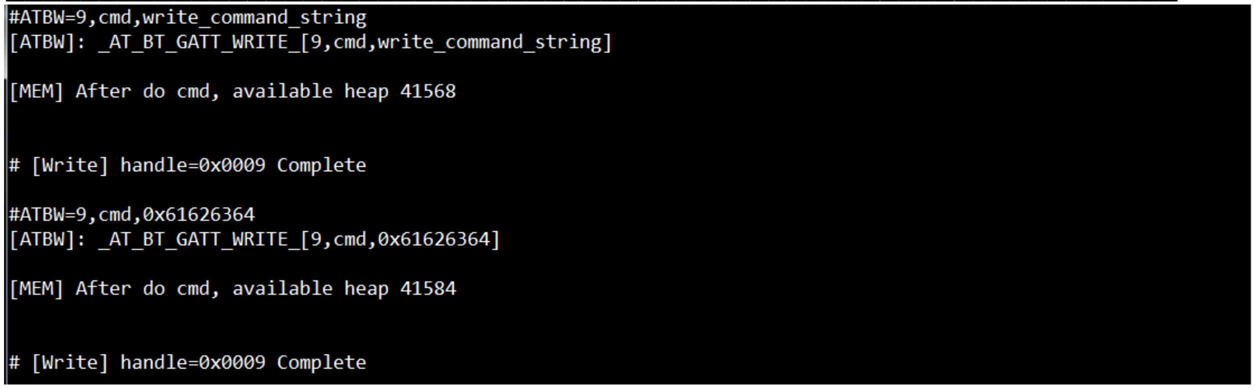


```
[example_bt_gatt_client] MTU: 100
[Service] UUID=FF01, handle=0x0004
[Service] Discovery Complete
[Included Service] UUID=FFAA, handle=0x0001 (context: UUID=FF01, handle=0x0004)
[Included Service] Discovery Complete (context: UUID=FF01, handle=0x0004)
[Characteristic] UUID=FF02, handle=0x0007, properties=0x02 (context: UUID=FF01, handle=0x0004)
[Characteristic] UUID=FF03, handle=0x0009, properties=0x04 (context: UUID=FF01, handle=0x0004)
[Characteristic] UUID=FF04, handle=0x000b, properties=0x08 (context: UUID=FF01, handle=0x0004)
[Characteristic] UUID=FF05, handle=0x000d, properties=0x10 (context: UUID=FF01, handle=0x0004)
[Characteristic] UUID=FF06, handle=0x0010, properties=0x20 (context: UUID=FF01, handle=0x0004)
[Characteristic] UUID=FF07, handle=0x0013, properties=0x08 (context: UUID=FF01, handle=0x0004)
[Characteristic] UUID=FF08, handle=0x0015, properties=0xff (context: UUID=FF01, handle=0x0004)
[Characteristic] Discovery Complete (context: UUID=FF01, handle=0x0004)
[Included Service] Discovery Complete (context: UUID=FFAA, handle=0x0001)
[Characteristic] UUID=FFBB, handle=0x0003, properties=0x06 (context: UUID=FFAA, handle=0x0001)
[Descriptor] Discovery Complete (context: UUID=FF04, handle=0x000b)
[Descriptor] Discovery Complete (context: UUID=FF03, handle=0x0009)
[Descriptor] Discovery Complete (context: UUID=FF02, handle=0x0007)
[Characteristic] Discovery Complete (context: UUID=FFAA, handle=0x0001)
[Descriptor] UUID=2902, handle=0x000e (context: UUID=FF05, handle=0x000d)
[Descriptor] Discovery Complete (context: UUID=FF05, handle=0x000d)
```

7.4.1 Write Command

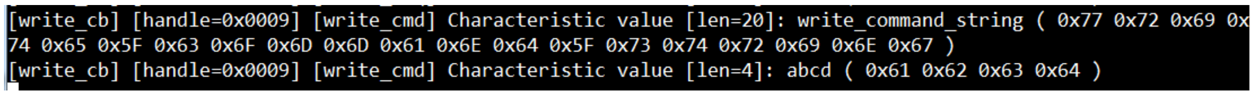
In this example, from the service discovery result, we can see that the characteristic value handle of UUID FF03 is 0x0009, and the characteristic property bit Write Without Response (0x04) is set. Use **ATBW=handle,cmd,data** to write data.

On the GATT Client side:



```
#ATBW=9,cmd,write_command_string
[ATBW]: _AT_BT_GATT_WRITE_[9,cmd,write_command_string]

[MEM] After do cmd, available heap 41568


# [Write] handle=0x0009 Complete

#ATBW=9,cmd,0x61626364
[ATBW]: _AT_BT_GATT_WRITE_[9,cmd,0x61626364]

[MEM] After do cmd, available heap 41584


# [Write] handle=0x0009 Complete
```

On the GATT Server side:

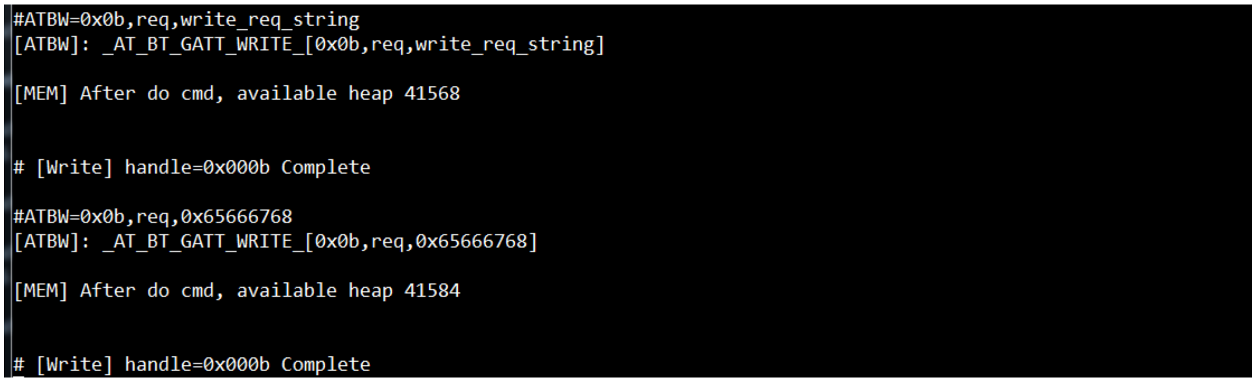


```
[write_cb] [handle=0x0009] [write_cmd] Characteristic value [len=20]: write_command_string ( 0x77 0x72 0x69 0x
74 0x65 0x5F 0x63 0x6F 0x6D 0x6D 0x61 0x6E 0x64 0x5F 0x73 0x74 0x72 0x69 0x6E 0x67 )
[write_cb] [handle=0x0009] [write_cmd] Characteristic value [len=4]: abcd ( 0x61 0x62 0x63 0x64 )
```

7.4.2 Write Request

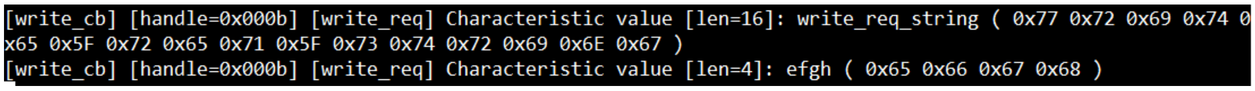
In this example, from the service discovery result, we can see that the characteristic value handle of UUID FF04 is 0x000b, and the characteristic property bit Write (0x08) is set. Use **ATBW=handle,req,data** to write data.

On the GATT Client side:



```
#ATBW=0x0b,req,write_req_string
[ATBW]: _AT_BT_GATT_WRITE_[0x0b,req,write_req_string]

[MEM] After do cmd, available heap 41568


# [Write] handle=0x000b Complete

#ATBW=0x0b,req,0x65666768
[ATBW]: _AT_BT_GATT_WRITE_[0x0b,req,0x65666768]

[MEM] After do cmd, available heap 41584


# [Write] handle=0x000b Complete
```

On the GATT Server side:



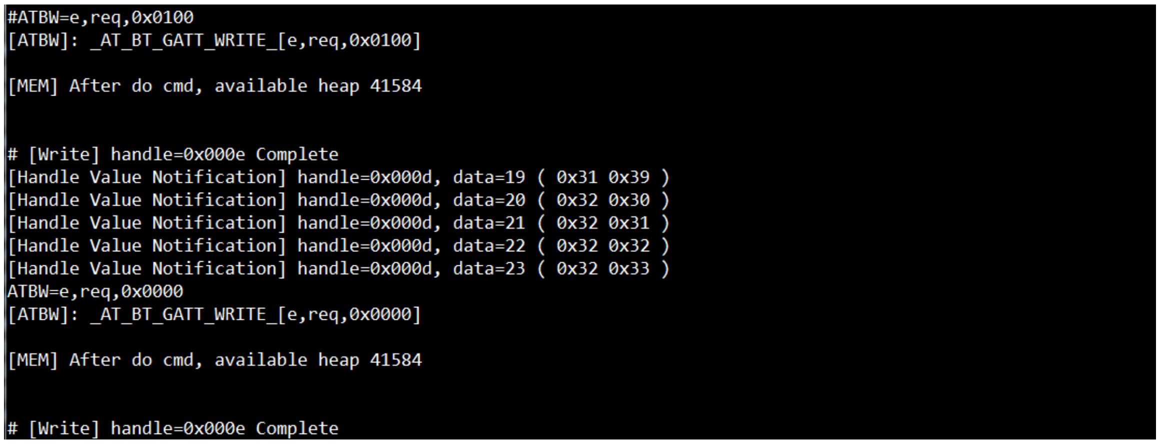
```
[write_cb] [handle=0x000b] [write_req] Characteristic value [len=16]: write_req_string ( 0x77 0x72 0x69 0x74 0
x65 0x5F 0x72 0x65 0x71 0x5F 0x73 0x74 0x72 0x69 0x6E 0x67 )
[write_cb] [handle=0x000b] [write_req] Characteristic value [len=4]: efgh ( 0x65 0x66 0x67 0x68 )
```

7.4.3 Write Descriptor to enable Notification/Indication

In this example, from the service discovery result, we can see that the characteristic value handle of UUID FF05 is 0x000b, and the characteristic property bit Notify (0x10) is set. Its CCCD

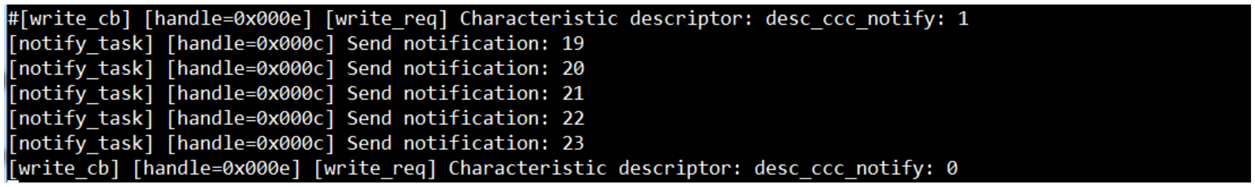
(Client Characteristic Configuration Descriptor, UUID 2902) handle is 0x000e. To enable notification, user needs to send write request to the handle of CCCD. Value 0x0001 is for Notify, and value 0x0002 is for Indicate. In this case, we set the CCCD to 0x0001 (0x0100: little endian) to enable notification, and 0x0000 to disable. Once the CCCD is set to 0x0001, GATT Client can receive notification data from GATT Server.

On the GATT Client side:



```
#ATBW=e,req,0x0100
[ATBW]: _AT_BT_GATT_WRITE_[e,req,0x0100]

[MEM] After do cmd, available heap 41584


# [Write] handle=0x000e Complete
[Handle Value Notification] handle=0x000d, data=19 ( 0x31 0x39 )
[Handle Value Notification] handle=0x000d, data=20 ( 0x32 0x30 )
[Handle Value Notification] handle=0x000d, data=21 ( 0x32 0x31 )
[Handle Value Notification] handle=0x000d, data=22 ( 0x32 0x32 )
[Handle Value Notification] handle=0x000d, data=23 ( 0x32 0x33 )
ATBW=e,req,0x0000
[ATBW]: _AT_BT_GATT_WRITE_[e,req,0x0000]

[MEM] After do cmd, available heap 41584


# [Write] handle=0x000e Complete
```

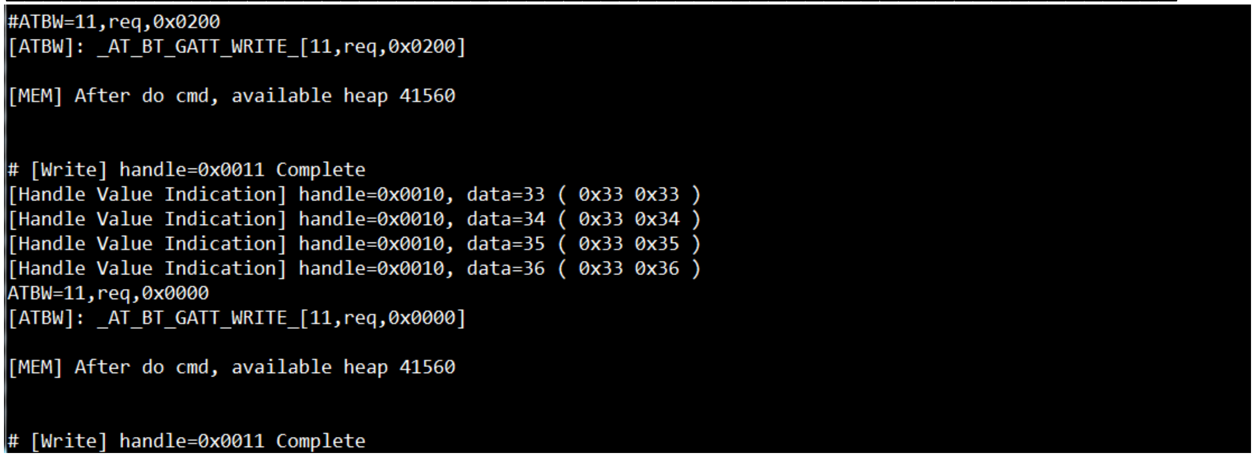
On the GATT Server side:



```
#[write_cb] [handle=0x000e] [write_req] Characteristic descriptor: desc_ccc_notify: 1
[notify_task] [handle=0x000c] Send notification: 19
[notify_task] [handle=0x000c] Send notification: 20
[notify_task] [handle=0x000c] Send notification: 21
[notify_task] [handle=0x000c] Send notification: 22
[notify_task] [handle=0x000c] Send notification: 23
[write_cb] [handle=0x000e] [write_req] Characteristic descriptor: desc_ccc_notify: 0
```

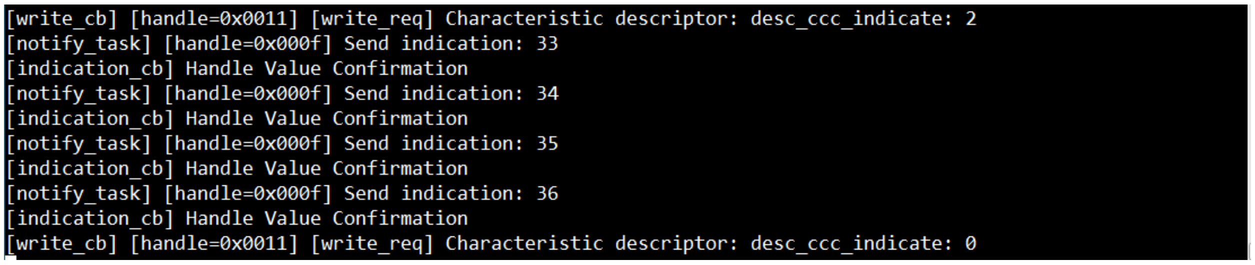
Characteristic value handle of UUID FF06 is 0x0010, and the characteristic property bit Indicate (0x20) is set. Its CCCD (Client Characteristic Configuration Descriptor, UUID 2902) handle is 0x0011. To enable indication, user needs to send write request to the handle of CCCD. Value 0x0002 is for Indicate. In this case, we set the CCCD to 0x0002 (0x0200: little endian) to enable indication, and 0x0000 to disable. Once the CCCD is set to 0x0002, GATT Client can receive indication data from GATT Server.

On the GATT Client side:



```
#ATBW=11,req,0x0200
[ATBW]: _AT_BT_GATT_WRITE_[11,req,0x0200]

[MEM] After do cmd, available heap 41560


# [Write] handle=0x0011 Complete
[Handle Value Indication] handle=0x0010, data=33 ( 0x33 0x33 )
[Handle Value Indication] handle=0x0010, data=34 ( 0x33 0x34 )
[Handle Value Indication] handle=0x0010, data=35 ( 0x33 0x35 )
[Handle Value Indication] handle=0x0010, data=36 ( 0x33 0x36 )
ATBW=11,req,0x0000
[ATBW]: _AT_BT_GATT_WRITE_[11,req,0x0000]

[MEM] After do cmd, available heap 41560


# [Write] handle=0x0011 Complete
```

On the GATT Server side:



```
[write_cb] [handle=0x0011] [write_req] Characteristic descriptor: desc_ccc_indicate: 2
[notify_task] [handle=0x000f] Send indication: 33
[indication_cb] Handle Value Confirmation
[notify_task] [handle=0x000f] Send indication: 34
[indication_cb] Handle Value Confirmation
[notify_task] [handle=0x000f] Send indication: 35
[indication_cb] Handle Value Confirmation
[notify_task] [handle=0x000f] Send indication: 36
[indication_cb] Handle Value Confirmation
[write_cb] [handle=0x0011] [write_req] Characteristic descriptor: desc_ccc_indicate: 0
```

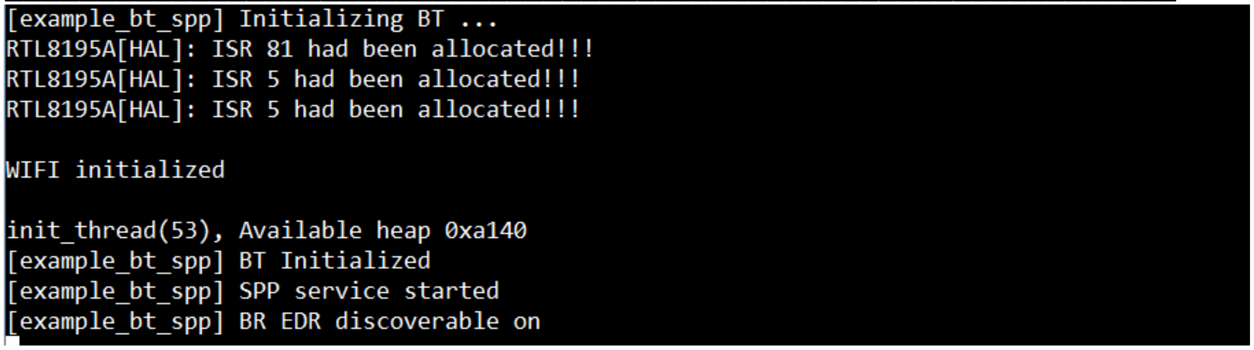
8 SPP (Serial Port Profile)

8.1 How to Run the Example

To run the example of bt_spp, set CONFIG_BT and CONFIG_EXAMPLE_BT_SPP flag in \project\realtek_ameba1_va0_example\inc\platform_opts.h

```
//Config in platform_opts.h
#define CONFIG_BT       1
…
#define CONFIG_EXAMPLE_BT_SPP          1
```

Once see the message **[example_bt_spp] SPP service started** and **[example_bt_spp] BR EDR discoverable on**, it means that the example is ready.

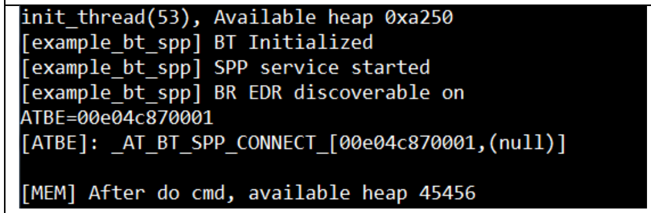
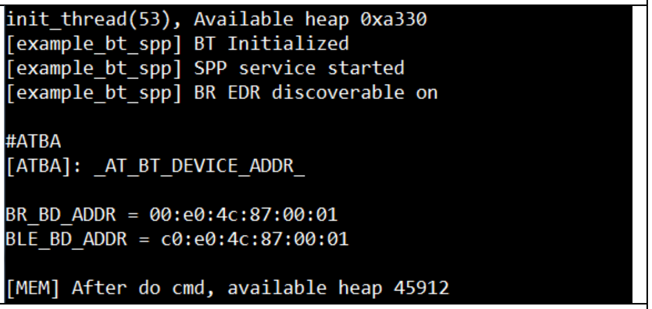


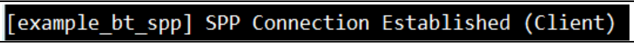
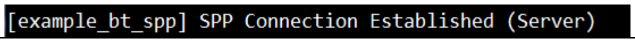
```
[example_bt_spp] Initializing BT ...
RTL8195A[HAL]: ISR 81 had been allocated!!!
RTL8195A[HAL]: ISR 5 had been allocated!!!
RTL8195A[HAL]: ISR 5 had been allocated!!!

WIFI initialized

init_thread(53), Available heap 0xa140
[example_bt_spp] BT Initialized
[example_bt_spp] SPP service started
[example_bt_spp] BR EDR discoverable on
```

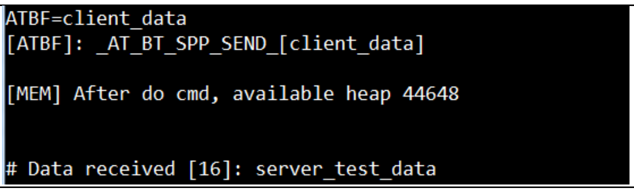
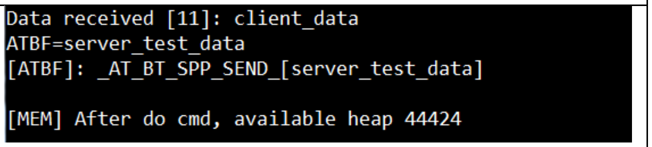
User can use two Ameba running example_bt_spp to test it.

Let one Ameba be server, and use AT command **ATBE=BR_BD_ADDR** on another Ameba to connect it.

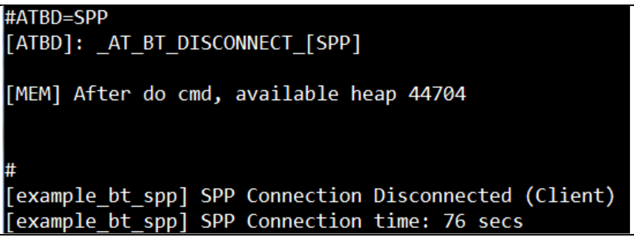
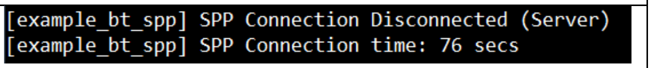
Client	**Server**
Connect to Server: **ATBE**	Get BT address: **ATBA**

 `init_thread(53), Available heap 0xa250`
`[example_bt_spp] BT Initialized`
`[example_bt_spp] SPP service started`
`[example_bt_spp] BR EDR discoverable on`
`ATBE=00e04c870001`
`[ATBE]: _AT_BT_SPP_CONNECT_[00e04c870001,(null)]`
 	
`[MEM] After do cmd, available heap 45456`
	 `init_thread(53), Available heap 0xa330`
	`[example_bt_spp] BT Initialized`
	`[example_bt_spp] SPP service started`
	`[example_bt_spp] BR EDR discoverable on`
	
	`#ATBA`
	`[ATBA]: _AT_BT_DEVICE_ADDR_`
	
	`BR_BD_ADDR = 00:e0:4c:87:00:01`
	`BLE_BD_ADDR = c0:e0:4c:87:00:01`
	
	`[MEM] After do cmd, available heap 45912`

Connection Established	
 `[example_bt_spp] SPP Connection Established (Client)`	 `[example_bt_spp] SPP Connection Established (Server)`
Send Data: **ATBF**	

 `ATBF=client_data`
`[ATBF]: _AT_BT_SPP_SEND_[client_data]`
 	
`[MEM] After do cmd, available heap 44648`
 	
`# Data received [16]: server_test_data`
	 `Data received [11]: client_data`
	`ATBF=server_test_data`
	`[ATBF]: _AT_BT_SPP_SEND_[server_test_data]`
	
	`[MEM] After do cmd, available heap 44424`

Disconnect: **ATBD=SPP**	

 `#ATBD=SPP`
`[ATBD]: _AT_BT_DISCONNECT_[SPP]`
 	
`[MEM] After do cmd, available heap 44704`
 	
`#`
`[example_bt_spp] SPP Connection Disconnected (Client)`
`[example_bt_spp] SPP Connection time: 76 secs`
	 `[example_bt_spp] SPP Connection Disconnected (Server)`
	`[example_bt_spp] SPP Connection time: 76 secs`