AN0077

Application Note

BLE Application Note

Introduction

This application note introduces how to use AT32WB415 wireless Bluetooth module to customize BLE-related functions, how to execute communication between wireless Bluetooth module and MCU, and how the MCU behaves after it receives a request from wireless Bluetooth module. In addition, this application note outlines AT command protocol, and introduces how to add custom services and characteristics to the Bluetooth module profile, as well as how to handle these demand commands from the wireless Bluetooth module on the MCU side.

In addition, this document also introduces how to control wireless Bluetooth module functions through AT command, allowing users to change the basic configurations on BLE side without modifying the code.

Applicable products:

Part number	AT32WB415xx

Contents

List of tables

List of figures

1 Introduction to Bluetooth

One key reason for the incredible success of Bluetooth® technology is the tremendous flexibility it provides developers. Offering two radio options, Bluetooth technology provides developers with a versatile set of full-stack, fit-for-purpose solutions to meet the ever-expanding needs for wireless connectivity.

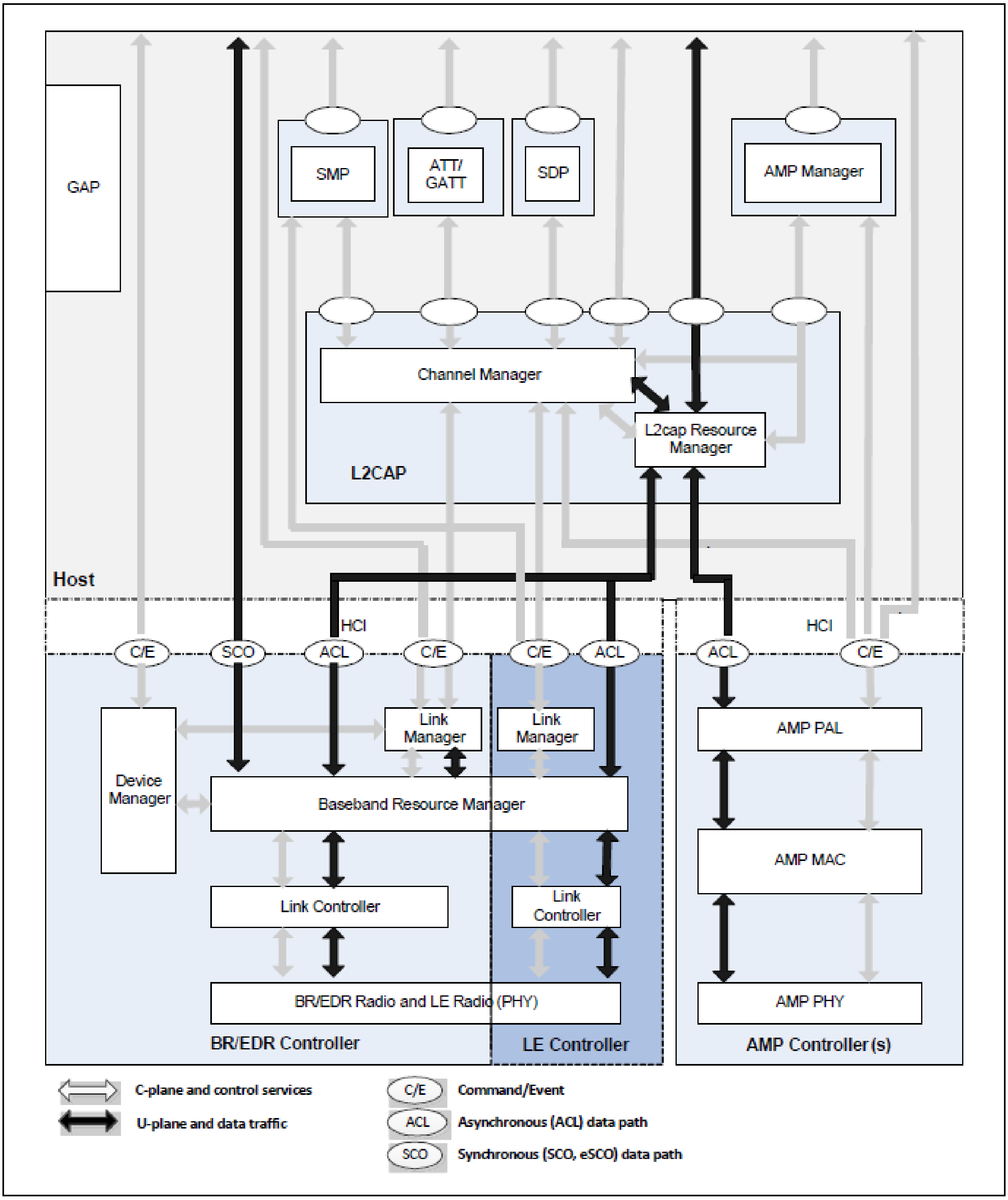
Whether a product streams high-quality audio between a smartphone and speaker, transfers data between a tablet and medical device, or sends messages between thousands of nodes in a building automation solution, the Bluetooth Low Energy (LE) and Bluetooth Classic radios are designed to meet the unique needs of developers worldwide.

This application note focuses on Bluetooth Low Energy (hereinafter referred to as BLE) rather than classic Bluetooth (hereinafter referred to as BR/EDR). For details about BR/EDR, please visit the official website of Bluetooth SIG.

The Bluetooth Low Energy (BLE) radio is designed for very low power operation. Transmitting data over 40 channels in the 2.4 GHz unlicensed ISM frequency band, the BLE radio provides developers a tremendous amount of flexibility to build products that meet the unique connectivity requirements of their market. BLE supports multiple communication topologies, expanding from point-to-point to broadcast and, most recently, mesh, enabling Bluetooth technology to support the creation of reliable, large-scale device networks. While initially known for its device communications capabilities, BLE is now also widely used as a device positioning technology to address the increasing demand for high accuracy indoor location services. BLE, which initially supports simple presence and proximity features, now also supports Bluetooth® direction finding and will soon support high-precision distance measurements.

The architecture of BLE is shown in Figure 1.

Figure 1. Bluetooth core system architecture



In this application, the modified parts of code are all in the Host block and only LE controller block is used, and the entire BLE system is implemented by the wireless Bluetooth module. The part that will actually be modified is GAP and GATT in the Host block. The following sections will introduce GAP and GATT and the influences of modifying the two small blocks.

1.1 Generic Access Profile (GAP)

GAP is an acronym for the Generic Access Profile, and it controls connections and advertising in Bluetooth. GAP is what makes your device visible to the outside world, and determines how two devices can (or cannot) interact with each other.

1.1.1 Device role

GAP defines various roles for devices, but the two key concepts to keep in mind are Central devices and Peripheral devices.

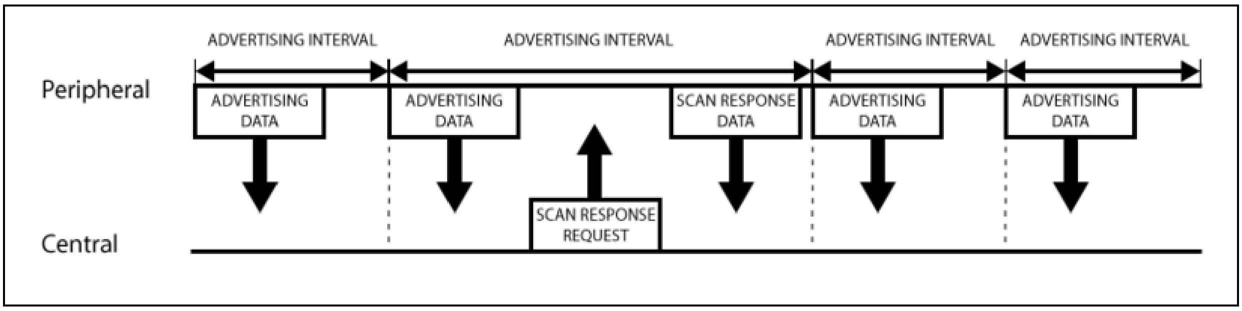
Peripheral devices are small, low power, resource constrained devices. Central devices are usually the mobile phone or tablet that you connect to with far more processing power and memory.

1.1.2 Advertising and scan response data

There are two ways to send advertising out with GAP, i.e., Advertising Data payload and Scan Response payload. Both payloads are identical and can contain up to 31 bytes of data, but only the advertising data payload is mandatory, since this is the payload that will be constantly transmitted out from the device to let central devices in range know that it exists.

The scan response payload is an optional secondary payload that central devices can request, and allows device designers to fit a bit more information in the advertising payload such a strings for a device name, etc.

Figure 2. Advertising and scan response



1.1.3 Broadcast network topology

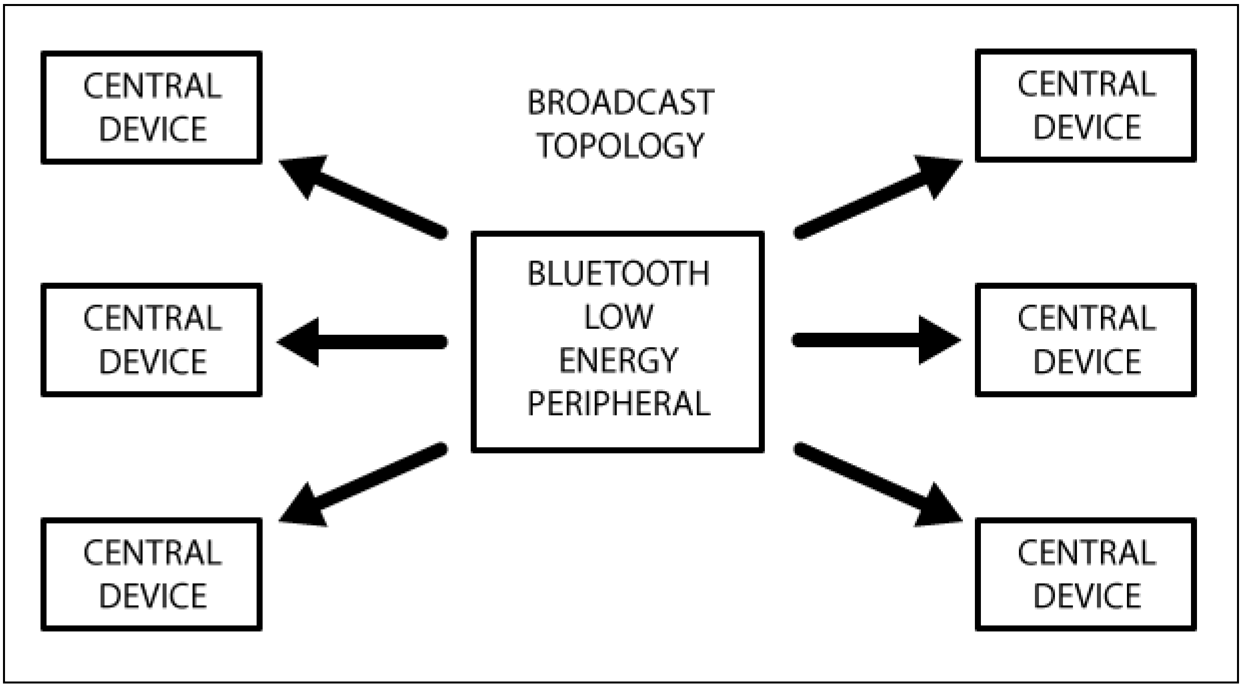
While most peripherals advertise themselves so that a connection can be established and GATT services and characteristics can be used (which allows for much more data to be exchanged in both directions), there are situations where you only want to advertise data.

The main use case here is where you want a peripheral to send data to more than one device at a time. This is only possible using the advertising packet since data sent and received in connected mode can only be seen by those two connected devices.

By including a small amount of custom data in the 31 byte advertising or scan response payloads, you can use a low cost Bluetooth Low Energy peripheral to send data one-way to any devices in listening range, as shown in the figure below. This is known as Broadcasting in Bluetooth Low Energy.

Once you establish a connection between your peripheral and a central device, the advertising process will generally stop and you will typically no longer be able to send advertising packets out anymore, and you will use GATT services and characteristics to communicate in both directions.

Figure 3. Broadcast network topology



1.2 GATT

GATT is an acronym for the Generic Attribute Profile, and it defines the way that two Bluetooth Low Energy devices transfer data back and forth using concepts called Services and Characteristics. It makes use of a generic data protocol called the Attribute Protocol (ATT), which is used to store Services, Characteristics and related data in a simple lookup table using 16-bit IDs for each entry in the table.

GATT comes into play once a dedicated connection is established between two devices, meaning that you have already gone through the advertising process governed by GAP.

The most important thing to keep in mind with GATT and connections is that connections are exclusive. It means that a BLE peripheral can only be connected to one central device at a time! As soon as a peripheral connects to a central device, it will stop advertising itself and other devices will no longer be able to see it or connect to it until the existing connection is broken.

Establishing a connection is also the only way to allow two-way communication, where the central device can send meaningful data to the peripheral and vice versa.

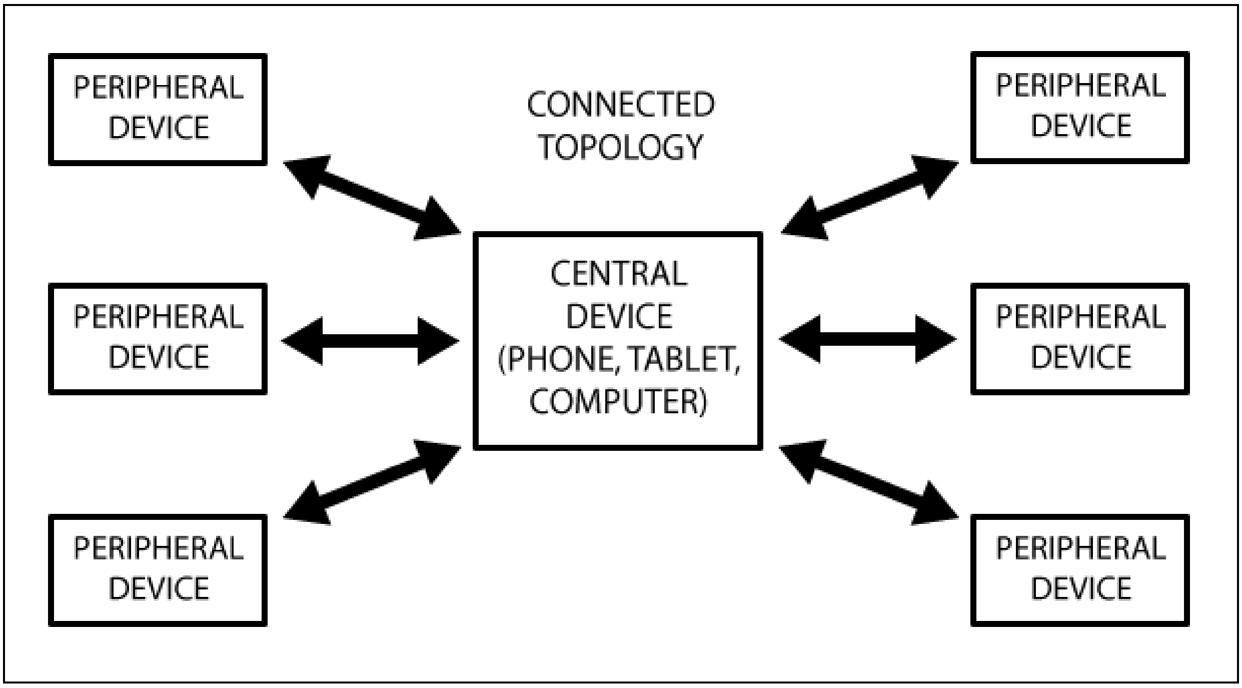
1.2.1 Connected network topology

The following figure should explain the way that Bluetooth Low Energy devices work in a connected environment. A peripheral can only be connected to one central device (such as a mobile phone) at a time, but the central device can be connected to multiple peripherals.

If data needs to be exchanged between two peripherals, a custom mailbox system will need to be implemented where all messages pass through the central device.

Once a connection is established between a peripherals and central device, however, communication can take place in both directions, which is different from the one-way broadcasting approach using only advertising data and GAP.

Figure 4. Connected network topology



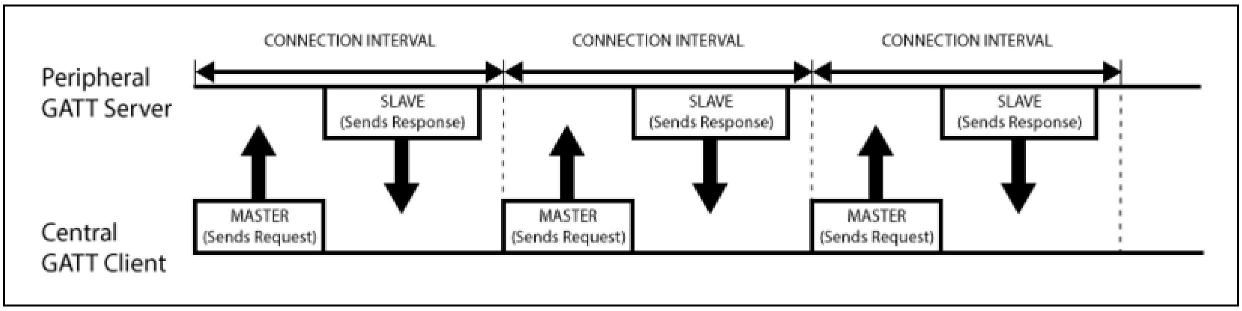
1.2.2 GATT Transactions

An important concept to understand with GATT is the server/client relationship. The peripheral is known as the GATT Server, which holds the ATT lookup data and service and characteristic definitions, and the GATT Client (the phone/tablet), which sends requests to this server. All transactions are started by the GATT Client, which receives response from the GATT Server.

When establishing a connection, the peripheral will suggest a "Connection Interval" to the central device, and the central device will try to reconnect every connection interval to see if any new data is available, etc. It is important to keep in mind that this connection interval is really just a suggestion, though! Your central device may not be able to honor the request because it is busy communicating with another peripheral or the required system resources just are not available.

The following figure should illustrate the data exchange process between a peripheral (the GATT Server) and a central device (the GATT Client), with the main device initiating every transaction.

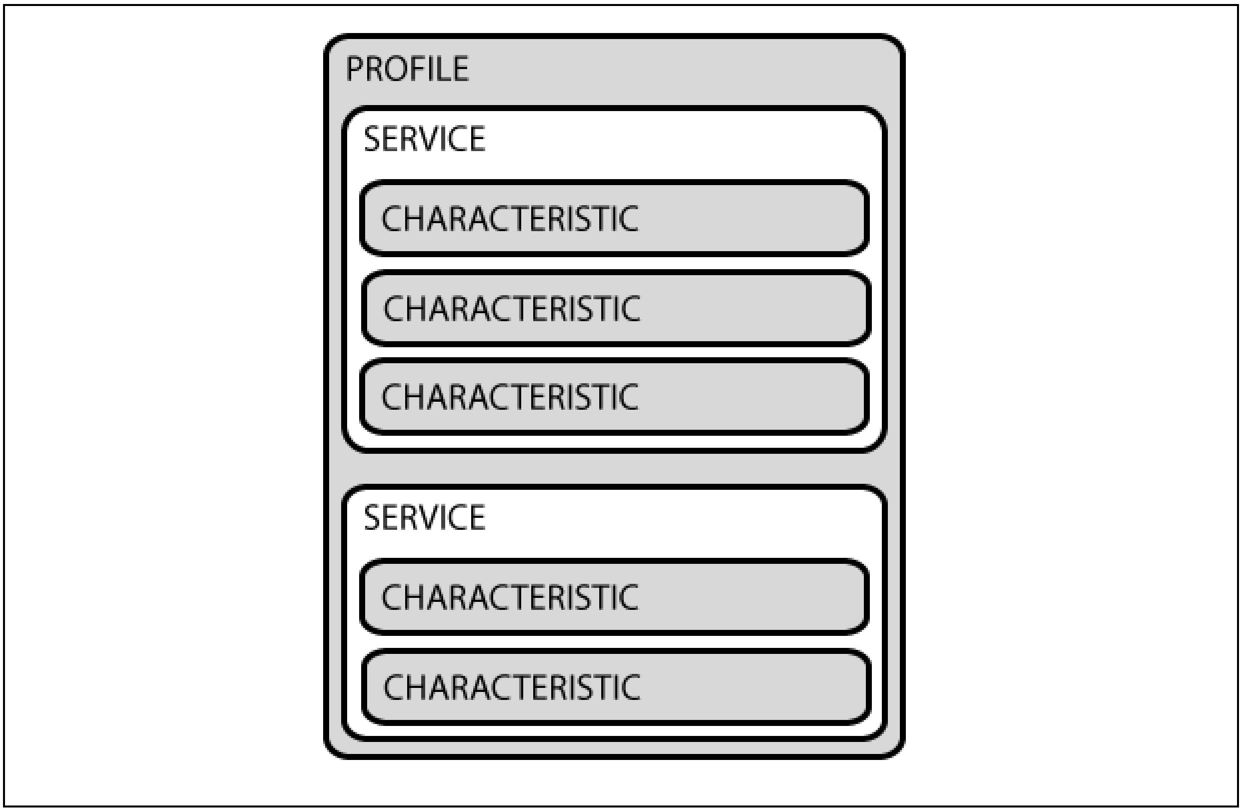
Figure 5. GATT Transactions



1.2.3 Services and characteristics

GATT transactions in BLE are based on high-level, nested objects called Profiles, Services and Characteristics, which can be seen in the figure below.

Figure 6. Profile architecture



1.2.3.1 Profile

A Profile does not actually exist on the BLE peripheral itself; it is simply a pre-defined collection of Services that has been compiled either by the Bluetooth SIG or by the peripheral designers. The Heart Rate Profile, for example, combines the Heart Rate Service and the Device Information Service. The complete list of officially adopted GATT-based profiles can be seen here: Profiles Overview.

1.2.3.2 Service

Services are used to break data up into logical entities, and contain specific chunks of data called characteristics. A service can have one or more characteristics, and each service distinguishes itself from other services by means of a unique numeric ID called UUID, which can be either 16-bit (for officially adopted BLE Services) or 128-bit (for custom services).

A full list of officially adopted BLE services can be seen on the "Service" page of the Bluetooth Developer Portal. If you look at the Heart Rate Service, for example, we can see that this officially adopted service has a 16-bit UUID of 0x180D, and contains up to three characteristics, though only the first one is mandatory: Heart Rate Measurement, Body Sensor Location and Heart Rate Control Point.

1.2.3.3 Characteristics

The lowest level concept in GATT transactions is the Characteristic, which encapsulates a single data point (though it may contain an array of related data, such as X/Y/Z values from a 3-axis accelerometer, etc.).

Similarly to Services, each Characteristic distinguishes itself via a pre-defined 16-bit or 128-bit

UUID, and you're free to use the standard characteristics defined by the Bluetooth SIG or define your own custom characteristics which only your peripheral and SW understands.

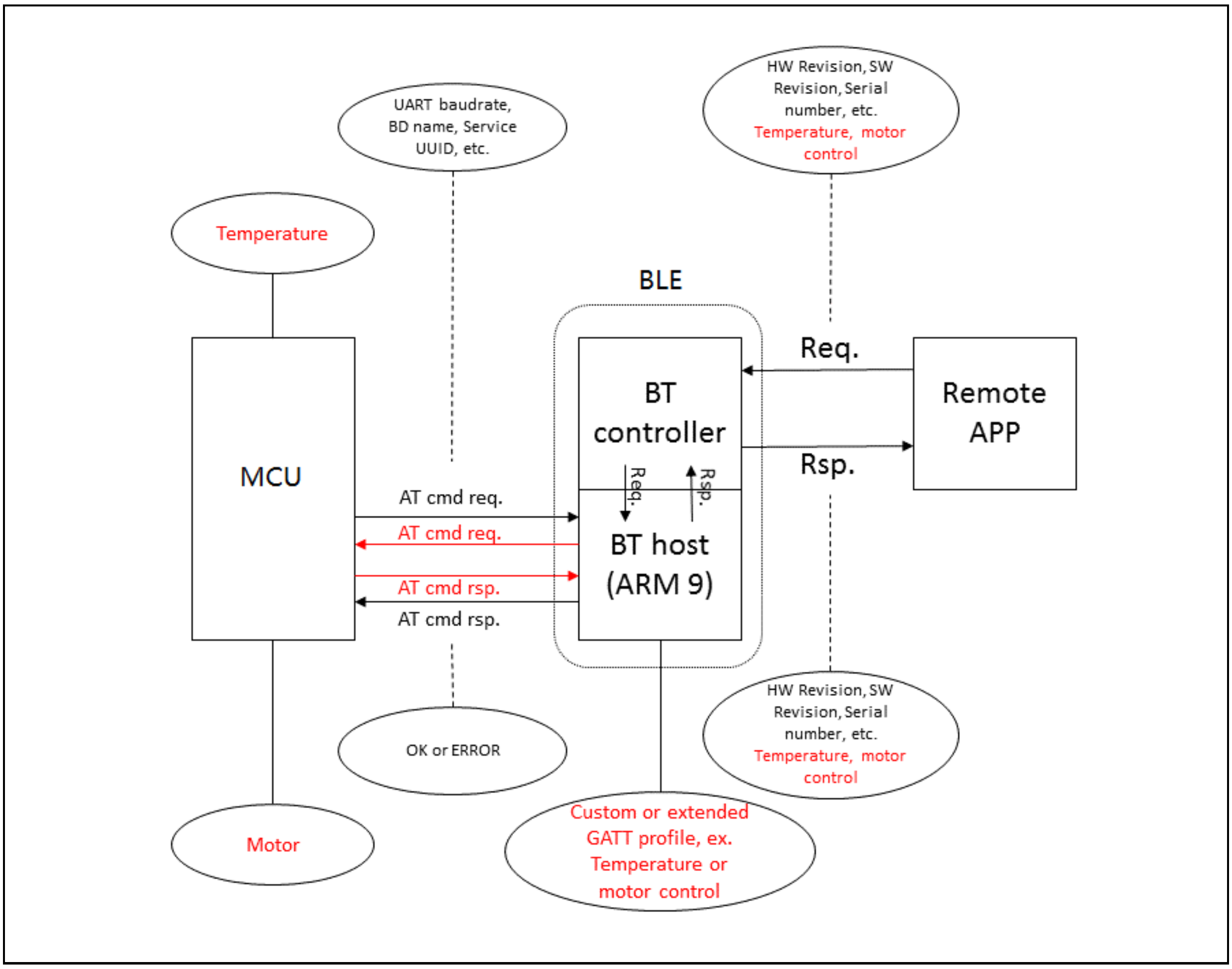
As an example, the Heart Rate Measurement characteristic is mandatory for the Heart Rate Service, and uses a UUID of 0x2A37. It starts with a single 8-bit value describing the HRM data format (whether the data is UINT8 or UINT16, etc.), and then goes on to include the heart rate measurement data that matches this config byte.

Characteristics are the main point that you will interact with your BLE peripheral, so it is important to understand the concept. They are also used to send data back to the BLE peripheral, since you are also able to write to characteristic. You could implement a simple UART-type interface with a custom “UART Service” and two characteristics, one for the TX channel and one for the RX channel, where one characteristic might be configured as read only and the other would have write privileges.

1.3 System framework

AT32WB415 actually consists of MCU and wireless Bluetooth module (BLE) that communicates through UART interface. After receiving a request from remote APP, BLE obtains required information from MCU or performs operations through AT command; or the MCU sends AT command request through UART to change the configuration on BLE side. No matter which direction the request is sent, users can expand AT command according to the needs to implement various control methods.

Figure 7. System framework



2 Add custom services to BLE

In this routine, there are already necessary services for GATT, and these services can be obtained through remote APP, but users need to customize services to implement other desired functions. In this application, a custom service is written for users. Users can also add other services following this routine.

In addition, this is an ARM9 project, and users need to install Legacy Support for compilation. Please download at www2.keil.com/mdk5/legacy/.

2.1 Add profiles to project

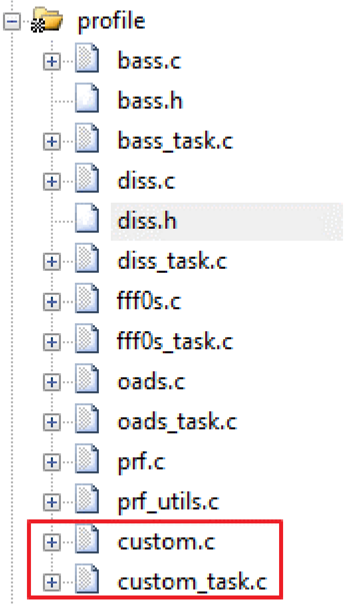
When adding a custom service, the following six files are required:

- custom.c
- custom.h
- custom_task.c
- custom_task.h
- app_custom.c
- app_custom.h
- Put these files in the following directory (users need to create a folder):
- custom.c and custom_task.c: sdk\ble_stack\com\profiles\custom\src
- custom.h and custom_task.h: sdk\ble_stack\com\profiles\custom\api
- app_custom.c and app_custom.h: projects\ble_app_gatt\app

2.2 Configure profiles in project

1. Open Keil, and then add custom.c and custom_task.c to “profile”.

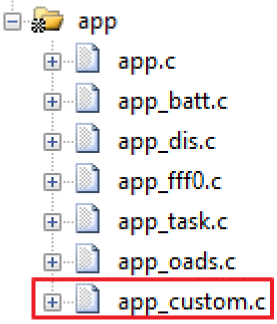
Figure 8. Files in profile



profile
bass.c
bass.h
bass_task.c
diss.c
diss.h
diss_task.c
fff0s.c
fff0s_task.c
oads.c
oads_task.c
prf.c
prf_utils.c
custom.c
custom_task.c

2. Add app_custom.c to “app”.

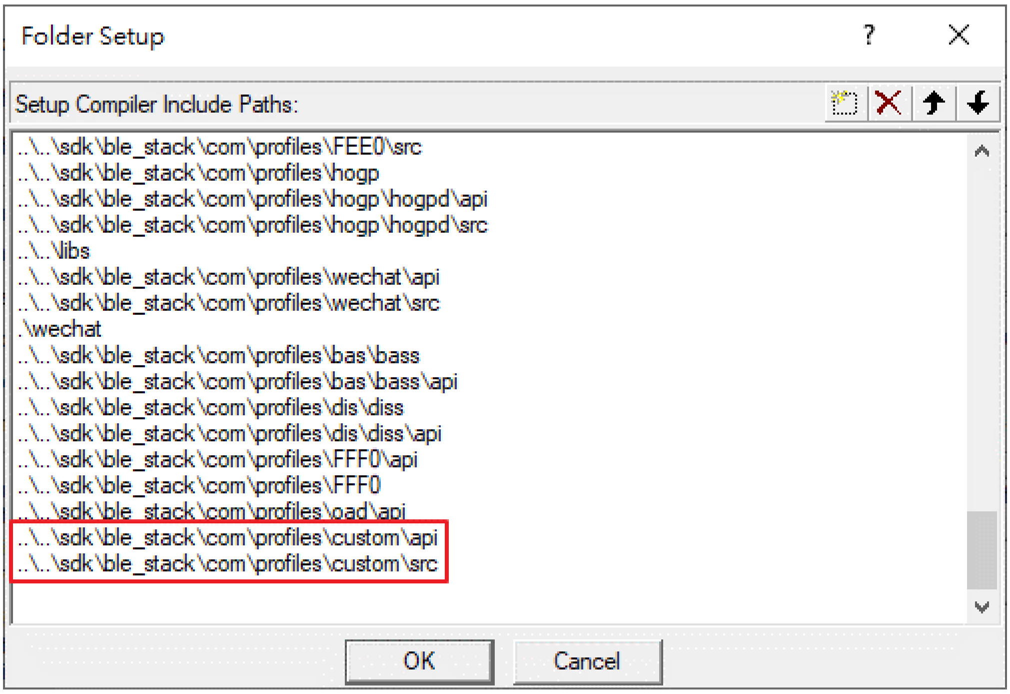
Figure 9. Files in app



app
app.c
app_batt.c
app_dis.c
app_fff0.c
app_task.c
app_oads.c
app_custom.c

3. Add the corresponding profile path to “Include Paths” of Keil C/C++.

Figure 10. Include Paths



```
..\..\sdk\ble_stack\com\profiles\FEE0\src
..\..\sdk\ble_stack\com\profiles\hogp
..\..\sdk\ble_stack\com\profiles\hogp\hogpd\api
..\..\sdk\ble_stack\com\profiles\hogp\hogpd\src
..\..\libs
..\..\sdk\ble_stack\com\profiles\wechat\api
..\..\sdk\ble_stack\com\profiles\wechat\src
.\wechat
..\..\sdk\ble_stack\com\profiles\bas\bass
..\..\sdk\ble_stack\com\profiles\bas\bass\api
..\..\sdk\ble_stack\com\profiles\dis\diss
..\..\sdk\ble_stack\com\profiles\dis\diss\api
..\..\sdk\ble_stack\com\profiles\FFF0\api
..\..\sdk\ble_stack\com\profiles\FFF0
..\..\sdk\ble_stack\com\profiles\oad\api
..\..\sdk\ble_stack\com\profiles\custom\api
..\..\sdk\ble_stack\com\profiles\custom\src
```

2.3 Add custom services to current software architecture

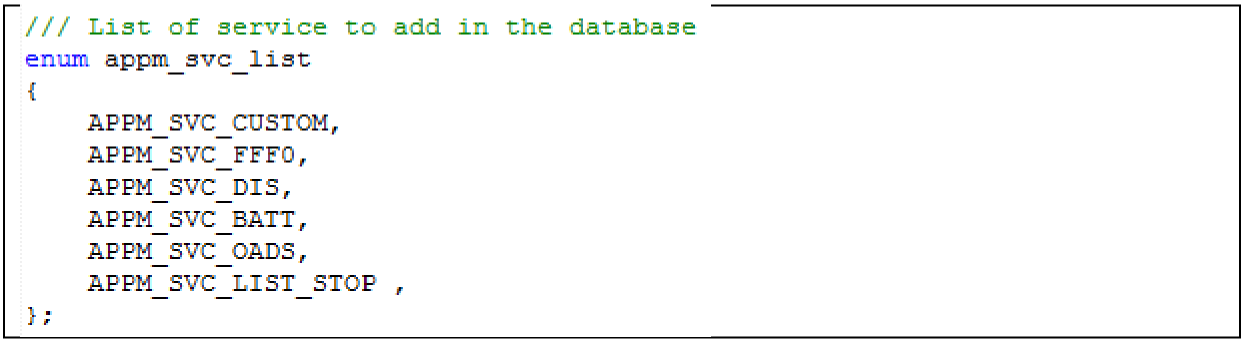
1. Find the appm_msg_handler function in app_task.c, and add message case for custom ID processing.

Figure 11. Entry point for processing new task ID

```
case (TASK_ID_CUSTOM):
{
    // Call the Health Thermometer Module
    msg_pol = appm_get_handler(&app_custom_table_handler, msgid, param, src_id);
} break;
```

2. Find the appm_svc_list in app.c.

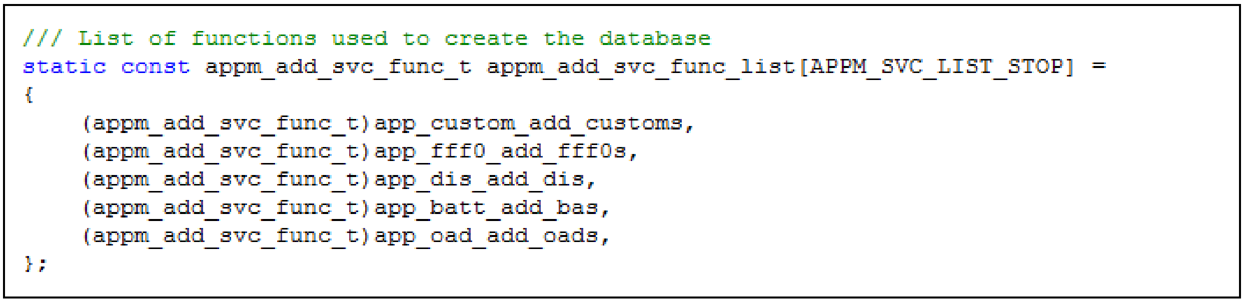
Figure 12. List of services



```
/// List of service to add in the database
enum appm_svc_list
{
    APPM_SVC_CUSTOM,
    APPM_SVC_FFF0,
    APPM_SVC_DIS,
    APPM_SVC_BATT,
    APPM_SVC_OADS,
    APPM_SVC_LIST_STOP ,
};
```

3. Add a list of functions in app.c to create a database.

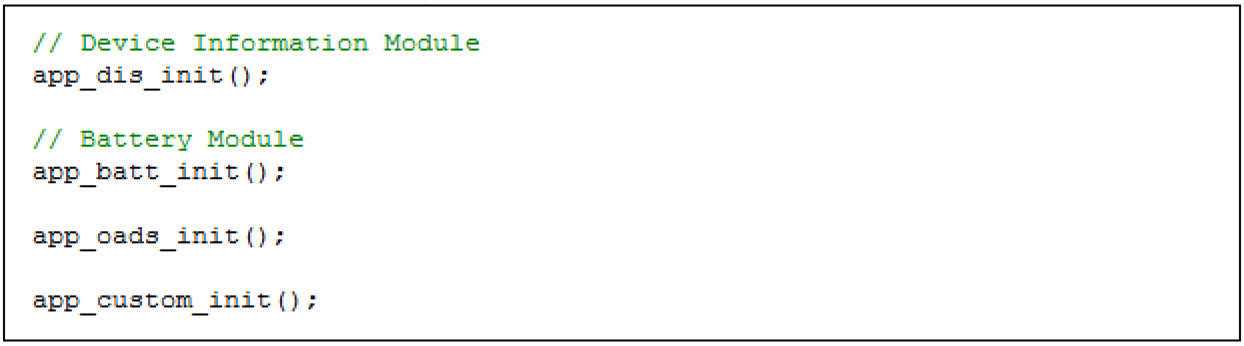
Figure 13. List of functions



```
/// List of functions used to create the database
static const appm_add_svc_func_t appm_add_svc_func_list[APPM_SVC_LIST_STOP] =
{
    (appm_add_svc_func_t)app_custom_add_customs,
    (appm_add_svc_func_t)app_fff0_add_fff0s,
    (appm_add_svc_func_t)app_dis_add_dis,
    (appm_add_svc_func_t)app_batt_add_bas,
    (appm_add_svc_func_t)app_oad_add_oads,
};
```

4. Find appm_init function in app.c and add in app_custom_init function.

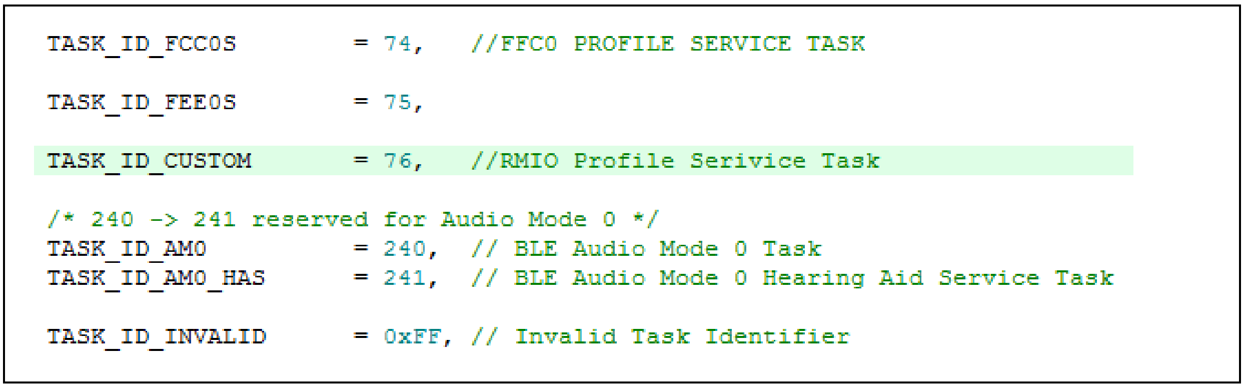
Figure 14. Initialize custom service



```
// Device Information Module
app_dis_init();

// Battery Module
app_batt_init();

app_oads_init();

app_custom_init();
```

5. Add custom service ID to TASK_API_ID of rwip_task.h.

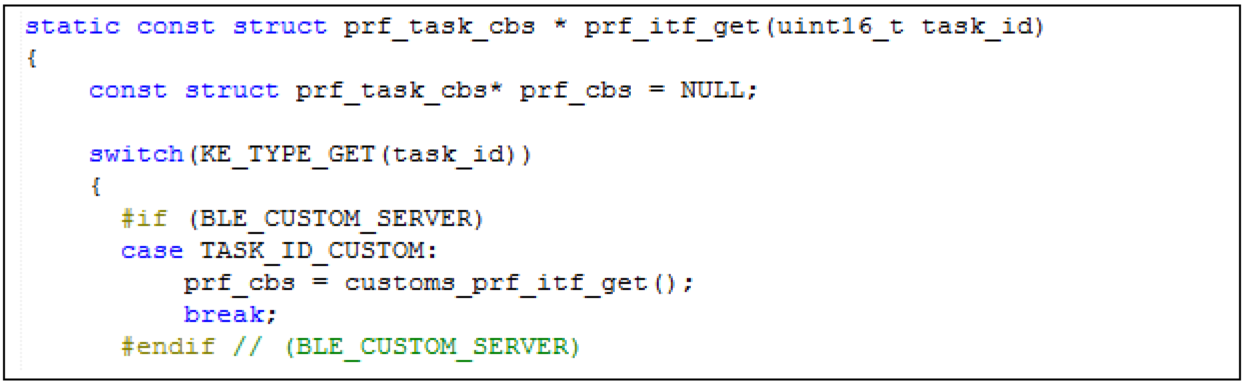
Figure 15. Add task ID



```
TASK_ID_FCC0S        = 74,   //FFC0 PROFILE SERVICE TASK

TASK_ID_FEE0S        = 75,

TASK_ID_CUSTOM       = 76,   //RMIO Profile Serivice Task

/* 240 -> 241 reserved for Audio Mode 0 */
TASK_ID_AM0          = 240,  // BLE Audio Mode 0 Task
TASK_ID_AM0_HAS      = 241,  // BLE Audio Mode 0 Hearing Aid Service Task

TASK_ID_INVALID      = 0xFF, // Invalid Task Identifier
```

6. Add customs_prf_itf_get function call to prf.c.

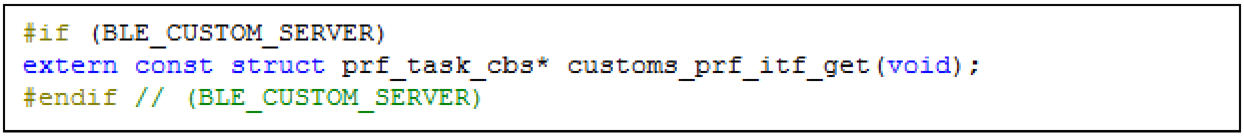
Figure 16. Call custom_prf_itf_get()



```
static const struct prf_task_cbs * prf_itf_get(uint16_t task_id)
{
    const struct prf_task_cbs* prf_cbs = NULL;

    switch(KE_TYPE_GET(task_id))
    {
      #if (BLE_CUSTOM_SERVER)
      case TASK_ID_CUSTOM:
          prf_cbs = customs_prf_itf_get();
          break;
      #endif // (BLE_CUSTOM_SERVER)
```

7. Add custom_prf_itf_get function declaration to prf.c.

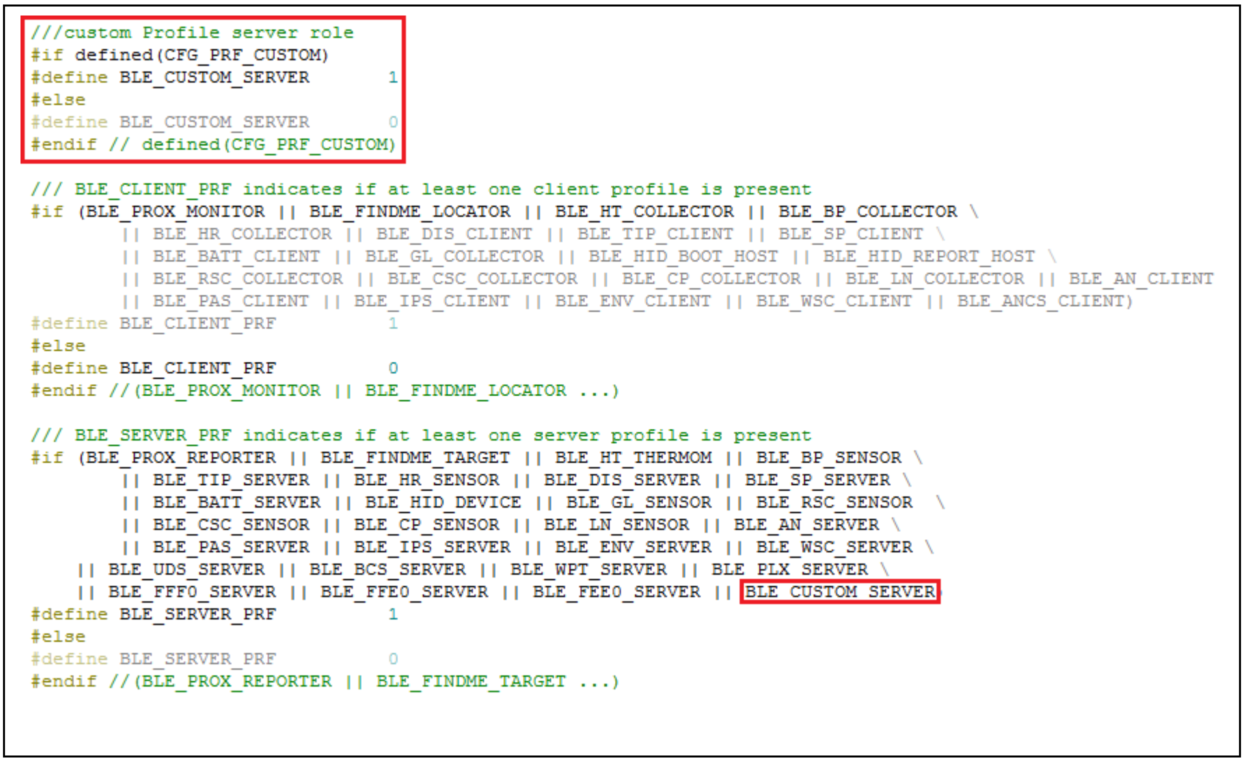
Figure 17. Declare custom_prf_itf_get()



```
#if (BLE_CUSTOM_SERVER)
extern const struct prf_task_cbs* customs_prf_itf_get(void);
#endif // (BLE_CUSTOM_SERVER)
```

8. Add the following definitions to rwprf_config.h.

Figure 18. Open the macro for custom service and conditions listed as servo profile



```
///custom Profile server role
#if defined(CFG_PRF_CUSTOM)
#define BLE_CUSTOM_SERVER        1
#else
#define BLE_CUSTOM_SERVER        0
#endif // defined(CFG_PRF_CUSTOM)

/// BLE_CLIENT_PRF indicates if at least one client profile is present
#if (BLE_PROX_MONITOR || BLE_FINDME_LOCATOR || BLE_HT_COLLECTOR || BLE_BP_COLLECTOR \
        || BLE_HR_COLLECTOR || BLE_DIS_CLIENT || BLE_TIP_CLIENT || BLE_SP_CLIENT \
        || BLE_BATT_CLIENT || BLE_GL_COLLECTOR || BLE_HID_BOOT_HOST || BLE_HID_REPORT_HOST \
        || BLE_RSC_COLLECTOR || BLE_CSC_COLLECTOR || BLE_CP_COLLECTOR || BLE_LN_COLLECTOR || BLE_AN_CLIENT
        || BLE_PAS_CLIENT || BLE_IPS_CLIENT || BLE_ENV_CLIENT || BLE_WSC_CLIENT || BLE_ANCS_CLIENT)
#define BLE_CLIENT_PRF           1
#else
#define BLE_CLIENT_PRF           0
#endif //(BLE_PROX_MONITOR || BLE_FINDME_LOCATOR ...)

/// BLE_SERVER_PRF indicates if at least one server profile is present
#if (BLE_PROX_REPORTER || BLE_FINDME_TARGET || BLE_HT_THERMOM || BLE_BP_SENSOR \
        || BLE_TIP_SERVER || BLE_HR_SENSOR || BLE_DIS_SERVER || BLE_SP_SERVER \
        || BLE_BATT_SERVER || BLE_HID_DEVICE || BLE_GL_SENSOR || BLE_RSC_SENSOR  \
        || BLE_CSC_SENSOR || BLE_CP_SENSOR || BLE_LN_SENSOR || BLE_AN_SERVER \
        || BLE_PAS_SERVER || BLE_IPS_SERVER || BLE_ENV_SERVER || BLE_WSC_SERVER \
    || BLE_UDS_SERVER || BLE_BCS_SERVER || BLE_WPT_SERVER || BLE_PLX_SERVER \
    || BLE_FFF0_SERVER || BLE_FFE0_SERVER || BLE_FEE0_SERVER || BLE_CUSTOM_SERVER)
#define BLE_SERVER_PRF           1
#else
#define BLE_SERVER_PRF           0
#endif //(BLE_PROX_REPORTER || BLE_FINDME_TARGET ...)
```

The BLE_CUSTOM_SERVER macro definition is used in custom.c, custom.h, custom_task.c and custom_task.h. The compiler can compile custom services only when this macro is open.

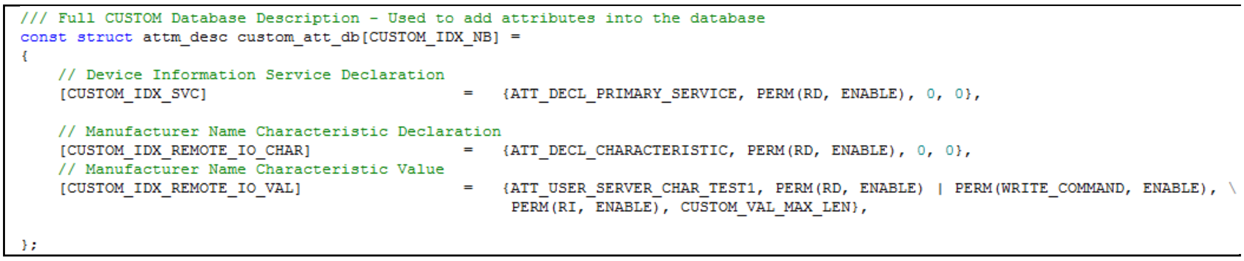
2.4 BLE interface description

1. Custom service implements a readable and writable characteristic, and its UUID and related attributes are shown in the following table.

Table 1. Characteristics of custom service

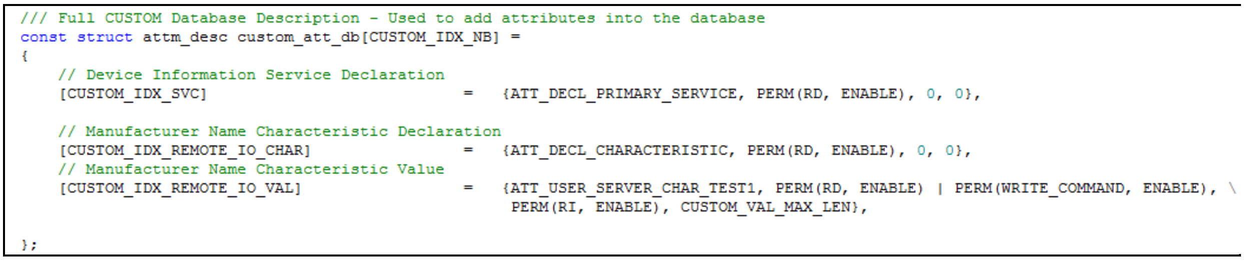
UUID	Characteristic permission	Data length to be sent/received
0xC101	Read/Write without response	1 byte

Set permissions in the ATT database of custom service.



```
/// Full CUSTOM Database Description - Used to add attributes into the database
const struct attm_desc custom_att_db[CUSTOM_IDX_NB] =
{
    // Device Information Service Declaration
    [CUSTOM_IDX_SVC]                            =   {ATT_DECL_PRIMARY_SERVICE, PERM(RD, ENABLE), 0, 0},

    // Manufacturer Name Characteristic Declaration
    [CUSTOM_IDX_REMOTE_IO_CHAR]                 =   {ATT_DECL_CHARACTERISTIC, PERM(RD, ENABLE), 0, 0},
    // Manufacturer Name Characteristic Value
    [CUSTOM_IDX_REMOTE_IO_VAL]                  =   {ATT_USER_SERVER_CHAR_TEST1, PERM(RD, ENABLE) | PERM(WRITE_COMMAND, ENABLE), \
                                                     PERM(RI, ENABLE), CUSTOM_VAL_MAX_LEN},

};
```

Figure 19. ATT database of custom service



```
/// Full CUSTOM Database Description - Used to add attributes into the database
const struct attm_desc custom_att_db[CUSTOM_IDX_NB] =
{
    // Device Information Service Declaration
    [CUSTOM_IDX_SVC]                            =   {ATT_DECL_PRIMARY_SERVICE, PERM(RD, ENABLE), 0, 0},

    // Manufacturer Name Characteristic Declaration
    [CUSTOM_IDX_REMOTE_IO_CHAR]                 =   {ATT_DECL_CHARACTERISTIC, PERM(RD, ENABLE), 0, 0},
    // Manufacturer Name Characteristic Value
    [CUSTOM_IDX_REMOTE_IO_VAL]                  =   {ATT_USER_SERVER_CHAR_TEST1, PERM(RD, ENABLE) | PERM(WRITE_COMMAND, ENABLE), \
                                                     PERM(RI, ENABLE), CUSTOM_VAL_MAX_LEN},

};
```

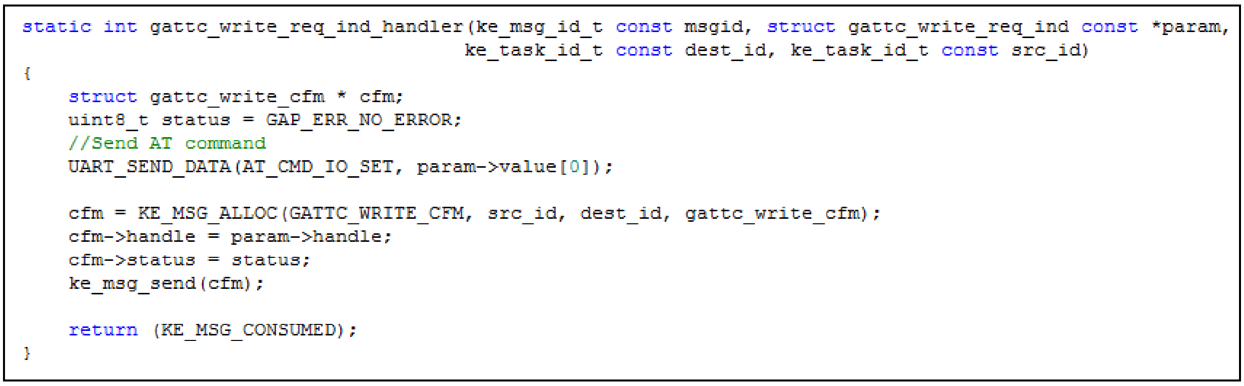
The second parameter of the structure can set the permission of custom service or characteristic. The permissions are defined as follows.

Table 2. Permission definitions

Code symbol	Description
RD	Read
WRITE_REQ	Write
WRITE_COMMAND	Write without response
NTF	Notification
IND	Indication

2. Data sending function is located in custom_task.c, which is implemented by using gattc_write_req_ind_handler() function.

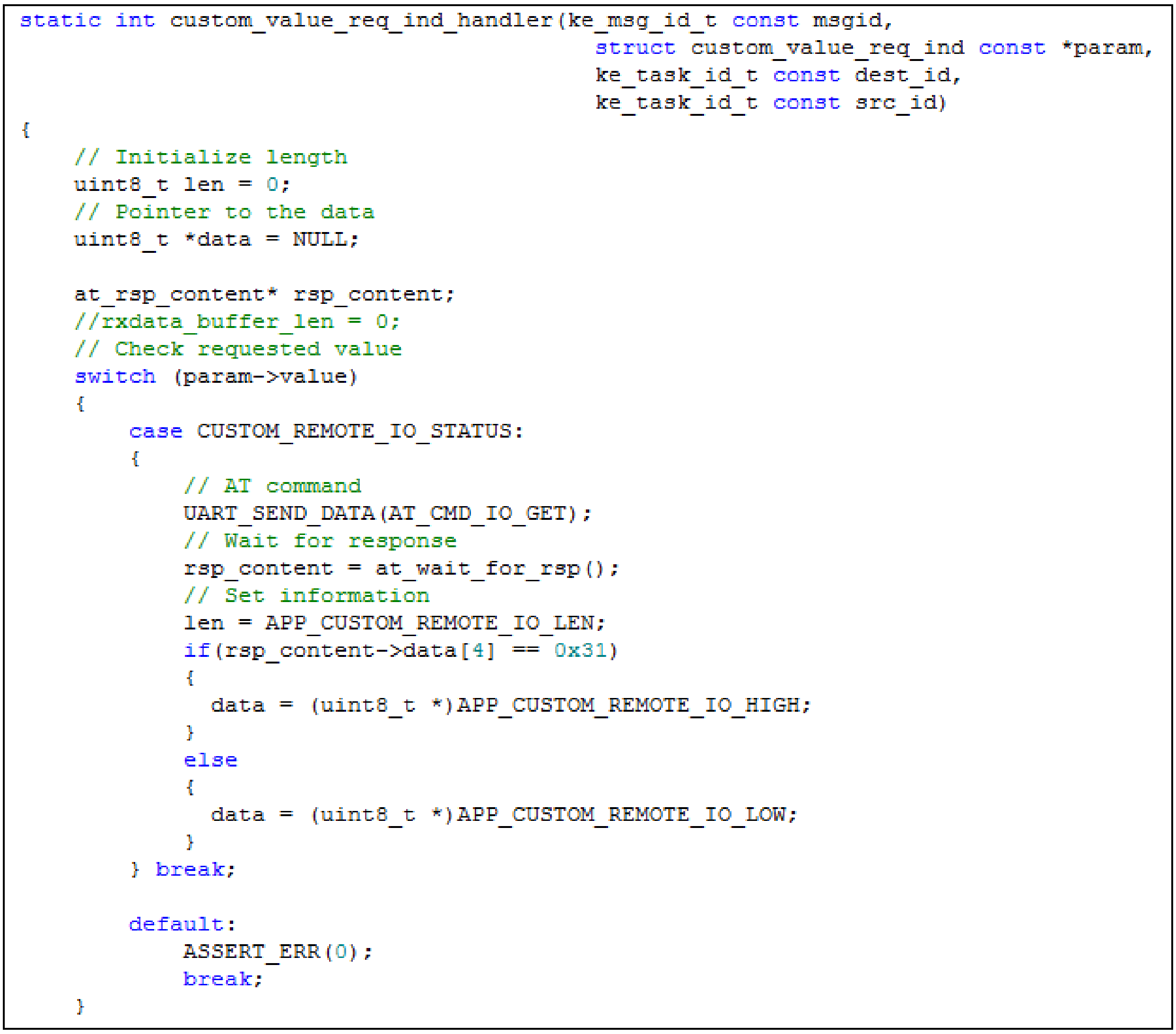
Figure 20. Data sending function



```
static int gattc_write_req_ind_handler(ke_msg_id_t const msgid, struct gattc_write_req_ind const *param,
                                       ke_task_id_t const dest_id, ke_task_id_t const src_id)
{
    struct gattc_write_cfm * cfm;
    uint8_t status = GAP_ERR_NO_ERROR;
    //Send AT command
    UART_SEND_DATA(AT_CMD_IO_SET, param->value[0]);

    cfm = KE_MSG_ALLOC(GATTC_WRITE_CFM, src_id, dest_id, gattc_write_cfm);
    cfm->handle = param->handle;
    cfm->status = status;
    ke_msg_send(cfm);

    return (KE_MSG_CONSUMED);
}
```

3. Data receiving function is located in app_custom.c, which is implemented by using custom_value_req_ind_handler() function. More cases can be added through switch in a similar way.

Figure 21. Data receiving function



```
static int custom_value_req_ind_handler(ke_msg_id_t const msgid,
                                           struct custom_value_req_ind const *param,
                                           ke_task_id_t const dest_id,
                                           ke_task_id_t const src_id)
{
    // Initialize length
    uint8_t len = 0;
    // Pointer to the data
    uint8_t *data = NULL;

    at_rsp_content* rsp_content;
    //rxdata_buffer_len = 0;
    // Check requested value
    switch (param->value)
    {
        case CUSTOM_REMOTE_IO_STATUS:
        {
            // AT command
            UART_SEND_DATA(AT_CMD_IO_GET);
            // Wait for response
            rsp_content = at_wait_for_rsp();
            // Set information
            len = APP_CUSTOM_REMOTE_IO_LEN;
            if(rsp_content->data[4] == 0x31)
            {
              data = (uint8_t *)APP_CUSTOM_REMOTE_IO_HIGH;
            }
            else
            {
              data = (uint8_t *)APP_CUSTOM_REMOTE_IO_LOW;
            }
        } break;

        default:
            ASSERT_ERR(0);
            break;
    }
```

3 AT command

3.1 Introduction

The Hayes command set (also known as the AT command set) is a specific command language originally developed for the Hayes Smartmodem 300. The command set consists of a series of short text strings that can be combined to produce commands for operations such as dialing, hanging up, and changing the parameters of the connection. The vast majority of dial-up modems use the Hayes command set in numerous variations.

The Hayes command set can subdivide into four groups:

1. Basic command set: A capital character followed by a digit. For example, M1.
2. Extended command set: An "&" (ampersand) and a capital character followed by a digit. This extends the basic command set. For example, &M1.
3. Proprietary command set: Usually starting either with a backslash ("\") or with a percent sign ("%"); these commands vary widely among modem manufacturers.
4. Register commands: Sr=n, where “r” is the number of the register to be changed, and “n” is the new value that is assigned.

3.2 BLE command

In this application note, only the basic command set is used. There are also some important characters for modem initialization.

1) AT - "Attention": Each command string is prefixed with "AT", and a number of discrete modem commands can be concatenated after the "AT".
2) Z - reset: Reset the modem to its initial state.
3) (a comma): Pause the software for one second, or many seconds if there are multiple commas.
4) ^M - Send a Carriage Return character to modem. It is a control character (transmitting this character is actually transmitting a byte, and the content is CR in ASCII).

AT command set lists implemented in this application are shown below.

Table 3. AT command set list(send from MCU)

	Send from MCU	Response from BLE	Note
Wrong command or command not supported		ERROR	When the BLE receives a command not supported or wrong command, it returns ERROR, and MCU/BLE will send a new AT command. E.g., If MCU sends a wrong command ATT, BLE will return ERROR.
Startup test: AT	AT	OK	A. It is used to confirm whether the BLE is ready. B. After receiving this command, MCU returns OK and confirms to start AT

				command, thus to avoid MCU sending AT command before the completion of BLE power-on initialization, causing malfunctions. E.g., Test to confirm that BLE is in AT command mode: if MCU sends AT, BLE will return OK.
Set UART baud rate and save in Flash: AT+BAUD	9600bps	AT+BAUD1	OK9600	A. Default baud rate: 9,600bps B. After BLE responds to the baud rate, the new baud rate is saved in Flash, and BLE communicates with MCU at the new baud rate. Power on again and reset, BLE will continue communication at the set baud rate C. After BLE responds to the baud rate, it immediately switches to the new baud rate for communication. E.g., When the baud rate is set to 115,200bps, MCU sends AT+BAUD5 and BLE returns OK115200; then BLE communicates with MCU at 115,200bps. Power on again and reset, BLE will continue communication at 115,200bps.
	19200bps	AT+BAUD2	OK19200	
	38400bps	AT+BAUD3	OK38400	
	57600bps	AT+BAUD4	OK57600	
	115200bps	AT+BAUD5	OK115200	
Set UART baud rate and save in SRAM: AT+BAUDS	9600bps	AT+BAUDS1	OK9600	A. After BLE responds to the baud rate, it communicates with MCU at the new baud rate. Power on again, BLE will communicate at the baud rate saved in Flash.
	19200bps	AT+BAUDS2	OK19200	
	38400bps	AT+BAUDS3	OK38400	
	57600bos	AT+BAUDS4	OK57600	
	115200bps	AT+BAUDS5	OK115200	

			B. After responding to the baud rate, BLE communicates with MCU at the new baud rate immediately. E.g., When the baud rate is set to 19,200bps, MCU sends AT+BAUDS2 and BLE returns OK19200; then BLE communicates with MCU at 19200bps. Power on again and reset, BLE will communicate with MCU at the baud rate saved in Flash.
Modify BD name and save in Flash: AT+NAME	AT+NAMExxxx	OKxxxx	A. Default name: SerialSPP B. After BLE responds to the BD name, the new BD name is saved in Flash, and BLE continue advertising with the new BD name. Power on again and reset, BLE will continue communication with the new BD name. C. Support up to 20-char BD name. E.g., When BD name is changed to Serial-GATT, MCU sends AT+NAMESerial-GATT and BLE returns OKSerial-GATT; then BLE advertises with the name of Serial-GATT. Power on again and reset, BLE will continue using the name of Serial-GATT.
Modify BD name and save in SRAM: AT+NAMES	AT+NAMESxxxx	OKxxxx	A. After responding to the BD name, BLE continue advertising with the new BD name.

				Power on again and reset, BLE will continue communication with the BD name saved in Flash. B. Support up to 20-char BD name. E.g., When BD name is changed to Serial-GATT, MCU sends AT+NAMESSerial-GATT and BLE returns OKSerial-GATT; then BLE advertises with the name of Serial-GATT. Power again and reset, BLE will use the BD name saved in Flash.
Set advertising interval and save in Flash: AT+ADVI	100ms	AT+ADVI1	OK100	A. Default advertising interval: 100ms B. After BLE returns OK, the new advertising interval is saved in Flash and used for advertising packet. Power again and reset, the new advertising interval will be used. E.g., When the advertising interval is set to 100ms, MCU sends AT+ADVI1 and BLE returns OK100; then the advertising interval is 100ms. Power again and reset, BLE will continue using the advertising interval of 100ms.
	250ms	AT+ADVI2	OK250	
	500ms	AT+ADVI3	OK500	
	1600ms	AT+ADVI4	OK1600	
	3200ms	AT+ADVI5	OK3200	
Set advertising interval and save in SRAM: AT+ADVIS	100ms	AT+ADVIS1	OK100	After BLE returns OK, the new interval is used for advertising packet. Power on again and reset, the advertising interval saved in Flash will be used. E.g., When the advertising interval is set to 100ms,
	250ms	AT+ADVIS2	OK250	
	500ms	AT+ADVIS3	OK500	
	1600ms	AT+ADVIS4	OK1600	
	3200ms	AT+ADVIS5	OK3200	

				MCU sends AT+ADVIS1 and BLE returns OK100; then the advertising interval is 100ms. Power on again and reset, BLE will use the advertising interval saved in Flash.
Read Flash: AT+RFLASH		AT+RFLASHad	OKadda	A. Default 256 byte data value: FF ad(address): 1char da(data): 1 char B. BLE returns OK followed by address and the corresponding data. E.g., When reading the data of “address:00”, MCU sends AT+RFLASH00 and BLE returns OK00FF. The data read from “address:00” is FF.
Write Flash: AT+WFLASH		AT+WFLASHad, da	OKadda	A. Default reserved for MCU accessing 256 byte data: FF ad(address): 1char da(data): 1 char B. BLE returns OK followed by address and the corresponding data. E.g., When writing data:AA of “address:00”, MCU sends AT+WFLASH00AA and BLE returns OK00AA. The data written to “address:00” is AA.

Table 4. AT command set list(send from BLE)

	Send from BLE	Response from MCU	Note
Read remote IO level: AT+IOGET	AT+IOGET	OKIOx	X= 0 or 1. X=0: low level X=1: high level
Write remote IO level: AT+IOSET	AT+IOSETx	OKIOx	X= 0 or 1. X=0: low level X=1: high level

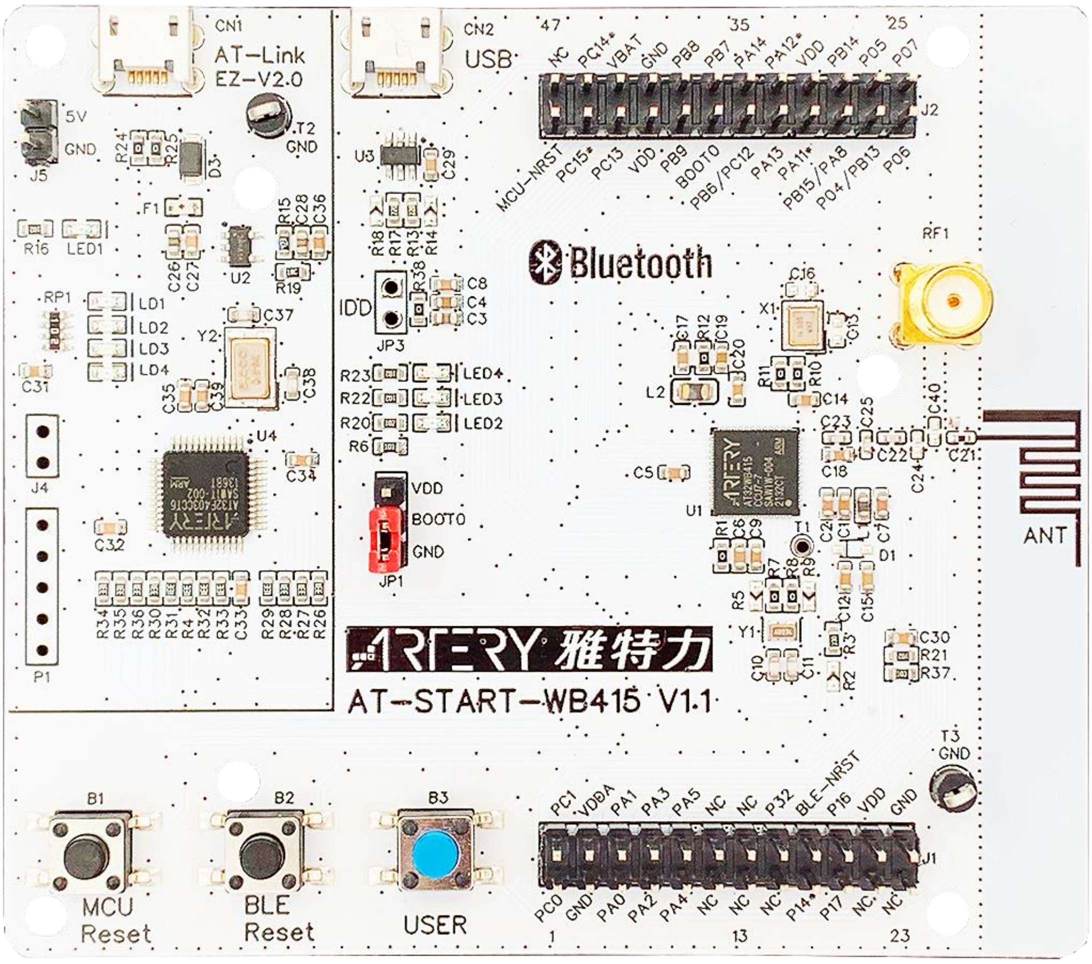
4 BLE application case

This application case shows how to use BLE to operate AT32WB415 on smartphones, including IO control and IO data reading.

4.1 Hardware

1) AT-START-WB415 Board
2) Smartphone with LightBlue APP
3) Micro USB cable

Figure 22. AT-START-WB415 Board



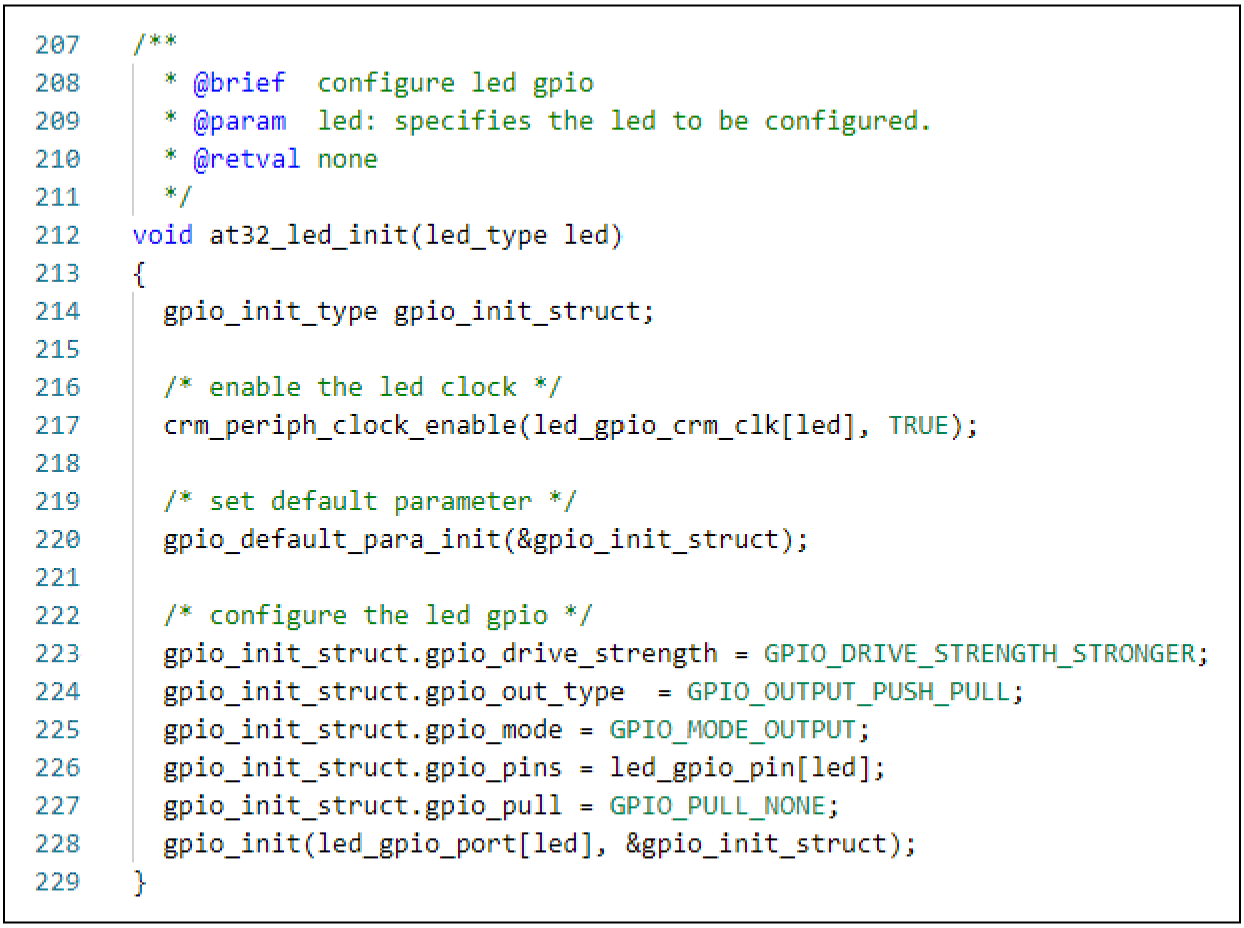
4.2 Software resources

4.2.1 MCU operations

IO control and data reading refer to the operations on MCU peripherals. In the code, users need to complete initialization and write functions to be executed after receiving the command. This application note takes GPIO control as an example, and users can follow this architecture for subsequent development.

1. First, configure the corresponding GPIO. In this case, LED2(PB7) on AT-START-WB415 is used as the controlled pin.

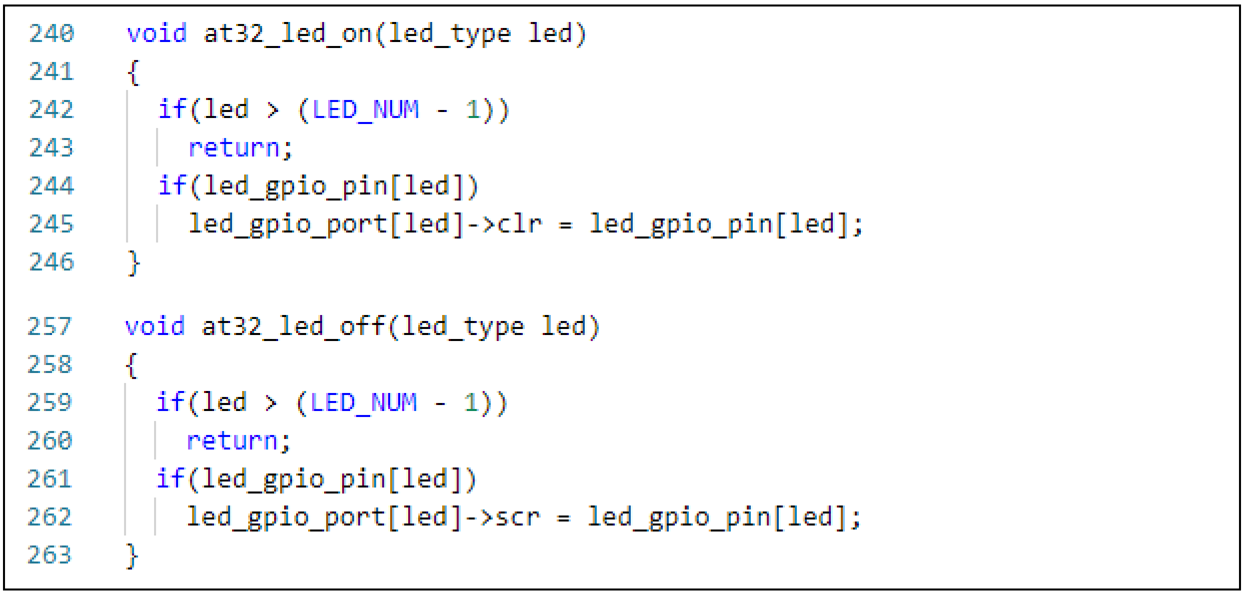
Figure 23. Initialize LED function



```c
/**
  * @brief  configure led gpio
  * @param  led: specifies the led to be configured.
  * @retval none
  */
void at32_led_init(led_type led)
{
  gpio_init_type gpio_init_struct;

  /* enable the led clock */
  crm_periph_clock_enable(led_gpio_crm_clk[led], TRUE);

  /* set default parameter */
  gpio_default_para_init(&gpio_init_struct);

  /* configure the led gpio */
  gpio_init_struct.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER;
  gpio_init_struct.gpio_out_type  = GPIO_OUTPUT_PUSH_PULL;
  gpio_init_struct.gpio_mode = GPIO_MODE_OUTPUT;
  gpio_init_struct.gpio_pins = led_gpio_pin[led];
  gpio_init_struct.gpio_pull = GPIO_PULL_NONE;
  gpio_init(led_gpio_port[led], &gpio_init_struct);
}
```

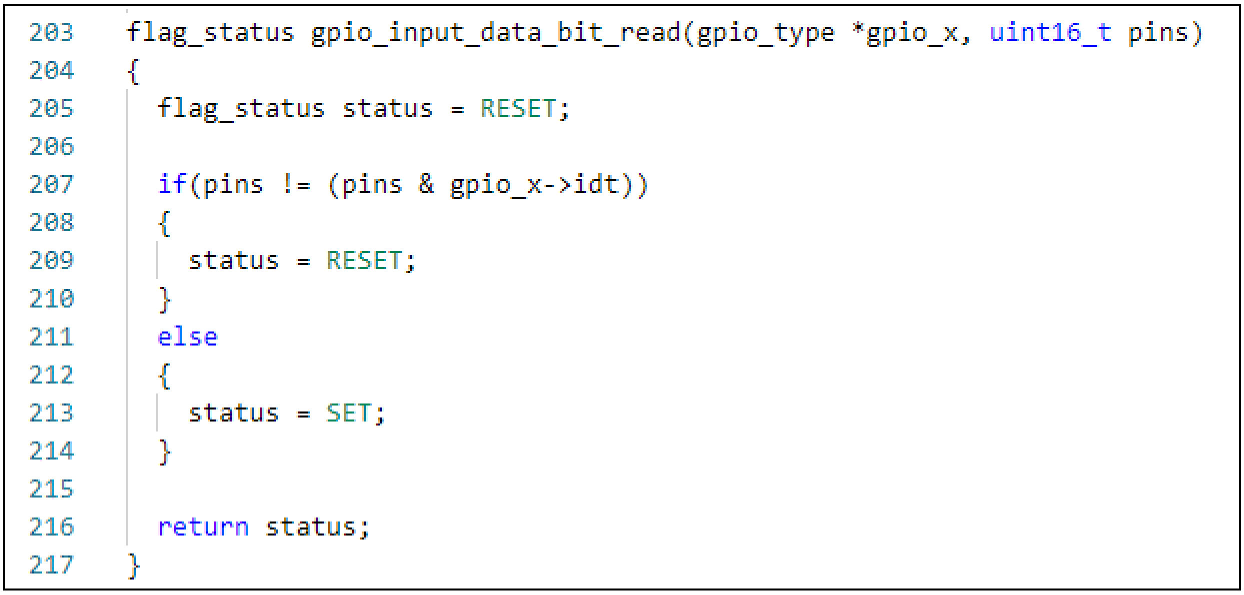
2. Write the code to read and write LED.

Figure 24. Write LED



```c
void at32_led_on(led_type led)
{
  if(led > (LED_NUM - 1))
    return;
  if(led_gpio_pin[led])
    led_gpio_port[led]->clr = led_gpio_pin[led];
}

void at32_led_off(led_type led)
{
  if(led > (LED_NUM - 1))
    return;
  if(led_gpio_pin[led])
    led_gpio_port[led]->scr = led_gpio_pin[led];
}
```

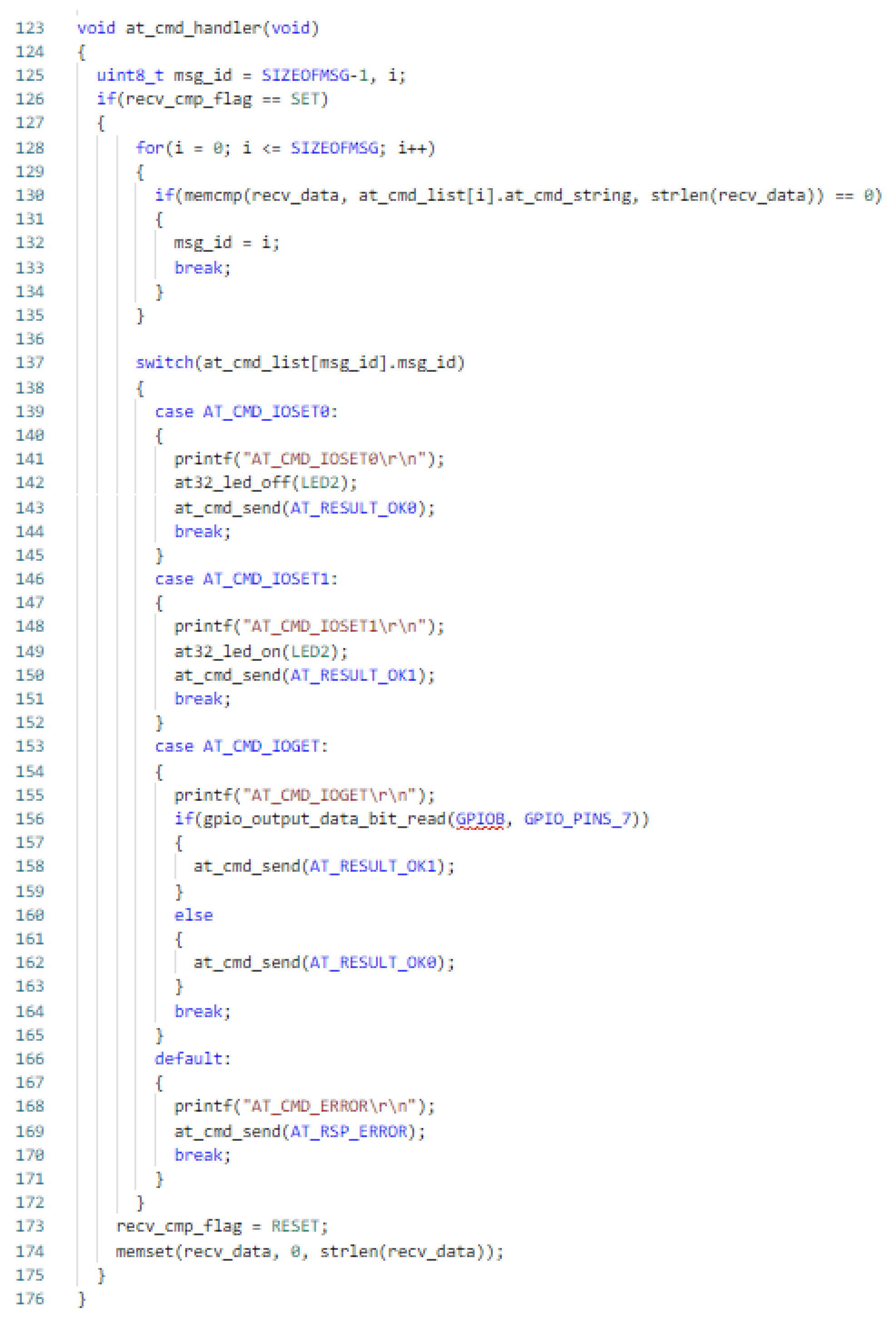
Figure 25. Read LED



```
flag_status gpio_input_data_bit_read(gpio_type *gpio_x, uint16_t pins)
{
  flag_status status = RESET;

  if(pins != (pins & gpio_x->idt))
  {
    status = RESET;
  }
  else
  {
    status = SET;
  }

  return status;
}
```

3. Call the at_cmd_handler function in the main loop to decode the AT Command and perform corresponding operations for different commands.

Figure 26. Call GPIO write and read function

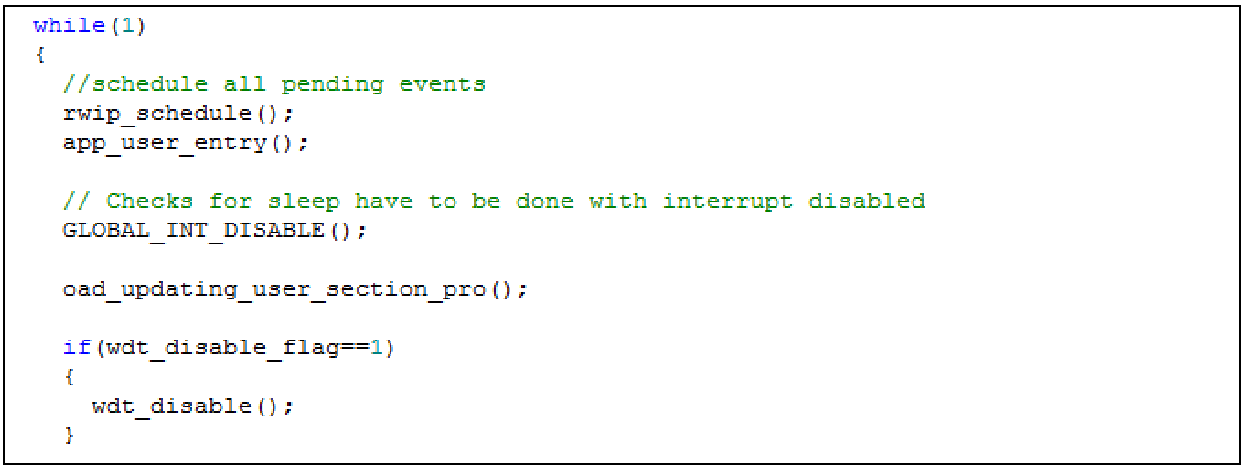


```
void at_cmd_handler(void)
{
  uint8_t msg_id = SIZEOFMSG-1, i;
  if(recv_cmp_flag == SET)
  {
      for(i = 0; i <= SIZEOFMSG; i++)
      {
        if(memcmp(recv_data, at_cmd_list[i].at_cmd_string, strlen(recv_data)) == 0)
        {
          msg_id = i;
          break;
        }
      }

      switch(at_cmd_list[msg_id].msg_id)
      {
        case AT_CMD_IOSET0:
        {
          printf("AT_CMD_IOSET0\r\n");
          at32_led_off(LED2);
          at_cmd_send(AT_RESULT_OK0);
          break;
        }
        case AT_CMD_IOSET1:
        {
          printf("AT_CMD_IOSET1\r\n");
          at32_led_on(LED2);
          at_cmd_send(AT_RESULT_OK1);
          break;
        }
        case AT_CMD_IOGET:
        {
          printf("AT_CMD_IOGET\r\n");
          if(gpio_output_data_bit_read(GPIOB, GPIO_PINS_7))
          {
            at_cmd_send(AT_RESULT_OK1);
          }
          else
          {
            at_cmd_send(AT_RESULT_OK0);
          }
          break;
        }
        default:
        {
          printf("AT_CMD_ERROR\r\n");
          at_cmd_send(AT_RSP_ERROR);
          break;
        }
      }
    recv_cmp_flag = RESET;
    memset(recv_data, 0, strlen(recv_data));
  }
}
```

4.2.2 BLE receives requests

The command processing on Bluetooth side mainly relies on the app_user_entry() function in app.c. After the uart_rx_done flag is set, entry the at_result_to_prefix() to perform decoding to determine whether the received data is AT command and determine the corresponding command number; then entry the corresponding case according to the command number, execute the corresponding request event, and then respond to MCU side.

Figure 27. Poll app_user_entry() in main loop

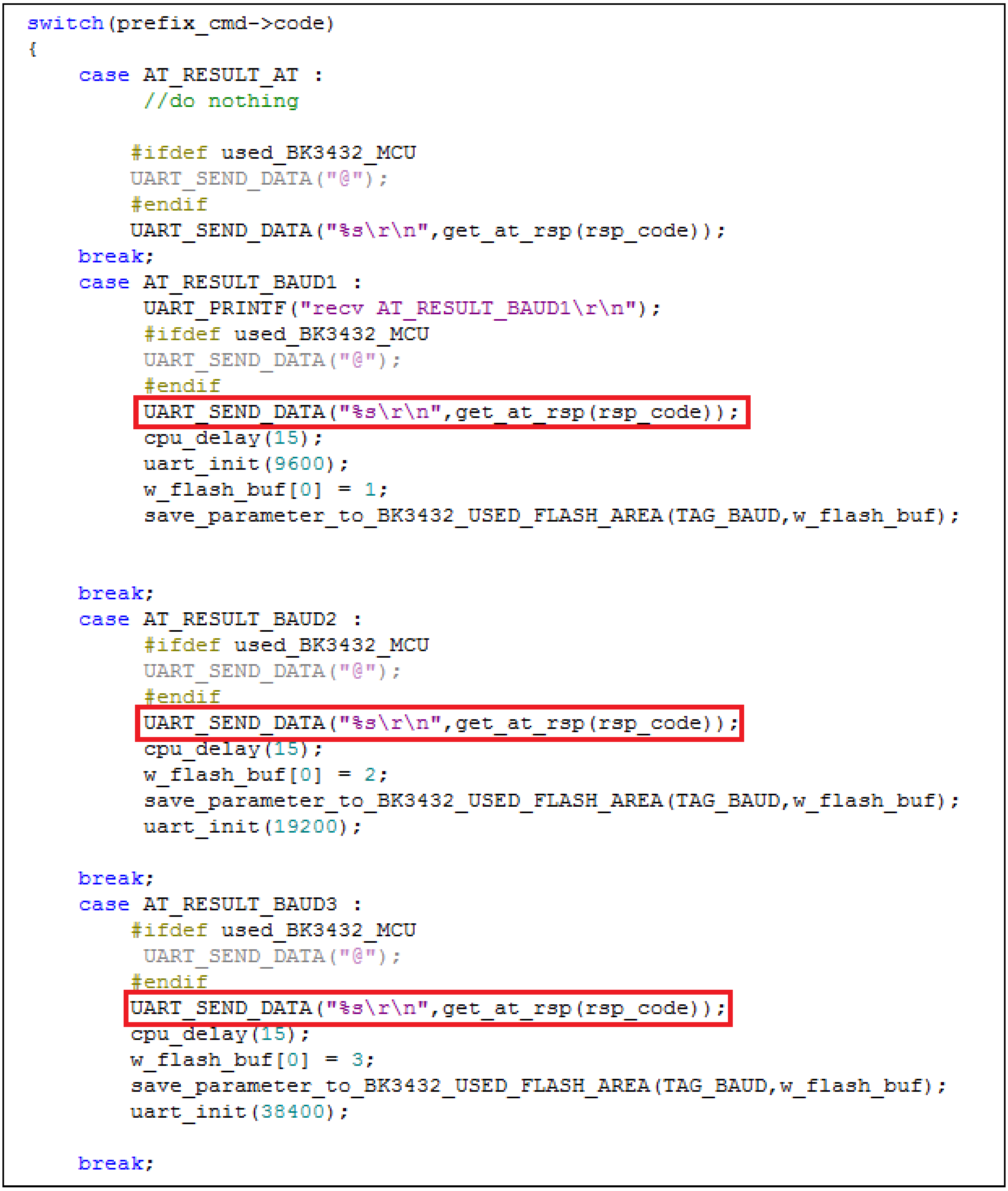


```
while(1)
{
  //schedule all pending events
  rwip_schedule();
  app_user_entry();

  // Checks for sleep have to be done with interrupt disabled
  GLOBAL_INT_DISABLE();

  oad_updating_user_section_pro();

  if(wdt_disable_flag==1)
  {
    wdt_disable();
  }
```

Figure 28. Decode received data

```
if(uart_rx_done == 1)
{
    // uint8_t baud_change = 0;
    uint8_t len;
    uint8_t rsp_code;
    //uint8_t idx;
    extern uint8_t rxdata_buffer_len;
    at_prefix_t *prefix_cmd;
    uint8_t w_flash_buf[2];
    //len = strlen((char*)rxdata_buffer);
    len = rxdata_buffer_len;
    rxdata_buffer_len = 0;
    if(rxdata_buffer[len-1] == '\n')
    {
        //AT command finish
        //UART_PRINTF("finish\r\n");
        memcpy(&AT_cmd_buf[recv_AT_cmd_idx],rxdata_buffer,len);
        //UART_PRINTF("%s\r\n",AT_cmd_buf);
        AT_cmd_len += len;
        recv_AT_cmd_idx = 0;
    }
    else
    {
        //command not finish
        memcpy(&AT_cmd_buf[recv_AT_cmd_idx],rxdata_buffer,len);
        recv_AT_cmd_idx = len;
        AT_cmd_len += len;
        uart_rx_done = 0;
        //UART_PRINTF("not finish\r\n");
        return;

    }

    //dispatch AT-COMMAND
    prefix_cmd = at_result_to_prefix((char*)AT_cmd_buf, AT_cmd_len);
    uart_rx_done = 0;
    without_prefix_len = AT_cmd_len-prefix_cmd->prefix_len;
```

Figure 29. Select corresponding case, execute event and respond

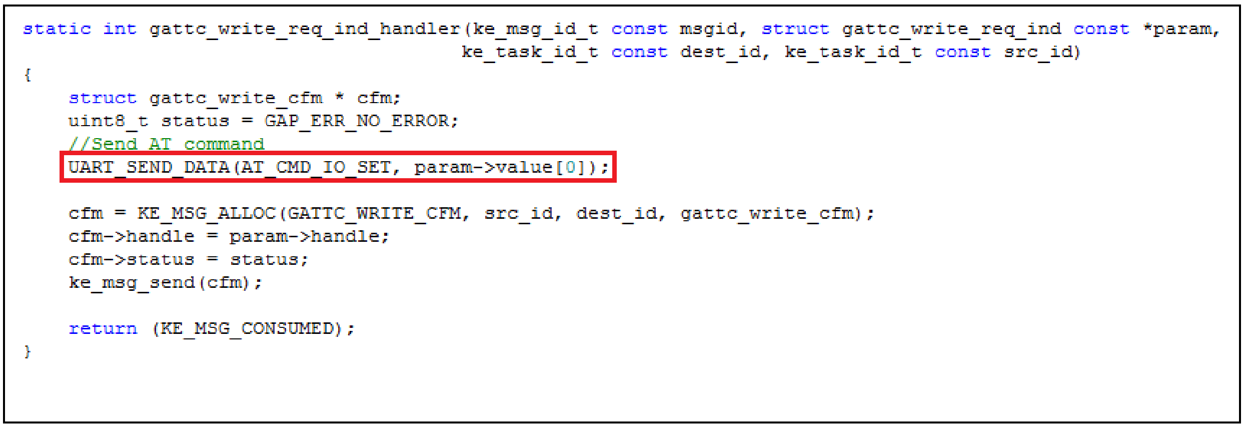


```
switch(prefix_cmd->code)
{
    case AT_RESULT_AT :
         //do nothing

        #ifdef used_BK3432_MCU
        UART_SEND_DATA("@");
        #endif
        UART_SEND_DATA("%s\r\n",get_at_rsp(rsp_code));
    break;
    case AT_RESULT_BAUD1 :
         UART_PRINTF("recv AT_RESULT_BAUD1\r\n");
         #ifdef used_BK3432_MCU
         UART_SEND_DATA("@");
         #endif
         UART_SEND_DATA("%s\r\n",get_at_rsp(rsp_code));
         cpu_delay(15);
         uart_init(9600);
         w_flash_buf[0] = 1;
         save_parameter_to_BK3432_USED_FLASH_AREA(TAG_BAUD,w_flash_buf);


    break;
    case AT_RESULT_BAUD2 :
         #ifdef used_BK3432_MCU
         UART_SEND_DATA("@");
         #endif
         UART_SEND_DATA("%s\r\n",get_at_rsp(rsp_code));
         cpu_delay(15);
         w_flash_buf[0] = 2;
         save_parameter_to_BK3432_USED_FLASH_AREA(TAG_BAUD,w_flash_buf);
         uart_init(19200);

    break;
    case AT_RESULT_BAUD3 :
        #ifdef used_BK3432_MCU
         UART_SEND_DATA("@");
        #endif
        UART_SEND_DATA("%s\r\n",get_at_rsp(rsp_code));
        cpu_delay(15);
        w_flash_buf[0] = 3;
        save_parameter_to_BK3432_USED_FLASH_AREA(TAG_BAUD,w_flash_buf);
        uart_init(38400);

    break;
```

4.2.3 BLE sends requests

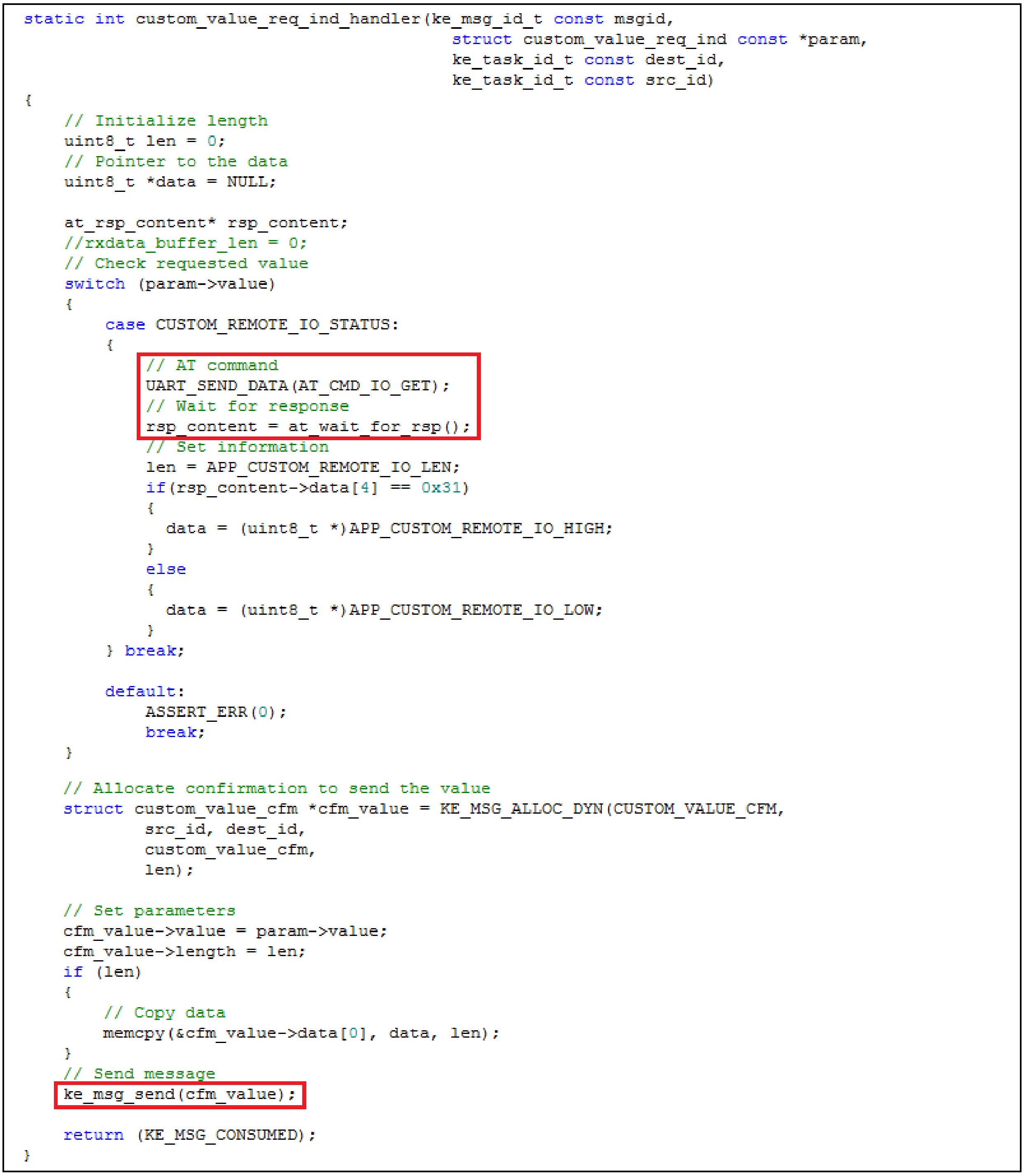
The parts that send a request are added according to the implementation of characteristic. In this application, they are read remote IO level and write remote IO level. Both parts send AT command to MCU through the UART_SEND_DATA() function. The “write” in this application is set as “Write without response” in Profile, so there is no need to wait for the response from MCU. As for “read” in this application, the value should be added to the response of GATT, so users must wait for the MCU to respond. In the code, the at_wait_for_rsp() function is used, and wait to obtain the responded data. After obtaining the data from MCU, send the data to the smartphone through the ke_msg_send() function.

Figure 30. Send write IO command



```
static int gattc_write_req_ind_handler(ke_msg_id_t const msgid, struct gattc_write_req_ind const *param,
                                       ke_task_id_t const dest_id, ke_task_id_t const src_id)
{
    struct gattc_write_cfm * cfm;
    uint8_t status = GAP_ERR_NO_ERROR;
    //Send AT command
    UART_SEND_DATA(AT_CMD_IO_SET, param->value[0]);

    cfm = KE_MSG_ALLOC(GATTC_WRITE_CFM, src_id, dest_id, gattc_write_cfm);
    cfm->handle = param->handle;
    cfm->status = status;
    ke_msg_send(cfm);

    return (KE_MSG_CONSUMED);
}
```

Figure 31. Send read IO command and send back data



```
static int custom_value_req_ind_handler(ke_msg_id_t const msgid,
                                         struct custom_value_req_ind const *param,
                                         ke_task_id_t const dest_id,
                                         ke_task_id_t const src_id)
{
    // Initialize length
    uint8_t len = 0;
    // Pointer to the data
    uint8_t *data = NULL;

    at_rsp_content* rsp_content;
    //rxdata_buffer_len = 0;
    // Check requested value
    switch (param->value)
    {
        case CUSTOM_REMOTE_IO_STATUS:
        {
            // AT command
            UART_SEND_DATA(AT_CMD_IO_GET);
            // Wait for response
            rsp_content = at_wait_for_rsp();
            // Set information
            len = APP_CUSTOM_REMOTE_IO_LEN;
            if(rsp_content->data[4] == 0x31)
            {
              data = (uint8_t *)APP_CUSTOM_REMOTE_IO_HIGH;
            }
            else
            {
              data = (uint8_t *)APP_CUSTOM_REMOTE_IO_LOW;
            }
        } break;

        default:
            ASSERT_ERR(0);
            break;
    }

    // Allocate confirmation to send the value
    struct custom_value_cfm *cfm_value = KE_MSG_ALLOC_DYN(CUSTOM_VALUE_CFM,
            src_id, dest_id,
            custom_value_cfm,
            len);

    // Set parameters
    cfm_value->value = param->value;
    cfm_value->length = len;
    if (len)
    {
        // Copy data
        memcpy(&cfm_value->data[0], data, len);
    }
    // Send message
    ke_msg_send(cfm_value);

    return (KE_MSG_CONSUMED);
}
```

4.2.4 Software download

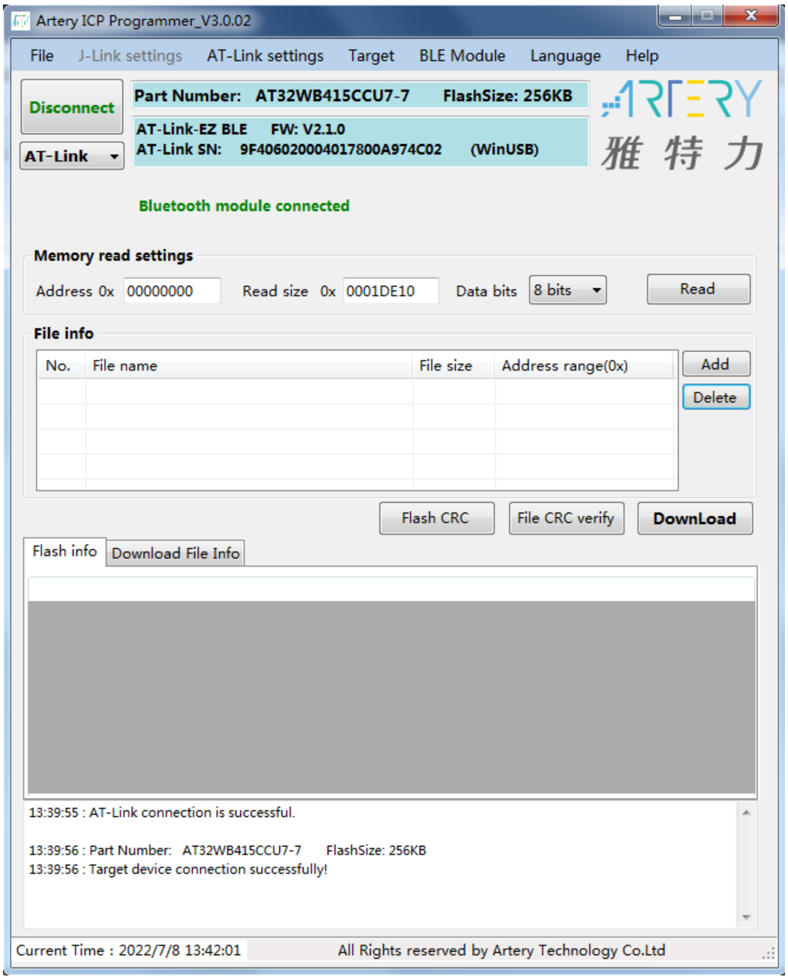
After compiling the code of Bluetooth and MCU, download software to WB415 board through ICP Tool. Users need to import wb415_ble_app_merge.bin (BLE side code) and Template.hex (MCU side code). The download process is as follows:

1. Connect AT-Link to PC via USB.
2. Open the host computer software Artery ICP Programmer Tool and connect to the AT32WB415 chip.
3. Select BLE side code. Click the “Add” button in “File info” and select files to be downloaded. The default path after BLE side code compilation is “output->app” of the project. Select

wb415_ble_app_merge.bin, and enter the download start address “0x00000000”.

4. Select MCU side code. Click the “Add” button in “File info” and select files to be downloaded. The default path after MCU side code compilation is the “Objects” folder of the project. Then, select Template.hex.
5. Click to download.
6. After the download is complete, the host computer software will prompt download & verification completion.

Figure 32. Host computer software connects to AT32WB415 chip



Part Number: AT32WB415CCU7-7 FlashSize: 256KB

AT-Link-EZ BLE FW: V2.1.0

AT-Link SN: 9F406020004017800A974C02 (WinUSB)

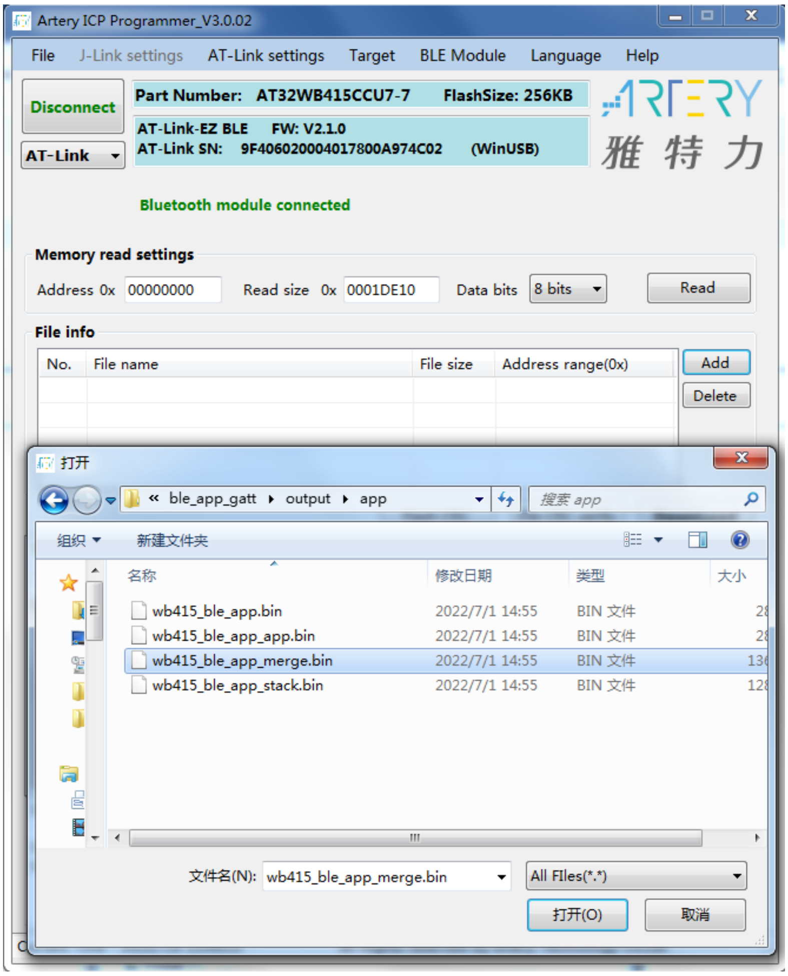
Bluetooth module connected

Address 0x 00000000 Read size 0x 0001DE10 Data bits 8 bits

```
13:39:55 : AT-Link connection is successful.

13:39:56 : Part Number:  AT32WB415CCU7-7     FlashSize: 256KB
13:39:56 : Target device connection successfully!
```

Current Time : 2022/7/8 13:42:01 All Rights reserved by Artery Technology Co.Ltd

Figure 33. Add BLE files



Artery ICP Programmer_V3.0.02

Part Number: AT32WB415CCU7-7 FlashSize: 256KB

AT-Link-EZ BLE FW: V2.1.0

AT-Link SN: 9F406020004017800A974C02 (WinUSB)

Bluetooth module connected

Memory read settings

Address 0x 00000000 Read size 0x 0001DE10 Data bits 8 bits

File info

No.	File name	File size	Address range(0x)

打开

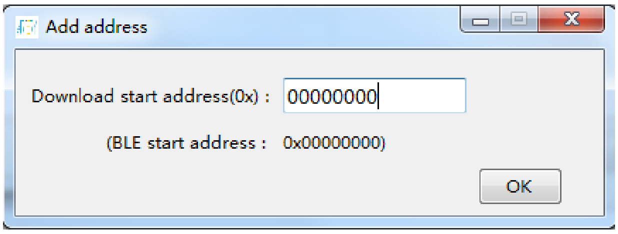
« ble_app_gatt ▸ output ▸ app

名称	修改日期	类型
wb415_ble_app.bin	2022/7/1 14:55	BIN 文件
wb415_ble_app_app.bin	2022/7/1 14:55	BIN 文件
wb415_ble_app_merge.bin	2022/7/1 14:55	BIN 文件
wb415_ble_app_stack.bin	2022/7/1 14:55	BIN 文件

文件名(N): wb415_ble_app_merge.bin

All Files(*.*)

Figure 34. Modify BLE download start address

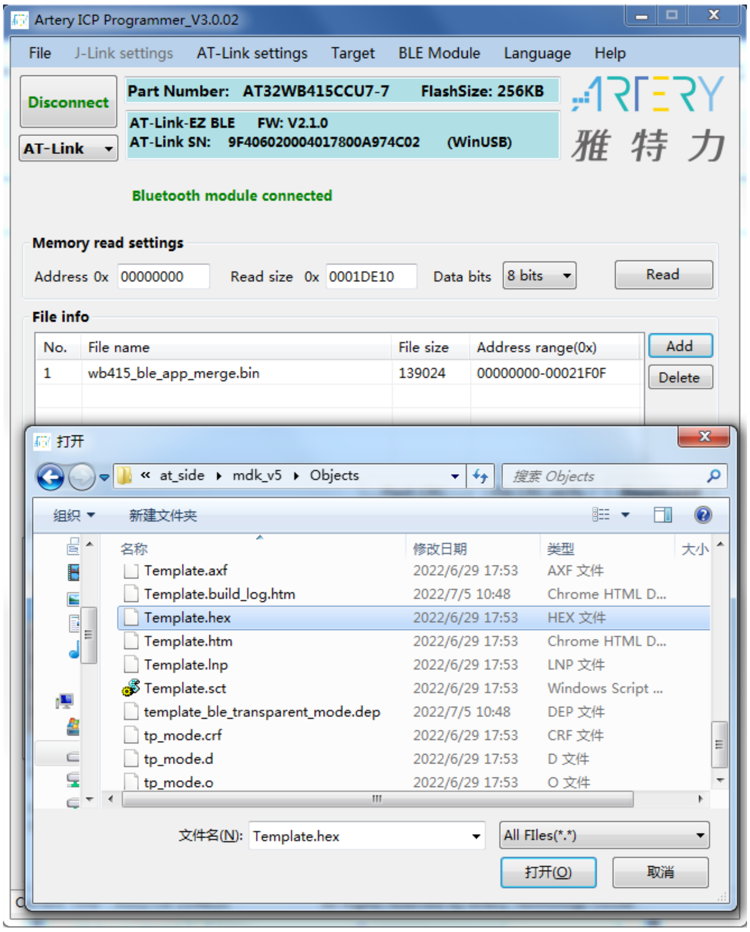


Add address

Download start address(0x) : 00000000

(BLE start address : 0x00000000)

Figure 35. Add MCU files



Part Number: AT32WB415CCU7-7 FlashSize: 256KB

AT-Link-EZ BLE FW: V2.1.0

AT-Link SN: 9F406020004017800A974C02 (WinUSB)

Bluetooth module connected

Memory read settings

Address 0x 00000000 Read size 0x 0001DE10 Data bits 8 bits

File info

No.	File name	File size	Address range(0x)
1	wb415_ble_app_merge.bin	139024	00000000-00021F0F

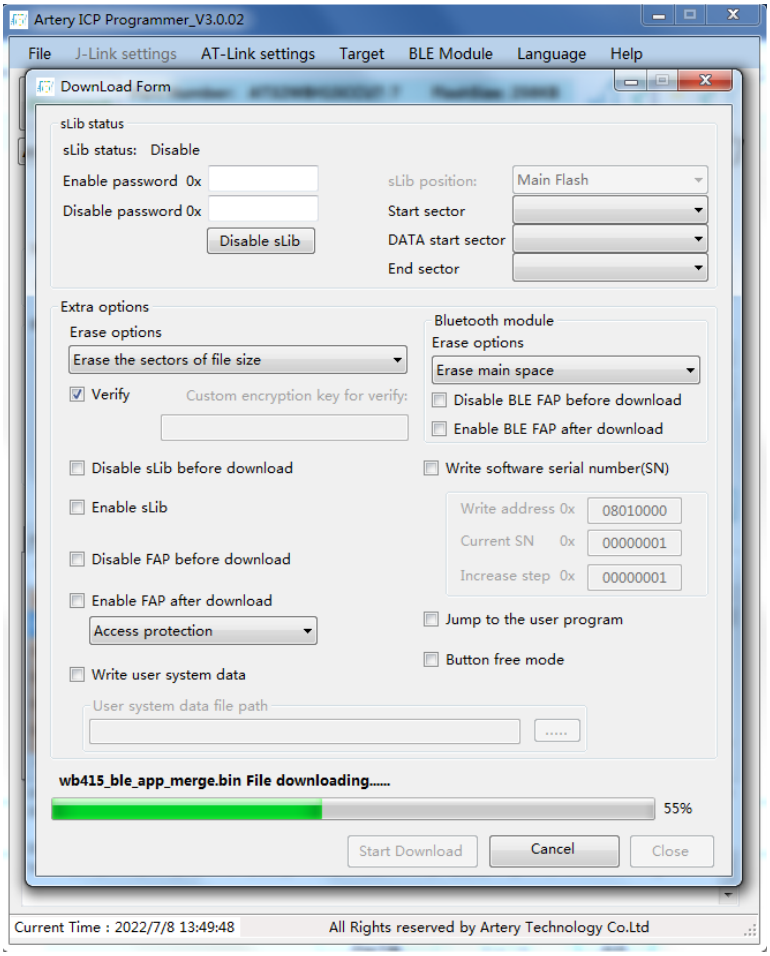
« at_side ▸ mdk_v5 ▸ Objects

搜索 Objects

名称	修改日期	类型	大小
Template.axf	2022/6/29 17:53	AXF 文件	
Template.build_log.htm	2022/7/5 10:48	Chrome HTML D...	
Template.hex	2022/6/29 17:53	HEX 文件	
Template.htm	2022/6/29 17:53	Chrome HTML D...	
Template.lnp	2022/6/29 17:53	LNP 文件	
Template.sct	2022/6/29 17:53	Windows Script ...	
template_ble_transparent_mode.dep	2022/7/5 10:48	DEP 文件	
tp_mode.crf	2022/6/29 17:53	CRF 文件	
tp_mode.d	2022/6/29 17:53	D 文件	
tp_mode.o	2022/6/29 17:53	O 文件	

文件名(N): Template.hex All Files(*.*)

Figure 36. Click to download



sLib status

sLib status: Disable

Enable password 0x

Disable password 0x

sLib position: Main Flash

Start sector

DATA start sector

End sector

Extra options

Erase options

Erase the sectors of file size

☑ Verify

Custom encryption key for verify:

☐ Disable sLib before download

☐ Enable sLib

☐ Disable FAP before download

☐ Enable FAP after download

Access protection

☐ Write user system data

User system data file path

Bluetooth module

Erase options

Erase main space

☐ Disable BLE FAP before download

☐ Enable BLE FAP after download

☐ Write software serial number(SN)

Write address 0x 08010000

Current SN 0x 00000001

Increase step 0x 00000001

☐ Jump to the user program

☐ Button free mode

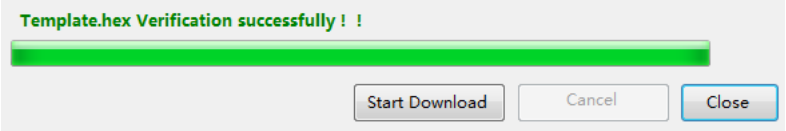
wb415_ble_app_merge.bin File downloading......

55%

Current Time : 2022/7/8 13:49:48

All Rights reserved by Artery Technology Co.Ltd

Figure 37. Download & verification completion



Template.hex Verification successfully！！

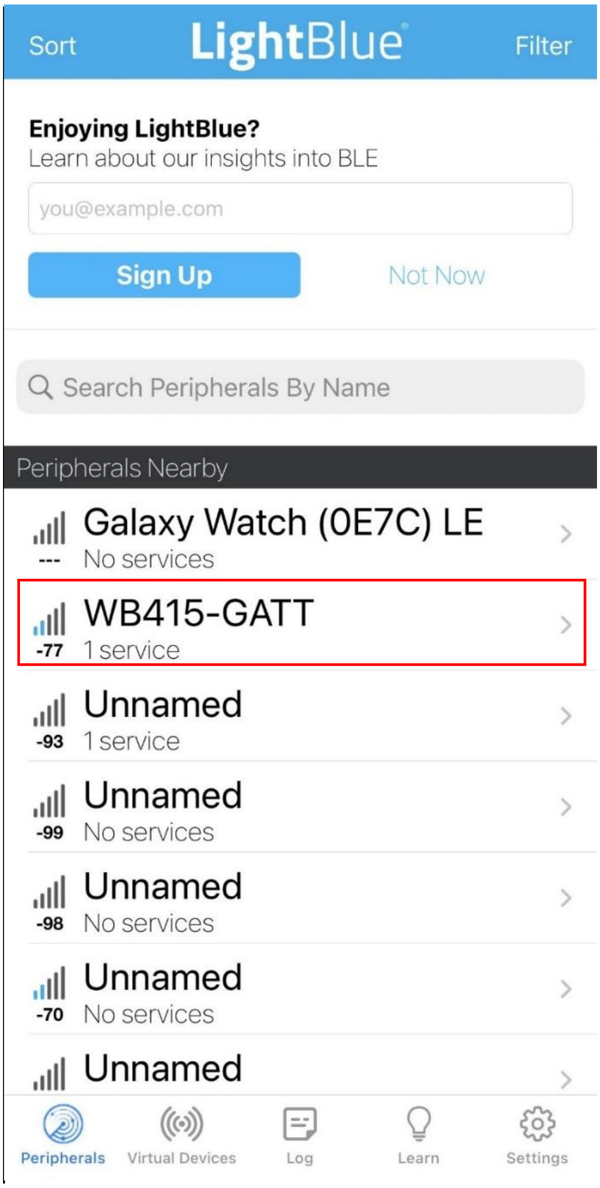
4.3 AT command mode

It is recommended to install a Bluetooth tool/software with Bluetooth device operation function on the smartphone. This application note takes LightBlue APP as an example.

Perform the following steps to verify that the AT command mode in this application works properly.

1. Open LightBlue APP and find the Bluetooth device called WB415-GATT and then connect to it.

Figure 38. Search WB415-GATT



Enjoying LightBlue?
Learn about our insights into BLE

Peripherals Nearby

---	Galaxy Watch (0E7C) LE	No services
-77	WB415-GATT	1 service
-93	Unnamed	1 service
-99	Unnamed	No services
-98	Unnamed	No services
-70	Unnamed	No services
	Unnamed	

2. Check to confirm that it is connected. Click UUID:C0DE and confirm that the 0xC101 is set. The 0xC101 is the service and characteristic used in AT command mode.

Figure 39. Connection status and 0xC101 characteristics

WB415-GATT

UUID: B4F39B81-64B6-8854-58EC-AD321CCE989C

Connected

ADVERTISEMENT DATA

UUID: F000

0xF002
Properties: Write Without Response

0xF001
Properties: Notify

UUID: C0DE

0xC101
Properties: Read Write Without Response

Device Information

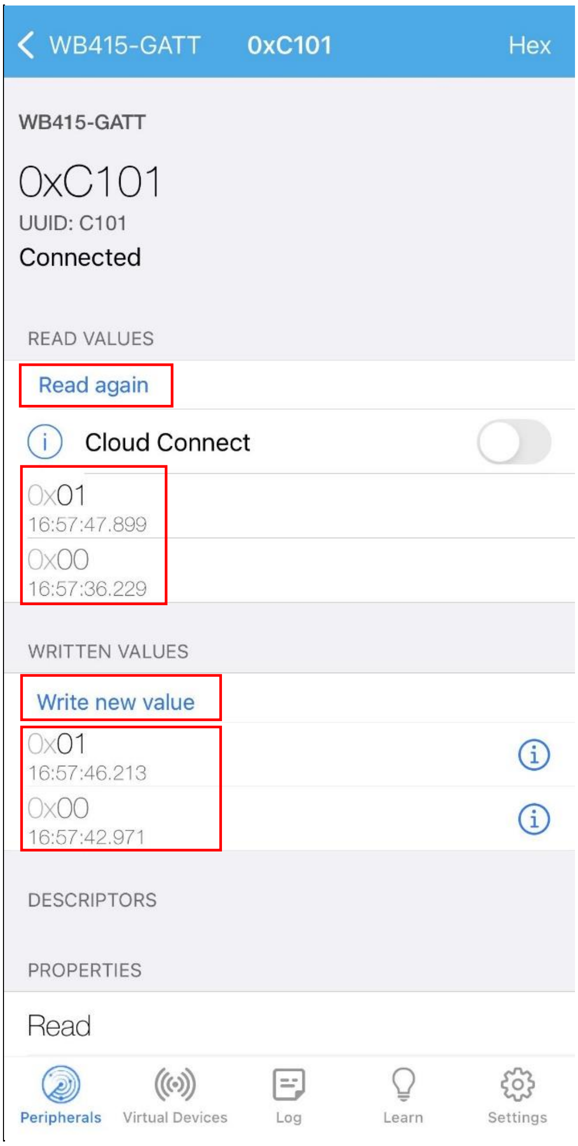
Manufacturer Name String
BEKEN SAS

Model Number String
BK-BLE-1.0

Serial Number String

3. There are two available functions, i.e., READ VALUES and WRITTEN VALUES.
4. Click “Read again” to obtain IO status data, and the returned data is 0x00 or 0x01 (represents LED off or LED on) to indicate IO high level or IO low level.
5. Click “Write new value”, and write 0 or 1 to configure IO low level or IO high level. Two statuses of LED2 can be seen on AT-START-WB415 board. LED2 is on when the circuit is low-level.

Figure 40. Read/write IO data



WB415-GATT 0xC101 Hex

WB415-GATT

0xC101

UUID: C101

Connected

READ VALUES

Read again

Cloud Connect

0x01
16:57:47.899

0x00
16:57:36.229

WRITTEN VALUES

Write new value

0x01
16:57:46.213

0x00
16:57:42.971

DESCRIPTORS

PROPERTIES

Read

4.4 Transparent mode

The transparent command simplifies development process, so users do not need to implement services and characteristics but only focus on application development on the MCU side. Other required functions can be realized by defining the format of transparent data. In transparent mode, no “CR + LF” is added at the end of each data. This application note introduces two interfaces, i.e., WB415 USART2 used for connection with mobile app, and USB interface used for custom HID.

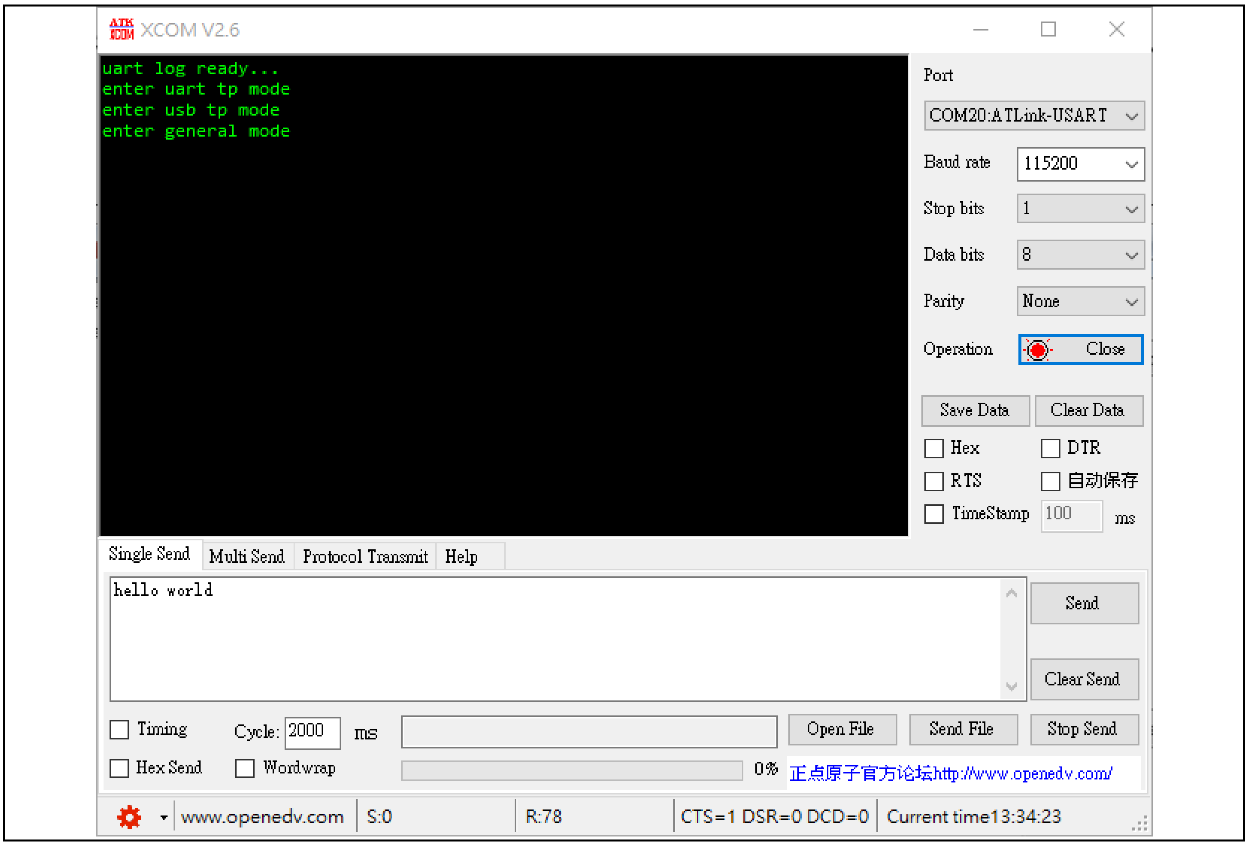
4.4.1 UART interface

1. Use the USER key on AT-START-WB415 to switch AT command mode and transparent mode (UART and USB). The current mode information is print out through USART2_TX(PA2). LED3 indicates the current mode. In AT command mode, LED3 is off; in transparent mode, LED3 is on.

Note: Transparent mode conflicts with AT command mode, which means that custom services are unavailable in transparent mode.

2. Press USER key on WB415 to enter transparent mode, and LED3 is on; or confirm whether the current mode is transparent mode according to the message print out via USART2.

Figure 41. Switch to transparent mode



XCOM V2.6

```
uart log ready...
enter uart tp mode
enter usb tp mode
enter general mode
```

Port: COM20:ATLink-USART

Baud rate: 115200

Stop bits: 1

Data bits: 8

Parity: None

Operation: ● Close

☐ Hex ☐ DTR

☐ RTS ☐ 自动保存

☐ TimeStamp 100 ms

hello world

☐ Timing Cycle: 2000 ms

☐ Hex Send ☐ Wordwrap 0%

正点原子官方论坛http://www.openedv.com/

www.openedv.com S:0 R:78 CTS=1 DSR=0 DCD=0 Current time13:34:23

3. Use LightBlue to connect to WB415, and find the F000 service that includes F001 and F002 characteristics. The service and characteristics are used in transparent mode.

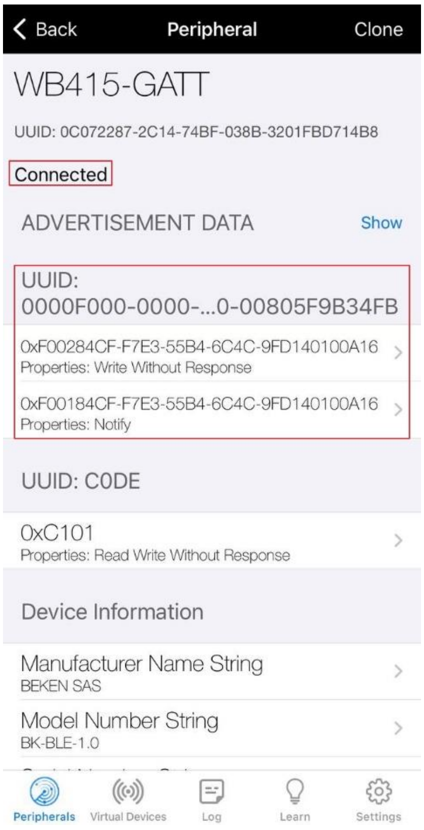


Figure 42. LightBlue connects to WB415

WB415-GATT

UUID: 0C072287-2C14-74BF-038B-3201FBD714B8

Connected

ADVERTISEMENT DATA

UUID:
0000F000-0000-...0-00805F9B34FB

0xF00284CF-F7E3-55B4-6C4C-9FD140100A16
Properties: Write Without Response

0xF00184CF-F7E3-55B4-6C4C-9FD140100A16
Properties: Notify

UUID: C0DE

0xC101
Properties: Read Write Without Response

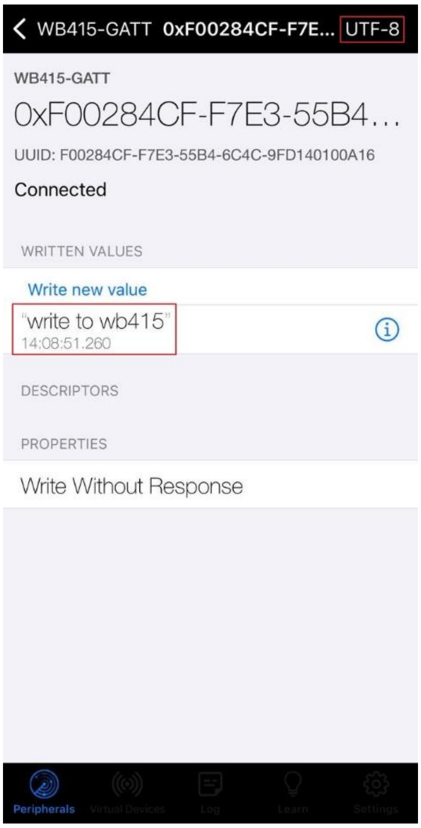
Device Information

Manufacturer Name String
BEKEN SAS

Model Number String
BK-BLE-1.0

4. Transmit data to MCU through transparent mode: Enter 0xF002 and switch the data mode to UTF-8 String in the upper right. Click “Write new value” to input any string, and the string will be output to the serial port assistant through USART2 TX of WB415.

Figure 43. LightBlue write data



WB415-GATT 0xF00284CF-F7E... UTF-8

WB415-GATT

0xF00284CF-F7E3-55B4...

UUID: F00284CF-F7E3-55B4-6C4C-9FD140100A16

Connected

WRITTEN VALUES

Write new value

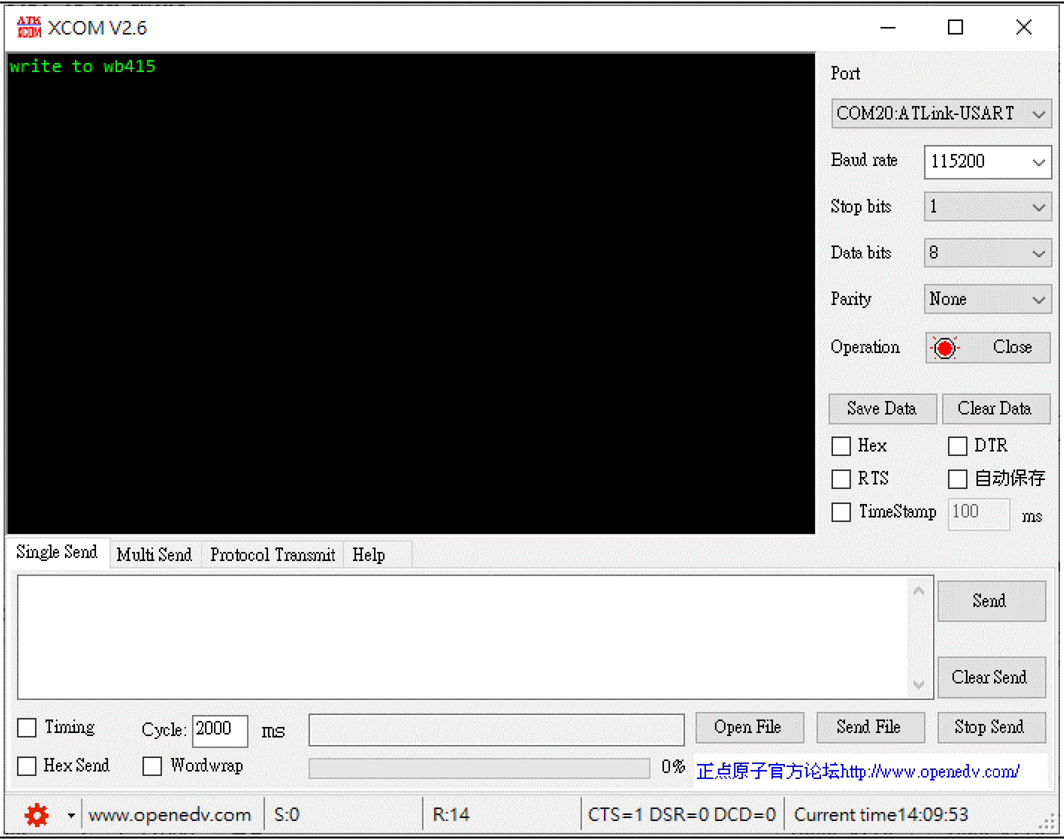
"write to wb415"
14:08:51.260

DESCRIPTORS

PROPERTIES

Write Without Response

Figure 44. WB415 prints the received data

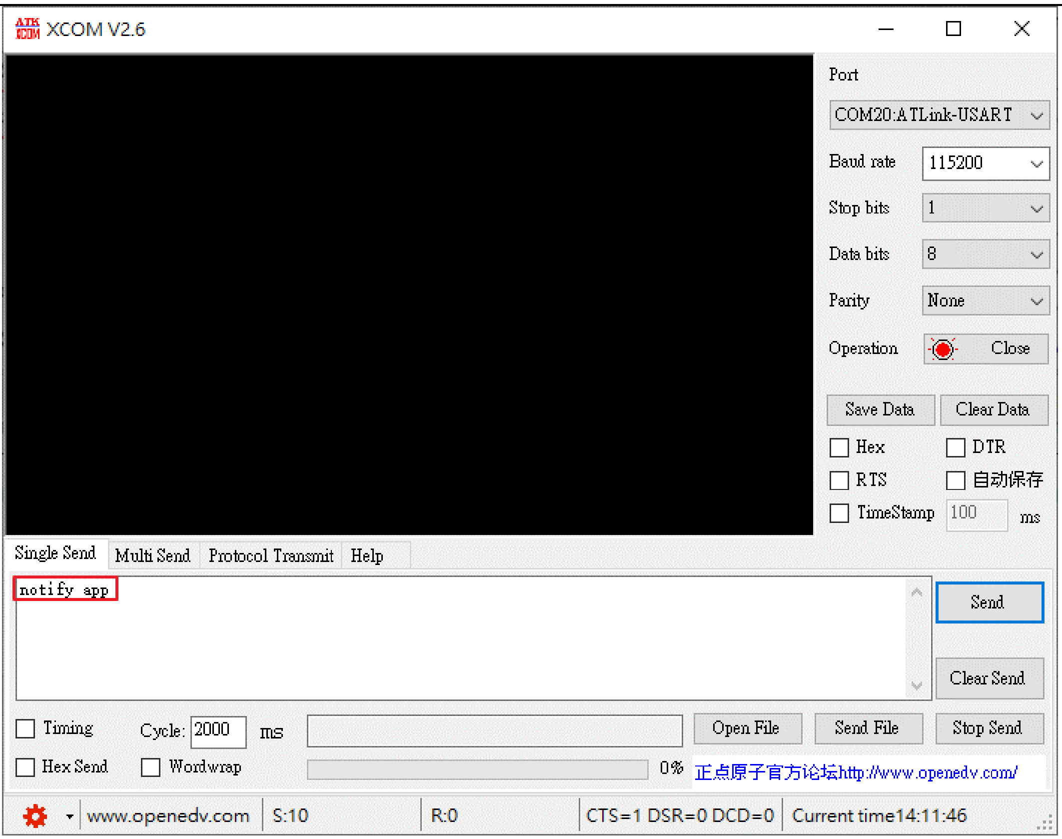


```
write to wb415
```

Port: COM20:ATLink-USART
Baud rate: 115200
Stop bits: 1
Data bits: 8
Parity: None
Operation: Close
☐ Hex ☐ DTR
☐ RTS ☐ 自动保存
☐ TimeStamp 100 ms
☐ Timing Cycle: 2000 ms
☐ Hex Send ☐ Wordwrap 0%
www.openedv.com S:0 R:14 CTS=1 DSR=0 DCD=0 Current time14:09:53

5. Transmit data to a smartphone through transparent mode: Enter 0xF001 and click “Listen for notifications”, and the first data to be received is “Notification Start”. Then, click “Send” after the serial port assistant has printed all strings, and 0xF001 will display these strings.

Figure 45. Input data to WB415

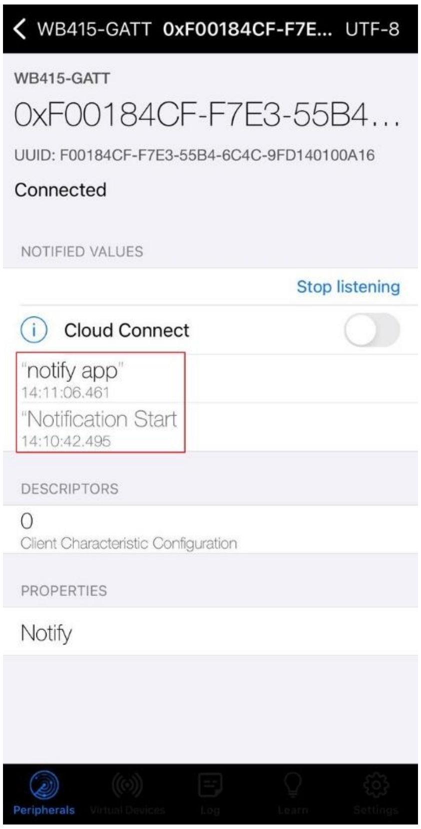


Port: COM20:ATLink-USART
Baud rate: 115200
Stop bits: 1
Data bits: 8
Parity: None
Operation: Close
☐ Hex ☐ DTR
☐ RTS ☐ 自动保存
☐ TimeStamp 100 ms

```
notify app
```

☐ Timing Cycle: 2000 ms
☐ Hex Send ☐ Wordwrap 0%
www.openedv.com S:10 R:0 CTS=1 DSR=0 DCD=0 Current time14:11:46

Figure 46. LightBlue receives data from WB415



WB415-GATT 0xF00184CF-F7E... UTF-8

WB415-GATT

0xF00184CF-F7E3-55B4...

UUID: F00184CF-F7E3-55B4-6C4C-9FD140100A16

Connected

NOTIFIED VALUES

Stop listening

Cloud Connect

"notify app"
14:11:06.461

"Notification Start
14:10:42.495

DESCRIPTORS

0
Client Characteristic Configuration

PROPERTIES

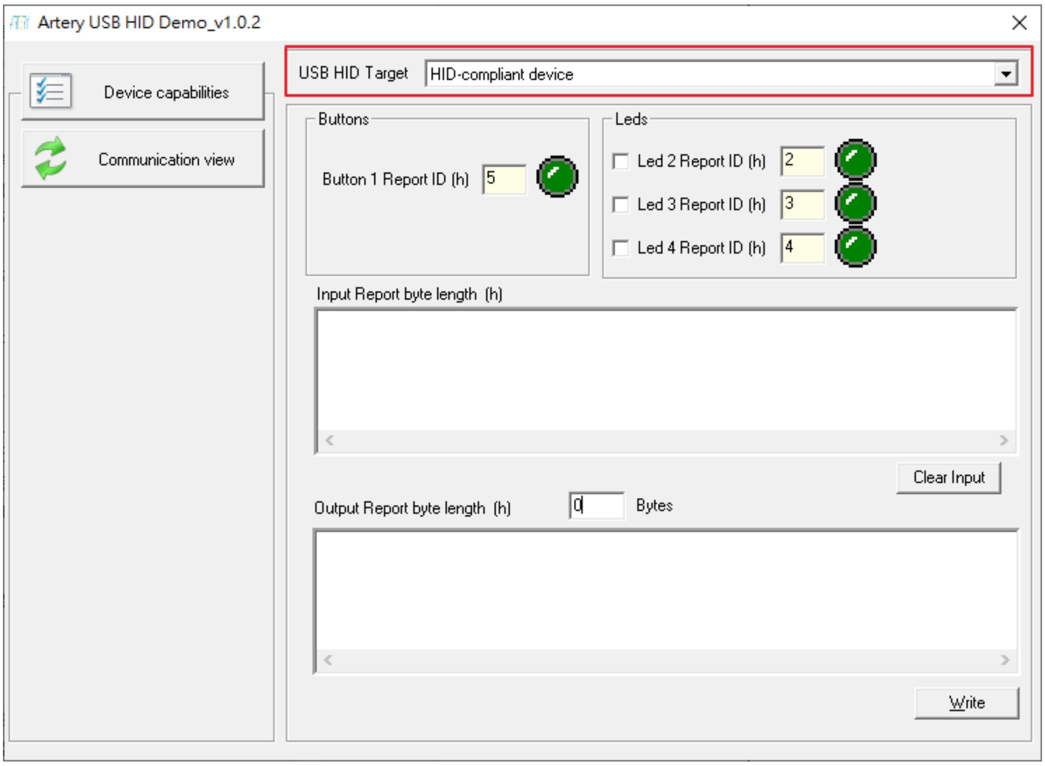
Notify

4.4.2 USB interface

The demo of transparent mode via USB is based on the custom HID demo in BSP; refer to AN0097 for details. Exactly the same as transparent mode via UART on the BT end, it uses Artery USB HID Demo host computer to connect to WB415, and the transmit and receive data also have 0xF001 and 0xF002 features. The process is as below:

1. Switch WB415 to USB transparent mode; press the USER key to switch among UART, USB and General mode.
2. Connect USB cable to WB415, open the host computer and select USB HID target.

Figure 47. Select USB HID Target



USB HID Target: HID-compliant device

Button 1 Report ID (h): 5

Led 2 Report ID (h): 2

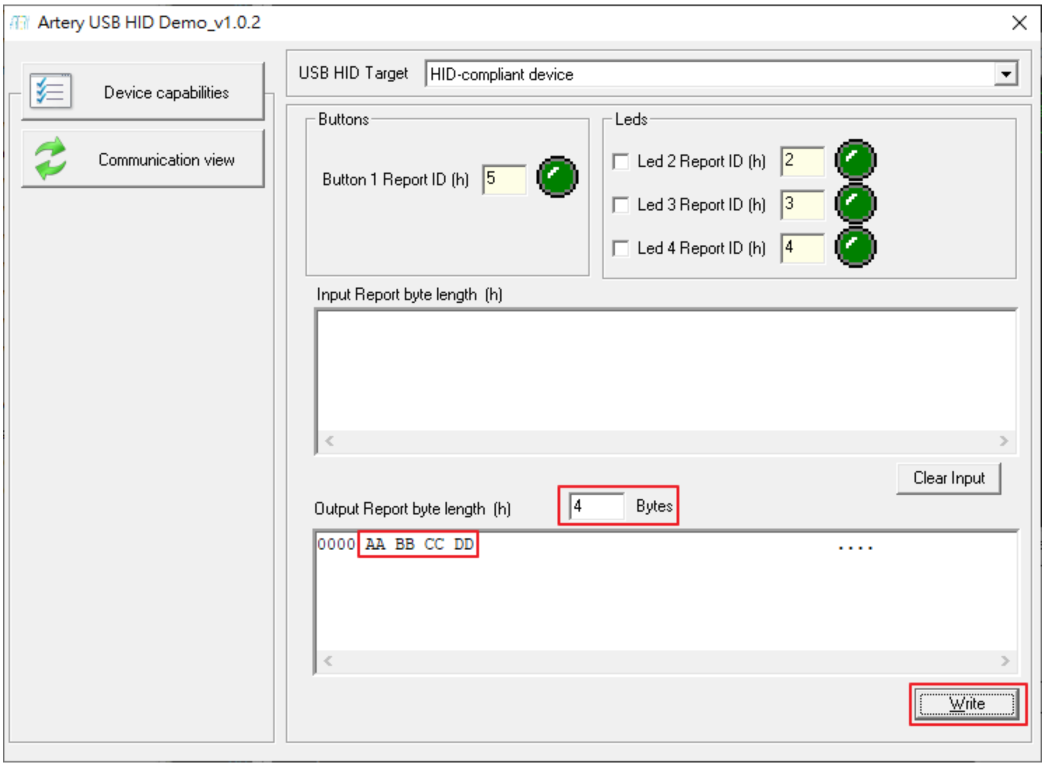
Led 3 Report ID (h): 3

Led 4 Report ID (h): 4

Output Report byte length (h): 0 Bytes

3. Fill in the data length before sending data to APP; then click "Write" and check 0xF001 on mobile APP to view the received data.

Figure 48. Fill in data length



USB HID Target: HID-compliant device

Button 1 Report ID (h): 5

Led 2 Report ID (h): 2

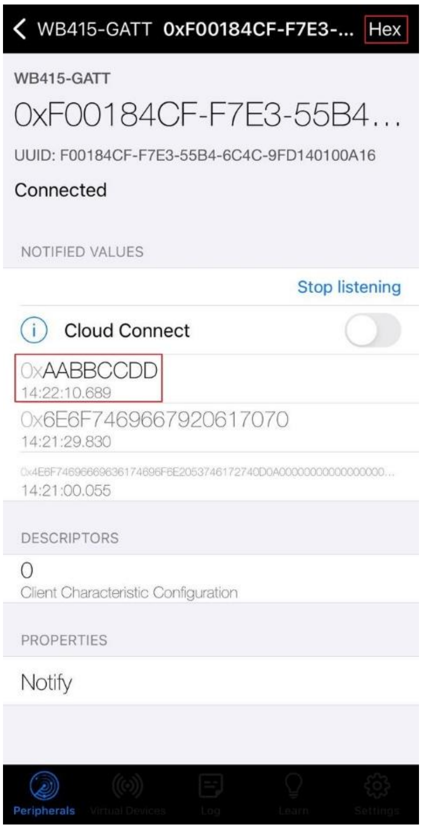
Led 3 Report ID (h): 3

Led 4 Report ID (h): 4

Output Report byte length (h): 4 Bytes

```
0000 AA BB CC DD                                   ....
```

Figure 49. Received data on mobile APP


WB415-GATT 0xF00184CF-F7E3-...

WB415-GATT

0xF00184CF-F7E3-55B4...

UUID: F00184CF-F7E3-55B4-6C4C-9FD140100A16

Connected

NOTIFIED VALUES

0xAABBCCDD
14:22:10.689

0x6E6F7469667920617070
14:21:29.830

0x4E6F74696669636174696F6E2053746172740D0A000000000000000000...
14:21:00.055

DESCRIPTORS

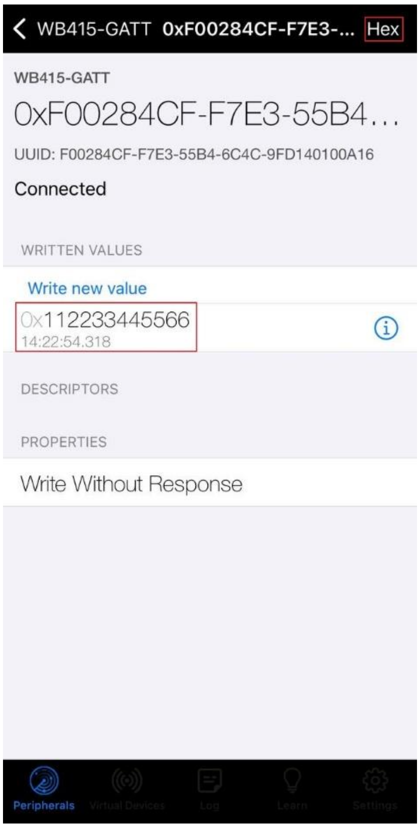
0
Client Characteristic Configuration

PROPERTIES

Notify

4. The same as UART transparent mode, the mobile APP sends data to USB host computer; then find the 0xF002 and write data. The data sent from mobile is displayed in the “Input Report” on the USB host computer.

Figure50. Send data to USB host computer


WB415-GATT 0xF00284CF-F7E3-...

WB415-GATT

0xF00284CF-F7E3-55B4...

UUID: F00284CF-F7E3-55B4-6C4C-9FD140100A16

Connected

WRITTEN VALUES

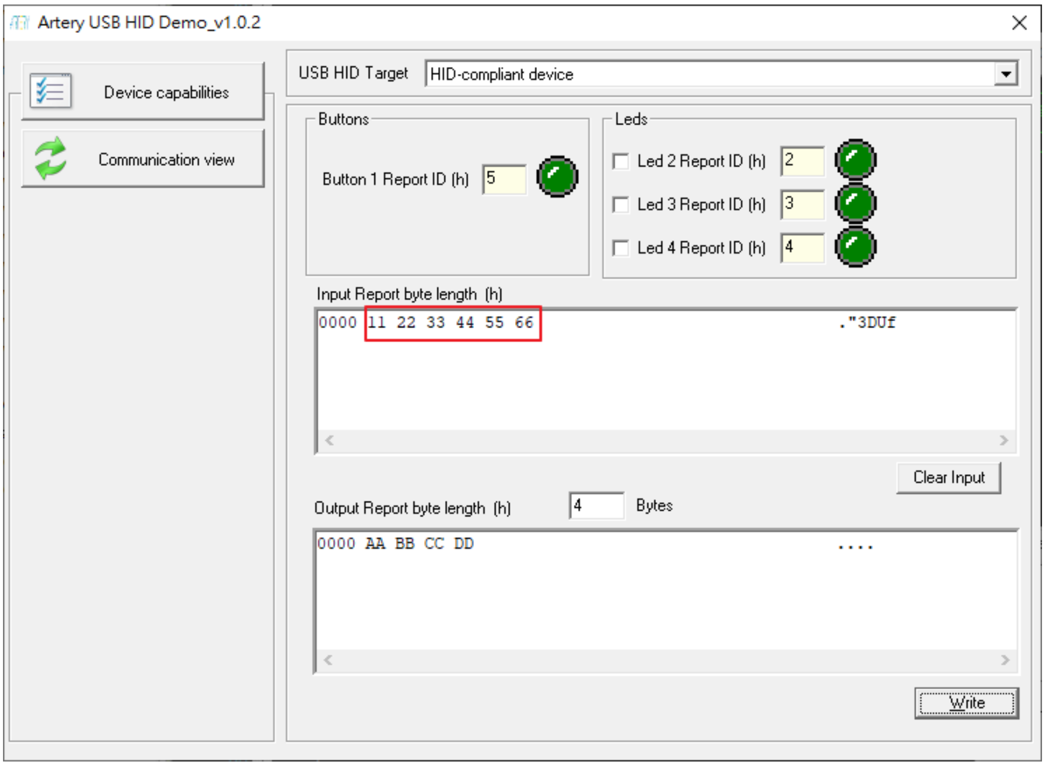
0x112233445566
14:22:54.318

DESCRIPTORS

PROPERTIES

Write Without Response

Figure 51. Received data in Input Report on the host computer



USB HID Target: HID-compliant device

Button 1 Report ID (h): 5

☐ Led 2 Report ID (h): 2

☐ Led 3 Report ID (h): 3

☐ Led 4 Report ID (h): 4

Input Report byte length (h)

```
0000 11 22 33 44 55 66                    ."3DUf
```

Output Report byte length (h): 4 Bytes

```
0000 AA BB CC DD                          ....
```

5 Revision history

Table 5. Document revision history

Date	Version	Revision note
2021.12.30	2.0.0	Initial release
2022.04.18	2.0.1	1. Use functions inside BSP to call LED ON or LED OFF1; 2. Modify the values returned in BLE Get IO state: “H” and “L” of ASCII changed to “1” and “0”.
2022.04.25	2.0.2	1. Update the photo of WB415 board; 2. Add instructions on MCU side code download.
2022.06.15	2.0.3	Add application cases of transparent mode.
2022.07.08	2.0.4	Update screenshots of ICP tool and XCOM interfaces.
2022.11.16	2.0.5	Added transparent mode via USB.

IMPORTANT NOTICE – PLEASE READ CAREFULLY

Purchasers understand and agree that purchasers are solely responsible for the selection and use of Artery's products and services.

Artery's products and services are provided "AS IS" and Artery provides no warranties express, implied or statutory, including, without limitation, any implied warranties of merchantability, satisfactory quality, non-infringement, or fitness for a particular purpose with respect to the Artery's products and services.

Notwithstanding anything to the contrary, purchasers acquires no right, title or interest in any Artery's products and services or any intellectual property rights embodied therein. In no event shall Artery's products and services provided be construed as (a) granting purchasers, expressly or by implication, estoppel or otherwise, a license to use third party's products and services; or (b) licensing the third parties' intellectual property rights; or (c) warranting the third party's products and services and its intellectual property rights.

Purchasers hereby agrees that Artery's products are not authorized for use as, and purchasers shall not integrate, promote, sell or otherwise transfer any Artery's product to any customer or end user for use as critical components in (a) any medical, life saving or life support device or system, or (b) any safety device or system in any automotive application and mechanism (including but not limited to automotive brake or airbag systems), or (c) any nuclear facilities, or (d) any air traffic control device, application or system, or (e) any weapons device, application or system, or (f) any other device, application or system where it is reasonably foreseeable that failure of the Artery's products as used in such device, application or system would lead to death, bodily injury or catastrophic property damage.

Any inconsistency of the sold ARTERY products with the statement and/or technical features specification described in this document will immediately cause the invalidity of any warranty granted by ARTERY products or services stated in this document by ARTERY, and ARTERY disclaims any responsibility in any form.

© 2022 ARTERY Technology – All Rights Reserved