Qinheng Low Power Bluetooth

Software Development Reference Manual

V1.8

January 3, 2024

Table of contents

Preface

This manual briefly introduces the software development of Qinheng low-power Bluetooth. It includes the software development platform, software The basic framework of the development and the low-power Bluetooth protocol stack. For easy understanding, this manual uses the CH58x chip as an example. This manual is for introduction. The software development of other low-power Bluetooth chips of our company can also refer to this manual.

CH58x is a RISC-V chip that integrates two independent full-speed USB host and device controllers and transceivers. 12-bit ADC, touch key detection module, RTC, power management, etc. For more information about CH58x, please refer to CH583DS1.PDF manual document.

1. Overview

1.1 Introduction

Bluetooth supports two wireless technologies since version 4.0:

Bluetooth Basic Rate/Enhanced Data Rate (commonly known as BR/EDR Classic Bluetooth)

Bluetooth Low Energy

The Bluetooth Low Energy protocol was created to transmit very small packets of data at a time, so it consumes significantly less power than Classic Bluetooth. decline.

Devices that support both classic Bluetooth and low-power Bluetooth are called dual-mode devices, such as mobile phones.

Devices that use the same 10-bit DAC are called single-mode devices. These devices are mainly used in low-power applications, such as applications powered by button batteries.

1.2 Basic Introduction to Bluetooth Low Energy Protocol Stack

The Bluetooth low energy protocol stack result is shown in Figure 1-1.

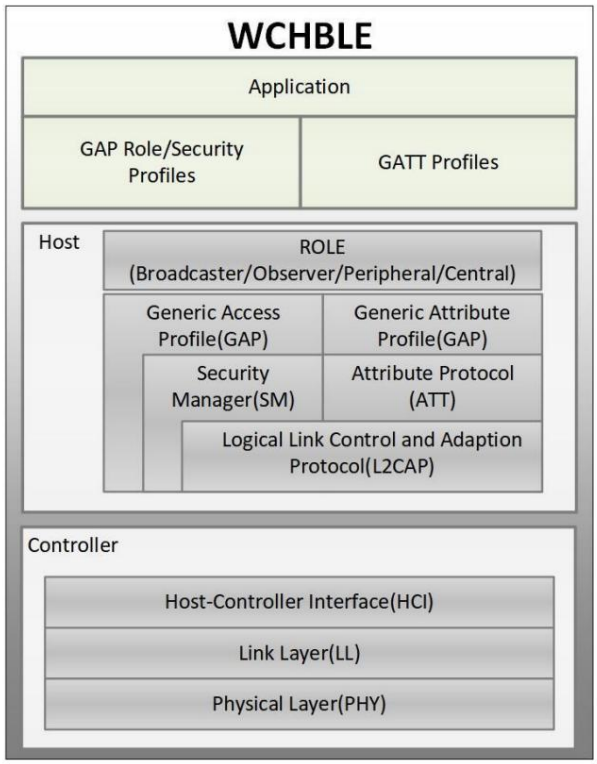


Figure 1-1

The protocol stack consists of Host (host protocol layer) and Controller (control protocol layer), and these two parts are generally are executed separately.

Configuration and application are in the Generic Access Profile (GAP) and Generic Attribute Profile (GATT) of the protocol stack implemented in the layer.

The physical layer (PHY) is the bottom layer of the BLE protocol stack. It specifies the basic radio frequency parameters of BLE communication, including signal frequency and rate, modulation scheme, etc.

The physical layer uses Gaussian frequency shift keying (GFSK) modulation in the 2.4GHz channel.

There are three implementation schemes for the physical layer of BLE 5.0, namely, 1Mbps uncoded physical layer, 2Mbps uncoded physical layer and 1Mbps coded physical layer. The 1Mbps uncoded physical layer is compatible with the physical layer of the BLE 4.0 series protocol, while the other two physical layers extend the communication rate and communication distance respectively.

The LinkLayer controls the device to be in one of the five states: standby, advertising, listening/scanning, initiating, and connected. Around these states, BLE devices can perform operations such as broadcasting and connecting. The LinkLayer defines the data packet format, timing specifications, and interface protocols in various states. The Generic Access Profile is the external interface layer for the internal functions of BLE devices. It

specifies three aspects: GAP roles, modes and procedures, and security issues. It mainly manages the broadcasting, connection, and device binding of Bluetooth devices. Broadcaster - a device that is always broadcasting and cannot be connected Observer - a device that can scan

for broadcasting devices, but cannot initiate a connection Slave - a

broadcasting device that can be connected and can act as a slave in a single link layer

connection Host - can scan for broadcasting devices and initiate connections, and act as a host in a single

link layer or multiple link layers

The Logical Link Control and Adaptation Protocol (LLCP) is an adapter between the host and the controller, providing data encapsulation services. It connects the application layer upward and the controller layer downward, so that the upper application operation does not need to care about the data details of the controller.

The Security Manager provides pairing and key distribution services to achieve secure connections and data exchange. The Attribute Transfer

Protocol defines the concept of attribute entities, including UUID, handle, and attribute value, and specifies the operation methods and details of attribute reading, writing, and notification. The Generic Attribute Profile

defines the service framework and protocol structure of using ATT.

The communication of application data between two devices is realized through the GATT layer of the protocol stack.

GATT Server - A device that provides data services to GATT Clients GATT Client -

A device that reads and writes application data from a GATT Server

2. Development Platform

2.1 Overview

CH58x is a 32-bit RISC-V microcontroller with integrated BLE wireless communication. It integrates 2Mbps low-power Bluetooth communication on chip.
module, two full-speed USB host and device controller transceivers, 2 SPI, RTC and other rich peripheral resources.
Taking the CH58x development platform as an example, other low-power Bluetooth chips of our company can also refer to this manual.

2.2 Configuration

CH58x is a true single-chip solution. The controller, host, configuration files and applications are all implemented on CH58x.
Please refer to the Central and Peripheral examples.

2.3 Software Overview

The software development kit includes the following six main components:

- TMOS
- HAL
- BLE Stack
- Profiles
- RISC-V Core
- The Application

package provides four GAP configuration files:

- Peripheral role
- Central role
- Broadcaster role
- Observer role

Some GATT profiles and applications are also provided.
Please refer to CH58xEVT software package for details.

3. Task Management System (TMOS)

3.1 Overview

The Bluetooth low energy protocol stack and applications are based on TMOS (Task Management Operating System). It is a control loop. Through TMOS, the execution mode of events can be set. TMOS is the scheduling core, BLE protocol stack, profile definition, and all applications are implemented around it. TMOS is not an operating system in the traditional sense, but a Implement a system resource management mechanism with multi-tasking as the core.

For a task, the unique task ID, task initialization, and the events that can be executed under the task are all or missing.

3.2 Task Initialization

First, to ensure that TMOS continues to run, TMOS_SystemProcess() needs to be executed in the last loop of main(). The initialization task needs to call the tmosTaskID TMOS_ProcessEventRegister(pTaskEventHandlerFn eventCb) function to register the event callback function in TMOS and generate a unique 8-bit task ID. After the service is initialized, the task ID increases in sequence, and the smaller the task ID, the higher the priority. The protocol stack task must have the highest priority. Priority.

```
1. halTaskID = TMOS_ProcessEventRegister( HAL_ProcessEvent );
```

3.3 Task events and event execution

TMOS is scheduled by polling. The system clock is generally derived from RTC and the unit is 625ÿs. The custom event (Event) is added to the task list of TMOS by registering a task (Task) , and then TMOS schedules and runs it. After the task is initialized, TMOS will poll the task event in a loop, and the event flag is stored in a 16-bit variable. Each bit corresponds to a unique event in the same task. A flag bit of 1 means the event corresponding to the bit is running, and a flag bit of 0 means the event corresponding to the bit is running. Each Task user can customize up to 15 events, 0x8000 is the SYS_EVENT_MSG reserved by the system Events, i.e. system message delivery events, cannot be defined. Please refer to Section 3.5 for details .

The basic structure of task execution is shown in Figure 3.1. TMOS polls the task according to its priority to see if there is an event that needs to be executed. System tasks, such as sending and receiving responses in 2.4G automatic mode and related transactions in Bluetooth, have the highest priority. After the system task is completed, if there is a user event, the corresponding callback function will be executed. When a cycle ends, If there is time to spare, the system will enter idle or sleep mode.

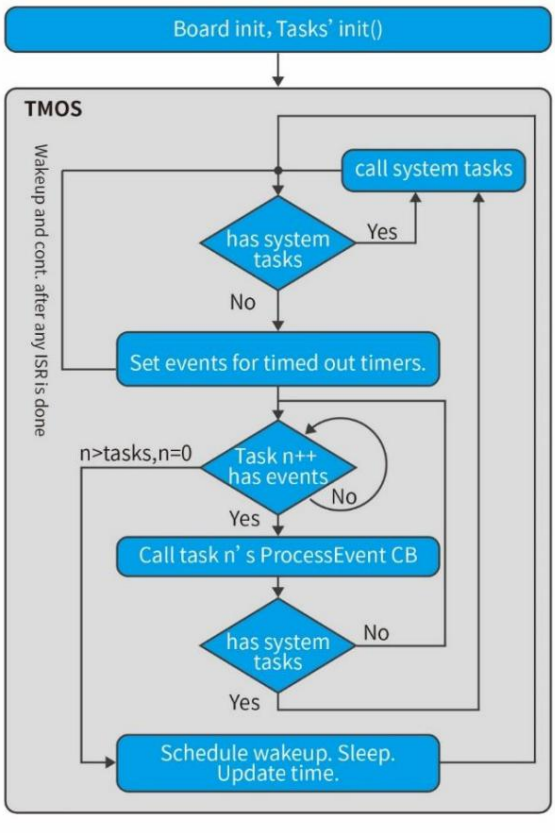


Figure 3.1 TMOS task management diagram

To illustrate the general code format of TMOS event processing, take the HAL_TEST_EVENT event of the HAL layer as an example. The same applies if you replace HAL_TEST_EVENT with other events. If you want to define a TEST event in the HAL layer, you can After the task initialization of the HAL layer is completed, add the event HAL_TEST_EVEN in the task callback function. Its basic format is as follows:

```
1. if ( events & HAL_TEST_EVENT )
2. {
3. PRINT( "* \n" );
4. return events ^ HAL_TEST_EVENT;
5. }
```

After the event is executed, the corresponding 16-bit event variable needs to be returned to clear the event to prevent repeated processing of the same event. The above code clears the HAL_TEST_EVENT flag by return events ^ HAL_TEST_EVENT;.

After the event is added, call the tmos_set_event(halTaskID, HAL_TEST_EVENT) function to execute it immediately. The event corresponding to the row is executed only once. halTaskID is the task selected for execution, HAL_TEST_EVENT is The corresponding event under the task.

```
tmos_start_task( halTaskID, HAL_TEST_EVENT, 1000 );
```

If you do not want to execute an event immediately, you can call the tmos_start_task(tmosTaskID taskID, tmosEvents event, tmosTimer time) function. Its function is similar to tmos_set_event. The difference is that after setting the event you want to execute, After the task ID and event flag of the task to be executed, a third parameter needs to be added: the timeout period for the event execution.

After the timeout period is reached, the task is executed once. Then, the next task execution time can be defined in the event to execute a certain task in a timed loop.
event.

```
1. if ( events & HAL_TEST_EVENT )
2. {
3.         PRINT( "* \n" );
4. tmos_start_task( halTaskID, HAL_TEST_EVENT, MS1_TO_SYSTEM_TIME( 1000 ));
5.         return events ^ HAL_TEST_EVENT;
6. }
```

At this time, there is only one timer event HAL_TEST_EVENT in TMOS. After the system completes this time, it will enter the empty state.
If the sleep function is turned on, the device will enter sleep mode. The actual effect is shown in Figure 3.2.

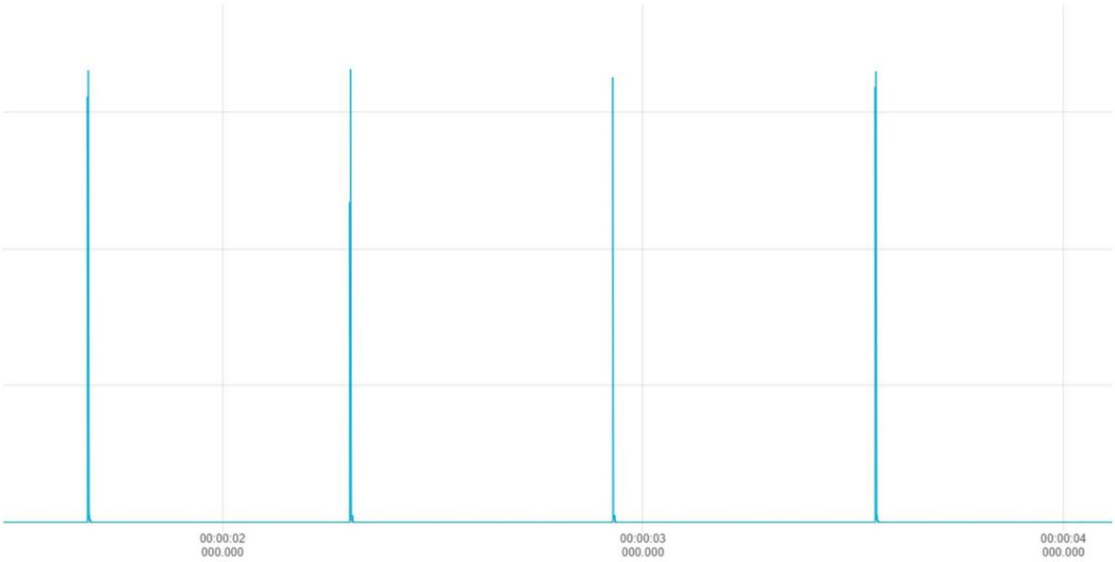


Figure 3.2 Scheduled tasks

3.4 Memory Management

TMOS uses a separate memory block, and users can customize the memory address and size.
This memory is used for software management. You can analyze the memory usage by turning on the Bluetooth binding function and encryption function.
Since the Bluetooth Low Energy protocol stack also uses this memory, it needs to be tested under the maximum expected operating conditions.

3.5 TMOS Data Transfer

TMOS provides a communication scheme for receiving and sending data for different task transfers. The type of data is arbitrary.
And if there is enough memory, the length can be arbitrary.

To send a data, follow these steps:

1. Use the tmos_msg_allocate() function to apply for memory for the data to be sent. If the application is successful, the memory is returned
address, or NULL on failure.
2. Copy the data into memory.
3. Call the tmos_msg_send() function to send the data pointer to the specified task.

```
1. // Register Key task ID
```

```
2. HAL_KEY_RegisterForKeys(centralTaskId)
3. // Send the address to the task
4.         msgPtr = ( keyChange_t * ) tmos_msg_allocate( sizeof(keyChange_t));
5.         if ( msgPtr )
6.         {
7.             msgPtr->hdr.event = KEY_CHANGE;
8.             msgPtr->state = state;
9.             msgPtr->keys = keys;
10.            tmos_msg_send( registeredKeysTaskID, ( uint8_t * ) msgPtr );
11. }
```

After the data is sent successfully, SYS_EVENT_MSG is set to valid, and the system will execute the SYS_EVENT_MSG event. In practice, the data is retrieved by calling the tmos_msg_receive() function. After the data has been processed, it must be retrieved using The tmos_msg_deallocate() function releases memory. Please refer to the example for details.

Assuming that the message is sent to the task ID of central, the system event of central will receive this message.

```
1. uint16_t Central_ProcessEvent( uint8_t task_id, uint16_t events ){
2. if ( events & SYS_EVENT_MSG ) {
3.         uint8_t *pMsg;
4.         if ( (pMsg = tmos_msg_receive( centralTaskId )) != NULL ){
5.             central_ProcessTMOSMsg( (tmos_event_hdr_t *)pMsg );
6.             // Release the TMOS message
7.                 tmos_msg_deallocate( pMsg );
8. }
```

Query the KEY_CHANGE event:

```
1. static void central_ProcessTMOSMsg(tmos_event_hdr_t *pMsg)
2. {
3.                 switch ( pMsg->event ) {
4.                   case KEY_CHANGE:{
5.                ...
```

4. Brief analysis of application examples

4.1 Overview

The low-power Bluetooth EVT example includes a simple BLE project: Peripheral. Burning this project into the CH58x chip can implement a simple low-power Bluetooth slave device.

4.2 Project Preview

After loading the .WVPROJ file, you can see the project file in the left window of MounRiverStudio:

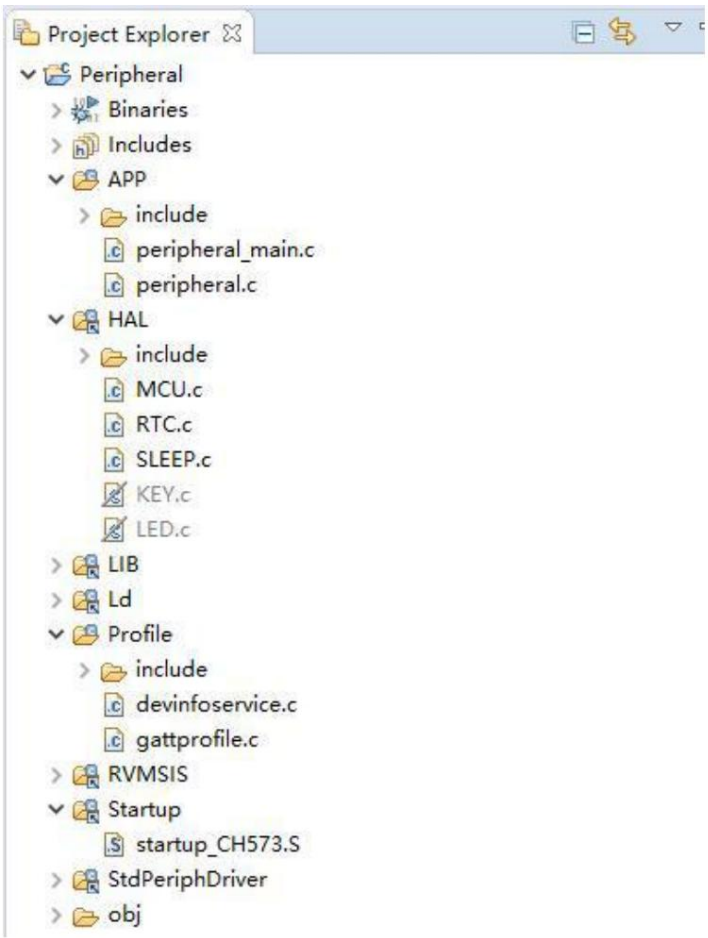


Figure 4.1 Project files

Files can be divided into the

following categories: 1. APP – Source files and header files related to the application can be placed here, and the main function of the routine is also here.

2. HAL – This folder contains the source code and header files of the HAL layer, which is the Bluetooth protocol stack and chip hardware.

Driver interaction layer.

3. LIB – is the protocol stack library file for low-power Bluetooth.

4. LD – Linker script.

5. Profile – This file contains the source code and header files of the GAP role profile, GAP security profile and GATT profile, as

well as the header files required by the GATT service. Please refer to Section 5 for details .

6. RVMSIS – Source code and header files for RISC-V core access.

7. Startup – startup file.

8. StdPeriphDriver – includes the low-level driver files of chip peripherals.

9. obj – files generated by the compiler, including map files and hex files.

4.3 Starting from main()

The Main() function is the starting point of the program. This function first initializes the system clock and then configures the IO port status. state to prevent the floating state from causing unstable working current; then initialize the serial port for printing debugging, and finally initialize TMOS And low-power Bluetooth. The main() function of the Peripheral project is as follows:

```
int main( void )
{
#if (defined (DCDC_ENABLE)) && (DCDC_ENABLE == TRUE)
    PWR_DCDCCfg( ENABLE );
#endif
    SetSysClock( CLK_SOURCE_PLL_60MHz ); //Set the system clock
    GPIOA_ModeCfg( GPIO_Pin_All, GPIO_ModeIN_PU ); //Configure IO port
    GPIOB_ModeCfg(GPIO_Pin_All, GPIO_ModeIN_PU);
#ifdef DEBUG
    GPIOA_SetBits(bTXD1); //Configure the serial port
    GPIOA_ModeCfg(bTXD1, GPIO_ModeOut_PP_5mA);
    UART1_DefInit(); //Initialize the serial port
#endif
    PRINT("%s\n",VER_LIB);
    CH58X_BLEInit(); //Initialize Bluetooth library
    HAL_Init() ;
    GAPRole_PeripheralInit( );
    Peripheral_Init() ;
    while(1){
        TMOS_SystemProcess() ; //Main loop
}
}
```

4.4 Application Initialization

4.4.1 Initializing the Bluetooth Low Energy Library

The low-power Bluetooth library initialization function CH58X_BLEInit() configures the library's memory through the configuration parameter bleConfig_t. Clock, transmit power and other parameters, and then pass the configuration parameters into the library through the BLE_LibInit() function.

4.4.2 HAL layer initialization

Register the HAL layer task to initialize the hardware parameters, such as RTC clock, sleep wake-up, RF calibration, etc.

```
void HAL_Init()
{
    halTaskID = TMOS_ProcessEventRegister( HAL_ProcessEvent );
    HAL_TimeInit();
```

```
5. #if (defined HAL_SLEEP) && (HAL_SLEEP == TRUE)
6.     HAL_SleepInit();
7. #endif
8. #if (defined HAL_LED) && (HAL_LED == TRUE)
9.     HAL_LedInit() ;
10. #endif
11. #if (defined HAL_KEY) && (HAL_KEY == TRUE)
12. HAL_KeyInit() ;
13. #endif
14. #if (defined BLE_CALIBRATION_ENABLE) && (BLE_CALIBRATION_ENABLE == TRUE)
15. tmos_start_task( halTaskID, HAL_REG_INIT_EVENT, MS1_TO_SYSTEM_TIME( BLE_CA
    LIBRATION_PERIOD ) );          // Add a calibration task. A single calibration takes less than 10ms.
16. #endif
17. tmos_start_task( halTaskID, HAL_TEST_EVENT, 1000 );          // Add a test task
18. }
```

4.4.3 Bluetooth Low Energy Slave Initialization

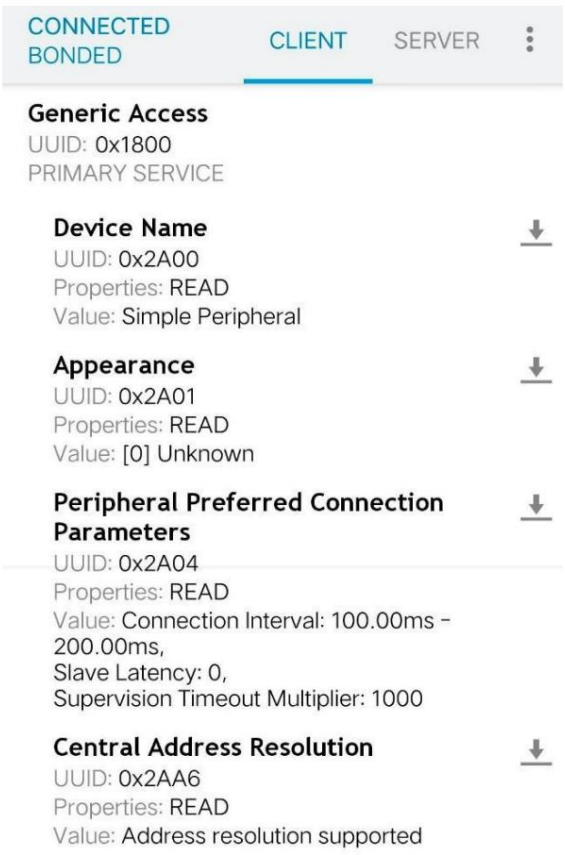
This process consists of two parts:

1. Initialization of the GAP role, which is completed by the low-power Bluetooth library;

2. Initialize the low-power Bluetooth slave application, including slave task registration, parameter configuration (such as broadcast parameters, connection parameters, binding parameters, etc.), registration of GATT layer services, and registration of callback functions. See Section **5.5.3.2** for details .

Figure 4.2 shows the complete property table of the example Peripheral, which can be used as a reference for low-power Bluetooth communication.

Please refer to Chapter 5 for more information.



Generic Attribute
UUID: 0x1801
PRIMARY SERVICE

Service Changed
UUID: 0x2A05
Properties: INDICATE
Descriptors:
Client Characteristic Configuration
UUID: 0x2902
Value: Indications enabled

Device Information
UUID: 0x180A
PRIMARY SERVICE

System ID
UUID: 0x2A23
Properties: READ
Value: (0x) 00-00-00-00-00-00-00-00

Model Number String
UUID: 0x2A24
Properties: READ
Value: Model Number

Serial Number String
UUID: 0x2A25
Properties: READ
Value: Serial Number

Firmware Revision String
UUID: 0x2A26
Properties: READ
Value: Firmware Revision

Hardware Revision String
UUID: 0x2A27
Properties: READ
Value: Hardware Revision

Software Revision String
UUID: 0x2A28
Properties: READ
Value: Software Revision

Manufacturer Name String
UUID: 0x2A29
Properties: READ
Value: Manufacturer Name

IEEE 11073-20601 Regulatory Certification Data List
UUID: 0x2A2A
Properties: READ
Value: (0x) FE-00-65-78-70-65-72-69-6D-65-6E-74-61-6C

PnP ID
UUID: 0x2A50
Properties: READ
Value: Bluetooth SIG Company: Reserved ID <0x07D7>
Product Id: 0
Product Version: 272

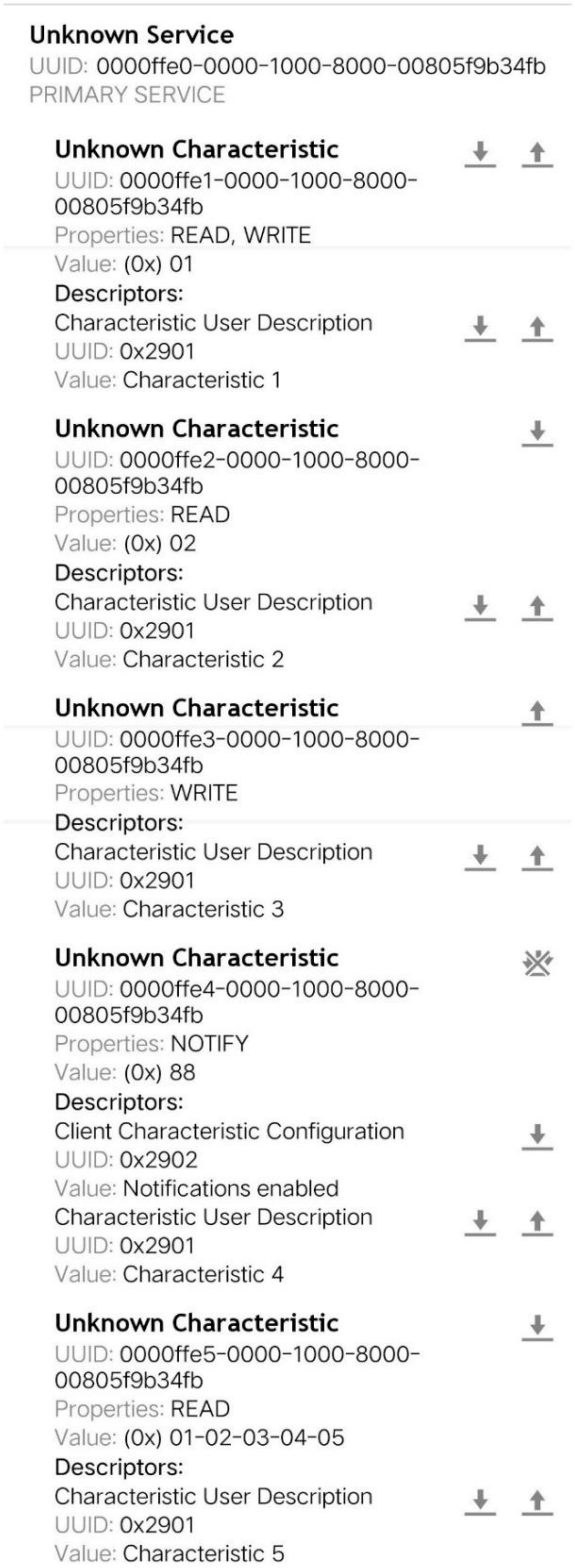


Figure 4.2 Attribute table

4.5 Event Handling

After initialization is complete, By turning on the event (i.e. setting a bit in the event), the application task will process The following subsections describe possible sources of events for events in Peripheral_ProcessEvent.

4.5.1 Timed Events

As shown in the following program segment (the program segment is located in the example peripheral.c), the application contains a program called The TMOS timer causes the SBP_PERIODIC_EVT event to occur periodically.

After SBP_PERIODIC_EVT processing is completed, the timer timeout value is set to SBP_PERIODIC_EVT (the default value is 5000ms).

A periodic event occurs every 5 seconds, and the performPeriodicTask() function is called to implement the function.

```
1. if(events & SBP_PERIODIC_EVT)
2. {
3.     // Restart timer
4.     if(SBP_PERIODIC_EVT_PERIOD)
5.     {
6.     tmos_start_task(Peripheral_TaskID, SBP_PERIODIC_EVT,
       SBP_PERIODIC_EVT_PERIOD);
7.     }
8. // Perform periodic application task
9. performPeriodicTask();
10. return (events ^ SBP_PERIODIC_EVT);
11. }
```

This periodic event handling is just an example, but it highlights how to perform custom actions in periodic tasks.

Before processing a periodic event, a new TMOS timer will be started to set the next periodic task.

4.5.2 TMOS Message Passing

TMOS messages may come from various layers of the BLE protocol stack. For more information, see 3.5 TMOS Data Transfer.

4.6 Callbacks

The application code can be written in the event handling code segment or in the callback function, such as simpleProfileChangeCB() and peripheralStateNotificationCB(). Between the protocol stack and the application

The communication is implemented by callback functions, such as simpleProfileChangeCB(), which can notify the application of changes in characteristic values.

Use the program.

5. Bluetooth Low Energy Protocol Stack

5.1 Overview

The code of the Bluetooth Low Energy protocol stack is in the library file, and the original code is not provided. However, users should understand these layers functionality as they interact directly with the application.

5.2 Generic Access Profile (GAP)

5.2.1 Overview The GAP layer of the Bluetooth Low Energy protocol stack defines the following states of the device, as shown in Figure 5.1

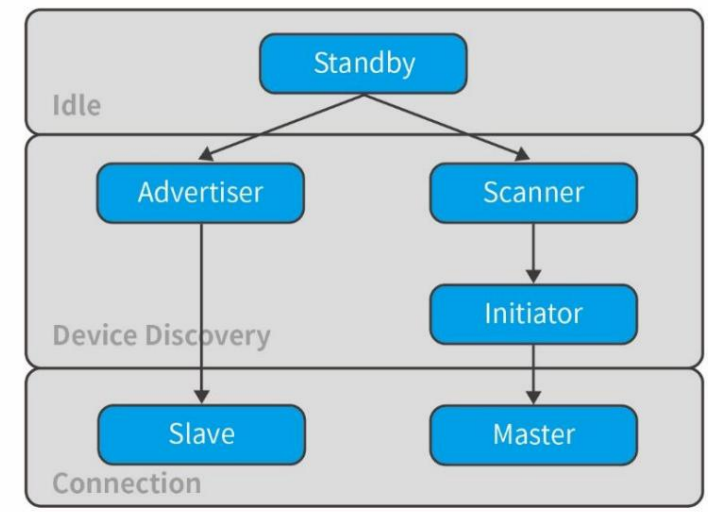


Figure 5.1 GAP status

Among them: Standby: The idle state when the low-power Bluetooth protocol stack is not enabled;

Advertiser: The device uses specific data for broadcasting, which may include the device's name, address and other data. The advertisement indicates that the device is available for connection.

Scanner: After receiving the broadcast data, the scanning device sends a scan request packet to the broadcaster, and the broadcaster returns a scan response packet. The scanner reads the broadcaster's information and determines whether it can be connected. This process describes the process of discovering devices.

Initiator: When establishing a connection, the connection initiator must specify the device address for the connection. If the addresses match, A connection will be established with the broadcaster. The connection initiator will initialize the connection parameters when establishing the connection.

Master or Slave: If the device is a broadcaster before connecting, it is a slave when connecting. If it is the initiator before the connection, it will be the host after the connection.

5.2.1.1 Connection Parameters This section describes the connection parameters when the connection is established. These connection parameters can be modified by both the host and the slave. Connection Interval – Bluetooth low energy uses a frequency hopping scheme, and devices send and receive data on a specific channel at a specific time. A data transmission and reception between two devices becomes a connection event. The connection interval is the time between two connection events, and its time unit is 1.25ms. The range of the connection interval is 7.5ms~4s.

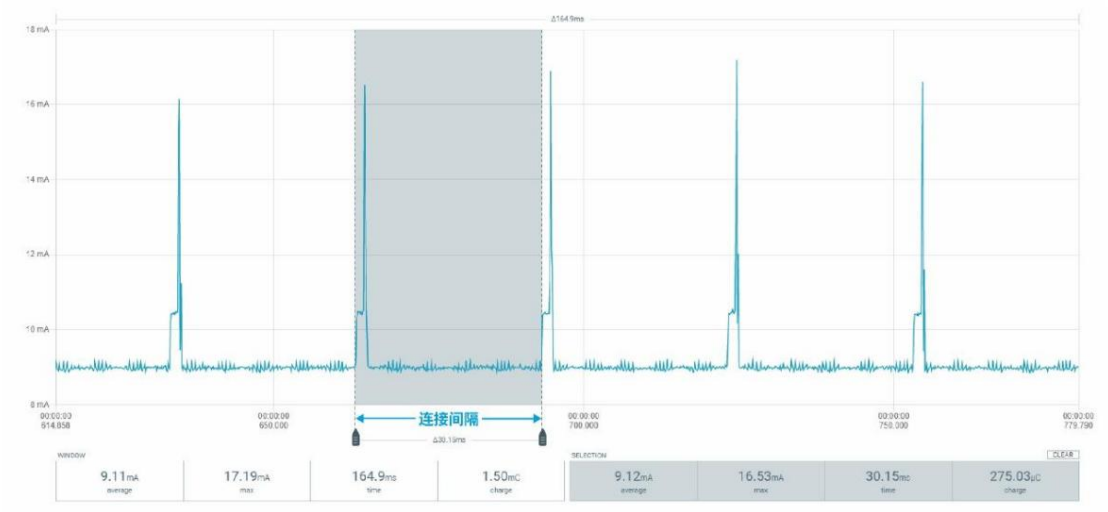


Figure 5.2 Connection events and connection intervals

Different applications may require different connection intervals. A smaller connection interval will reduce the data response time and correspondingly increase the power consumption.

Slave Latency – This parameter allows the slave to skip some connection events. If the device has no data to send, the slave latency can skip the connection event and stop the RF during the connection event, thereby reducing power consumption. The value of the slave latency represents the maximum number of events that can be skipped, and its range is 0~499. However, the effective connection interval must be less than 16s. For more information about the effective connection interval, please refer to 5.2.1.2

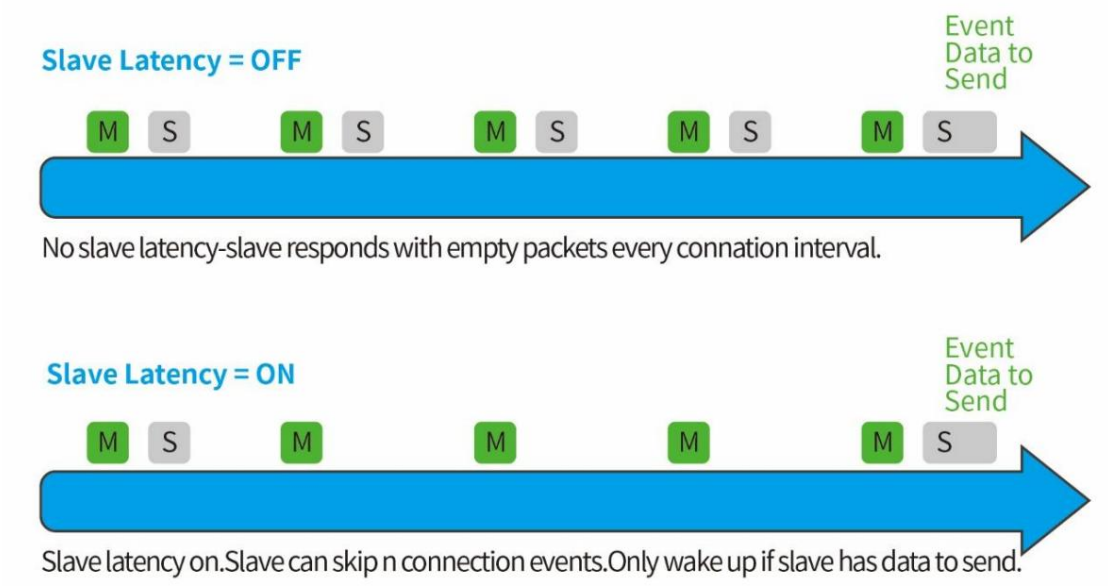


Figure 5.3 Slave device delay

Supervision Time-out - This parameter is the maximum time between two valid connection events. If there is no valid connection event after this time, the connection is considered disconnected and the device returns to the unconnected state. The range of supervision timeout is 10 (100ms) ~ 3200 (32s). The timeout must be greater than the valid connection interval.

5.2.1.2 Valid Connection Interval The

valid connection interval is the time between two connection events when the slave device is delayed enabled and there is no data transmission. If slave delay is not enabled or its value is 0, the effective connection interval is the configured connection interval.

The calculation formula is

as follows: Effective connection interval = connection interval! " (1 + slave device delay)

When the connection interval is 80 (100ms) Slave

device delay: 4 Effective

connection interval: (100ms) " (1 + 4) = 500ms Then in the absence of data transmission,

the slave will initiate a connection event every 500ms.

5.2.1.3 Connection Parameter Considerations Reasonable

connection parameters help optimize the power consumption and performance of low-power Bluetooth. The following summarizes the trade-offs in connection parameter settings: Reducing

the connection interval will:

Increase the power consumption of

the master and slave devices Increase the throughput

between the two devices Reduce the time it takes for data to travel

between the two devices

Increasing the connection interval will:

Reduce the power consumption of the master and slave

devices Reduce the throughput between the two devices Increase

the time it takes for data to travel between the two devices Reducing the slave device latency or setting it to 0 will:

Increase the power consumption of

the slave device Reduce the time required for the master to send data to the

slave device Increasing the slave

device latency will: Reduce the power consumption of the slave device when there is no data to

send to the master Increase the time required for the master to send data to the slave device

5.2.1.4 Connection Parameter Update In some

applications, the slave may need to change the connection parameters during the connection according to the application.

Parameter Update Request is sent to the host to update the connection parameters. For Bluetooth 4.0, the L2CAP layer of the protocol stack handles this request.

The request contains the following

parameters: Minimum

connection interval Maximum

connection interval Slave

delay Supervision timeout

These parameters are the connection parameters requested by the slave, but the master can reject the request.

5.2.1.5 Terminating a Connection The

master or slave may terminate a connection for any reason. When either device initiates a connection termination, the other device must

The two devices respond to terminate the connection before disconnecting.

5.2.2 GAP Abstraction Layer Applications

can implement corresponding BLE functions, such as broadcasting and connection, by calling the API functions of the GAP layer.

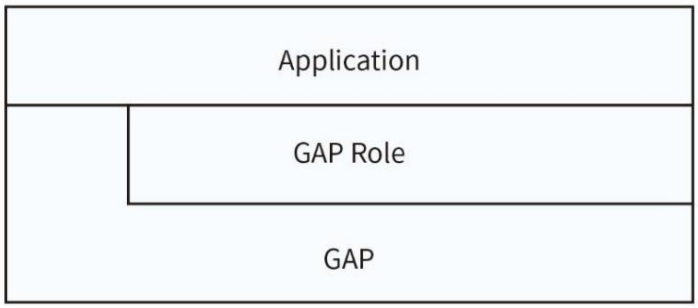


Figure 5.4 GAP abstract layer

5.2.3 GAP layer configuration

Most of the functions of the GAP layer are implemented in the library. Users can find the corresponding function declarations in CH58xBLE_LIB.h. bright.

Section 8.1 defines the GAP API, which can be accessed through GAPRole_SetParameter() and GAPRole_GetParamenter() To set and detect parameters, such as broadcast interval, scan interval, etc. The GAP layer configuration example is as follows:

```
1. // Setup the GAP Peripheral Role Profile
2. {
3.         uint8_t initial_advertising_enable = TRUE;
4.         uint16_t desired_min_interval = DEFAULT_DESIRED_MIN_CONN_INTERVAL;
5.         uint16_t desired_max_interval = DEFAULT_DESIRED_MAX_CONN_INTERVAL;
6.         GAPRole_SetParameter( GAPROLE_MIN_CONN_INTERVAL, sizeof( uint16_t ), &de
   sired_min_interval );
7.         GAPRole_SetParameter( GAPROLE_MAX_CONN_INTERVAL, sizeof( uint16_t ), &de
   sired_max_interval );
8.
9. }
```

5.3 GAPRole Task

As described in Section 4.4 , GAPRole is a separate task (GAPRole_PeripheralInit). Most of the GAPRole code runs in the Bluetooth library, simplifying the application layer program. This task is performed by the application during initialization. Start and configure. If there is a callback, the application can register the callback function with the GAPRole task.

Depending on the device configuration, the GAP layer can perform the following four roles:

Broadcaster - broadcast only and cannot be connected

Observer - only scans for broadcasts and cannot establish connections

Peripheral - can broadcast and establish a connection as a slave at the link layer

Central - can scan broadcasts and can also act as a host to establish single or multiple connections at the link layer

The following introduces the two roles of peripheral device (Peripheral) and central device (Central).

5.3.1 Peripheral Role

The general steps to initialize a peripheral device are as follows:

1. Initialize the GAPRole parameters as shown in the following code.

```
1. // Setup the GAP Peripheral Role Profile
```

```
2. {
3.     uint8_t initial_advertising_enable = TRUE;
4.     uint16_t desired_min_interval = DEFAULT_DESIRED_MIN_CONN_INTERVAL;
5.     uint16_t desired_max_interval = DEFAULT_DESIRED_MAX_CONN_INTERVAL;
6.
7.       // Set the GAP Role Parameters
8.     GAPRole_SetParameter( GAPROLE_ADVERT_ENABLED, sizeof( uint8_t ), &initial_
    advertising_enable );
9.     GAPRole_SetParameter( GAPROLE_SCAN_RSP_DATA, sizeof ( scanRspData ), scanR
    spData );
10. GAPRole_SetParameter( GAPROLE_ADVERT_DATA, sizeof( advertData ), advertDat
    a );
11. GAPRole_SetParameter( GAPROLE_MIN_CONN_INTERVAL, sizeof( uint16_t ), &desi
    red_min_interval );
12. GAPRole_SetParameter( GAPROLE_MAX_CONN_INTERVAL, sizeof( uint16_t ), &desi
    red_max_interval );
13. }
14.
15. // Set the GAP Characteristics
16. GGS_SetParameter( GGS_DEVICE_NAME_ATT, GAP_DEVICE_NAME_LEN, attDeviceName
    );
17.
18. // Set advertising interval
19. {
20.     uint16_t advInt = DEFAULT_ADVERTISING_INTERVAL;
twenty one.
twenty two.     GAP_SetParamValue( TGAP_DISC_ADV_INT_MIN, advInt );
twenty three.     GAP_SetParamValue( TGAP_DISC_ADV_INT_MAX, advInt );
twenty four. }
```

2. Initialize the GAPRole task, including passing the function pointer to the application callback function.

```
1. if (events & SBP_START_DEVICE_EVT){
2. // Start the Device
3.     GAPRole_PeripheralStartDevice( Peripheral_TaskID, &Peripheral_BondMgrCBs,
    &Peripheral_PeripheralCBs );
4.     return ( events ^ SBP_START_DEVICE_EVT );
5. }
```

3. Send GAPRole command from the application layer

The application layer updates the connection parameters.

```
1. // Send connect param update request
2.         GAPRole_PeripheralConnParamUpdateReq(peripheralConnList.connHandle,
3.                                              DEFAULT_DESIRED_MIN_CONN_INTERVAL,
4.                                              DEFAULT_DESIRED_MAX_CONN_INTERVAL,
5.                                              DEFAULT_DESIRED_SLAVE_LATENCY,
6.                                              EFAULT_DESIRED_CONN_TIMEOUT,
7.                                              Peripheral_TaskID);
```

The protocol stack receives the command, executes the parameter update operation, and returns the corresponding status.

4. The GAPRole task passes the protocol stack and GAP-related events to the application layer.

The Bluetooth protocol stack receives the disconnection command and passes it to the GAP layer.

The GAP layer receives the command and passes it directly to the application layer through the callback function.

```
1. static void peripheralStateNotificationCB( gapRole_States_t newState, gapRol
       eEvent_t * pEvent )
2. {
3.                 switch (newState & GAPROLE_STATE_ADV_MASK)
4.                 {
5.                     . . .
6.                     case GAPROLE_ADVERTISING:
7.                     if( pEvent->gap.opcode == GAP_LINK_TERMINATED_EVENT )
8.                     {
9. ...
```

5.3.2 Central Role

The general operation of initializing the central device is as follows:

1. Initialize the GAPRole parameters as shown in the following code.

```
1. uint8_t scanRes = DEFAULT_MAX_SCAN_RES;
2. GAPRole_SetParameter( GAPROLE_MAX_SCAN_RES, sizeof( uint8_t ), &scanRes );
```

2. Initialize the GAPRole task, including passing the function pointer to the application callback function.

```
1. if ( events & START_DEVICE_EVT )
2. {
3.                 // Start the Device
4.         GAPRole_CentralStartDevice( centralTaskId, &centralBondCB, &centralRoleCB
       );
5.           return ( events ^ START_DEVICE_EVT );
6. }
```

3. Send GAPRole command from the application layer

The application layer calls the application function and sends the GAP command.

```
GAPRole_CentralStartDiscovery(DEFAULT_DISCOVERY_MODE,
                              DEFAULT_DISCOVERY_ACTIVE_SCAN,
                              DEFAULT_DISCOVERY_WHITE_LIST );
```

The GAP layer sends a command to the Bluetooth protocol stack. The protocol stack receives the command, performs a scan operation, and returns the corresponding status.

4. The GAPRole task passes the protocol stack and GAP-related events to the application layer.

The Bluetooth protocol stack receives the disconnection command and passes it to the GAP layer.

The GAP layer receives the command and passes it directly to the application layer through the callback function.

```
static void centralEventCB(gapRoleEvent_t *pEvent)
{
    switch ( pEvent->gap.opcode )
    {
...
        case GAP_DEVICE_DISCOVERY_EVENT:
        {
            uint8_t i;
          // See if peer device has been discovered
            for ( i = 0; i < centralScanRes; i++ )
            {
          if (tmos_memcmp( PeerAddrDef, centralDevList[i].addr, B_ADDR_LEN))
            break;
    }
...
```

5.4 GAP Binding Management

The GAPBondMgr protocol handles security management in Bluetooth Low Energy connections, allowing certain data to be transmitted only after authentication. It can then be read and written.

Table 5.1 GAP binding management terms

Terminology Description	
Pairing: The process of key interaction	
Encryption: Data is encrypted after pairing, or re-encrypted	
verify ÿAuthenticationÿ	The pairing process is completed with Man in the Middle (MITM) protection
Binding stores the encryption key in non-volatile memory for use in the next encryption sequence	
Authorization ÿAuthorizationÿ	In addition to authentication, an additional application-level key exchange is performed
Out-of-band (OOB) keys are not exchanged wirelessly, but rather via a wireless connection such as a serial port or NFC. This also provides MITM protection.	
Man-in-the-Middle (MITM) protection. This prevents the encryption from being cracked by listening to the key transmitted over the air.	
Direct connection (JustWorks) is a matchmaking method without a middleman.	

The general process for establishing a secure connection is as follows:

1. Key pairing (including the following two methods).

   A. JustWorks, sending keys wirelessly

   B. MITM, sending keys through a middleman

2. Encrypt the connection via a key.

3. Bind key and store key.

4. When you connect again, encrypt the connection using the stored key.

5.4.1 Close pairing

```
1. uint8_t pairMode = GAPBOND_PAIRING_MODE_NO_PAIRING; 2.
GAPBondMgr_SetParameter( GAPBOND_PERI_PAIRING_MODE, sizeof ( uint8_t ), &pa
    irMode );
```

When pairing is disabled, the stack will reject any pairing attempts.

5.4.2 Direct pairing without binding

```
1. uint8_t mitm = FALSE;
2. uint8_t bonding = FALSE;
3. uint8_t pairMode = GAPBOND_PAIRING_MODE_WAIT_FOR_REQ;
4. GAPBondMgr_SetParameter( GAPBOND_PERI_PAIRING_MODE, sizeof ( uint8_t ), &pai
    rMode );
5. GAPBondMgr_SetParameter( GAPBOND_PERI_MITM_PROTECTION, sizeof ( uint8_t ), &
    mitm );
6. GAPBondMgr_SetParameter( GAPBOND_PERI_BONDING_ENABLED, sizeof ( uint8_t ), &
    bonding );
```

It should be noted that to enable the pairing function, you also need to configure the IO function of the device, that is, whether the device supports display output

If the device does not support keyboard input but is configured to allow password input via the device, pairing cannot be established.

```
1. uint8_t ioCap = GAPBOND_IO_CAP_DISPLAY_ONLY;
2. GAPBondMgr_SetParameter( GAPBOND_PERI_IO_CAPABILITIES, sizeof ( uint8_t ), &
    ioCap );
```

5.4.3 Pairing and Binding via a Middleman

```
1. uint32_t passkey = 0; // passkey "000000"
2. uint8_t pairMode = GAPBOND_PAIRING_MODE_WAIT_FOR_REQ;
3. uint8_t mitm = TRUE;
4. uint8_t bonding = TRUE;
5. uint8_t ioCap = GAPBOND_IO_CAP_DISPLAY_ONLY;
6. GAPBondMgr_SetParameter( GAPBOND_PERI_DEFAULT_PASSCODE, sizeof ( uint32_t ),
    &passkey );
```

```
7. GAPBondMgr_SetParameter( GAPBOND_PERI_PAIRING_MODE, sizeof ( uint8_t ), &pai
    rMode );
8. GAPBondMgr_SetParameter( GAPBOND_PERI_MITM_PROTECTION, sizeof ( uint8_t ), &
    mitm );
9. GAPBondMgr_SetParameter( GAPBOND_PERI_IO_CAPABILITIES, sizeof ( uint8_t ), &
    ioCap );
10. GAPBondMgr_SetParameter( GAPBOND_PERI_BONDING_ENABLED, sizeof ( uint8_t ), &
    bonding );
```

Use a middleman to pair and bind, and generate a key through a 6-digit password.

5.5 Generic Attribute Profile (GATT)

The GATT layer allows applications to communicate data between two connected devices. Data is transferred and stored in the form of characteristics. In GATT, when two devices connect, they will each play one of two roles:

· GATT Server – This device provides GATT clients with the ability to read or write to the characteristic database.

GATT Client – This device reads and writes data from a GATT server.

Figure 5.5 shows the relationship between the low-power Bluetooth server and the client, where the peripheral device (low-power Bluetooth module) is GATT server, the central device (smartphone) is the GATT client.

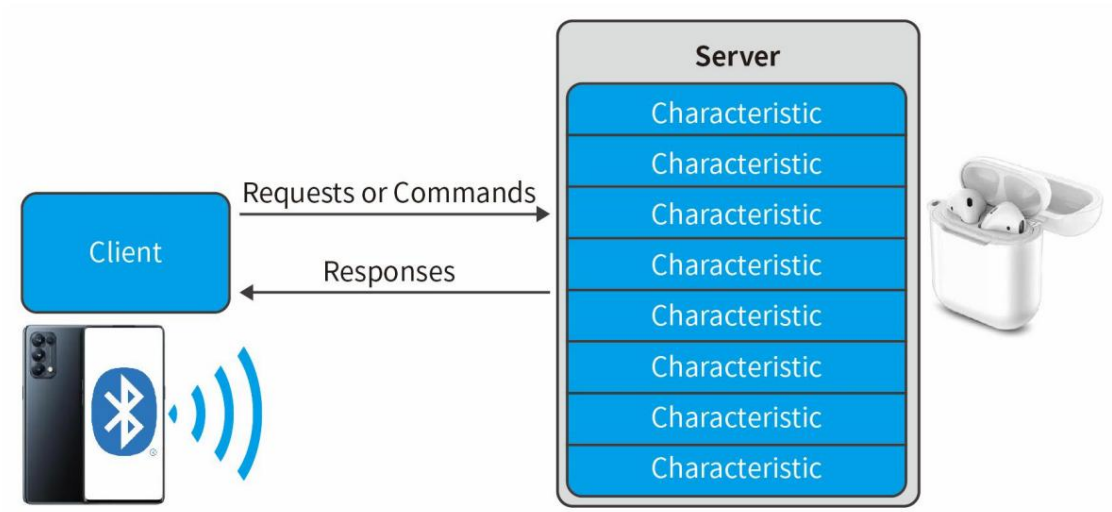


Figure 5.5 GATT server and client

Typically, the server and client roles of GATT are independent of the central device and peripheral device roles of GAP. A central device can be a GATT client or server. A device can also be a GATT server or client.
GATT server or client.

5.5.1 GATT Characteristics and Attributes

Typical features consist of the following attributes:

Characteristic Value: This value is the data value of the characteristic.

Characteristic Declaration: The properties, location, and type of the characteristic value.

Client Characteristic Configuration: Through this configuration, GATT service
The device can configure the attributes that need to be sent to the GATT server (notified), or send to the GATT server and expect Get a response (indicated).

CharacteristicUserDescription: An ASCII string describing the characteristic value.

These attributes are stored in the GATT server's attribute table. The following characteristics are associated with each attribute: Handle - The index of the attribute in the table. Each attribute has a unique handle. Type - This attribute indicates what the attribute represents. It is called a Universally Unique Identifier (UUID). Some UUIDs are defined by the Bluetooth SIG, and others can be customized by the user.

·Permissions - used to restrict the permissions and methods that GATT clients can use to access the value of this attribute.

·Value (pValue) - a pointer to the attribute value. Its length cannot be changed after initialization. Maximum size 512 characters Festival.

5.5.2 GATT Services and Protocols

GATT service is a collection of characteristics. The following is the corresponding gattprofile service in the Peripheral project (gattprofile service is used for testing and The sample profile for the demonstration (full source code in gattprofile.c and gattprofile.h) has a property table.

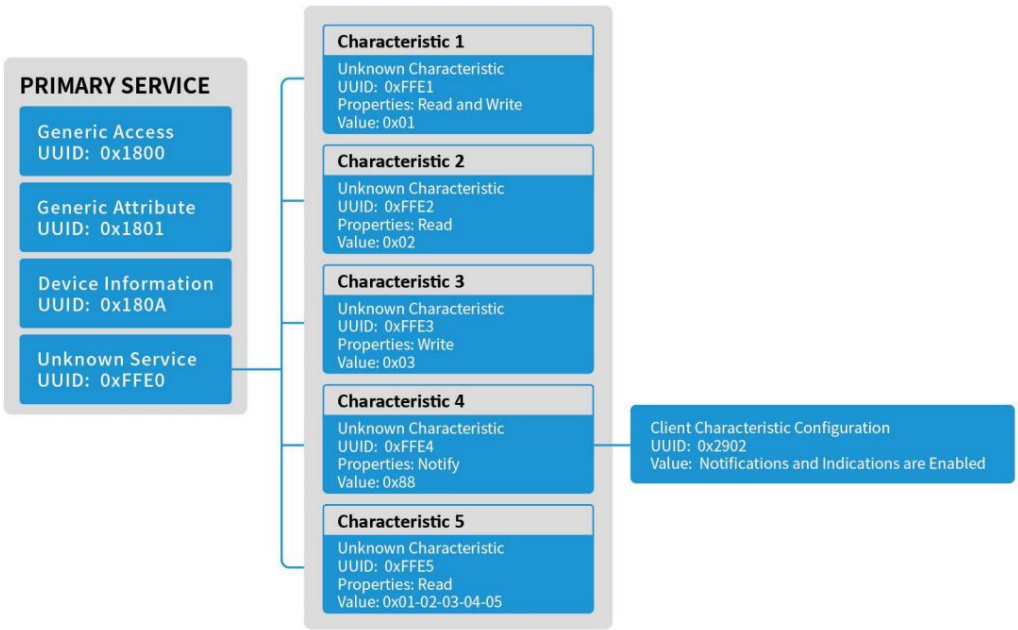


Figure 5.6 GATT attribute table

The Gattprofile contains the following five characteristics: · simpleProfilechar1 - 1 byte can be read or written from a GATT client device. · simpleProfilechar2 - 1 byte can be read from a GATT client device, but not written. · simpleProfilechar3 - 1 byte can be written from a GATT client device, but not read. · simpleProfilechar4 - can be configured to send a notification of 1 byte to a GATT client device, but not read or write. · simpleProfilechar5 - 5 bytes can be read from a GATT client device, but not written.

Here are some of the relevant properties: 0x02: Allow reading of characteristic values 0x04: Allow writing of characteristic values without response 0x08: Allow writing of characteristic values (with response) 0x10: Permission for characteristic value notification (without acknowledgment) 0x20: Allow characteristic value notification (with acknowledgment)

5.5.3 GATT Client Abstraction Layer

GATT clients do not have attribute tables because clients receive information rather than provide it. Most interfaces in the GATT layer are directly Connect from the application.

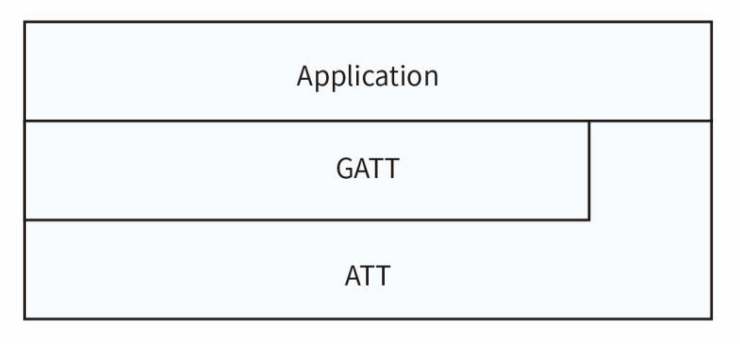


Figure 5.6 GATT client abstraction layer

5.5.3.1 Application of GATT layer

This section describes how to use the GATT client directly in your application. You can find the corresponding source code in the sample central.

1. Initialize the GATT client.

```
1.  // Initialize GATT Client
2. GATT_InitClient();
```

2. Register to receive incoming ATT indications and notifications.

```
1. // Register to receive incoming ATT Indications/Notifications
2. GATT_RegisterForInd( centralTaskId );
```

3. Execute the client program, such as GATT_WriteCharValue(), to send data to the server.

```
1. bStatus_t GATT_WriteCharValue( uint16_t connHandle, attWriteReq_t *pReq, uin
   t8_t taskId )
```

4. The application receives and processes the response from the GATT client. The following is the response for a "write" operation.

First, the protocol stack receives the write response and sends it to the application layer through the task TMOS message.

```
1. uint16_t Central_ProcessEvent( uint8_t task_id, uint16_t events)
2. {
3.     if ( events & SYS_EVENT_MSG )
4.         {
5.         uint8_t *pMsg;
6.             if ( (pMsg = tmos_msg_receive( centralTaskId )) != NULL )
7.             {
8.             central_ProcessTMOSMsg( (tmos_event_hdr_t *)pMsg );
9. ...
```

The application layer task queries the GATT message:

```
1. static void central_ProcessTMOSMsg(tmos_event_hdr_t *pMsg)
2. {
3.                 switch ( pMsg->event )
4.                 {
5.                       case GATT_MSG_EVENT:
6.                 centralProcessGATTMsg( (gattMsgEvent_t *) pMsg );
```

According to the received content, the application layer can perform corresponding functions:

```
1. static void centralProcessGATTMsg( gattMsgEvent_t *pMsg )
2. {
3. ...
4.           else if ( ( pMsg->method == ATT_WRITE_RSP ) ||
5.                ( ( pMsg->method == ATT_ERROR_RSP ) &&
6.                  ( pMsg->msg.errorRsp.reqOpcode == ATT_WRITE_REQ ) ) )
7.                 {
8.           //Application
9. ...
```

Clear the message after the application has completed processing:

```
1.           // Release the TMOS message
2.             tmos_msg_deallocate( pMsg );
3.         }
4.         // return unprocessed events
5.         return (events ^ SYS_EVENT_MSG);
6. }
```

5.5.4 GATT Server Abstraction Layer

As a GATT server, most GATT functions can be configured through GATTServApp.

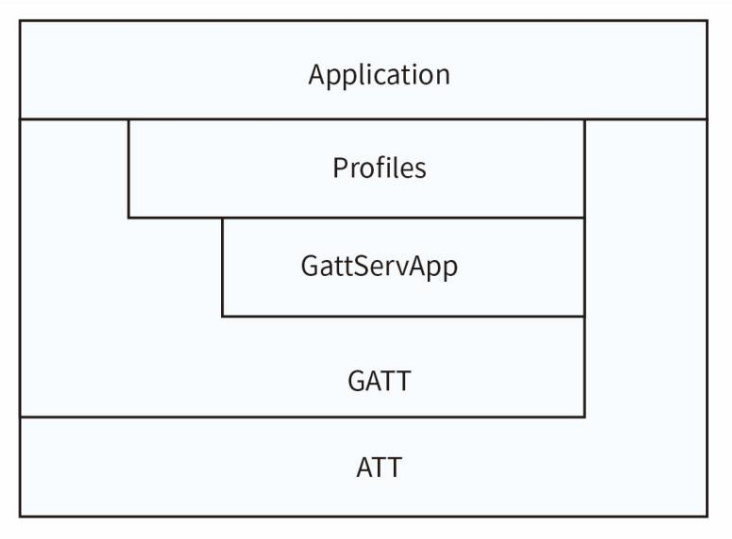


Figure 5.7 GATT server abstraction layer

The usage specifications of GATT

are as follows: 1. Create a GATT profile to configure the GATTServApp module. 2. Use the API interface in the GATTServApp module to operate the GATT layer.

5.5.4.1 GATTServApp Module The

GATTServApp module is used to store and manage the attribute table of the application. Various configuration files use this module to add their characteristic values to the attribute table. Its functions include: finding specific attributes, reading the characteristic values of the client, and modifying the characteristic values of the client. Please refer to the API section for details.

Each time the application is initialized, it will use the GATTServApp module to add services to build the GATT table. The content of each service includes: UUID, value, permission, and read/write permission. Figure 5.8 describes the process of adding services by the GATTServApp module.

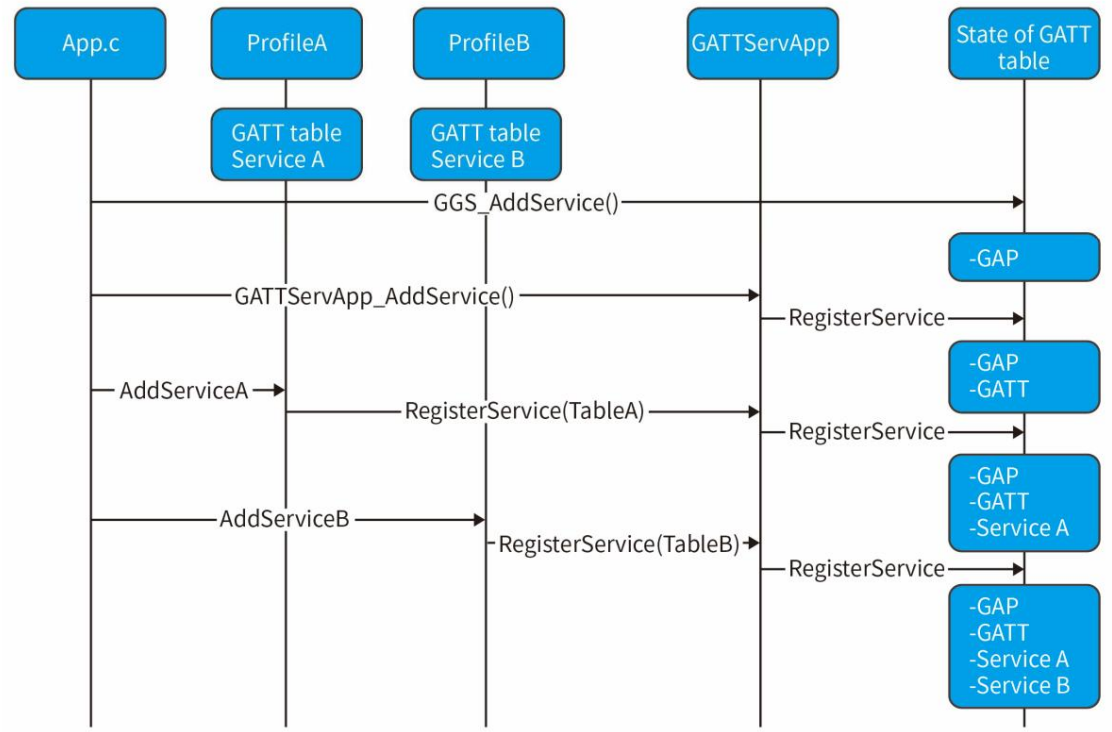


Figure 5.8 Property table initialization

The initialization of GATTServApp can be found in the Peripheral_Init() function.

```
1. // Initialize GATT attributes
2. GGS_AddService( GATT_ALL_SERVICES );                    // GAP
3. GATTServApp_AddService( GATT_ALL_SERVICES );            //GATT attributes
4. DevInfo_AddService();                                   // Device Information Service
5. SimpleProfile_AddService( GATT_ALL_SERVICES ); // Simple GATT Profile
```

5.5.4.2 Configuration file structure

This section introduces the basic structure of the profile and provides the usage of GATTProfile in Peripheral project.
Example.

5.5.4.2.1 Creating an attribute table

Each service must define a fixed-size attribute table to be passed to the GATT layer.

In Peripheral Engineering, the definition is as follows:

```
1. static gattAttribute_t simpleProfileAttrTbl[] =
2. ...
```

The format of each attribute is as follows:

```
1. typedef struct attAttribute_t
2. {
3.         gattAttrType_t type;          //!< Attribute type (2 or 16 octet UUIDs)
4.         uint8_t permissions; //!< Attribute permissions
5.         uint16_t handle;              //!< Attribute handle - assigned internally by
6.                                       //!< attribute server
7.         uint8_t *pValue;              //!< Attribute value - encoding of the octet
8.                                       //!< array is defined in the applicable
9.                                       //!< profile. The maximum length of an
10.                                      //!< attribute value shall be 512 octets.
11. } gattAttribute_t;
```

The elements in the attribute:

type - The UUID associated with the attribute.

```
1. typedef struct
2. {
3.         uint8_t len;              //!< Length of UUID (2 or 16)
4.         const uint8_t *uuid; //!< Pointer to UUID
5. } gattAttrType_t;
```

len can be 2 bytes or 16 bytes. *uuid can be a pointer to a
The number can also be a pointer to a custom UUID.

Permission – Configures whether a GATT client device can access the attribute's value. The configurable permissions are as follows:

– GATT_PERMIT_READ //readable

– GATT_PERMIT_WRITE // Writable

– GATT_PERMIT_AUTHEN_READ // Authentication required to read

– GATT_PERMIT_AUTHEN_WRITE //Authorization required to write

– GATT_PERMIT_AUTHOR_READ // Authorization required to read

– GATT_PERMIT_ENCRYPT_READ // Encrypted reading required

– GATT_PERMIT_ENCRYPT_WRITE // Encrypted write required

Handle – The handle assigned by GATTServApp. Handles are automatically assigned in order.

pValue – pointer to the property value. Its length cannot be changed after initialization. Maximum size 512 bytes.

Let's create the property table in the Peripheral project:

First create the service properties:

```
1.      // Simple Profile Service
2. {
3.          { ATT_BT_UUID_SIZE, primaryServiceUUID }, /* type */
4.          GATT_PERMIT_READ,                         /* permissions */
5.          0,                                        /* handle */
6.          (uint8_t *)&simpleProfileService          /* pValue */
7. },
```

This attribute is the primary service UUID (0x2800) defined by the Bluetooth SIG. GATT clients must read this attribute.

So set the permission to readable. pValue is a pointer to the UUID of the service, which is customized to 0xFFE0.

```
1. // Simple Profile Service attribute
2. static const gattAttrType_t simpleProfileService = { ATT_BT_UUID_SIZE, simpl
       eProfileServUUID };
```

Then create the characteristic's declaration, value, user description, and client characteristic configuration, as described in Section 5.5.1.

```
1.      // Characteristic 1 Declaration
2. {
3.          { ATT_BT_UUID_SIZE, characterUUID },
4.          GATT_PERMIT_READ,
5.          0,
6.          &simpleProfileChar1Props
7. },
```

The type of the attribute characteristic declaration needs to be set to the BluetoothSIG defined

Characteristic UUID value (0x2803), and the GATT client must read this UUID, so its permission is set to readable.

The value refers to the properties of this feature and is readable and writable.

```
1. // Simple Profile Characteristic 1 Properties
```

```
2. static uint8_t simpleProfileChar1Props = GATT_PROP_READ | GATT_PROP_WRITE;
3.
4. // Characteristic Value 1
5. {
6.         { ATT_BT_UUID_SIZE, simpleProfilechar1UUID },
7.         GATT_PERMIT_READ | GATT_PERMIT_WRITE,
8.         0,
9.         simpleProfileChar1
10. },
```

In the characteristic value, the type is set to a custom UUID (0xFFF1). Since the attribute of the characteristic value is readable and writable, The permission to set the value is readable and writable. pValue points to the location of the actual value, as follows:

```
1. // Characteristic 1 Value
2. static uint8_t simpleProfileChar1[SIMPLEPROFILE_CHAR1_LEN] = { 0 };
3.
4. // Characteristic 1 User Description
5. {
6.         { ATT_BT_UUID_SIZE, charUserDescUUID },
7.         GATT_PERMIT_READ,
8.         0,
9.         simpleProfileChar1UserDesp
10. },
```

In the user description, the type is set to the characteristic UUID value defined by Bluetooth SIG (0x2901), and its permissions are set to Readable. The value is a user-defined string, as follows:

```
1. // Simple Profile Characteristic 1 User Description
2. static uint8_t simpleProfileChar1UserDesp[] = "Characteristic 1\0";
3.
4. // Characteristic 4 configuration
5. {
6.         { ATT_BT_UUID_SIZE, clientCharCfgUUID },
7.         GATT_PERMIT_READ | GATT_PERMIT_WRITE,
8.         0,
9.         (uint8_t *)simpleProfileChar4Config
10. },
```

The type must be set to the client characteristic configuration UUID (0x2902) defined by Bletooth SIG. GATT clients must This needs to be read and written, so the permissions are set to read and write. pValue points to the address of the client characteristic configuration value.

```
1. static gattCharCfg_t simpleProfileChar4Config[4];
```

5.5.4.2.2 Adding Services

When the Bluetooth protocol stack is initialized, it must add the GATT services it supports. Services include the GATT services required by the protocol stack.
Services such as GGS_AddService and GATTServApp_AddService, as well as some user-defined services such as
SimpleProfile_AddService in the Peripheral project. Take SimpleProfile_AddService() as an example.
These functions do the following:

First, you need to define the client characteristic configuration, namely Client Characteristic Configuration (CCC).

```
1. static gattCharCfg_t simpleProfileChar4Config[4];
```

Then initialize CCC.

For each CCC in the profile, the GATTServApp_InitCharCfg() function must be called. This function uses
Initializes CCC with information from a previously bound connection. If the function cannot find the information, sets the initial values to default values.

```
1. // Initialize Client Characteristic Configuration attributes
2. GATTServApp_InitCharCfg(INVALID_CONNHANDLE, simpleProfileChar4Config);
```

Finally register the profile through GATTServApp.

The GATTServApp_RegisterService() function passes the simpleProfileAttrTb1 attribute table of the profile to
Passed to GATTServApp in order to add the profile's attributes to the application-wide attribute table managed by the stack.

```
1. // Register GATT attribute list and CBs with GATT Server App
2.     status = GATTServApp_RegisterService( simpleProfileAttrTbl,
3.                                           GATT_NUM_ATTRS( simpleProfileAttrTbl ),
4.                                           GATT_MAX_ENCRYPT_KEY_SIZE,
5.                                           &simpleProfileCBs );
```

5.5.4.2.3 Registering application callback function

In the Peripheral project, whenever the GATT client writes a characteristic value, the GATTProfile calls the application
To use the callback function, you first need to set the callback function during initialization.

```
1. bStatus_t SimpleProfile_RegisterAppCBs( simpleProfileCBs_t *appCallbacks)
2. {
3.     if ( appCallbacks )
4.     {
5.         simpleProfile_AppCBs = appCallbacks;
6.
7.         return ( SUCCESS );
8.     }
9.     else
10. {
11.         return (bleAlreadyInRequestedMode);
12. }
13. }
```

The callback function is as follows:

```
1. // Callback when a characteristic value has changed
2. typedef void (*simpleProfileChange_t)( uint8_t paramID );
3.
4. typedef struct
5. {
6.     simpleProfileChange_t     pfnSimpleProfileChange; // Called when chara
7. cteristic value changes
8. } simpleProfileCBs_t;
```

The callback function must point to an application of this type, as follows:

```
1. // Simple GATT Profile Callbacks
2. static simpleProfileCBs_t Peripheral_SimpleProfileCBs =
3. {
4.     simpleProfileChangeCB     // Charactersitic value change callback
5. };
```

5.5.4.2.4 Read and write callback functions

When the configuration file is read or written, a corresponding callback function is required. Its registration method is the same as the application callback function. For details, please refer to Peripheral Engineering.

5.5.4.2.5 Obtaining and setting configuration files

The configuration file contains functions for reading and writing characteristics. Figure 5.9 describes the logic of the application setting configuration file parameters.

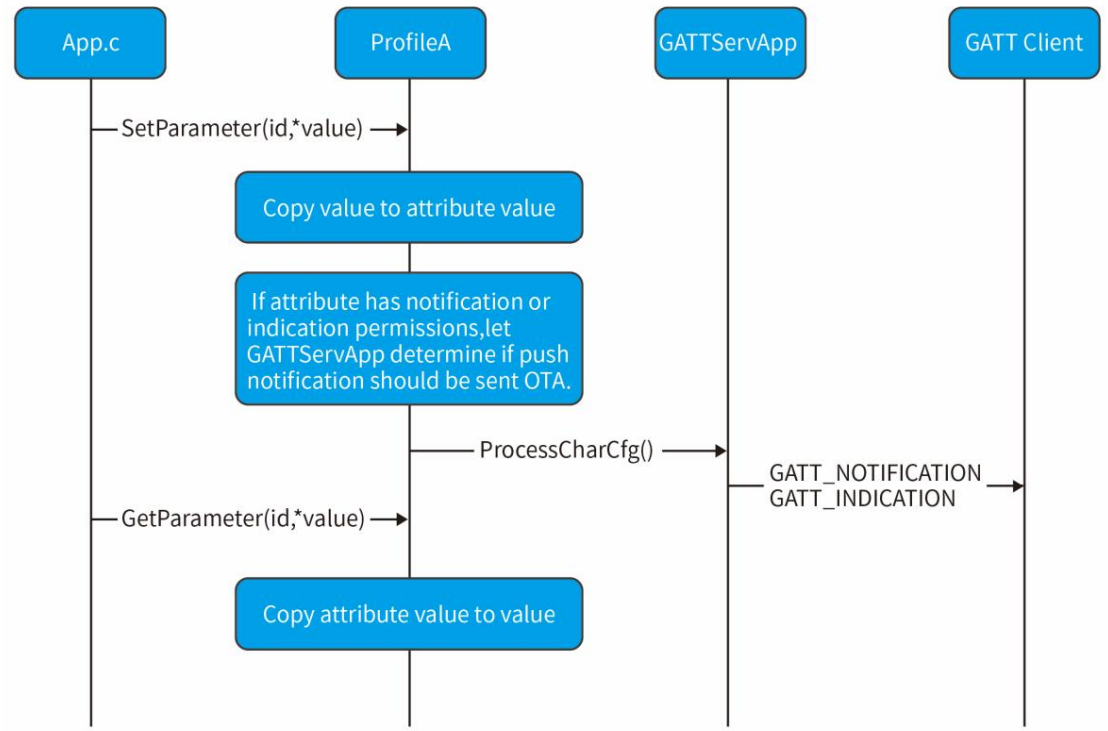


Figure 5.9 Getting and setting configuration file parameters

The application code is as follows:

```
1. SimpleProfile_SetParameter( SIMPLEPROFILE_CHAR1, SIMPLEPROFILE_CHAR1_LEN, ch
   arValue1 );
```

5.6 Logical Link Control and Adaptation Protocol

The Logical Link Control and Adaptation Protocol layer (L2CAP) sits on top of the HCI layer and transfers data between the upper layers of the host (GAP layer, GATT layer, application layer, etc.) and the lower protocol stack. This upper layer has the segmentation and reassembly capabilities of L2CAP, enabling higher-level protocols and applications to send and receive data packets with a length of 64KB. It is also capable of handling protocol multiplexing to provide multiple connections and multiple connection types (through an air interface), while providing quality of service support and grouped communication. The CH58x Bluetooth protocol stack supports an effective maximum MTU of 247.

5.7 Host and Controller Interaction

HCI (HostControllerInterface) connects the host and the controller and converts the host's operations into HCI instructions. The BLE CoreSpec supports four types of HCI: UART, USB, SDIO, and 3-Wire UART.

For a single Bluetooth chip with a full protocol stack, you only need to call the API interface function. At this time, HCI is a function call and callback. For products with only a controller, that is, the main control chip is used to operate the BLE chip, and the BLE chip is connected to the main control chip as an external chip. At this time, the main control chip only needs to interact with the BLE chip through standard HCI instructions (usually UART).

The HCI discussed in this guide is function calls and function callbacks.

6. Create a BLE application

6.1 Overview

After reading the previous chapters, you should understand how to implement a Bluetooth Low Energy application.
Getting started writing Bluetooth low energy applications, and some considerations.

6.2 Configuring the Bluetooth Protocol Stack

First, you need to determine the role of this device. We provide the following roles:

Central

Peripheral

Broadcaster

Observer

Selecting different roles requires calling different role initialization APIs. Please refer to Section 5.3 for details .

6.3 Defining Bluetooth Low Energy Behavior

Use the Bluetooth Low Energy protocol stack API to define system behaviors, such as adding profiles, adding GATT databases, configuring
Set to safe mode, etc. See Chapter 5 for details.

6.4 Defining Application Tasks

Make sure that the application contains callback functions for the protocol stack and event handlers from TMOS.
Add other tasks as described in the chapter.

6.5 Application Profile

Configure DCDC enable, RTC clock, sleep function, MAC address, low power Bluetooth protocol in the config.h file.
For details on the RAM size used by the stack, etc., see the config.h file.

It should be noted that WAKE_UP_RTC_MAX_TIME is the time to wait for the 32M crystal to stabilize.
The wake-up time is affected by factors such as body, voltage, stability, etc. You need to add a buffer time to the wake-up time to improve stability.

6.6 Limiting application processing during Bluetooth Low Energy operation

Due to the timing dependency of the Bluetooth Low Energy protocol, the controller must
If it is not processed in time, it will cause retransmission or disconnection. And TOMS is not multi-threaded, so when
When Bluetooth Low Energy is processing a transaction, other tasks must be stopped to allow the controller to process. So make sure not to occupy the
If a large number of events require complex processing, please refer to Section 3.3 to split them.

6.7 Interruptions

During the operation of low-power Bluetooth, the time needs to be calculated through the RTC timer, so during this period, do not disable the global
The interrupt service routine should not be too long, otherwise the low-power Bluetooth operation will be interrupted for a long time, which will cause the connection to be disconnected.
Connect and disconnect.

7. Create a simple RF application

7.1 Overview

RF applications are based on RF transmission and reception PHY to achieve wireless communication in the 2.4GHz band. Differences from BLE The problem is that the RF application does not establish the BLE protocol.

7.2 Configuring the protocol stack

First initialize the Bluetooth library:

```
1. CH58X_BLEInit() ;
```

Then configure the role of this device to RF Role:

```
1. RF_RoleInit ( );
```

7.3 Defining Application Tasks

Register the RF task, initialize the RF function, and register the RF callback function:

```
1. taskID = TMOS_ProcessEventRegister(RF_ProcessEvent);
2.         rfConfig.accessAddress = 0x71764129; // 0x55555555 and
3. 0xAAAAAAAA (It is recommended that no more than 24 bit inversions and no more than 6 consecutive 0s or 1s)
4.         rfConfig.CRCInit = 0x555555;
5.         rfConfig.Channel = 8;
6.         rfConfig.Frequency = 2480000;
7.         rfConfig.LLEMode = LLE_MODE_BASIC|LLE_MODE_EX_CHANNEL|LLE_MODE_NON_RSSI;
8. // Enable LLE_MODE_EX_CHANNEL to select rfConfig.Frequency as the communication frequency
9.         rfConfig.rfStatusCB = RF_2G4StatusCallBack;
10.        state = RF_Config( &rfConfig );
```

7.4 Application Configuration File

Configure DCDC enable, RTC clock, sleep function, MAC address, low power Bluetooth protocol in the config.h file.

For details on the RAM size used by the stack, etc., see the config.h file.

It should be noted that WAKE_UP_RTC_MAX_TIME is the time to wait for the 32M crystal to stabilize.

The wake-up time is affected by factors such as body, voltage, stability, etc. You need to add a buffer time to the wake-up time to improve stability.

7.5 RF Communications

7.5.1 Basic Mode

In Basic mode, you only need to keep the receiver in receiving mode, that is, call the RF_RX() function.

The problem is that after receiving the data, you need to call the RF_RX() function again to put the device in receiving mode again, and do not directly

Calling the RF transceiver function in the RF_2G4StatusCallBack() callback function may cause confusion in its status.

The communication diagram is as follows:

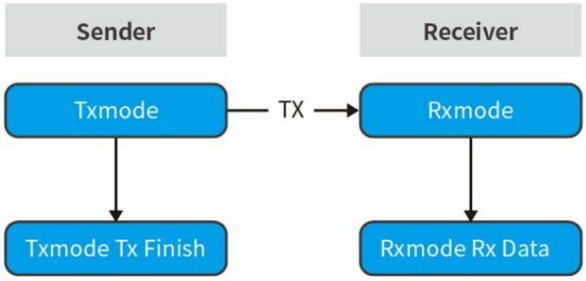


Figure 7.1 Basic mode communication diagram

The RF sending API is RF_Tx(), please refer to Section 8.6 for details .

When RF receives data, it will enter the callback function RF_2G4StatusCallBack() and get the received data in the callback function. data.

7.5.2 Auto Mode

Since the Basic mode is only one-way transmission, the user cannot know whether the communication is successful, thus generating the Auto mode. Mode.

The Auto mode adds a receiving response mechanism based on the Basic mode, that is, after the receiver receives the data, The data is sent to the sender to notify the sender that the data has been successfully received.

The communication diagram is as follows:

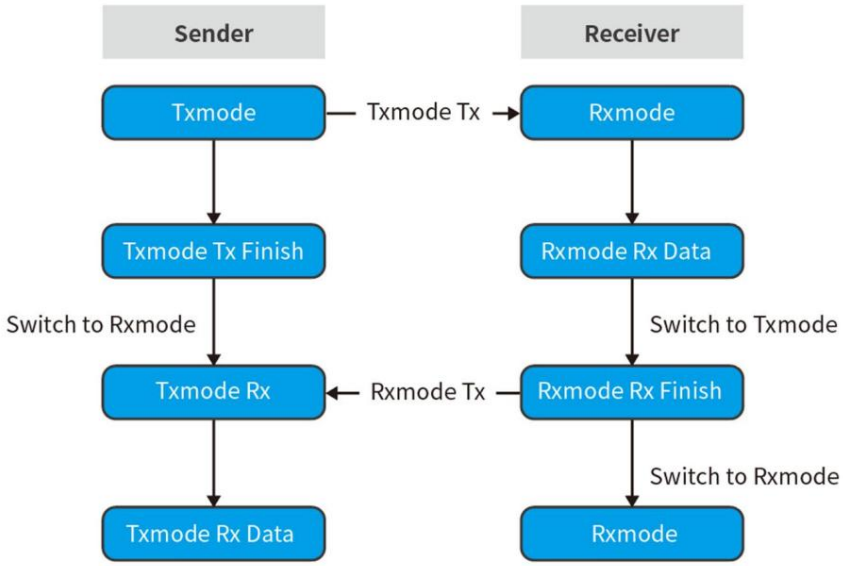


Figure 7.2 Auto mode communication diagram

In Auto mode, RF will automatically switch to receiving mode after sending data. The timeout period of this receiving mode is 3ms. If no data is received within 3ms, the receiving mode will be turned off. The received data and the timeout status are returned through the callback function. Back to the application layer.

7.5.2.1 Automatic frequency hopping

The automatic frequency hopping solution based on RF Auto mode design can effectively solve the interference problem of 2.4GHz channels.

To use the frequency hopping function, you need to actively enable the frequency hopping receiving or sending events:

```
1. // Turn on frequency hopping transmission
2. if( events & SBP_RF_CHANNEL_HOP_TX_EVT ){
```

```
3.          PRINT("\n---------- hop tx...\n");
4.          if( RF_FrequencyHoppingTx(16)){
5.              tmos_start_task(taskID             , SBP_RF_CHANNEL_HOP_TX_EVT ,100 );
6.          }
7.          return events^SBP_RF_CHANNEL_HOP_TX_EVT;
8. }
9. // Enable frequency hopping reception
10. if( events & SBP_RF_CHANNEL_HOP_RX_EVT ){
11.         PRINT("hop rx...\n");
12.         if( RF_FrequencyHoppingRx(200))
13.         {
14.             tmos_start_task(taskID             , SBP_RF_CHANNEL_HOP_RX_EVT ,400 );
15.         }
16.         else
17.         {
18.             RF_Rx(TX_DATA,10,0xFF,0xFF);
19.         }
20.         return events^SBP_RF_CHANNEL_HOP_RX_EVT;
twenty one. }
```

After configuring the RF communication mode to automatic mode, the sender starts the frequency hopping transmission event:

```
1. tmos_set_event(taskID, SBP_RF_CHANNEL_HOP_TX_EVT);
```

Receive and start frequency hopping reception event:

```
1. tmos_set_event(taskID, SBP_RF_CHANNEL_HOP_RX_EVT);
```

The frequency hopping function can be realized.

It should be noted that if the receiver is already in receiving mode, it is necessary to turn off RF first (call RF_Shut() function).

Then enable the frequency hopping receive event.

8. API

8.1 TMOS API

8.1.1 Instructions

1. bStatus_t TMOS_TimerInit(pfnGetSysClock fnGetClock)

TMOS clock initialization.

	describe
Parameter pfnGetSysClock	0: Select RTC as system clock Other valid values: other clock acquisition interfaces, such as SYS_GetSysTickCnt()
return	0: SUCCESS 1: FAILURE

1. tmosTaskID TMOS_ProcessEventRegister(pTaskEventHandlerFn eventCb)

Register event callback function, which is usually executed first when registering a task.

parameter	describe
eventCb	TMOS task callback function
return	The assigned ID value, OxFF means invalid

1. bStatus_t tmos_set_event(tmosTaskID taskID, tmosEvents event)

Immediately start the corresponding event in the task taskID, and execute it once when called.

parameter	describe
taskID	Task ID assigned by tmos
event	Events in the task
return	0: Success

1. bStatus_t tmos_start_task(tmosTaskID taskID, tmosEvents event, tmosTimer time)

After **a delay of time*625ÿs , start the corresponding event in the taskID task and execute it once for each call.**

parameter	describe
taskID	Task ID assigned by tmos
event	Events in the task
time	Delayed time
return	0: Success

1. bStatus_t tmos_stop_event(tmosTaskID taskID, tmosEvents event)

Stop an event. After calling this function, the event will not take effect.

parameter	describe
taskID	Task ID assigned by tmos
event	Events in the task
return	0: Success

1. bStatus_t tmos_clear_event(tmosTaskID taskID, tmosEvents event)

Clean up an event that has timed out. Note that this cannot be executed in its own event function.

parameter	describe
taskID	Task ID assigned by tmos
event	Events in the task
return	0: Success

1. bStatus_t tmos_start_reload_task(tmosTaskID taskID, tmosEvents event, tmosTimer time)

Delay time*625ÿs to execute event event, call a loop execution, unless tmos_stop_task is run to close Lose.

parameter	describe
taskID	Task ID assigned by tmos
event	Events in the task
time	Delayed time
return	0: Success

1. tmosTimer tmos_get_task_timer(tmosTaskID taskID, tmosEvents event)

Gets the number of ticks until the event expires.

parameter	describe
taskID	Task ID assigned by tmos
event	Events in the task
return	!0: The number of ticks until the event expires 0: Event not found

1. uint32_t TMOS_GetSystemClock(void)

Returns the tmos system running time in units of 625ÿs, such as 1600=1s.

parameter	describe

return	TMOS runtime

1. void TMOS_SystemProcess(void)

The system processing function of tmos needs to be continuously run in the main function.

1. bStatus_t tmos_msg_send(tmosTaskID taskID, uint8_t *msg_ptr)

Send a message to a task. When this function is called, the message event event of the corresponding task will be set to 1 immediately to take effect.

parameter	describe
taskID	tmos assigned task ID message
msg_ptr	pointer
returns	SUCCESS: Success INVALID_TASK: Invalid task ID INVALID_MSG_POINTER: Invalid message pointer

1. uint8_t *tmos_msg_receive(tmosTaskID taskID)

Receive a message.

parameter	describe
taskID	Task ID assigned by tmos
return	Message received or no message to be received (NULL)

1. uint8_t *tmos_msg_allocate(uint16_t len)

Allocate memory space for the message.

Parameter Description	
len	Message length
return	The requested buffer pointer NULL: Application failed

1. bStatus_t tmos_msg_deallocate(uint8_t *msg_ptr)

Release the memory space occupied by the message.

Parameter Description	
msg_ptr	Message pointer
return	0: Success

1. uint8_t tmos_snv_read(uint8_t id, uint8_t len, void *pBuf)

Read data from NV.

Note: The read and write operations of the NV area should be called before the TMOS system is running.

parameter	describe
id	Valid NV item ID Length
len	of read data
pBuf	Pointer to the data to be read
Return	SUCCESS NV_OPER_FAILED: Failed

1. void TMOS_TimerIRQHandler(void)

TMOS timer interrupt function.

The following functions save more memory space than C language library functions

1. uint32_t tmos_rand(void)

Generates pseudo-random numbers.

Parameters	describe
returned	Pseudo-random numbers

1. bool tmos_memcmp(const void *src1, const void *src2, uint32_t len)

Compares memory area src1 with the first len bytes of memory area src2.

parameter	describe
src1	Memory block pointer
src2	Memory block pointer
len	The number of bytes to be compared
return	1: Same 0: Different

1. bool tmos_isbufset(uint8_t *buf, uint8_t val, uint32_t len)

Compares the given data to see if they are all the given values.

parameter	describe
Buf	Buffer address
val	Numeric
len	The length of the data
return	1: Same 0: Different

1. uint32_t tmos_strlen(char *pString)

Computes the length of the string pString, up to but not including the terminating null character.

parameter	describe
pString	The string to calculate the length of
return	Length of the string

1. void tmos_memset(void * pDst, uint8_t Value, uint32_t len)

Copies the characters Value to the first len characters of the string pointed to by the parameter pDst.

	describe
Parameter pDst	The memory block to fill
Value	The value to be set
len	The number of characters to be set to the value
return	Pointer to the storage area pDst

1. void tmos_memcpy(void *dst, const void *src, uint32_t len)

Copies len bytes from memory area src to memory area dst.

parameter	describe
dst	The target array used to store the copied content, the type is cast to void* pointer
src	The data source to be copied, type cast to void* pointer
len	The number of bytes to be copied
return	Pointer to the destination storage area dst

8.2 GAP API

8.2.1 Instructions

1. bStatus_t GAP_SetParamValue(uint16_t paramID, uint16_t paramValue)

Set GAP parameter value. Use this function to change the default GAP parameter value.

Parameter Description	
paramID	Parameter ID, refer to 8.2.2
paramValue New parameter value	
Returns SUCCESS or INVALIDPARAMETER (invalid parameter ID)	

1. uint16 GAP_GetParamValue(uint16_t paramID)

Get the GAP parameter value.

	describe
Parameter	Parameter ID, refer to 8.1.2 GAP
paramID Return	parameter value; if the parameter ID is invalid, return 0xFFFF

8.2.2 Configuration Parameters

The following are commonly used parameter IDs. For detailed parameter IDs, please refer to CH58xBLE.LIB.h.

Parameter	describe
ID TGAP_GEN_DISC_ADV_MIN	The broadcast duration of the general broadcast mode, unit: 0.625ms (default Default value: 0)
TGAP_LIM_ADV_TIMEOUT	The broadcast duration of the limited discoverable broadcast mode, unit: 1s (default Recognition value: 180)
TGAP_DISC_ADV_INT_MIN	Minimum broadcast interval, unit: 0.625ms (default: 160)
TGAP_DISC_ADV_INT_MAX	Maximum broadcast interval, unit: 0.625ms (default: 160)
TGAP_DISC_SCAN	Scan duration, unit: 0.625ms (default value: 16384)
TGAP_DISC_SCAN_INT	Scan interval, unit: 0.625ms (default: 16)
TGAP_DISC_SCAN_WIND	Scan window, unit: 0.625ms (default: 16)
TGAP_CONN_EST_SCAN_INT	Scan interval for establishing a connection, unit: 0.625ms (default value: 16)
TGAP_CONN_EST_SCAN_WIND	Scan window for establishing a connection, unit: 0.625ms (default value: 16)
TGAP_CONN_EST_INT_MIN	The minimum connection interval for establishing a connection, unit: 1.25ms (default Value: 80)
TGAP_CONN_EST_INT_MAX	The maximum connection interval for establishing a connection, unit: 1.25ms (default Value: 80)
TGAP_CONN_EST_SUPERV_TIMEOUT Connection management timeout for establishing a connection, unit: 10ms (default Recognition value: 2000)	
TGAP_CONN_EST_LATENCY	Slave delay for establishing connection (default: 0)

8.2.3 Events

This section introduces the events related to the GAP layer. The relevant declarations can be found in the CH58xBLE_LIB.h file.

Some events are passed directly to the application, and some are handled by GAPRole and GAPBondMgr.

Regardless of which layer they are passed to, they are delivered as GAP_MSG_EVENTs with a header:

```
typedef struct
{
    tmos_event_hdr_t hdr;                //!< GAP_MSG_EVENT and status
    uint8_t opcode;                      //!< GAP type of command. Ref: @ref GAP
_MSG_EVENT_DEFINES
} gapEventHdr_t;
```

The following are common event names and event message formats. For details, please refer to CH58xBLE_LIB.h.

GAP_DEVICE_INIT_DONE_EVENT: This event is set when device initialization is complete.

```
typedef struct
{
```

```
3.      tmos_event_hdr_t hdr;                   //!< GAP_MSG_EVENT and status
4.      uint8_t opcode;                         //!<GAP_DEVICE_INIT_DONE_EVENT
5.      uint8_t devAddr[B_ADDR_LEN];            //!< Device's BD_ADDR
6.      uint16_t dataPktLen;                    //!< HC_LE_Data_Packet_Length
7.      uint8_t numDataPkts;                    //!< HC_Total_Num_LE_Data_Packets
8. } gapDeviceInitDoneEvent_t;
```

GAP_DEVICE_DISCOVERY_EVENT: This event is set when the device discovery process is completed.

```
1. typedef struct
2. {
3.      tmos_event_hdr_t hdr; //!< GAP_MSG_EVENT and status
4.      uint8_t opcode;             //!< GAP_DEVICE_DISCOVERY_EVENT
5.      uint8_t numDevs;            //!< Number of devices found during scan
6.      gapDevRec_t *pDevList; //!< array of device records
7. } gapDevDiscEvent_t;
```

· GAP_END_DISCOVERABLE_DONE_EVENT: This event is set when the broadcast ends.

```
1. typedef struct
2. {
3. tmos_event_hdr_t hdr; //!< GAP_MSG_EVENT and status
4. uint8_t opcode;                  //!< GAP_END_DISCOVERABLE_DONE_EVENT
5. } gapEndDiscoverableRspEvent_t;
```

GAP_LINK_ESTABLISHED_EVENT: This event is set after the connection is established.

```
1. typedef struct
2. {
3.      tmos_event_hdr_t hdr;           //!< GAP_MSG_EVENT and status
4.      uint8_t opcode;                 //!< GAP_LINK_ESTABLISHED_EVENT
5.      uint8_t devAddrType;            //!< Device address type: @ref GAP_ADDR_TYPE_
6. DEFINES
7.      uint8_t devAddr[B_ADDR_LEN]; //!< Device address of link
8.      uint16_t connectionHandle; //!< Connection Handle from controller used t
9. o ref the device
10. uint8_t connRole;               //!< Connection formed as Master or Slave
11. uint16_t connInterval;          //!< Connection Interval
12. uint16_t connLatency;           //!< Connection Latency
13. uint16_t connTimeout;           //!< Connection Timeout
14. uint8_t clockAccuracy;          //!< Clock Accuracy
15. } gapEstLinkReqEvent_t;
```

GAP_LINK_TERMINATED_EVENT: This event is set after the connection is terminated.

```
typedef struct
{
    tmos_event_hdr_t hdr; //!< GAP_MSG_EVENT and status
    uint8_t opcode;              //!< GAP_LINK_TERMINATED_EVENT
    uint16_t connectionHandle; //!< connection Handle
    uint8_t reason;              //!< termination reason from LL
    uint8_t connRole;
} gapTerminateLinkEvent_t;
```

GAP_LINK_PARAM_UPDATE_EVENT: This event is set after receiving a parameter update event.

```
typedef struct
{
    tmos_event_hdr_t hdr;          //!< GAP_MSG_EVENT and status
    uint8_t opcode;                //!< GAP_LINK_PARAM_UPDATE_EVENT
    uint8_t status;                //!< bStatus_t
    uint16_t connectionHandle; //!< Connection handle of the update
    uint16_t connInterval;         //!< Requested connection interval
    uint16_t connLatency;          //!< Requested connection latency
    uint16_t connTimeout;          //!< Requested connection timeout
} gapLinkUpdateEvent_t;
```

GAP_DEVICE_INFO_EVENT: This event is set by the discovered device during device discovery.

```
typedef struct
{
    tmos_event_hdr_t hdr;          //!< GAP_MSG_EVENT and status
    uint8_t opcode;                //!< GAP_DEVICE_INFO_EVENT
    uint8_t eventType;             //!< Advertisement Type: @ref GAP_ADVERTISEMEN
T_REPORT_TYPE_DEFINES
    uint8_t addrType;              //!< address type: @ref GAP_ADDR_TYPE_DEFINES
    uint8_t addr[B_ADDR_LEN]; //!< Address of the advertisement or SCAN_RSP
    int8_t rssi;                   //!< Advertisement or SCAN_RSP RSSI
uint8_t dataLen;                   //!< Length (in bytes) of the data field (evtD
ata)
    uint8_t *pEvtData;             //!< Data field of advertisement or SCAN_RSP
} gapDeviceInfoEvent_t;
```

8.3 GAPRole API

8.3.1 GAPRole Common Role API

8.3.1.1 Instructions

1. bStatus_t GAPRole_SetParameter(uint16_t param, uint16_t len, void *pValue)

Set the GAP role parameters.

Parameter Description	
param	Configuration parameter ID, see section 8.2.1.2 for details
len	The length of the data written
pValue	Pointer to the parameter value to be set. This pointer depends on the parameter ID and will be cast to Appropriate data type.
return	SUCCESS INVALIDPARAMETER: Invalid parameter bleInvalidRange: parameter length is invalid blePending: The last parameter update has not yet ended bleIncorrectMode: Mode error

1. bStatus_t GAPRole_GetParameter(uint16_t param, void *pValue)

Get GAP role parameters.

Parameter Description	
param	Configuration parameter ID, see section 8.2.1.2 for details
pValue	Pointer to where to get the parameter. This pointer depends on the parameter ID and will be cast to Change to the appropriate data type.
return	SUCCESS INVALIDPARAMETER: Invalid parameter

1. bStatus_t GAPRole_TerminateLink(uint16_t connHandle)

Disconnects the connection specified by the current connHandle.

Parameter Description	
connHandle connection handle	
return	SUCCESS bleIncorrectMode: Mode error

1. bStatus_t GAPRole_ReadRssiCmd(uint16_t connHandle)

Read the RSSI value of the connection specified by the current connHandle.

Parameter Description	
connHandle connection handle	
return	SUCCESS

	0x02: No valid connection

8.3.1.2 Common configurable parameters

	Read/Write Size Description	
Parameter	Read-only uint8 device address	
GAPROLE_BD_ADDR GAPRPLE_ADVERT_ENABLE	Readable and writable uint8	Enable or disable broadcast, enabled by default
GAPROLE_ADVERT_DATA	Readable and writable ÿ 240	broadcast data, all 0 by default.
GAPROLE_SCAN_RSP_DATA	Readable and writable ÿ240	Scan response data, all 0 by default
GAPROLE_ADV_EVENT_TYPE	Readable and writable uint8	Broadcast type, non-directional broadcast can be connected by default
GAPROLE_MIN_CONN_INTERV AL	Readable and writable uint16	minimum connection interval, range: 1.5ms~4s, The default value is 8.5ms.
GAPROLE_MAX_CONN_INTERV AL	Readable and writable uint16	maximum connection interval, range: 1.5ms~4s, The default value is 8.5ms.

8.3.1.3 Callback Function

```
1. /**
2. * Callback when the device has read a new RSSI value during a connection.
3. */
4. typedef void (*gapRolesRssiRead_t)(uint16_t connHandle, int8_t newRSSI )
```

This function is the callback function for reading RSSI. Its pointer points to the application so that GAPRole can return the event to the application.
To the application. Delivery is as follows:

```
1. // GAP Role Callbacks
2. static gapCentralRoleCB_t centralRoleCB =
3. {
4.     centralRssiCB,          // RSSI callback
5.     centralEventCB,         //Event callback
6.     centralHciMTUChangeCB   //MTU change callback
7. };
```

8.3.2 GAPRolePeripheral Role API

8.3.2.1 Instructions

```
1. bStatus_t GAPRole_PeripheralInit( void )
```

Bluetooth slave GAPRole task initialization.

Parameters	describe
returned	SUCCESS bleInvalidRange: Parameter out of range

```
1. bStatus_t GAPRole_PeripheralStartDevice( uint8_t taskid, gapBondCBs_t *pCB,g
       apRolesCBs_t *pAppCallbacks )
```

Initialize the Bluetooth slave device.

Parameter Description	
taskid	The task ID assigned by tmos
pCB	is bound to the callback function, including key callback and pairing status callback
pAppCallbacks GAPRole	callback function, including device status callback, RSSI callback, parameter update callback Tune
return	SUCCESS bleAlreadyInRequestedMode: The device has already been initialized

```
1. bStatus_t GAPRole_PeripheralConnParamUpdateReq( uint16_t connHandle,
2.                                                 uint16_t minConnInterval,
3.                                                 uint16_t maxConnInterval,
4.                                                 uint16_t latency,
5.                                                 uint16_t connTimeout,
6.                                                 uint8_t taskId)
```

Bluetooth slave connection parameters updated.

Note: Different from GAPRole_UpdateLink(), , This is the connection parameter negotiated between the slave and the host. and GAPRole_UpdateLink() allows the host to directly configure the connection parameters.

Parameter Description	
connHandle connection handle	
minConnInterval minimum connection interval	
maxConnInterval Maximum connection interval	
latency	Number of slave device delay events
connTimeout Connection timeout	
taskID	toms The assigned task ID
return	SUCCESS: Parameter upload successful BleNotConnected: No connection so parameters cannot be updated bleInvalidRange: parameter error

8.3.2.2 Callback Function

```
1. typedef struct
2. {
3.     gapRolesStateNotify_t pfnStateChange; //!< Whenever the device changes
4. state
5.     gapRolesRssiRead_t pfnRssiRead; //!< When a valid RSSI is read from
6. Controller
7.     gapRolesParamUpdateCB_t pfnParamUpdate; //!< When the connection
8. Parameteres are updated
9. } gapRolesCBs_t;
```

Slave status callback function:

```
1. /**
2.          * Callback when the device has been started. Callback event to
3.          * the Notify of a state change.
4. */
5. void (*gapRolesStateNotify_t )( gapRole_States_t newState,
       gapRoleEvent_t *pEvent);
```

The statuses are classified into the following categories:

GAPROLE_INIT //Wait for startup

GAPROLE_STARTED // Initialization completed but not broadcast yet

GAPROLE_ADVERTISING //Broadcasting

GAPROLE_WAITING //The device is started but not broadcasting, and is waiting to broadcast again

GAPROLE_CONNECTED //Connection status

GAPROLE_CONNECTED_ADV //Connected and broadcasting

GAPROLE_ERROR //Invalid status. If this status is used, it indicates an error.

Slave parameter update callback function:

```
1. /**
2.          * Callback when the connection parameters are updated.
3.          */
4. typedef void (*gapRolesParamUpdateCB_t)( uint16_t connHandle,
5.                                          uint16_t connInterval,
6.                                          uint16_t connSlaveLatency,
7.                                          uint16_t connTimeout );
```

This callback function is called when the parameters are updated successfully.

8.3.3 GAPRole Central Role API

8.3.3.1 Instructions

```
1. bStatus_t GAPRole_CentralInit( void )
```

Host GAPRole task initialized.

Parameters	describe
returned	SUCCESS bleInvalidRange: Parameter out of range

```
1. bStatus_t GAPRole_CentralStartDevice( uint8_t taskid, gapBondCBs_t *pCB, gap
       CentralRoleCB_t *pAppCallbacks )
```

Start the device in the host role. This function is usually called once during system startup.

parameter	describe
taskid	The task ID assigned by tmos
pCB	is bound to the callback function, including key callback and pairing status callback
pAppCallbacks GAPRole	callback function, including device status callback, RSSI callback, parameter update callback Tune
return	SUCCESS bleAlreadyInRequestedMode: The device has been started

1. bStatus_t GAPRole_CentralStartDiscovery(uint8_t mode, uint8_t activeScan, u int8_t whiteList)

Host scan parameter configuration.

Parameter Description	
mode	Scan mode, divided into: DEVDISC_MODE_NONDISCOVERABLE: Do not set DEVDISC_MODE_GENERAL: Scan for generic discoverable devices DEVDISC_MODE_LIMITED: Scan for limited discoverable devices DEVDISC_MODE_ALL: Scan all
activeScan	TRUE to enable scanning
whiteList	TRUE to scan only whitelisted devices
return	SUCCESS

1. bStatus_t GAPRole_CentralCancelDiscovery(void)

The host stops scanning.

Parameters	describe
returned	SUCCESS bleInvalidTaskID: No task is being scanned bleIncorrectMode: Not in scanning mode

1. bStatus_t GAPRole_CentralEstablishLink(uint8_t highDutyCycle, uint8_t white List, uint8_t addrTypePeer, uint8_t *peerAddr)

Connect to the peer device.

Parameter Description	
highDutyCycle TURE	Enable high duty cycle scanning
whiteList	TURE Use whitelist
addrTypePeer	The address type of the peer device, including: ADDRTYPE_PUBLIC: BD_ADDR ADDRTYPE_STATIC: static address

	ADDRTYPE_PRIVATE_NONRESOLVE: Unresolvable private address ADDRTYPE_PRIVATE_RESOLVE: Resolvable private address
peerAddr peer device address	
return	SUCCESS: Successfully connected bleIncorrectMode: Invalid profile bleNotReady: Scanning in progress bleAlreadyInRequestedMode: Temporarily unable to process bleNoResources: Too many connections

8.3.3.2 Callback Function

Pointers to these callback functions are passed from the application to the GAPRole so that the GAPRole can return events to the application.
Programs. They are passed as follows:

```
typedef struct
{
    gapRolesRssiRead_t rssiCB; //!< RSSI callback.
    pfnGapCentralRoleEventCB_t eventCB; //!< Event callback.
    pfnHciDataLenChangeEvCB_t ChangCB; //!< Length Change Event Callback.
} gapCentralRoleCB_t; // gapCentralRoleCB_t
```

Host RSSI callback function:

```
/**
* Callback when the device has read a new RSSI value during a connection.
*/
typedef void ( *gapRolesRssiRead_t )( uint16_t connHandle, int8_t newRSSI )
```

This function reports the RSSI to the application.

Host event callback function:

```
/**
    * Central Event Callback Function
    */
typedef void ( *pfnGapCentralRoleEventCB_t ) ( gapRoleEvent_t *pEvent );
//!< Pointer to event structure.
```

This callback is used to deliver GAP state change events to the application.

For callback events, please refer to Section 8.1.3 .

MTU interaction callback function:

```
typedef void (*pfnHciDataLenChangeEvCB_t)
(
    uint16_t connHandle,
    uint16_t maxTxOctets,
    uint16_t maxRxOctets
);
```

That is, the packet size for interacting with Bluetooth Low Energy.

8.4 GATT API

8.4.1 Instructions

8.4.1.1 Slave instructions

```
bStatus_t GATT_Indication( uint16_t connHandle, attHandleValueInd_t *pInd, u
    int8_t authenticated, uint8_t taskId )
```

The server indicates a characteristic value to the client and expects the attribute protocol layer to confirm that it has successfully received the indication.

It should be noted that memory needs to be released when a failure is returned.

Parameter Description	
connHandle connection handle	
pInd	Points to the command to be sent
authenticated Whether an authenticated connection is required	
taskId	Task ID assigned by tmos

```
bStatus_t GATT_Notification( uint16_t connHandle, attHandleValueNoti_t *pNot
    i, uint8_t authenticated )
```

The server notifies the client of the characteristic value, but does not expect any attribute protocol layer to confirm that the notification has been successfully received.

It should be noted that memory needs to be released when a failure is returned.

Parameter Description	
connHandle connection handle	
pInd	Point to the instruction to be notified
authenticated Whether an authenticated connection is required	

8.4.1.2 Host Commands

```
bStatus_t GATT_ExchangeMTU( uint16_t connHandle, attExchangeMTUReq_t *pReq,
    uint8_t taskId )
```

When the client supports a value greater than the default ATT_MTU value of the attribute protocol, the client uses this procedure to set ATT_MTU to Set to the maximum possible value that both devices can support.

Parameter Description	
connHandle connection handle	
pReq	Points to the command to be sent
taskID	The ID of the task being notified

1. bStatus_t GATT_DiscAllPrimaryServices(uint16_t connHandle, uint8_t taskId)

Discover all primary services on the server.

Parameter Description	
connHandle connection handle	
taskID	The ID of the task being notified

1. bStatus_t GATT_DiscPrimaryServiceByUUID(uint16_t connHandle, uint8_t *pUUID , uint8_t len, uint8_t taskId)

When only the UUID is known, the client can use this function to discover the primary service on the server.

There may be multiple primary services on the server, so the discovered primary service is identified by a UUID.

Parameter Description	
connHandle connection handle	
pUUID	Pointer to the UUID of the server to look up
len	Length of value
taskID	The ID of the task being notified

1. bStatus_t GATT_FindIncludedServices(uint16_t connHandle, uint16_t startHan dle, uint16_t endHandle, uint8_t taskId)

The client uses this function to find this service on the server. The service to be found is identified by the service handle scope.

Parameter Description	
connHandle connection handle	
startHandle starting handle	
endHandle End handle	
taskID	The ID of the task being notified

1. bStatus_t GATT_DiscAllChars(uint16_t connHandle, uint16_t startHandle, uint 16_t endHandle, uint8_t taskId)

A client can use this function to find all characteristic declarations on a server when only the service handle scope is known.

Parameter Description	
connHandle connection handle	
startHandle starting handle	

endHandle End handle	
taskID	The ID of the task being notified

1. bStatus_t GATT_DiscCharsByUUID(uint16_t connHandle, attReadByTypeReq_t *pRe
   q, uint8_t taskId)

This function can be used by a client to discover characteristics on a server when the service handle scope and characteristic UUID are known.

Parameter Description	
connHandle connection	handle
pReq	Pointer to the request to send
taskID	The ID of the task being notified

1. bStatus_t GATT_DiscAllCharDescs(uint16_t connHandle, uint16_t startHandle,
   uint16_t endHandle, uint8_t taskId)

When the handle scope of a feature is known, the client can use this procedure to find all feature descriptors in the feature definition. AttributeHandles and AttributeTypes.

Parameter Description	
connHandle connection	handle
startHandle starting handle	
endHandle End handle	
taskID	The ID of the task being notified

1. bStatus_t GATT_ReadCharValue(uint16_t connHandle, attReadReq_t *pReq, uint8
   _t taskId)

Once the client knows the characteristic handle, it can use this function to read the characteristic value from the server.

Parameter Description	
connHandle connection	handle
pReq	Pointer to the request to send
taskID	The ID of the task being notified

1. bStatus_t GATT_ReadUsingCharUUID(uint16_t connHandle, attReadByTypeReq_t *p
   Req, uint8_t taskId)

This function can be used to read the characteristic from the server when the client only knows the characteristic's UUID but not the characteristic's handle. Eigenvalue.

Parameter Description	
connHandle connection	handle
pReq	Pointer to the request to send

taskID	The ID of the task being notified

1. **bStatus_t GATT_ReadLongCharValue(uint16_t connHandle, attReadBlobReq_t *pRe q, uint8_t taskId)**

A server characteristic value was read, but the characteristic value was longer than could be sent in a single read response protocol.

Parameter Description	
connHandle	connection handle
pReq	Pointer to the request to send
taskID	The ID of the task being notified

1. **bStatus_t GATT_ReadMultiCharValues(uint16_t connHandle, attReadMultiReq_t * pReq, uint8_t taskId)**

Read multiple characteristic values from the server.

Parameter Description	
connHandle	connection handle
pReq	Pointer to the request to send
taskID	The ID of the task being notified

1. **bStatus_t GATT_WriteNoRsp(uint16_t connHandle, attWriteReq_t *pReq)**

When a client knows the characteristic handle, it can write to the characteristic without confirming whether the write was successful.

Parameter Description	
connHandle	connection handle
pReq	Pointer to the command to send

1. **bStatus_t GATT_SignedWriteNoRsp(uint16_t connHandle, attWriteReq_t *pReq)**

This function can be used to write characteristic values to the server when the client knows the characteristic handle and ATT confirms that it is not encrypted.

Characteristic Properties The authentication bit is enabled and both the server and client establish bindings.

Parameter Description	
connHandle	connection handle
pReq	Pointer to the command to send

1. **bStatus_t GATT_WriteCharValue(uint16_t connHandle, attWriteReq_t *pReq, uin t8_t taskId)**

When the client knows the characteristic handle, this function can write the characteristic value to the server. Only the first octet of the characteristic value can be written.

This function returns whether the write process is successful.

Parameter Description	
connHandle connection handle	
pReq	Pointer to the request to send
taskID	The ID of the task being notified

```
1. bStatus_t GATT_WriteLongCharDesc( uint16_t connHandle, attPrepareWriteReq_t
    *pReq, uint8_t taskId )
```

When the client knows the characteristic value handle but the characteristic value length is greater than the length defined in the Single Write Request Attribute Protocol, This function can be used.

Parameter Description	
connHandle connection handle	
pReq	Pointer to the request to send
taskID	The ID of the task being notified

8.4.2 Return

SUCCESS (0x00): The instruction executed as expected.

INVALIDPARAMETER (0x02): Invalid connection handle or request field.

MSG_BUFFER_NOT_AVAIL (0x04): HCI buffer is not available. Please try again later.

bleNotConnected (0x14): The device is not connected.

blePending (0x17):

– When returning to the client function, a server or GATT sub-procedure is in progress with a response pending.

– When returning to the server function, confirmation from the client is pending.

bleTimeout (0x16): The last transaction timed out. No ATT or GATT messages can be sent until reconnected.

bleMemAllocError (0x13): A memory allocation error occurred

bleLinkEncrypted (0x19): The link is encrypted. Do not send data containing authentication signatures over encrypted links. Name PDU.

8.4.3 Events

The application receives events from the protocol stack through TMOS messages (GATT_MSG_EVENT).

The following are common event names and event message formats. For details, please refer to CH58xBLE_LIB.h.

ATT_ERROR_RSP:

```
1. typedef struct
2. {
3.          uint8_t reqOpcode; //!< Request that generated this error response
4.          uint16_t handle; //!< Attribute handle that generated error response
5.          uint8_t errCode; //!< Reason why the request has generated error resp
6. onse
7. } attErrorRsp_t;
```

ATT_EXCHANGE_MTU_REQ:

```
typedef struct
{
    uint16_t clientRxMTU; //!< Client receive MTU size
} attExchangeMTUReq_t;
```

ATT_EXCHANGE_MTU_RSP:

```
typedef struct
{
    uint16_t serverRxMTU; //!< Server receive MTU size
} attExchangeMTURsp_t;
```

ATT_READ_REQ:

```
typedef struct
{
    uint16_t handle; //!< Handle of the attribute to be read (must be first
field)
} attReadReq_t;
```

ATT_READ_RSP:

```
typedef struct
{
    uint16_t len;       //!< Length of value
    uint8_t *pValue; //!< Value of the attribute with the handle given (0 t
o ATT_MTU_SIZE-1)
} attReadRsp_t;
```

ATT_WRITE_REQ:

```
typedef struct
{
    uint16_t handle; //!< Handle of the attribute to be written (must be fi
rst field)
    uint16_t len;       //!< Length of value
    uint8_t *pValue; //!< Value of the attribute to be written (0 to ATT_MT
U_SIZE-3)
    uint8_t sig;        //!< Authentication Signature status (not included (0)
    , valid (1), invalid (2)
    uint8_t cmd;        //!< Command Flag
```

```
} attWriteReq_t;
```

ATT_WRITE_RSP:

ATT_HANDLE_VALUE_NOTI:

```
typedef struct
{
    uint16_t handle; //!< Handle of the attribute that has been changed (mu
st be first field)
    uint16_t len;        //!< Length of value
    uint8_t *pValue; //!< Current value of the attribute (0 to ATT_MTU_SIZE
-3)
} attHandleValueNoti_t;
```

ATT_HANDLE_VALUE_IND:

```
typedef struct
{
    uint16_t handle; //!< Handle of the attribute that has been changed (mu
st be first field)
    uint16_t len;        //!< Length of value
    uint8_t *pValue; //!< Current value of the attribute (0 to ATT_MTU_SIZE
-3)
} attHandleValueInd_t;
```

ATT_HANDLE_VALUE_CFM:

– Empty msg field

8.4.4 GATT commands and corresponding ATT events

ATT response event GATT API call	
ATT_EXCHANGE_MTU_RSP GATT_ExchangeMTU	
ATT_FIND_INFO_RSP	GATT_DiscAllCharDescs
ATT_FIND_BY_TYPE_VALUE_RSP	GATT_DiscPrimaryServiceByUUID
ATT_READ_BY_TYPE_RSP	GATT_PrepareWriteReq GATT_ExecuteWriteReq GATT_FindIncludedServices GATT_DiscAllChars GATT_DiscCharsByUUID GATT_ReadUsingCharUUID
ATT_READ_RSP	GATT_ReadCharValue GATT_ReadCharDesc

ATT_READ_BLOB_RSP	GATT_ReadLongCharValue GATT_ReadLongCharDesc
ATT_READ_MULTI_RSP	GATT_ReadMultiCharValues
ATT_READ_BY_GRP_TYPE_RSP	GATT_DiscAllPrimaryServices
ATT_WRITE_RSP	GATT_WriteCharValue GATT_WriteCharDesc
ATT_PREPARE_WRITE_RSP	GATT_WriteLongCharValue GATT_ReliableWrites GATT_WriteLongCharDesc
ATT_EXECUTE_WRITE_RSP	GATT_WriteLongCharValue GATT_ReliableWrites GATT_WriteLongCharDesc

8.4.5 ATT_ERROR_RSP error code

ATT_ERR_INVALID_HANDLE (0x01): The given attribute handle value is not valid on this attribute server.

ATT_ERR_READ_NOT_PERMITTED (0x02): Unable to read attribute.

ATT_ERR_WRITE_NOT_PERMITTED (0x03): Unable to write attribute.

ATT_ERR_INVALID_PDU (0x04): The attribute PDU is invalid.

ATT_ERR_INSUFFICIENT_AUTHEN (0x05): This attribute requires authentication to be read or written enter.

ATT_ERR_UNSUPPORTED_REQ (0x06): The attribute server does not support the request received from the attribute client.

ATT_ERR_INVALID_OFFSET (0x07): The specified offset is beyond the end of the attribute.

ATT_ERR_INSUFFICIENT_AUTHOR (0x08): This attribute requires authorization to be read or written.

ATT_ERR_PREPARE_QUEUE_FULL (0x09): Too many queues are ready to write.

ATT_ERR_ATTR_NOT_FOUND (0x0A): The attribute was not found in the given attribute handle range.

ATT_ERR_ATTR_NOT_LONG (0x0B): Unable to read or write a blob using a read blob request or prepare write request. Write attributes.

ATT_ERR_INSUFFICIENT_KEY_SIZE (0x0C): The encryption key used to encrypt this link is of insufficient size.

ATT_ERR_INVALID_VALUE_SIZE (0x0D): The attribute value length is invalid for this operation.

ATT_ERR_UNLIKELY (0x0E): The requested attribute request encountered an error that is unlikely to occur, and Failed to complete as required.

ATT_ERR_INSUFFICIENT_ENCRYPT (0x0F): The attribute requires encryption to read or write.

ATT_ERR_UNSUPPORTED_GRP_TYPE (0x10): The attribute type is not supported as defined by the higher layer specification. Grouping attributes.

ATT_ERR_INSUFFICIENT_RESOURCES (0x11): Insufficient resources to complete the request.

8.5 GATTServApp API

8.5.1 Instructions

```
1. void GATTServApp_InitCharCfg( uint16_t connHandle, gattCharCfg_t *charCfgTbl
   )
```

Initialize the client feature configuration table.

Parameter Description	

connHandle connection handle	
charCfgTbl Client feature configuration table	
Return None	

```
1. uint16_t GATTServApp_ReadCharCfg( uint16_t connHandle, gattCharCfg_t *charCf
   gTbl )
```

Read the client's feature configuration.

Parameter Description	
connHandle connection handle	
charCfgTbl Client feature configuration table	
Returns the value of a property	

```
1. uint8_t GATTServApp_WriteCharCfg( uint16_t connHandle, gattCharCfg_t *charCf
   gTbl, uint16_t value )
```

Write characteristic configuration to the client.

Parameter Description	
connHandle connection handle	
charCfgTbl Client feature configuration table	
value	New value
return	SUCCESS FAILURE

```
1. bStatus_t GATTServApp_ProcessCCCWriteReq( uint16_t connHandle,
2.                                           gattAttribute_t *pAttr,
3.                                           uint8_t *pValue,
4.                                           uint16_t len,
5.                                           uint16_t offset,
6.                                           uint16_t validCfg );
```

Processes client characteristic configuration write requests.

Parameter Description	
connHandle connection handle	
pAttr	Pointer to the property
pvalue	Pointer to the data to be written
len	Data length
offset	The offset of the first eight bits of data to be written
validCfg valid configuration	
return	SUCCESS FAILURE

8.6 GAPBondMgr API

8.6.1 Instructions

1. **bStatus_t GAPBondMgr_SetParameter(uint16_t param, uint8_t len, void *pValue)**

Sets parameters for binding management.

parameter	describe
param	Configuration parameters
len	Write length
pValue	Pointer to the data to be written
returned	SUCCESS INVALIDPARAMETER: Invalid parameter

1. **bStatus_t GAPBondMgr_GetParameter(uint16_t param, void *pValue)**

Gets the parameters managed by the binding.

Parameter Description	
param	Configuration parameters
pValue	Points to the address of the read data
return	SUCCESS INVALIDPARAMETER: Invalid parameter

1. **bStatus_t GAPBondMgr_PasscodeRsp(uint16_t connectionHandle, uint8_t status, uint32_t passcode)**

Respond to password request.

Parameter Description	
connectionHandle connection handle	
status	SUCCESS: Password is available For more information, see SMP_PAIRING_FAILED_DEFINES integer
passcode return	value password
	SUCCESS: Binding record found and changed bleIncorrectMode: No connection found

8.6.2 Configuration Parameters

Common configuration parameters are shown in the following table. For detailed parameter IDs, please refer to CH58xBLE.LIB.h.

Parameter ID	Read and write size description
GAPBOND_PERI_PAIRI Readable uint8 pairing method, default is:	

NG_MODE	Writable		GAPBOND_PAIRING_MODE_WAIT_FOR_REQ
GAPBOND_PERI_DEFAUL LT_PASSCODE		uint32	Default man-in-the-middle protection key, range: 0-999999, default Think 0.
GAPBOND_PERI_MITM_ PROTECTION	Readable Writable	uint 8	Man-in-the-middle (MITM) protection. The default value is 0, which turns off man-in-the-middle Protect.
GAPBOND_PERI_IO_CA PABILITIES	Readable Writable	uint 8	I/O capabilities, the default is: GAPBOND_IO_CAP_DISPLAY_ONLY, that is, the device can only Reality.
GAPBOND_PERI_BONDI NG_ENABLED	Readable Writable	uint 8	If enabled, request to bind during pairing. Default is 0. No binding is requested.

8.7 RF PHY API

8.7.1 Instructions

1. bStatus_t RF_RoleInit(void)

RF protocol stack initialization.

Parameters	describe
returned	SUCCESS: Initialization successful

1. bStatus_t RF_Config(rfConfig_t *pConfig)

RF parameter configuration.

	describe
Parameter	Pointer to configuration parameters
pConfig Return	SUCCESS

1. bStatus_t RF_Rx(uint8_t *txBuf, uint8_t txLen, uint8_t pktRxType, uint8_t pktTxType)

RF receive data function: configure RF PHY to receive state, and reconfigure after receiving data.

parameter	describe
txBuf	In automatic mode, a pointer to the data returned after the RF receives the data
txLen	In automatic mode, the length of the data returned by RF after receiving the data (0-251)
pkRxType	Accepted packet type (0xFF: accept all types of packets)
pkTxType	In automatic mode, the data packet type returned by RF after receiving data
Return	SUCCESS

1. bStatus_t RF_Tx(uint8_t *txBuf, uint8_t txLen, uint8_t pktTxType, uint8_t p ktRxType)

RF send data function.

parameter	describe
txBuf	Pointer to RF transmit data
txLen	RF transmit data length (0-251)
pkTxType	The type of data packet sent
pkRxType	In automatic mode, the data type of the received data after the RF sends the data (0xFF: accept the received data) Typed packets)
return	SUCCESS

1. bStatus_t RF_Shut(void)

Turn off RF and stop transmitting or receiving.

Parameter Description	
return	SUCCESS

1. uint8_t RF_FrequencyHoppingTx(uint8_t resendCount)

RF transmitter turns on frequency hopping

	describe
Parameter resendCount	Maximum count of sending HOP_TX pdu (0: unlimited)
return	0: SUCCESS

1. uint8_t RF_FrequencyHoppingRx(uint32_t timeoutMS);

RF receiver turns on frequency hopping

parameter	describe
timeoutMS	Maximum time to wait for receiving HOP_TX pdu (Time = n * 1ms, 0: unlimited)
return	0: SUCCESS 1: Failed 2: LLEMode error (needs to be in automatic mode)

1. void RF_FrequencyHoppingShut(void)

Disable RF frequency hopping function

8.7.2 Configuration Parameters

The RF configuration parameter rfConfig_t is described as follows:

parameter	describe
LLEMode	LLE_MODE_BASIC: Basic mode, enters idle mode after sending or receiving. LLE_MODE_AUTO: Auto mode, automatically switches to receiving mode after sending is completed

	LLE_MODE_EX_CHANNEL: Switch to Frequency configuration band LLE_MODE_NON_RSSI: Set the first byte of received data to the packet type
Channel	RF communication channel (0-39)
Frequency	RF communication frequency (2400000KHz-2483500KHz), it is recommended not to use more than 24 times Bit flipped and no more than 6 consecutive 0s or 1s
AccessAddress	RF communication address
CRCInit	CRC initial value
RFStatusCB	RF status callback function
ChannelMap Channel map, each bit corresponds to a channel. A bit value of 1 means the channel is valid, otherwise it is not.	The channels are incremented bit by bit and channel 0 corresponds to bit 0.
Resv	reserve
HeartPeriod heartbeat packet interval, an integer multiple of 100ms	
HopPeriod The	Jump period (T=32n*RTC clock), default is 8
HopIndex frequency hopping channel interval value in the data channel selection algorithm, the default value is 17	
RxMaxlen	The maximum data length received in RF mode is 251 by default.
RxMaxlen	The maximum data length transmitted in RF mode is 251 by default.

8.7.3 Callback Functions

1. void RF_2G4StatusCallBack(uint8_t sta , uint8_t crc, uint8_t *rxBuf)

RF status callback function, this callback function will be entered when sending or receiving is completed.

Note: You cannot directly call the RF receiving or sending API in this function. You need to use the event method to call it.

Parameter Description	
sta	RF transmit and receive status
crc	Data packet status check, each bit represents a different status: bit0: data CRC check error; bit1: Packet type error;
rxBuf	Pointer to the received data
return	NULL

Revision History

Version	time	Revisions
V1.0	2021/3/22	Version Release
V1.1	2021/4/19	Added RF usage examples and
V1.2	2021/5/28	API content errata, added RF description
V1.3	2021/7/3	1. Name adjustment; 2. Modify the preface and development platform related describe; 3. Errata of Figure 3.1 .
V1.4	2021/11/2	1. Added section 7.5.2.1 to describe RF Frequency hopping function; 2. Added RF frequency hopping API description; 3. Optimize code display format.
V1.5	2022/5/6	1. Errata: Section 7.2 code reference is wrong error; 2. Errata: Section 5.3.2 is numbered incorrectly; Section 3.3.3 details the TMOS task execution structure.
V1.6	2022/8/19	1. Adjust the title of Chapter 4; 2. Added description of application routines; 3. Added section 5.5.2 to describe GATT services. Services and agreements; 4. Added TMOS API description; 5. Optimize content description; 6. Optimize Figure 4.2; 7. Sample code and routine synchronization; 8. Content Corrections.
V1.7	2022/9/30	1. Errata: Section 8.7.3 crc description is wrong 2. Header
V1.8	2024/1/3	adjustment 1. Errata: Figure 1.1 Errata 2. Errata: Text description

Version Statement and Disclaimer

The reference manual may contain a few inadvertent errors. Those found will be corrected regularly and updated in the next edition. and avoid such mistakes.