Qinheng Low Power Bluetooth Software Development Reference Manual

V1.8

January 3,2024

catalogs

preamble

This manual provides a brief introduction to the software development of Qinheng low-power Bluetooth. Including the software development platform, the basic framework of software development and low power bluetooth protocol stack. In order to facilitate the understanding, this manual is introduced with CH58x chip as an example, the software development of other low power bluetooth chips of our company can also refer to this manual.

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

1. summarize

1.1 present (sb for a job etc)

Bluetooth supports two wireless technologies since version 4.0:

- Bluetooth Basic Rate/Enhanced Data Rate (often referred to as BR/EDR Classic Bluetooth)

- Low Power Bluetooth

The Low Power Bluetooth protocol was created to transmit very small packets at a time, resulting in a significant reduction in power consumption compared to classic Bluetooth.

Devices that can support both classic Bluetooth and low-power Bluetooth are called dual-mode devices, such as cell phones. Devices that support only low-power Bluetooth are called single-mode devices. These devices are mainly used for low-power applications, such as those powered by coin cell batteries.

1.2 Low Power Bluetooth Protocol Stack Basic Introduction

The results of the low-power Bluetooth protocol stack are shown in Figure 1-1.

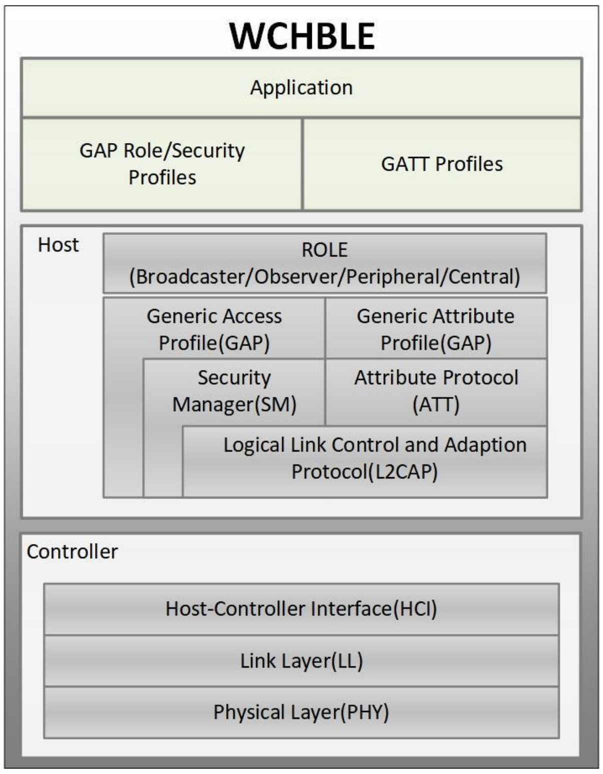

WCHBLE
Application
GAP Role/Security Profiles
GATT Profiles
Host
ROLE (Broadcaster/Observer/Peripheral/Central)
Generic Access Profile(GAP)
Generic Attribute Profile(GAP)
Security Manager(SM)
Attribute Protocol (ATT)
Logical Link Control and Adaption Protocol(L2CAP)
Controller
Host-Controller Interface(HCI)
Link Layer(LL)
Physical Layer(PHY)

Figure 1-1

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

Configurations and applications are implemented in the Generic Access Profile (GAP) and Generic Attribute Profile (GATT) layers of the protocol stack.

The Physical Layer (PHY) is the lowest layer of the BLE protocol stack, which specifies the basic RF parameters for BLE communication, including signal frequency, modulation scheme, and so on.

The physical layer is modulated using Gaussian Frequency Shift Keying (GFSK - Gauss frequency Shift Keying) technique in the 2.4GHz channel.

The physical layer of BLE 5.0 has three implementations, namely 1Mbps non-coded physical layer, 2Mbps non-coded physical layer and 1Mbps coded physical layer. The 1Mbps non-coded physical layer is compatible with the physical layer of the BLE 4.0 family of protocols, while the other two physical layers extend the communication rate and distance, respectively.

The LinkLayer controls the device in one of five states: ready, advertising, scanning, initiating, connected. Around these states, the BLE device can perform operations such as broadcasting and connecting, and the link layer defines the packet format, timing specification, and interface protocols in each state.

The Generic Access Profile (GAP) is the external interface layer for the internal functions of the BLE device. It specifies three aspects: GAP roles, modes and protocols, and security. It mainly manages the broadcast, connection and device binding of Bluetooth devices.

Broadcasters - devices that broadcast all the time that are not connectable

Observer - a device that can scan for broadcast devices, but cannot initiate the establishment of a connection

Slave - a broadcast device that can be attached as a slave in a single link layer connection

Host - can scan for broadcast devices and initiate connections, acting as a host in a single link layer or multiple link layers

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

Security Manager provides pairing and key distribution services to enable secure connections and data exchange.

Attribute Protocol (Attribute Protocol) defines the concept of attribute entities, including UUIDs, handles, and attribute values, and specifies the methods and details of operations such as reading, writing, and notification of attributes.

The Generic Attribute Profile (GAP) defines the structure of the service framework and protocols that use ATT, and the communication of application data from the two devices is realized through the GATT layer of the protocol stack.

GATT server - a device that provides data services to a GATT client
GATT client - a device that reads and writes application data from a
GATT server

2. development platform (computing)

2.1 summarize

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

2.2 configure

The CH58x is a true single chip solution, with the controller, host, profile and application all implemented on the CH58x. Refer to the Central and Peripheral routines.

2.3 Software Overview

The software development kit includes the following six main components:

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

The package provides four GAP profiles:

- Peripheralrole
- Centralrole
- Broadcasterrole
- Observerrole

A number of GATT configuration files as well as applications are also provided. Please refer to the CH58xEVT software package for details.

3. Task Management System (TMOS)

3.1 summarize

The low-power Bluetooth protocol stack and applications are based on TMOS (Task Management Operating System) which is a control loop through which events can be set to execute in a certain way.TMOS serves as the scheduling core around which the BLE protocol stack, profile definitions, and all applications are realized.TMOS is not an operating system in the traditional sense, but a system resource management mechanism that focuses on realizing multi-tasking. TMOS is not an operating system in the traditional sense, but a system resource management mechanism with multitasking as its core.

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

3.2 Task initialization

First of all, in order to ensure that TMOS continues to run, it is necessary to loop through TMOS_SystemProcess() at the end of main(). The initialization of the task needs to call tmosTaskID TMOS_ProcessEventRegister(pTaskEventHandlerFn eventCb) function to register the event□ call function into TMOS and generate a unique 8-bit task ID. Different tasks are initialized after the task ID is incremented, and the smaller the ID is, the higher the priority of the task is. The smaller the task ID, the higher the task priority. The stack task must have the highest priority.

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

3.3 Task events and execution of events

TMOS is scheduled by polling, and the system clock is usually derived from the RTC in 625 µs. User-defined events are added to the task chain list of TMOS by registering a task, and then TMOS schedules the task to run. After the task is initialized,TMOS polls the task events in a loop, and the event flags are stored in 16-bit variables, where each bit corresponds to a unique event in the same task. Each bit corresponds to a unique event within the same Task. A flag of 1 means that the event corresponding to that bit is running, while a flag of 0 means that it is not running.Each Task can have up to 15 user-defined events0x8000 is reserved for SYS_EVENT_MSG event, i.e., system messaging event, which 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 any event that needs to be executed, and the system tasks, such as end-of-transmission auto-receive answer in 2.4G auto mode and Bluetooth related transactions, have the highest priorityWhen the system task is finished,if there is a user event, the corresponding□ call function will be executedAt the end of a cycle, if there is still time available, the system enters idle or sleep mode.

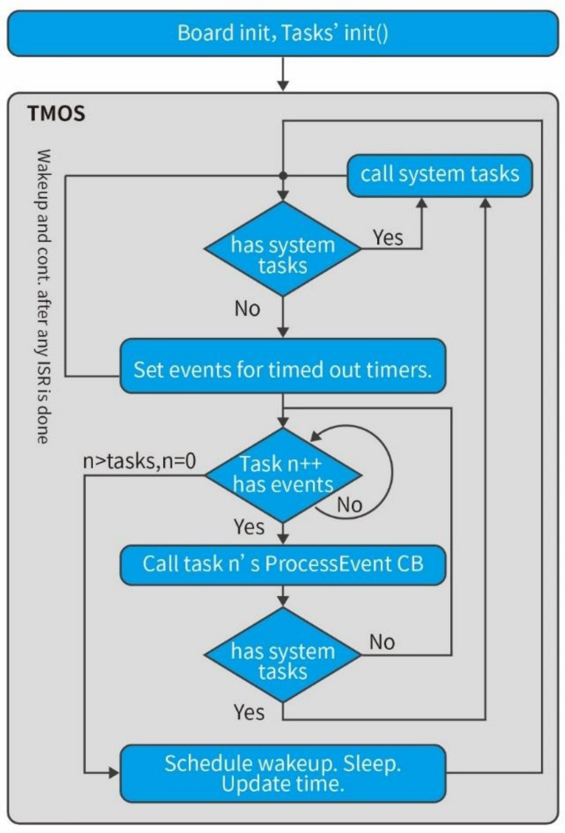

Board init, Tasks' init()
TMOS
Wakeup and cont. after any ISR is done
call system tasks
has system tasks
Yes
No
Set events for timed out timers.
n>tasks,n=0
Task n++ has events
No
Yes
Call task n' s ProcessEvent CB
has system tasks
No
Yes
Schedule wakeup. Sleep. Update time.

Figure 3.1 TMOS Task Management Diagram

In order to illustrate the common code format for TMOS to handle events, we take the HAL_TEST_EVENT event in the HAL layer as an example, where HAL_TEST_EVENT can be replaced by other events as well. If you want to define a TEST event in the HAL layer, you can add the event HAL_TEST_EVEN to the task回 call function after the task initialization in the HAL layer, and its basic format is as follows:

```
if ( events & HAL_TEST_EVENT )
{
PRINT("* \n").
return events^ HAL_TEST_EVENT.
}
```

The 16-bit event variable corresponding to回 needs to be returned to clear the event after the execution of the event is completed to prevent the same event from being processed repeatedly. The above code clears the HAL_TEST_EVENT flag by returning events^ HAL_TEST_EVENT;.

After the event is added, call tmos_set_event(halTaskID, HAL_TEST_EVENT) function to execute the corresponding event immediately, and the event will be executed only once. Where halTaskID is the task selected for execution and HAL_TEST_EVENT is the corresponding event under the task.

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

If you don't want to execute an event immediately, you can call tmos_start_task(tmosTaskID taskID, tmosEvents event, tmosTimer time), which is similar to tmos_set_event, but the difference lies in that after setting the task ID and the event flag of the task that you want to execute, you need to add a third parameter. need to add a third parameter: the timeout time of the event execution. That is, the event will

It is executed once after the timeout is reached. Then define the time of the next task execution in the event to loop through the event at regular intervals.

```c
if ( events & HAL_TEST_EVENT )
{
    PRINT( "* \n" ).
    tmos_start_task( halTaskID, HAL_TEST_EVENT, MS1_TO_SYSTEM_TIME( 1000 )).
    return events^ HAL_TEST_EVENT;
}
```

At this time, there is only one timed event HAL_TEST_EVENT in TMOS, and the system will enter the idle mode after executing this time, or will enter the sleep mode if the sleep function is turned on. The actual effect is shown in Figure 3.2.

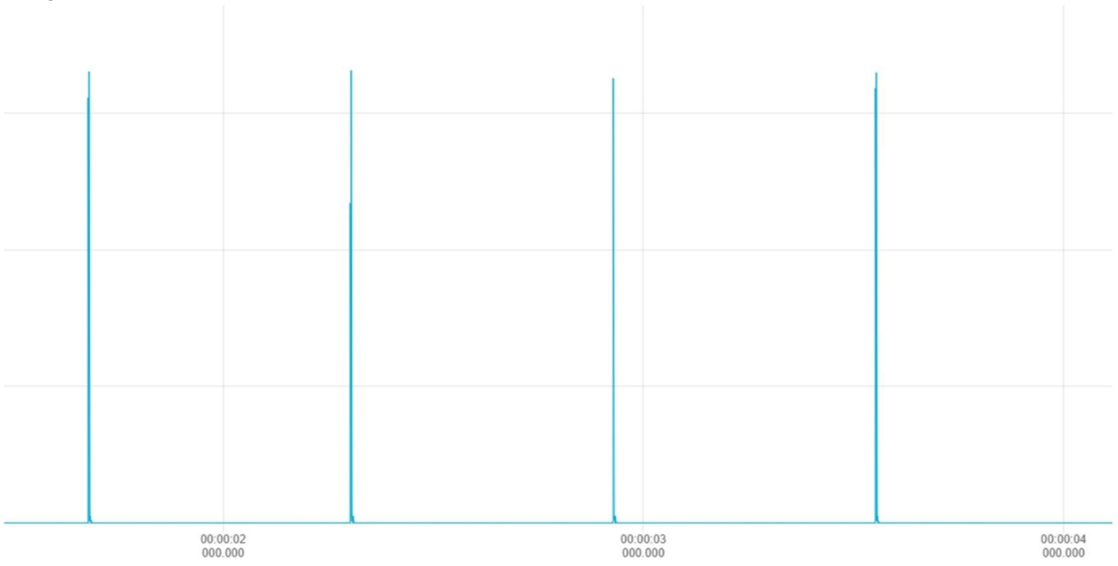


Figure 3.2 Timed Tasks

3.4 memory management

TMOS uses a separate piece of memory, which can be customized by the user in terms of address and size, and which is used for task event management in TMOS, and the memory usage can be analyzed by enabling Bluetooth bonding and encryption functions.

Since the protocol stack for low-power Bluetooth also uses this memory, it needs to be tested under 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 can be of arbitrary length with sufficient memory.

One data can be sent by following the steps below:

1. Use the tmos_msg_allocate() function to apply for memory for the sent data, and return memory address if the application succeeds, or NULL if it fails.
2. Copies the data into memory.
3. A pointer to call the tmos_msg_send() function to send data to the specified task.

```c
// Register Key task ID
```

```
HAL_KEY_RegisterForKeys( centralTaskId )
// Send the address to the task
        msgPtr= ( keyChange_t * ) tmos_msg_allocate( sizeof(keyChange_t) );
        if ( msgPtr )
        {
          msgPtr->hdr.event= KEY_CHANGE;
          msgPtr->state= state;
          msgPtr->keys= keys;
          tmos_msg_send( registeredKeysTaskID, ( uint8_t * ) msgPtr ).
}
```

After the data is successfully sent, SYS_EVENT_MSG is set to valid, at which time the system will execute the SYS_EVENT_MSG event and retrieve the data in practice by calling the tmos_msg_receive() function. The tmos_msg_deallocate() function must be used to free memory after data processing is complete. Please refer to the routines for details.

Assuming the message is sent to the central's task ID, the central's system events will receive the message.

```
uint16_t Central_ProcessEvent( uint8_t task_id, uint16_t events ){
if ( events & SYS_EVENT_MSG ) {
      uint8_t *pMsg.
      if ( (pMsg= tmos_msg_receive( centralTaskId )) ! = NULL ){
          central_ProcessTMOSMsg( (tmos_event_hdr_t *)pMsg ).
          // Release the TMOS message
            tmos_msg_deallocate( pMsg ).
}
```

Queries the KEY_CHANGE event:

```
static void central_ProcessTMOSMsg( tmos_event_hdr_t *pMsg )
{
            switch ( pMsg->event ){
              case KEY_CHANGE:{
          ...
```

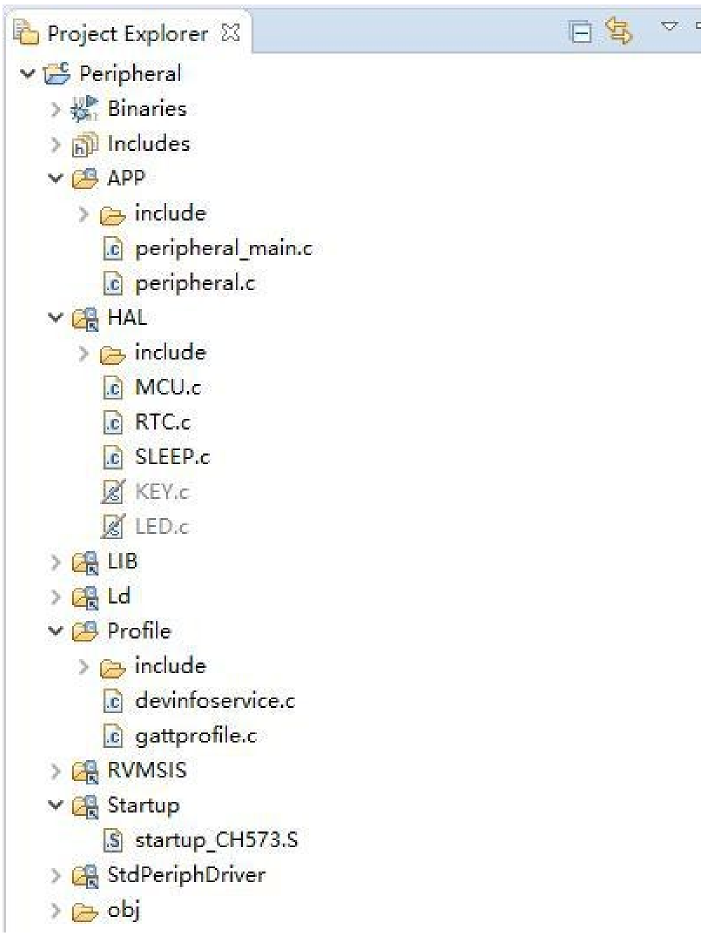
4. Application Example Analysis

4.1 summarize

The Low Power Bluetooth EVT routines include a simple BLE project: Peripheral, which can be burned into the CH58x chip to 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.


Project Explorer

```
Peripheral
  Binaries
  Includes
  APP
    include
    peripheral_main.c
    peripheral.c
  HAL
    include
    MCU.c
    RTC.c
    SLEEP.c
    KEY.c
    LED.c
  LIB
  Ld
  Profile
    include
    devinfoservice.c
    gattprofile.c
  RVMSIS
  Startup
    startup_CH573.S
  StdPeriphDriver
  obj
```

Figure 4.1 Project Documentation

The documents can be divided into the following categories:

1. APP – source and header files for the application can be placed here, as well as the main function of the routine.
2. HAL –This folder contains the source code and header files of HAL layer. This folder contains the source code and header files for the HAL layer, which is the layer where the Bluetooth protocol stack interacts with the chip hardware driver.
3. LIB - Protocol stack library file for Low Power Bluetooth.
4. LD – Link Script.
5. Profile – This file from contains the source code and header files for the GAP Role Profile, GAP Security Profile, and GATT as well as the header files required by the GATT service. Refer to Section 5 for details.
6. RVMSIS - Source code and headers for RISC-V kernel access.
7. Startup - Startup file.

8. StdPeriphDriver - includes the underlying driver files for the chip peripherals.
9. obj - Files generated by the compiler, including map files and hex files.

4.3 Starts at main()

Main() function is the starting point of the program, this function first initializes the system clock; then configure the IO port state, to prevent the float state leads to unstable operating current; then initialize the serial port for printing debugging, and finally initialize the TMOS and the low-power Bluetooth. main() function of the Peripheral project is shown below:

```
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 the 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.     //Initialize the serial port
#endif
PRINT("%s\n",VER_LIB).
CH58X_BLEInit( ); //Initialize the Bluetooth library.        //Initialize the Bluetooth library
HAL_Init( ).
   GAPRole_PeripheralInit( ).
Peripheral_Init( ).
while(1){
      TMOS_SystemProcess( ).             //Main loop
}
}
```

4.4 Application initialization

4.4.1 Low Power Bluetooth Library Initialization

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

4.4.2 HAL Layer Initialization

Registers HAL layer tasks to initialize hardware parameters such as RTC clock, sleep-wake, RF calibration, etc.

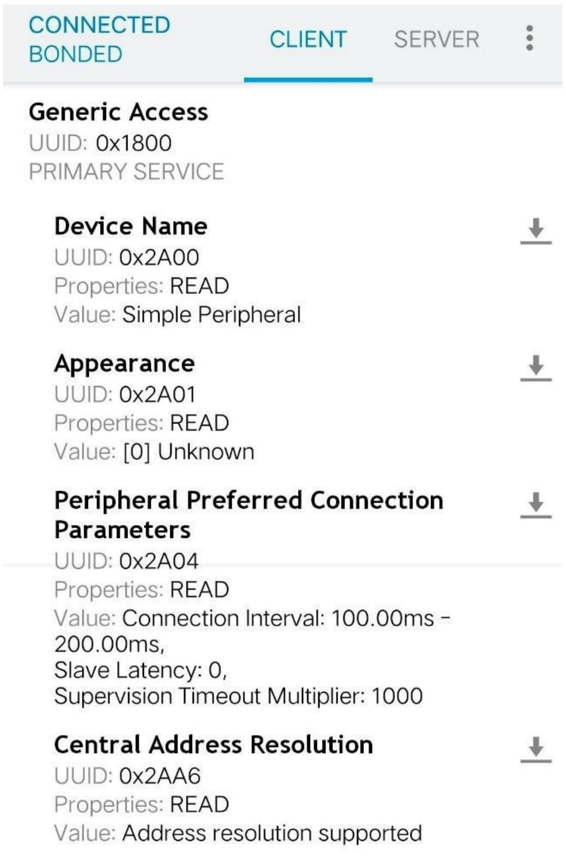
```
void HAL_Init()
{
     halTaskID= TMOS_ProcessEventRegister( HAL_ProcessEvent ).
     HAL_TimeInit().
#if (defined HAL_SLEEP) && (HAL_SLEEP== TRUE)
      HAL_SleepInit().
#endif
#if (defined HAL_LED) && (HAL_LED== TRUE)
      HAL_LedInit( ).
#endif
#if (defined HAL_KEY) && (HAL_KEY== TRUE)
HAL_KeyInit( ).
#endif
#if ( defined BLE_CALIBRATION_ENABLE ) && ( BLE_CALIBRATION_ENABLE== TRUE )

   tmos_start_task( halTaskID, HAL_REG_INIT_EVENT, MS1_TO_SYSTEM_TIME( BLE_CA
LIBRATION_PERIOD ) ); // Add calibration task, single calibration takes less than 10ms.        // Add
calibration task, single calibration takes less than 10ms.
#endif
tmos_start_task( halTaskID, HAL_TEST_EVENT, 1000 );                      // Add a test task
}
```

4.4.3 Low Power Bluetooth Slave Initialization

This process consists of two parts:

1. Initialization of GAP roles, this process is done by the Low Power Bluetooth library;
2. Initialization of the low-power Bluetooth slave application, including registration of slave tasks, parameter configuration (e.g., broadcast parameters, connection parameters, binding parameters, etc.) registration of GATT layer services, and registration of call functions. See Section 5.5.3.2 for details.

Figure 4.2 shows the complete property sheet of the routine Peripheral, which can be used as a reference when communicating with low-power Bluetooth. Please refer to Chapter 5 for detailed information.



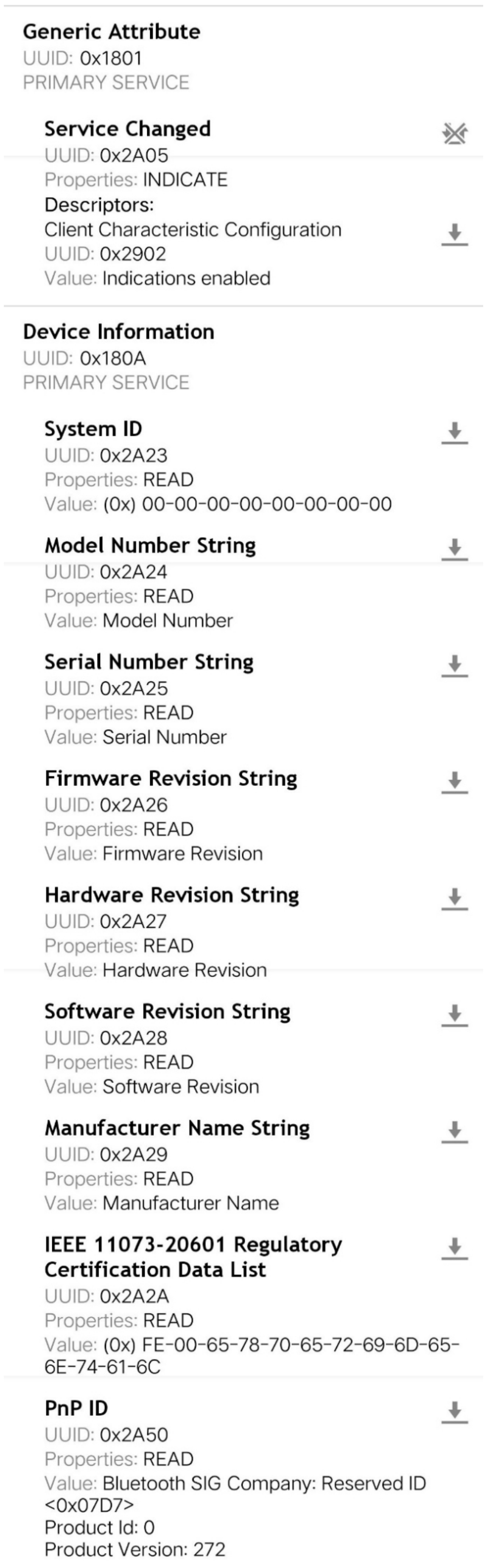
Generic Access
UUID: 0x1800
PRIMARY SERVICE

Device Name
UUID: 0x2A00
Properties: READ
Value: Simple Peripheral

Appearance
UUID: 0x2A01
Properties: READ
Value: [0] Unknown

Peripheral Preferred Connection Parameters
UUID: 0x2A04
Properties: READ
Value: Connection Interval: 100.00ms - 200.00ms,
Slave Latency: 0,
Supervision Timeout Multiplier: 1000

Central Address Resolution
UUID: 0x2AA6
Properties: READ
Value: Address resolution supported



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

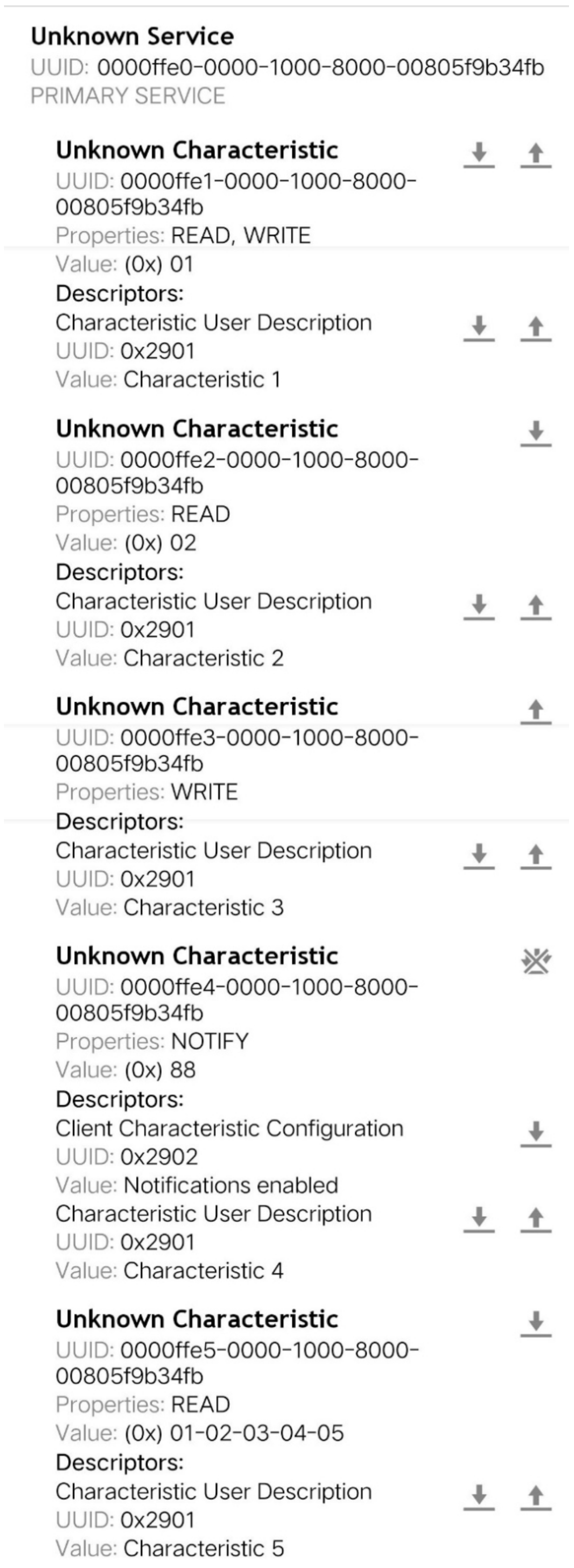
Hardware Revision String
UUID: 0x2A27
Properties: READ
Value: Hardware Revision

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



Unknown Service
UUID: 0000ffe0-0000-1000-8000-00805f9b34fb
PRIMARY SERVICE

Unknown Characteristic
UUID: 0000ffe1-0000-1000-8000-00805f9b34fb
Properties: READ, WRITE
Value: (0x) 01
Descriptors:
Characteristic User Description
UUID: 0x2901
Value: Characteristic 1

Unknown Characteristic
UUID: 0000ffe2-0000-1000-8000-00805f9b34fb
Properties: READ
Value: (0x) 02
Descriptors:
Characteristic User Description
UUID: 0x2901
Value: Characteristic 2

Unknown Characteristic
UUID: 0000ffe3-0000-1000-8000-00805f9b34fb
Properties: WRITE
Descriptors:
Characteristic User Description
UUID: 0x2901
Value: Characteristic 3

Unknown Characteristic
UUID: 0000ffe4-0000-1000-8000-00805f9b34fb
Properties: NOTIFY
Value: (0x) 88
Descriptors:
Client Characteristic Configuration
UUID: 0x2902
Value: Notifications enabled
Characteristic User Description
UUID: 0x2901
Value: Characteristic 4

Unknown Characteristic
UUID: 0000ffe5-0000-1000-8000-00805f9b34fb
Properties: READ
Value: (0x) 01-02-03-04-05
Descriptors:
Characteristic User Description
UUID: 0x2901
Value: Characteristic 5

Figure 4.2 Attribute Table

4.5 event processing

After initialization is complete and the event is opened (i.e., a bit is placed in the event), the application task will process the event in Peripheral_ProcessEvent, and the following subsections describe the possible sources of the event.

4.5.1 timed event

In the program segment shown below (which is located in the routine peripheral.c) the application contains a TMOS event named SBP_PERIODIC_EVT. The TMOS timer causes the SBP_PERIODIC_EVT event to occur periodically. The timer timeout value is set to after SBP_PERIODIC_EVT is processed (default 5000ms) The periodic event occurs every 5 seconds and the function performPeriodicTask() is called.

```
if(events & SBP_PERIODIC_EVT)
{
    // Restart timer
     if(SBP_PERIODIC_EVT_PERIOD)
     {
     tmos_start_task(Peripheral_TaskID, SBP_PERIODIC_EVT.
 SBP_PERIODIC_EVT_PERIOD).
     }
 // Perform periodic application task
   performPeriodicTask();
   return (events^ SBP_PERIODIC_EVT).
}
```

This periodic event processing is just an example, but highlights how custom actions can be performed in periodic tasks. A new TMOS timer will be started before processing the periodic event to be used to set the next periodic task.

4.5.2 TMOS Messaging

TMOS messages may originate from various layers of the BLE stack, see 3.5 TMOS Data Transfer for more information on this section.

4.6 回 harmonize

The application code can be written either in event handling snippets or in回 call functions such as simpleProfileChangeCB() and peripheralStateNotificationCB(). The communication between the stack and the application is realized by the回 call functions, e.g. simpleProfileChangeCB() can notify the application about the change of the feature value.

5. Low Power Bluetooth Protocol Stack

5.1 summarize

The code for the Low Power Bluetooth stack is in the library files and the original code will not be provided. However users should be aware of the functionality of these layers as they interact directly with the application.

5.2 General Access Profile (GAP)

5.2.1 summarize

The GAP layer of the low-power Bluetooth stack defines the following states of the device, as shown in

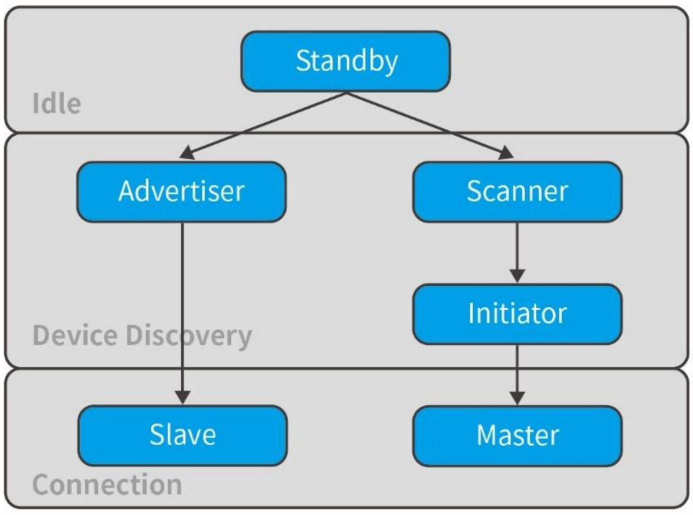


Figure 5.1

Figure 5.1 GAP Status

Among them:

Standby: The idle state in which the low-power Bluetooth protocol stack is not enabled;

Advertiser: The device broadcasts using specific data, the broadcast can contain data such as the name and address of the device. The broadcast can indicate that this device can be connected.

Scanner: When receiving broadcast data, the scanning device sends a scanning request packet to the broadcaster, which will return to回 to scan the corresponding packet. The scanner reads the information from the broadcaster and determines if it is connectable. This procedure describes the process of discovering a device.

Initiator: When establishing a connection, the connection initiator must specify the address of the device to be used for the connection, and if the address matches, 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 the connection, it is a slave at the time of the connection; if the device is an initiator before the connection, it is a host after the connection.

5.2.1.1 connection parameter

This section describes the connection parameters at the time of connection establishment, which can be modified by both the master and the slave.

- ConnectionInterval - Low Power Bluetooth uses a frequency hopping scheme where devices send and receive data on a characterized channel at specific times. A single data transmission and reception between two devices becomes a connection event. The ConnectionInterval is the time between two connection events, and its time unit is 1.25ms. The range of ConnectionInterval is 7.5ms~4s.

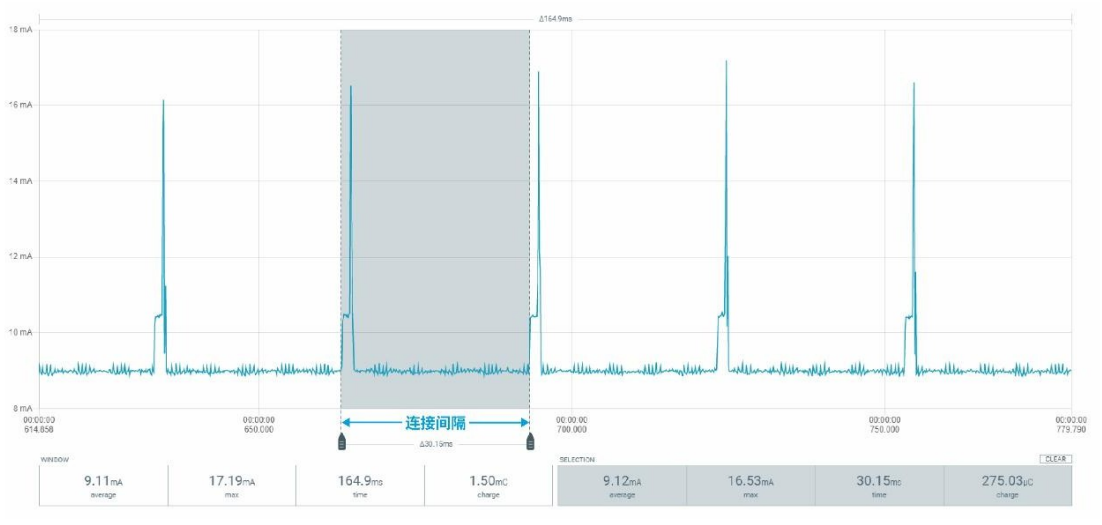


Figure 5.2 Connection Events and Connection Intervals

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

-SlaveLatency – This parameter allows the slave to skip some of the connection events. If the device has no data to send, the slave latency can skip connection events and stop RF during connection events, thus reducing power consumption. The value of the slave latency indicates the maximum number of events that can be skipped, ranging from 0 to 499, provided that the active connection interval is less than 16 s. For the active connection interval, refer to 5.2.1.2.

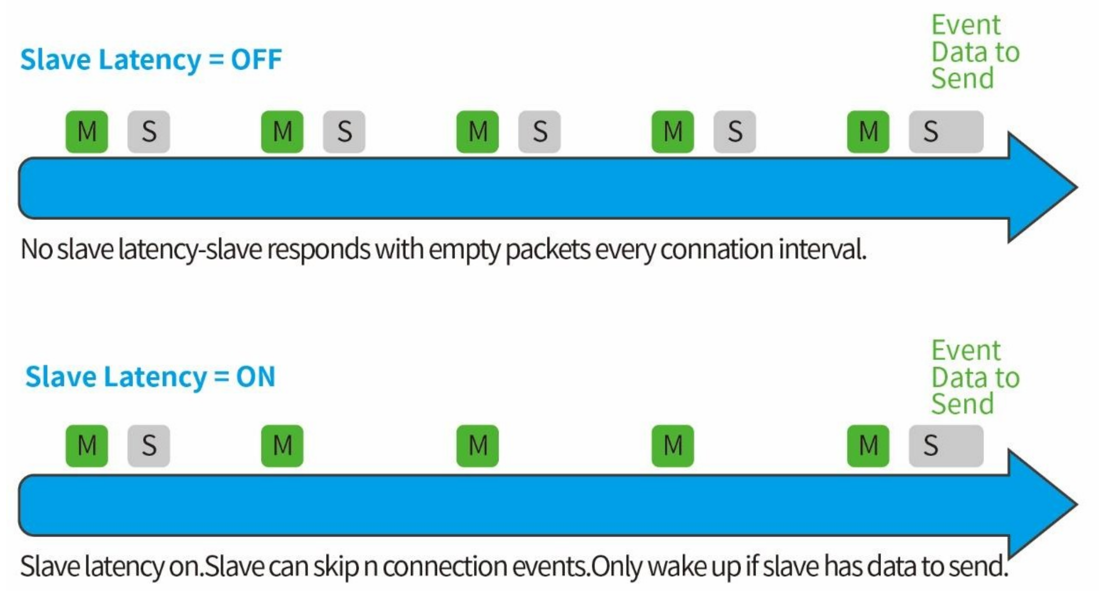

Slave Latency = OFF
Event Data to Send
M S M S M S M S M S
No slave latency-slave responds with empty packets every connation interval.
Slave Latency = ON
Event Data to Send
M S M M M M S
Slave latency on.Slave can skip n connection events.Only wake up if slave has data to send.

Figure 5.3 Slave Delay

- SupervisionTime-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 is returned to the回 unconnected state. The supervised timeout can range from 10 (100ms) to 3200 (32s) The timeout must be greater than the valid connection interval.

5.2.1.2 Active Connection Interval

With slave delay enabled and no data transfer, the effective connection interval is the time between two connection events. If slave delay is not enabled or has a value of 0, the effective connection interval is the configured connection interval.

The formula for its calculation is as follows:

$$\text{Effective Connection Interval} = (\text{Connection Interval}) \times (1 + \text{slave device delay})$$

(coll.) fail (a student)
Connection Interval: 80 (100ms)
Slave Delay: 4
Valid connection interval: $(100\text{ms})\times(1+4)=500\text{ms}$

Then, in the absence of data transfer, the slave will initiate a connection event every 500ms.

5.2.1.3 Notes on Connection Parameters

Proper connection parameters help optimize the power consumption as well as the performance of low-power Bluetooth, and the following summarizes the tradeoffs in connection parameter settings:

Reducing the connection interval will:

- Increased host-slave power consumption
- Increased throughput between two devices
- Reducing the tiit takes for data to travel to and from the two devices increasing the connection interval will:
- Reduced host-slave power consumption
- Reduced throughput between two devices
- Increase the time it takes for data to travel to and from both devices Decrease the slave device delay or set it to 0 will:
- Increase in power consumption of slaves
- Reduce the tiit takes for the master to send data to the slave Increase the slave device latency will:
- Reduces power consumption of slaves when there is no data to be sent to the master
- Increase the time it takes for the master to send data to the slave

5.2.1.4 Connection parameter update

In some applications, the slave may need to change connection parameters during the connection based on the application program. The slave can send a connection parameter update request to the host to update the connection parameters. For Bluetooth 4.0,the L2CAP layer of the protocol stack will handle this request.

The request contains the following parameters:

- Minimum connection interval
- Maximum connection interval
- Delay from equipment
- Supervisory time-outs

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

5.2.1.5 Terminate connection

The master or slave can terminate the connection for any reason. When either device enables termination of the connection, the other device must respond to terminate the connection before the two devices disconnect.

5.2.2 GAP Abstraction Layer

Applications can implement responsive BLE functionality, such as broadcasts and connections, by calling API functions in the GAP layer.

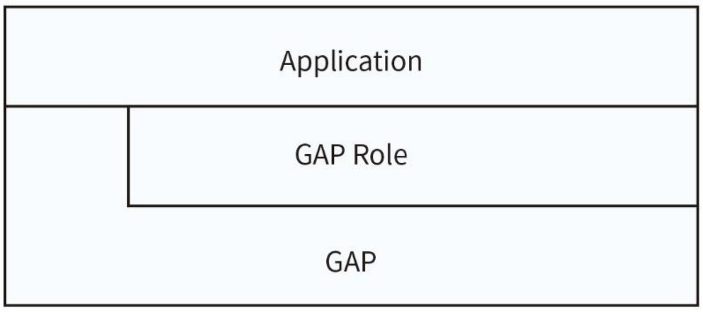


Figure 5.4 GAP Abstraction Layer

5.2.3 GAP Layer Configuration

Most of the functions of the GAP layer are implemented in libraries, and the user can find the corresponding function sounds in CH58xBLE_LIB.h.

Bright.

Section 8.1 defines the GAP API, can be used to set and detect parameters such as broadcast intervals, scanning intervals, etc. via GAPRole_SetParameter() and GAPRole_GetParamenter().An example of GAP layer configuration is shown below:

```
// Setup the GAP Peripheral Role Profile
{
        uint8_t initial_advertising_enable= TRUE;
        uint16_t desired_min_interval= DEFAULT_DESIRED_MIN_CONN_INTERVAL;
        uint16_t desired_max_interval= DEFAULT_DESIRED_MAX_CONN_INTERVAL;
        GAPRole_SetParameter( GAPROLE_MIN_CONN_INTERVAL, sizeof( uint16_t ), &de
    sired_min_interval );
        GAPRole_SetParameter( GAPROLE_MAX_CONN_INTERVAL, sizeof( uint16_t ), &de
    sired_max_interval ).

}
```

5.3 GAPRole Tasks

As described in Section 4.4, GAPRole is a separate task (GAPRole_PeripheralInit), and most of the GAPRole code runs in the Bluetooth library to simplify the application layer program. This task is started and configured by the application during initialization. If the回 call exists, the application can register the回 call function with the GAPRole task.

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

- Broadcaster - only broadcasts can't be connected

- Observer - only scans broadcasts and cannot establish a connection

- Peripheral - broadcastable and can be used as a slave to establish a connection at the link layer

- Central - can scan for broadcasts and also act as a host to establish single or multiple at the link layer The following describes the roles of the Peripheral and Central.

5.3.1 Peripheral Role

The routine steps for initializing a peripheral device are as follows:

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

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

```
{
     uint8_t initial_advertising_enable= TRUE;
     uint16_t desired_min_interval= DEFAULT_DESIRED_MIN_CONN_INTERVAL;.
     uint16_t desired_max_interval= DEFAULT_DESIRED_MAX_CONN_INTERVAL;.

      // Set the GAP Role Parameters.
     GAPRole_SetParameter( GAPROLE_ADVERT_ENABLED, sizeof( uint8_t ), &initial_
advertising_enable ).
     GAPRole_SetParameter( GAPROLE_SCAN_RSP_DATA, sizeof ( scanRspData ), scanR
spData ).
   GAPRole_SetParameter( GAPROLE_ADVERT_DATA, sizeof(advertData ), advertDat
a ).
   GAPRole_SetParameter( GAPROLE_MIN_CONN_INTERVAL, sizeof( uint16_t ), &desi
red_min_interval );
   GAPRole_SetParameter( GAPROLE_MAX_CONN_INTERVAL, sizeof( uint16_t ), &desi
red_max_interval );
}

// Set the GAP Characteristics
   GGS_SetParameter( GGS_DEVICE_NAME_ATT, GAP_DEVICE_NAME_LEN, attDeviceName
);

// Set advertising interval
{
       uint16_t advInt= DEFAULT_ADVERTISING_INTERVAL.

       GAP_SetParamValue( TGAP_DISC_ADV_INT_MIN, advInt ).
       GAP_SetParamValue( TGAP_DISC_ADV_INT_MAX, advInt ).
}
```

2. Initializes the GAPRole task, including passing the function pointer to the application回 call function.

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

3. Sends a GAPRole from the application layer to command the application layer to perform a connection parameter update.

```
// Send connect param update request
    GAPRole_PeripheralConnParamUpdateReq(peripheralConnList.connHandle,
                                          DEFAULT_DESIRED_MIN_CONN_INTERVAL, the
                                          DEFAULT_DESIRED_MAX_CONN_INTERVAL,
                                          DEFAULT_DESIRED_SLAVE_LATENCY,
                                          EFAULT_DESIRED_CONN_TIMEOUT, the
                                          Peripheral_TaskID).
```

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

4. The GAPRole task passes GAP-related events from the stack to the application layer. The Bluetooth stack receives the connection disconnect command and passes it to the GAP layer.

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

```
static void peripheralStateNotificationCB( gapRole_States_t newState, gapRol
 eEvent_t * pEvent )
{
            switch ( newState & GAPROLE_STATE_ADV_MASK )
            {
                 . . .
                 case GAPROLE_ADVERTISING.
                 if( pEvent->gap.opcode== GAP_LINK_TERMINATED_EVENT )
                 {
...
```

5.3.2 Central Role (Central Device Role)

The general operations for initializing the center device are as follows:

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

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

2. Initializes the GAPRole task, including passing the function pointer to the application⊡ call function.

```
if ( events & START_DEVICE_EVT )
{
            // Start the Device
     GAPRole_CentralStartDevice( centralTaskId, &centralBondCB, &centralRoleCB
);
        return ( events^ START_DEVICE_EVT ).
}
```

3. Send GAPRole commands from the application layer

   The application layer calls an application function to send a 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, which receives the command, performs the scanning operation, and returns回 the corresponding state

4. The GAPRole task passes GAP-related events from the stack to the application layer. The Bluetooth stack receives the disconnect command and passes it to the GAP layer. The GAP layer receives the command and passes it directly to the application layer via the回 call 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 low-power Bluetooth connections, enabling certain data to be read and written only after authentication.

Table 5.1 GAP Binding Management Terminology

nomenclature	descriptive
Pairing	key interaction process
Encryption	Data is encrypted or re-encrypted after pairing
check and verify proof (Authentication)	The matching process is done under the protection of a middleman (MITM: Manin the Middle).
Bonding	Store the encryption key in non-volatile memory for the next encryption sequence
authorize authorization (Authorization)	Additional application-level key exchange in addition to authentication
Outside the Box (OOB)	Keys are not exchanged wirelessly, but through a serial port or NFC, for example. Other sources are exchanged. This also provides MITM protection.
Intermediaries (MITM)	Man-in-the-middle protection. This prevents eavesdropping on wirelessly transmitted keys to break encryption
Directconnection(JustWorks)	Middleman-free matchmaking.

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

1. Key pairing (both of the following)
   A. JustWorks, Send Keys Wirelessly
   B. MITM, sending keys through an intermediary
2. The connection is encrypted with a key.
3. Bind the key and store the key.
4. When connecting again, the connection is encrypted using the stored key.

5.4.1 Close Pairing

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

When pairing is turned off, the stack will reject any pairing attempts.

5.4.2 Directly paired but not bound

```
uint8_t mitm= FALSE.
uint8_t bonding= FALSE.
uint8_t pairMode= GAPBOND_PAIRING_MODE_WAIT_FOR_REQ.
GAPBondMgr_SetParameter( GAPBOND_PERI_PAIRING_MODE, sizeof ( uint8_t ), &pai
  rMode ).
GAPBondMgr_SetParameter( GAPBOND_PERI_MITM_PROTECTION, sizeof ( uint8_t ), &
  mitm ).
GAPBondMgr_SetParameter( GAPBOND_PERI_BONDING_ENABLED, sizeof ( uint8_t ), &
  bonding ).
```

Note 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 andkeyboardinput.Ifthedevicedoesnotactuallysupportkeyboardinput,butisconfiguredto enterapasswordthroughthedevice,thepairingcannotbeestablished.

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

5.4.3 Binding through intermediary pairing

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

```
GAPBondMgr_SetParameter( GAPBOND_PERI_PAIRING_MODE, sizeof ( uint8_t ), &pai rMode );

GAPBondMgr_SetParameter( GAPBOND_PERI_MITM_PROTECTION, sizeof ( uint8_t ), &
    mitm ).

GAPBondMgr_SetParameter( GAPBOND_PERI_BONDING_ENABLED, sizeof ( uint8_t ), &
    bonding ).
```

Pairing and binding is done using a middleman, and keys are generated using a 6-digit passphrase.

5.5 Generic Attribute Profile (GATT)

The GATT layer is used by applications to communicate data between two connected devices, where data is passed and stored in the form of features. I n GATT, when two devices are connected, they will variously play one of the following two roles:

- GATT Server - This device provides a GATT client to read or write to the feature database.
- GATT Client - The device reads and writes data from a GATT server.

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

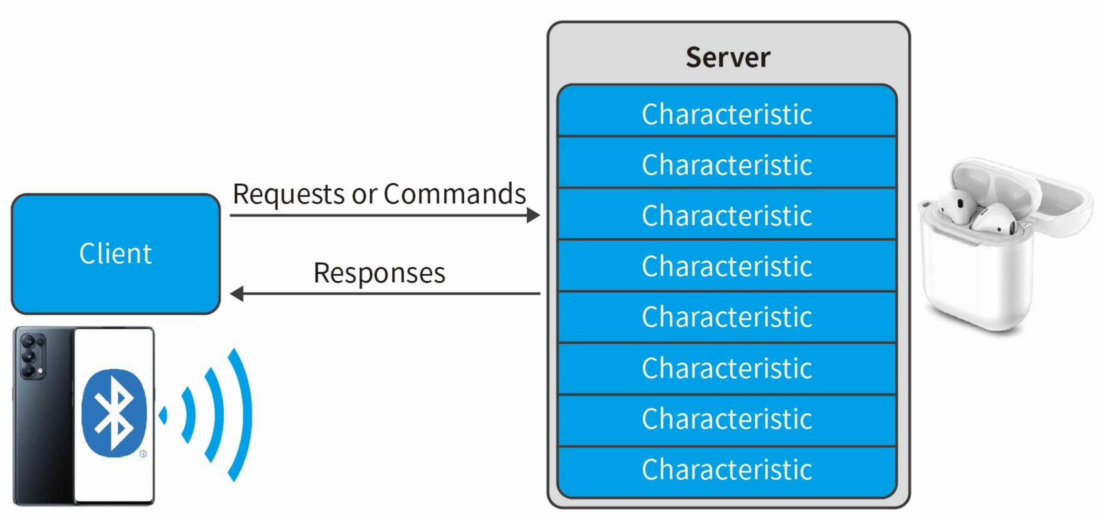


Figure 5.5 GATT Server and Client

Typically the GATT server and client roles are independent of the GAP peripheral device role for the central device. The peripheral device can be a GATT client or server, and the central device can be a GATT server or client. A device can also act as a GATT server or client at the same time.

5.5.1 GATT Features and Attributes

Typical characteristics consist of the following attributes:

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

- Characteristic Declaration: stores the attributes, location, and type of the characteristic value.

Client Characteristic Configuration:This configuration allows the GATT server to configure attributes that need to be sent to the GATT server (notified) or sent to the GATT server and expect a□ response (indicated)

- CharacteristicUserDescription: an ASCII string describing the characteristic value.

These attributes are stored in the attribute table of the GATT server and 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 property represents and is called a Universally Unique Identifier (UUID) Some UUIDs are defined by the BluetoothSIG, others can be customized by the user.

- Permissions - Used to restrict how a GATT client can access the value of this attribute.

- Value (pValue) - A pointer to the value of an attribute whose length cannot be changed after initialization. Maximum size 512 bytes

5.5.2 GATT Services and Agreements

GATT services are collections of features.

The following is a table of attributes in the Peripheral project that corresponds to the gattprofile service (the gattprofile service is a sample configuration file used for testing and demonstration purposes; the full source code is in gattprofile.c and gattprofile.h).

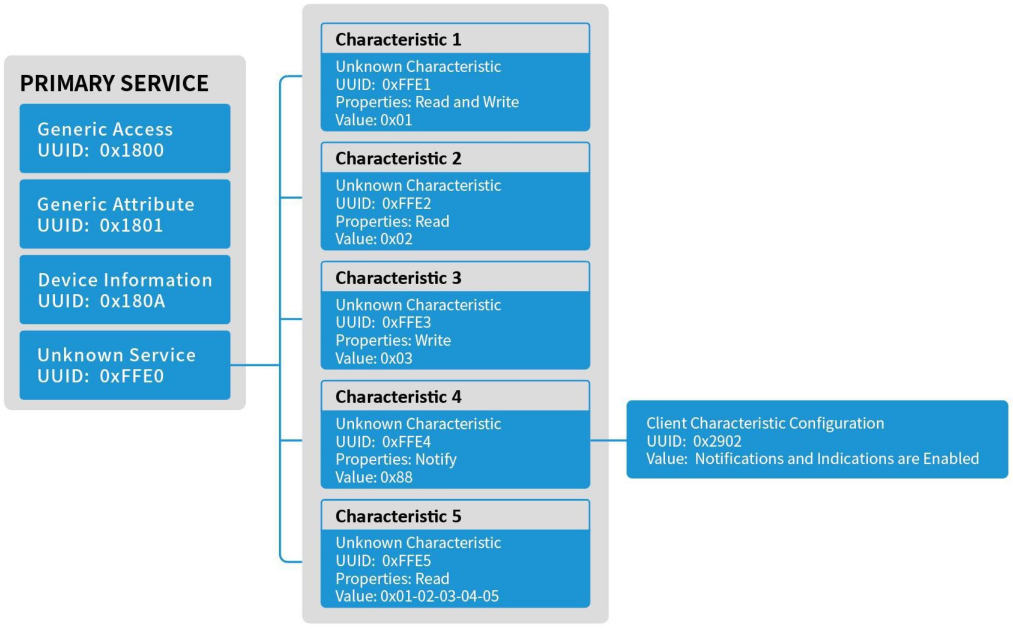

PRIMARY SERVICE
Generic Access UUID: 0x1800
Generic Attribute UUID: 0x1801
Device Information UUID: 0x180A
Unknown Service UUID: 0xFFE0
Characteristic 1
Unknown Characteristic UUID: 0xFFE1 Properties: Read and Write Value: 0x01
Characteristic 2
Unknown Characteristic UUID: 0xFFE2 Properties: Read Value: 0x02
Characteristic 3
Unknown Characteristic UUID: 0xFFE3 Properties: Write Value: 0x03
Characteristic 4
Unknown Characteristic UUID: 0xFFE4 Properties: Notify Value: 0x88
Characteristic 5
Unknown Characteristic UUID: 0xFFE5 Properties: Read Value: 0x01-02-03-04-05
Client Characteristic Configuration UUID: 0x2902 Value: Notifications and Indications are Enabled

Figure 5.6 GATT Attribute Table

Gattprofile contains the following five features:

- simpleProfilechar1 - 1 byte can be read from or written to the GATT client device.

- simpleProfilechar2 - 1 byte can be read from the GATT client device, but not written.

- simpleProfilechar3 - 1 byte can be written from the GATT client device, but not read.

-simpleProfilechar4 - can be configured to send a 1-byte notification to the GATT client device, but be read or written.

- simpleProfilechar5 - 5 bytes can be read from the GATT client device, but not written.

Here are some of the relevant properties:

- 0x02: Allow reading of feature values

- 0x04: Allow feature values to be written without response

- 0x08: Allow writing of feature values (with response)

- 0x10: License for eigenvalue notification (no acknowledgement)

- 0x20: Allow eigenvalue notification (with acknowledgement)

5.5.3 GATT Client Abstraction Layer

GATT clients do not have attribute tables because clients receive information rather than provide it. most of the interfaces to the GATT layer come directly from the application.

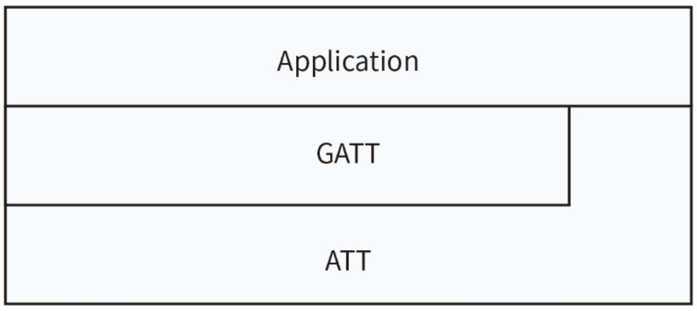


Figure 5.6 GATT Client Abstraction Layer

5.5.3.1 Application of the GATT layer

This section describes how to use the GATT clientdirectly in your application.The corresponding source code canbe found intheexampleprogramCentral.

1. Initialize the GATT client.

```
// Initialize GATT Client
GATT_InitClient().
```

2. Register the relevant information to receive incoming ATT instructions and notifications.

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

3. Execute a client-side program such as GATT_WriteCharValue(), which sends data to the server.

```
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, here is the responseto the " write" operation.Firsttheprotocolstackreceivesthewriteresponseand sends it to the application layer viaaTaskTMOS message.

```
uint16_t Central_ProcessEvent( uint8_t task_id, uint16_t events )
{
        if ( events & SYS_EVENT_MSG )
            {
           uint8_t *pMsg.
                if ( (pMsg= tmos_msg_receive( centralTaskId )) ! = NULL )
                {
                central_ProcessTMOSMsg( (tmos_event_hdr_t *)pMsg ).
...
```

The application layer task queries the GATT message:

```
static void central_ProcessTMOSMsg( tmos_event_hdr_t *pMsg )
{
              switch ( pMsg->event )
              {
                   case GATT_MSG_EVENT.
              centralProcessGATTMsg( (gattMsgEvent_t *) pMsg ).
```

Based on the received content, the application layer can make corresponding functions:

```
static void centralProcessGATTMsg( gattMsgEvent_t *pMsg )
{
...
        else if ( ( pMsg->method== ATT_WRITE_RSP ||
            ( ( pMsg->method== ATT_ERROR_RSP ) &&
              ( pMsg->msg.errorRsp.reqOpcode== ATT_WRITE_REQ )                  ) )
             {
        //Application
...
```

The application clears the message when processing is complete:

```
         // Release the TMOS message
          tmos_msg_deallocate( pMsg ).
        }
        // return unprocessed events
        return (events^ SYS_EVENT_MSG).
}
```

5.5.4 GATT Server Abstraction Layer

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

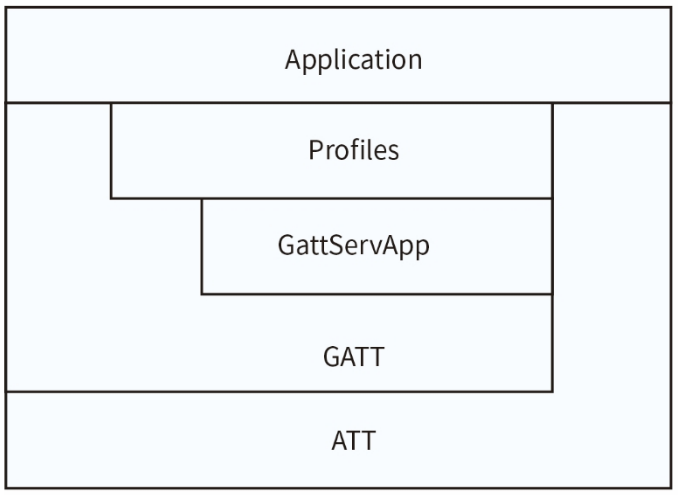


Figure 5.7 GATT Server Abstraction Layer

The specification for the use of GATT is as follows:

1. Create a GATT to configure the GATTServApp module.
2. Use the API interface in the GATTServApp module to operate on the GATT layer.

5.5.4.1 GATTServApp Module

The GATTServApp module is used to store and manage an application's property sheet, which is used by various profiles to add their feature values to the property sheet. Its functions include finding specific properties, reading client-side feature values, and modifying client-side feature values. Please refer to the API section for details.

With each initialization, the application uses the GATTServApp module to add services to build the GATT table. The contents of each service include the UUID, value, permissions, and read/write permissions. Figure 5.8 depicts the GATTServApp module adding services.

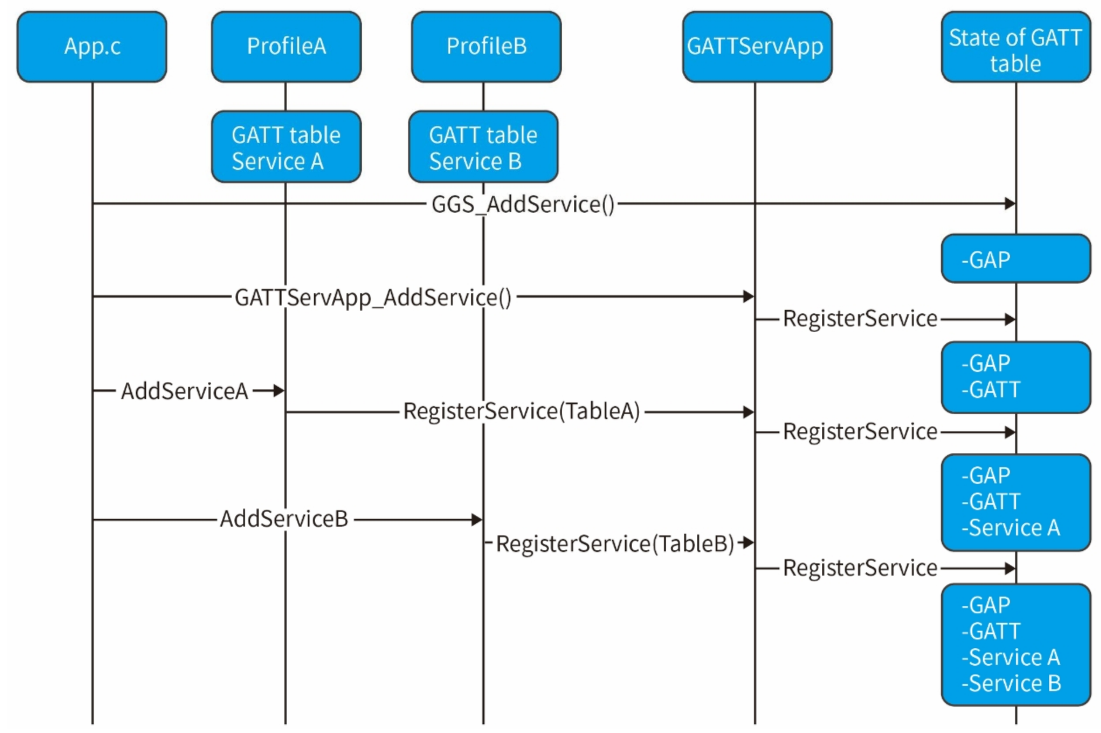


Figure 5.8 Attribute Table Initialization

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

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

5.5.4.2 Configuration File Architecture

This section describes the basic architecture of the profile and provides an example of the use of the GATTProfile in the Peripheral project.

5.5.4.2.1 Creating a Property Table

Each service must define a fixed-size attribute table to be passed to the GATT layer.
I n the Peripheral project, the definition is as follows:

```
static gattAttribute_t simpleProfileAttrTbl[=
...
```

The format of each attribute is as follows:

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

The individual elements in the attribute:

- type - The UUID associated with the attribute.

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

w h e r e len can be 2 bytes or 16 bytes. *uuid can be a number pointing to a number stored in the Bluetooth SIG or a UUID pointer for customization.

- Permission — Configures whether the GATT client device can access the values of the attributes. The configurable permissions are listed below:

—— GATT_PERMIT_READ //readable

—— GATT_PERMIT_WRITE // Writable

—— GATT_PERMIT_AUTHEN_READ // Authentication required for reads

—— GATT_PERMIT_AUTHEN_WRITE //authentication write required

—— GATT_PERMIT_AUTHOR_READ // Authorization to read is required.

—— GATT_PERMIT_ENCRYPT_READ // Encrypted read required

—— GATT_PERMIT_ENCRYPT_WRITE // encrypted write required

- Handle — Handle assigned by GATTServApp, handles are automatically assigned in order.

- pValue - Pointer to the value of the attribute. Its length cannot be changed after initialization. The maximum size is 512 bytes.

The following creates the property sheet in the Peripheral project: first create the service properties:

```
    // Simple Profile Service
{
           { ATT_BT_UUID_SIZE, primaryServiceUUID             }, /* type */
           GATT_PERMIT_READ.                                      /* permissions */
           0,                                                     /* handle */
            (uint8_t *)&simpleProfileService                          /* pValue */
},
```

This attribute is the primary service UUID (0x2800) as defined by the Bluetooth SIG. The GATT client must read this attribute, so set the permissions to readable. pValue is a pointer to the UUID of the service, customized as 0xFFE0.

```
// Simple Profile Service attribute
static const gattAttrType_t simpleProfileService= { ATT_BT_UUID_SIZE, simple
    eProfileServUUID }
```

The feature's declaration, value, user description, and client-side feature configuration are then created, as described in section 5.5.1.

```
        // Characteristic 1 Declaration
{
          { ATT_BT_UUID_SIZE, characterUUID }
          GATT_PERMIT_READ.
          0,
          &simpleProfileChar1Props
},
```

The type of the Characteristic Declaration needs to be set to the BluetoothSIG defined Characteristic UUID value (0x2803) which must be read by the GATT client, so its permissions are set to readable. The declared value refers to the attributes of the feature, which are readable and writable.

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

```
static uint8_t simpleProfileChar1Props= GATT_PROP_READ| GATT_PROP_WRITE;

//     Characteristic Value 1
   {
            { ATT_BT_UUID_SIZE, simpleProfilechar1UUID },.
            GATT_PERMIT_READ| GATT_PERMIT_WRITE,.
           0,
            simpleProfileChar1
   },
    }
```

In the feature value, the type is set to a custom UUID (0xFFF1) and the permissions of the value are set to read and write since the attributes of the feature value are read and write. pValue points to the location of the actual value as follows:

```
// Characteristic 1 Value
static     uint8_t simpleProfileChar1[SIMPLEPROFILE_CHAR1_LEN]= { 0 };

// Characteristic 1 User Description
   {
           { ATT_BT_UUID_SIZE, charUserDescUUID },.
           GATT_PERMIT_READ.
          0,
           simpleProfileChar1UserDesp
   }, }
```

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

```
// Simple Profile Characteristic 1 User Description
static uint8_t simpleProfileChar1UserDesp[]= "Characteristic 1\0";

// Characteristic 4 configuration
{
        { ATT_BT_UUID_SIZE, clientCharCfgUUID }
           GATT_PERMIT_READ| GATT_PERMIT_WRITE,.
           0.
           (uint8_t *)simpleProfileChar4Config
},
```

This type must be set to the Client Feature Configuration UUID defined by the Bletooth SIG (0x2902) which must be read and written by the GATT client, so the permissions are set to read and write. pValue Points to the address of the Client Feature Configuration value.

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

5.5.4.2.2 Add Service

When the Bluetooth stack is initialized, the GATT services it supports must be added. The services include the GATT services required by the stack, such as GGS_AddService and GATTServApp_AddService, as well as user-defined services such as SimpleProfile_AddService in the Peripheral project. AddService(), for example, these functions perform the following actions:

First the Client Characteristic Configuration i.e. Client Characteristic Configuration (CCC)needs to be defined.

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

Then initialize the CCC.

For each CCC in the configuration file, the GATTServApp_InitCharCfg() function must be called. This function initializes the CCC with information from a previously bound connection.If the function cannot find the information, it sets the initial value to the default.

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

Finally, register the configuration file through GATTServApp.

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

```
// Register GATT attribute list and CBs with GATT Server App
     status= GATTServApp_RegisterService( simpleProfileAttrTbl,
                                          GATT_NUM_ATTRS( simpleProfileAttrTbl ),
                                           GATT_MAX_ENCRYPT_KEY_SIZE, the
                                           &simpleProfileCBs ).
```

5.5.4.2.3 Register the application回call function

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

```
bStatus_t SimpleProfile_RegisterAppCBs( simpleProfileCBs_t *appCallbacks )
{
     if ( appCallbacks )
     {
       simpleProfile_AppCBs= appCallbacks;

       return ( SUCCESS ).
     }
     else
{
       return ( bleAlreadyInRequestedMode ).
}
}
```

回The call functions are as follows:

```
// Callback when a characteristic value has changed
typedef void (*simpleProfileChange_t)( uint8_t paramID ).

typedef struct
{
    simpleProfileChange_t                    pSimfnpleProfileChange; // Called when chara
  cteristic value changes
} simpleProfileCBs_t.
```

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

```
// Simple GATT Profile Callbacks
static simpleProfileCBs_t =
{
      simpleProfileChangeCB          // Charactersitic value change callback
}
```

5.5.4.2.4 Read and write回call function

When the configuration file is read/written, the corresponding回function is required, and its registration method is consistent with the application回function, specifically refer to the Peripheral project.

5.5.4.2.5 Getting and Setting Configuration Files

Configuration files contain read and write feature functionality, and Figure 5.9 depicts the logic by which the application sets configuration file parameters.

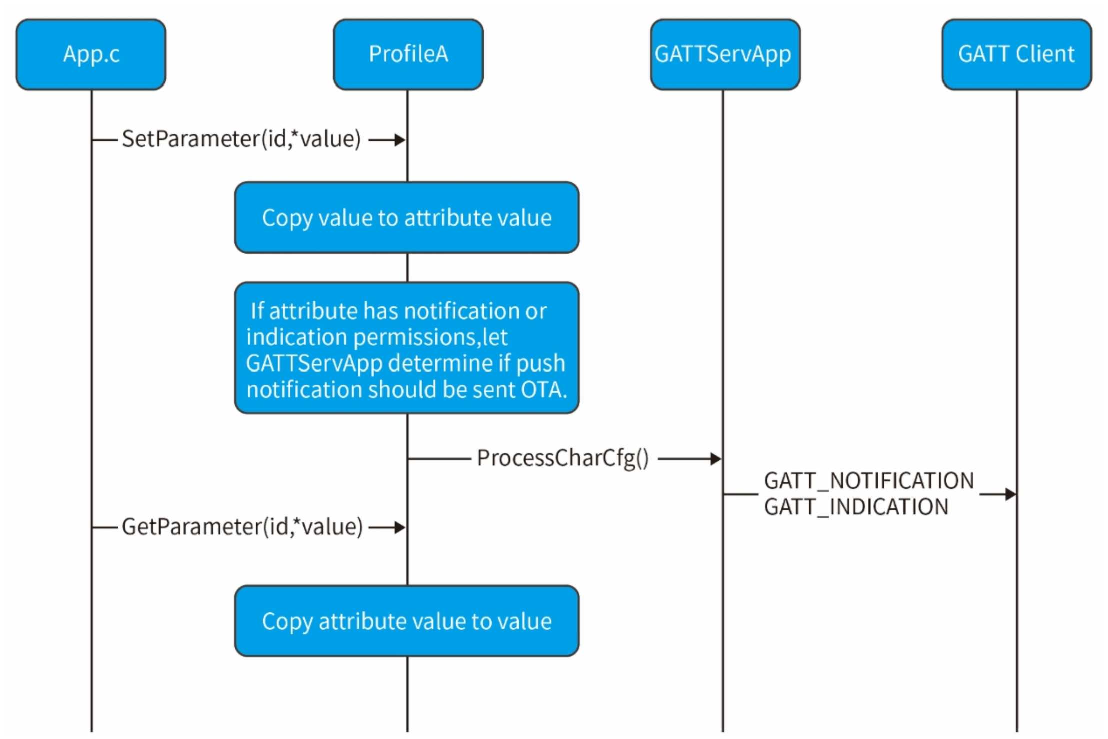

App.c
ProfileA
GATTServApp
GATT Client
SetParameter(id,*value)
Copy value to attribute value
If attribute has notification or indication permissions,let GATTServApp determine if push notification should be sent OTA.
ProcessCharCfg()
GATT_NOTIFICATION
GATT_INDICATION
GetParameter(id,*value)
Copy attribute value to value

Figure 5.9 Getting and Setting Profile Parameters

The application code is as follows:

```
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 host's upper layers (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 packets in 64KB lengths. It is also capable of handling multiplexing of protocols to provide multiple connections and multiple connection types (over a single air interface) while providing quality of service support and group communication.The CH58x Bluetooth stack supports an effective maximum MTU of 247.

5.7 Host-Controller Interaction

HCI (HostControllerInterface) which connects the host and the controller, and translates the host's operations into HCI commands to the controller.There are four types of HCI supported by BLE CoreSpec: UART, USB, SDIO, and 3-Wire UART.For a single Bluetooth chip with a full protocol stack, you only need to call API For a single Bluetooth chip with a full protocol stack, you only need to call the API function, at this time, HCI is a function call to and回; for products with only a controller, i.e., the main controller chip is used to operate the BLE chip, and the BLE chip is connected to the main controller chip as a plug-in chip. In this case, the main controller only needs to interact with the BLE chip through standard HCI commands (usually UART).

The HCIs discussed in this guide are all function calls versus function回 calls.

6. Create a BLE application

6.1 summarize

After reading the previous chapters, you should understand how to implement a low-power Bluetooth application. This chapter describes how to start writing a low-power Bluetooth application, as well as some considerations.

6.2 Configuring the Bluetooth Protocol Stack

First you need to determine the role of this device, we offer the following roles:

- Central
- Peripheral
- Broadcaster
- Observer

Selecting different roles requires calling different role initialization APIs, see Section 5.3 for details.

6.3 Defining Low Power Bluetooth Behavior

Use the Low Power Bluetooth stack's APIs to define system behavior, such as adding configuration files, adding a GATT database, configuring security modes, and so on. See Chapter 5 for details.

6.4 Defining application tasks

Make sure that the application contains the□ call functions to the protocol stack and event handlers TMOS. You can refer to adding additional tasks as described in Chapter 3
.

6.5 Application Configuration File

Configure DCDC enable, RTC clock, sleep function, MAC address, RAM size for low power Bluetooth stack, etc. in config.h file.

Note that WAKE_UP_RTC_MAX_TIME is the time to wait for the 32M crystal to stabilize. This stabilization time is affected by the crystal, voltage, stabilization, and other factors. You need to add a buffer to the wakeup time to improve stability.

6.6 Limit application processing during low-power Bluetooth operation

Due to the time-dependent nature of the low-power Bluetooth protocol, the controller must process each connection event or broadcast event before it arrives. Failure to process it in time can result in retransmission or connection disconnection. And TOMS is not multi-threaded, so when Low Power Bluetooth has a transaction to process, other tasks must be stopped for the controller to process. So make sure that your application does not take up a large number of events, and if complex processing is required, refer to section 3.3 to split them up.

6.7 disruptions

During the low-power Bluetooth operation, the time needs to be calculated by the RTC timer, so during this period, do not disable the global interrupt, and the time occupied by a single interrupt service program should not be too long, otherwise the long-term interruption of the low-power Bluetooth operation will lead to connection disconnection.

7. Create a simple RF application

7.1 summarize

RF applications are based on RF transmit and receive PHYs and realize wireless communication in the 2.4GHz band. The difference with BLE is that the RF application does not establish the protocol of BLE.

7.2 Configuring the protocol stack

First initialize the Bluetooth library:

```
CH58X_BLEInit( ).
```

Then configure the role of this device as RF Role:

```
RF_RoleInit ( ).
```

7.3 Defining application tasks

Register RF tasks, initialize RF functions, and register RF's回call functions:

```
taskID= TMOS_ProcessEventRegister( RF_ProcessEvent ).
        rfConfig.accessAddress= 0x71764129; // Disable the use of 0x5555555555 as well as the
0xAAAAAAAAAA (no more than 24 bit inversions and no more than 6 consecutive 0's or 1's are recommended).
        rfConfig.CRCInit= 0x555555555;
        rfConfig.Channel= 8;
        rfConfig.Frequency= 2480000;
        rfConfig.LLEMode= LLE_MODE_BASIC|LLE_MODE_EX_CHANNEL|LLE_MODE_NON_RSSI;
    // Enabling LLE_MODE_EX_CHANNEL indicates that rfConfig.Frequency is selected as the communication frequency.
        rfConfig.rfStatusCB= RF_2G4StatusCallBack;
        state= RF_Config( &rfConfig );
```

7.4 Application Configuration File

Configure DCDC enable, RTC clock, sleep function, MAC address, RAM size for low power Bluetooth stack, etc. in config.h file.

Note that WAKE_UP_RTC_MAX_TIME is the time to wait for the 32M crystal to stabilize. This stabilization time is affected by crystal, voltage, stability, etc. You need to add a buffer to the wakeup time to improve stability.

7.5 RF Communications

7.5.1 Basic mode

In Basic mode it is only necessary to keep the receiver in receive mode all the time, i.e. call the RF_RX() function. However, it should be noted that after receiving the data, you need to call RF_RX() function again to make the device in receive mode again, and do not call RF send/receive function directly in RF_2G4StatusCallBack()回call function, which may cause its status confusion.

The communication schematic is shown below:

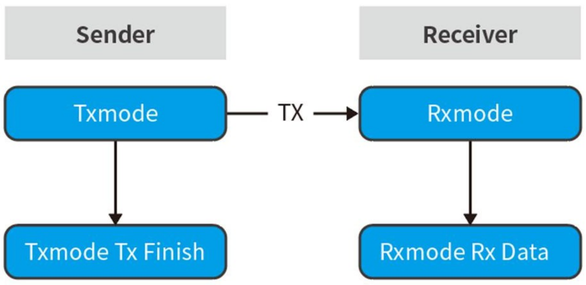


Figure 7.1 Basic Mode Communication Diagram

The API for RF transmission is RF_Tx(), please refer to section 8.6 for details.

If RF receives data, it will enter the call function RF_2G4StatusCallBack() and get the received data in the回 call function.

7.5.2 Auto mode

Since Basic mode is only a one-way transmission, the user has no way of knowing whether the communication

was successful or not, resulting in Auto mode.

Style.

Auto mode adds the mechanism of receiving response to Basic mode, i.e., after the receiver receives the data, the Data will be sent to the sender to notify the sender that the data has been successfully received.

The communication schematic is shown below:

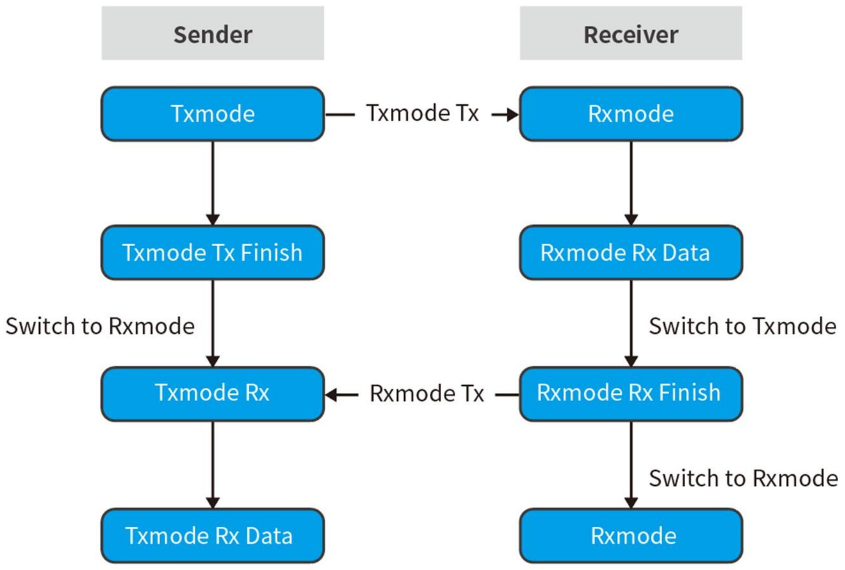


Figure 7.2 Auto Mode Communication Diagram

In Auto mode, RF will automatically switch to receive mode after sending data, the timeout of this receive mode is 3ms, if no data is received within 3ms, the receive mode will be closed. The received data and the timeout status are returned to回 function.

回 to the application layer.

7.5.2.1 automatic frequency hopping

Based on the RF Auto mode design, the automatic frequency hopping solution can effectively solve the interference problem of 2.4GHz channel. To use the FH function, you need to actively turn on the FH receive or FH transmit event:

```
// Enable frequency-hopping transmission
 if( events & SBP_RF_CHANNEL_HOP_TX_EVT ){
      PRINT("\n------------- hop tx... \n").
      if( RF_FrequencyHoppingTx( 16 ) ){
        tmos_start_task( taskID , SBP_RF_CHANNEL_HOP_TX_EVT ,100 );
      }
      return events^SBP_RF_CHANNEL_HOP_TX_EVT.
}
// Enable frequency
if( events & SBP_RF_CHANNEL_HOP_RX_EVT ){
      PRINT("hop rx... \n").
      If( RF_FrequencyHoppingRx( 200 ) )
{
        tmos_start_task( taskID , SBP_RF_CHANNEL_HOP_RX_EVT ,400 );
}
      else
{
        RF_Rx( TX_DATA,10,0xFF,0xFF ).
}
      return events^SBP_RF_CHANNEL_HOP_RX_EVT.
}
```

After configuring the RF communication mode as auto mode, the sender turns on the frequency hopping send event:

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

The receiving hair turns on the frequency hopping reception event:

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

The frequency hopping function can be realized.

Note that the receiver needs to turn off the RF (call RF_Shut() function)before turning on the FH receive event if it is already in receive mode.

8. API

8.1 TMOS API

8.1.1 directives

```
bStatus_t TMOS_TimerInit( pfnGetSysClock fnGetClock )
```

TMOS clock initialization.

parameters	descriptive
pfnGetSysClock	0: Select RTC as system clock Other valid values: other clock fetch interfaces such as SYS_GetSysTickCnt()
return (to)回	0: SUCCESS 1: FAILURE

```
tmosTasklD TMOS_ProcessEventRegister( pTaskEventHandIerFn eventCb )
```

The registration event回 call function, generally used to register the task to be executed first.

parameters	descriptive
eventCb	TMOS Tasks回 Call Functions
return (to)回	Assigned ID value, OxFF means invalid

```
bStatus_t tmos_set_event( tmosTaskID taskID, tmosEvents event )
```

Immediately starts the corresponding event in the taskID task, and executes it once.

parameters	descriptive
taskID	tmos Assigned Task ID
event	Events in the mission
return (to)回	0: Success

```
bStatus_t tmos_start_task( tmosTaskID taskID, tmosEvents event, tmosTimer time )
```

Delay time*625μs to start the corresponding event event in taskID task and execute it once.

parameters	descriptive
taskID	tmos Assigned Task ID
event	Events in the mission
time	Time delayed
return (to)回	0: Success

```
1. bStatus_t tmos_stop_event( tmosTaskID taskID, tmosEvents event )
```

Stops an event, which will not take effect after this function is called.

parameters	descriptive
taskID	tmos Assigned Task ID
event	Events in the mission
return (to)回	0: Success

```
1. bStatus_t tmos_clear_event( tmosTaskID taskID, tmosEvents event )
```

Cleans up an event that has timed out, taking care not to execute it within its own event function.

parameters	descriptive
taskID	tmos Assigned Task ID
event	Events in the mission
return (to)回	0: Success

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

Delay time*625μs to execute the event event, call a loop to execute it once, unless running tmos_stop_task to turn it off.

parameters	descriptive
taskID	tmos Assigned Task ID
event	Events in the mission
time	Time delayed
return (to)回	0: Success

```
1. tmosTimer tmos_get_task_timer( tmosTaskID taskID, tmosEvents event )
```

Gets the number of ticks the event is away from the expiration event.

parameters	descriptive
taskID	tmos Assigned Task ID
event	Events in the mission
return (to)回	!0: number of ticks before the event expires 0: Incident not found

```
1. uint32_t TMOS_GetSystemClock( void )
```

Return to回 tmos System runtime in 625μs, e.g. 1600=1s.

parameters	descriptive

return (to)回	TMOS runtime

```
1. void TMOS_SystemProcess( void )
```

The system handler functions of tmos need to be run continuously in the main function.

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

Sends a message to a task. When this function is called, the corresponding task's message event will be set to 1

parameters	descriptive
taskID	tmos Assigned Task ID
msg_ptr	message pointer
return (to)回	SUCCESS: Success INVALID_TASK: Task ID Invalid INVALID_MSG_POINTER: Invalid message pointer

```
1. uint8_t *tmos_msg_receive( tmosTaskID taskID )
```

Receive messages.

parameters	descriptive
taskID	tmos Assigned Task ID
return (to)回	Message received or no message to be received (NULL)

```
1. uint8_t *tmos_msg_allocate( uint16_t len )
```

Requests memory space for the message.

parameters	descriptive
len	Length of the message
return (to)回	Pointer to the requested buffer NULL: Application failed

```
1. bStatus_t tmos_msg_deallocate( uint8_t *msg_ptr )
```

Free the memory space occupied by the message.

parameters	descriptive
msg_ptr	message pointer
return (to)回	0: Success

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

Reads data from the NV.

Note: Read and write operations in the NV area should be called before the TMOS system is running.

parameters	descriptive
id	Valid NV Project ID
len	The length of the read data
pBuf	Pointer to the data to be read
return (to)回	SUCCESS NV_OPER_FAILED: failed

```
void TMOS_TimerIRQHandler( void )
```

TMOS Timer Interrupt Functions.

The following functions are more memory efficient than the C library functions

```
uint32_t tmos_rand( void )
```

Generate pseudo-random numbers.

parameters	descriptive
return (to)回	pseudorandom number

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

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

parameters	descriptive
src1	memory block pointer
src2	memory block pointer
len	Number of bytes to be compared
return (to)回	1: Same 0: Different

```
bool tmos_isbufset( uint8_t *buf, uint8_t val, uint32_t len )
```

Compares whether the given data are all the given values.

parameters	descriptive
Buf	buffer address
val	numerical value
len	Length of data
return (to)回	1: Same 0: Different

```
1. uint32_t tmos_strlen( char *pString )
```

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

parameters	descriptive
pString	The string whose length is to be calculated
return (to)回	Length of the string

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

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

parameters	descriptive
pDst	Memory block to be filled
Value	The value to be set
len	Number of characters to be set to this value
return (to)回	Pointer to storage area pDst

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

Copy len bytes from storage area src to storage area dst.

parameters	descriptive
dst	Target array for storing the contents of the copy, type-forced conversion to void* pointer
src	Data source to be copied, type-forced conversion to void* pointer
len	Number of bytes to be copied
return (to)回	Pointer to target storage area dst

8.2 GAP API

8.2.1 directives

```
1. bStatus_t GAP_SetParamValue( uint16_t paramID, uint16_t paramValue )
```

Sets the GAP parameter values. Use this function to change the default GAP parameter values.

parameters	descriptive
paramID	ID of the parameter, refer to 8.2.2.
paramValue	New parameter values
return (to)回	SUCCESS o r INVALIDPARAMETER (invalid parameter ID)

```
1. uint16 GAP_GetParamValue( uint16_t paramID )
```

G e t s the value of the GAP parameter.

parameters	descriptive
paramID	ID of the parameter, refer to 8.1.2.
return (to)囘	Parameter value of the GAP; if the parameter ID is invalid return囘 0xFFFF

8.2.2 Configuration parameters

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

Parameter ID	descriptive
tgap_gen_disc_adv_min	Broadcast duration of general-purpose broadcast mode, unit: 0.625ms (default) (Recognized value: 0)
tgap_lim_adv_timeout	Time-limited discoverable broadcast mode broadcast duration, unit:1s (default) (Recognized 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: 16384)
TGAP_DISC_SCAN_INT	Scan interval, unit:0.625ms (default: 16)
TGAP_DISC_SCAN_WIND	Scanning window, unit:0.625ms (default: 16)
TGAP_CONN_EST_SCAN_INT	Scanning interval for establishing connection, unit:0.625ms (default value.) 16)
TGAP_CONN_EST_SCAN_WIND	Scanning window for establishing connection, unit:0.625ms (default.) 16)
tgap_CONN_est_int_min	Minimum connection interval to establish connection, unit:1.25ms (default) (Value: 80)
TGAP_CONN_EST_INT_MAX	Maximum connection interval to establish connection, unit:1.25ms (default) (Value: 80)
tgap_CONN_EST_SUPERV_TIMEOUT	Connection management timeout for connection establishment, unit:10ms (default) (Recognized value: 2000)
TGAP_CONN_EST_LATENCY	Slave device delay for establishing connection (default: 0)

8.2.3 event

This section describes the events related to the GAP layer, which can be declared in the CH58xBLE_LIB.h file. Some of these events are passed directly to the application and some are handled by GAPRole and GAPBondMgr. They will be passed as GAP_MSG_EVENT with a header, regardless of the layer to which they are passed:

```
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 the format of the event delivery message. Refer to CH58xBLE_LIB.h for details.

- GAP_DEVICE_INIT_DONE_EVENT: Set this event when the device initialization is completed.

```
typedef struct
{
    tmos_event_hdr_t hdr.                       //! < GAP_MSG_EVENT and status
     uint8_t opcode.                             //! < GAP_DEVICE_INIT_DONE_EVENT
     uint8_t devAddr[B_ADDR_LEN];.               //! < Device's BD_ADDR
     uint16_t dataPktLen;                        //! < HC_LE_Data_Packet_Length
     uint8_t numDataPkts.                        //! < HC_Total_Num_LE_Data_Packets
 } gapDeviceInitDoneEvent_t;
```

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

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

- GAP_END_DISCOVERABLE_DONE_EVENT: Set this event when the broadcast ends.

```
typedef struct
{
   tmos_event_hdr_t hdr; //! < GAP_MSG_EVENT and status
uint8_t opcode.                   //! < GAP_END_DISCOVERABLE_DONE_EVENT
} gapEndDiscoverableRspEvent_t;
```

- GAP_LINK_ESTABLISHED_EVENT: Set this event after the connection is established.

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

-GAP_LINK_TERMINATED_EVENT: this event is set after the connection is disconnected.

```
typedef struct
{
      tmos_event_hdr_t hdr.          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: Set this event after receiving a parameter update event.

```
typedef struct
{
      tmos_event_hdr_t hdr.             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: Discovered devices place this event during device discovery.

```
typedef struct
{
      tmos_event_hdr_t hdr.           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 directives

```
bStatus_t GAPRole_SetParameter( uint16_t param, uint16_t len, void *pValue )
```

Set the GAP role parameters.

parameters	descriptive
param	Configuration parameter ID, see section 8.2.1.2 for details.
len	Length of data written
pValue	Pointer to the value of the setup parameter. The pointer depends on the parameter ID and will be forced to be converted to a The appropriate data type.
return (to)回	SUCCESS INVALIDPARAMETER: Parameter is invalid. bleInvalidRange: parameter length is invalid blePending: last parameter update not finished bleIncorrectMode: mode error

```
bStatus_t GAPRole_GetParameter( uint16_t param, void *pValue )
```

Get GAP role parameters.

parameters	descriptive
param	Configuration parameter ID, see section 8.2.1.2 for details.
pValue	A pointer to the location of the get parameter. This pointer depends on the parameter ID and will be forced to turn the Switch to the appropriate data type.
return (to)回	SUCCESS INVALIDPARAMETER: invalid parameter

```
bStatus_t GAPRole_TerminateLink( uint16_t connHandle )
```

Disconnects the connection specified by the current connHandle.

parameters	descriptive
connHandle	connection handle
return (to)回	SUCCESS bleIncorrectMode: mode error

```
bStatus_t GAPRole_ReadRssiCmd( uint16_t connHandle )
```

Reads the RSSI value for the current connHandle specified connection.

parameters	descriptive
connHandle	connection handle
return (to)回	SUCCESS

	0x02:No valid connection

8.3.1.2 Common Configurable Parameters

parameters	Read/Write	magnitude	descriptive
GAPROLE_BD_ADDR	read-only (computing)	uint8	device address
GAPRPLE_ADVERT_ENABLE	readable and writable	uint8	Enable or disable broadcasting, enabled by default
GAPROLE_ADVERT_DATA	readable and writable	≤240	Broadcast data, defaults to all 0's.
GAPROLE_SCAN_RSP_DATA	readable and writable	≤240	Scanning answer data, default all 0
GAPROLE_ADV_EVENT_TYPE	readable and writable	uint8	Broadcast type, non-directional broadcasts can be connected by default
GAPROLE_MIN_CONN_INTERVAL	readable and writable	uint16	Minimum connection interval, range: 1.5ms~4s. Default 8.5ms.
GAPROLE_MAX_CONN_INTERVAL	readable and writable	uint16	Maximum connection interval, range: 1.5ms~4s. Default 8.5ms.

8.3.1.3 回invoke a 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 is the回call to read the RSSI and its pointer points to the application so that GAPRole can return the event to回to the application. It is passed in the following way:

```
// GAP Role Callbacks
static gapCentralRoleCB_t =
{
    centralRssiCB, // RSSI callback      // RSSI callback
    centralEventCB, // Event callback    // Event callback
    centralHciMTUChangeCB              // MTU change callback
};
```

8.3.2 GAPRolePeripheral Role API

8.3.2.1 directives

```
bStatus_t GAPRole_PeripheralInit( void )
```

Bluetooth slave GAPRole task initialization.

parameters	descriptive
return (to)回	SUCCESS bleInvalidRange: parameter out of range

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

Bluetooth slave device initialization.

parameters	descriptive
taskid	tmos Assigned Task ID
pCB	Binding□ call function, including key□ call, pairing status□ call
pAppCallbacks	GAPRole□ call function, including the status of the device□ call, RSSI□ call, parameter update□ harmonize
return (to)□	SUCCESS bleAlreadyInRequestedMode:The device has already been initialized

```
bStatus_t GAPRole_PeripheralConnParamUpdateReq( uint16_t connHandle,
                                                              uint16_t minConnInterval, uint16_t
minConnInterval, uint16_t
                                                              uint16_t maxConnInterval, uint16_t
maxConnInterval, uint16_t
                                                              uint16_t latency, the
                                                              uint16_t connTimeout,
                                                              uint8_t taskId)
```

Bluetooth slave connection parameters are updated.

Note: Unlike GAPRole_UpdateLink(), which is a negotiated connection parameter between the slave and the host, GAPRole_UpdateLink() is a direct configuration of the connection parameter by the host.

parameters	descriptive
connHandle	connection handle
minConnInterval	Minimum connection interval
maxConnInterval	Maximum connection interval
latency	Number of delayed events from devices
connTimeout	Connection timeout
taskID	Task ID assigned by toms
return (to)□	SUCCESS: Parameters uploaded successfully BleNotConnected: no connection so parameters can't be updated bleInvalidRange: parameter error

8.3.2.2 □invoke a function

```
typedef struct
{
     gapRolesStateNotify_t pfnStateChange; //! < Whenever the device changes
state
     gapRolesRssiRead_t pfnRssiRead; //! < When a valid RSSI is read from
controller
     gapRolesParamUpdateCB_t pfnParamUpdate; //! < When the connection
parameteres are updated
} gapRolesCBs_t.
```

Slave Status回call function:

```
/**
       * Callback when the device has been started. callback event to
       * :: the Notify of a state change.
*/
void (*gapRolesStateNotify_t )( gapRole_States_t newState,
  gapRoleEvent_t *pEvent).
```

Among them, the states are categorized as follows:

- GAPROLE_INIT //waiting for startup
- GAPROLE_STARTED //Initialization completed but not broadcasted.
- GAPROLE_ADVERTISING //being broadcasted
- GAPROLE_WAITING //Device started but not broadcasting, waiting to broadcast again.
- GAPROLE_CONNECTED // Connection Status
- GAPROLE_CONNECTED_ADV //connected status and on broadcast
- GAPROLE_ERROR//Invalid state, if this state indicates an error

Slave parameters update the回call function:

```
/**
       * :: Callback when the connection parameteres are updated.
       */
typedef void (*gapRolesParamUpdateCB_t)( uint16_t connHandle,
                                                    uint16_t connInterval,
                                                    uint16_t connSlaveLatency, uint16_t
                                                    connSlaveLatency, uint16_t connSlaveLatency
                                                    uint16_t connTimeout );
```

Called on a successful parameter update. 回call function.

8.3.3 GAPRole Central Role API

8.3.3.1 directives

```
bStatus_t GAPRole_CentralInit( void )
```

Host GAPRole task initialization.

parameters	descriptive
return (to)回	SUCCESS bleInvalidRange: parameter out of range

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

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

parameters	descriptive
taskid	tmos Assigned Task ID
pCB	Binding□ call function, including key□ call, pairing status□ call
pAppCallbacks	GAPRole□ call function, including the status of the device□ call, RSSI□ call, parameter update□ harmonize
return (to)□	SUCCESS bleAlreadyInRequestedMode: the device is already started up

```
bStatus_t GAPRole_CentralStartDiscovery( uint8_t mode, uint8_t activeScan, u int8_t whiteList )
```

Host scanning parameter configuration.

parameters	descriptive
mode	Scanning modes, divided into: DEVDISC_MODE_NONDISCOVERABLE: no setting DEVDISC_MODE_GENERAL: scan for generic discoverable devices DEVDISC_MODE_LIMITED: scan for limited discoverable devices DEVDISC_MODE_ALL: Scanning for all
activeScan	TRUE to enable scanning
whiteList	TRUE for whitelisted devices only
return (to)□	SUCCESS

```
bStatus_t GAPRole_CentralCancelDiscovery( void )
```

The host stops scanning.

parameters	descriptive
return (to)□	SUCCESS bleInvalidTaskID: no task is being scanned bleIncorrectMode: not in scanning mode

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

Connect with the opposite end of the device.

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

	ADDRTYPE_PRIVATE_NONRESOLVE: non-resolvable private address ADDRTYPE_PRIVATE_RESOLVE: resolvable private address
peerAddr	peer-to-peer device address
return (to)回	SUCCESS: successful connection bleIncorrectMode: invalid profile bleNotReady: scanning in progress bleAlreadyInRequestedMode: can't be processed at the moment bleNoResources: too many connections

8.3.3.2 回invoke a function

These pointers to the回 call functions are passed from the application to the GAPRole so that the GAPRole can return events to回 to the application. They are passed as follows:

```c
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回 call function:

```c
/**
* 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 RSSI to the

application. Host Events回 call function:

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

This回 call is used to pass GAP state change events to the application.

回The tuning event can be found in section 8.1.3.

MTU Interactive回 call function:

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

i.e., the size of the packet that interacts with low-power Bluetooth.

8.4 GATT API

8.4.1 directives

8.4.1.1 slave command

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

The server indicates a feature value to the client and expects the Attribute Protocol layer to acknowledge that the indication has been successfully received. Note that memory needs to be freed when the return to回fails.

parameters	descriptive
connHandle	connection handle
pInd	Points to the command to be sent
authenticated	Whether an authenticated connection is required
taskId	tmos Assigned Task ID

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

The server notifies the client of the feature value, but does not expect any attribute protocol layer confirmation that the notification has been successfully received. Note that memory needs to be freed when the return to回fails.

parameters	descriptive
connHandle	connection handle
pInd	Points to instructions to be notified
authenticated	Whether an authenticated connection is required

8.4.1.2 host command

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

When the value supported by the client is greater than thevalueofthe attribute protocol's default ATT_MTU,theclientuses thisproceduretoset the ATT_MTU to the maximum possible value that can be supported by both devices.

parameters	descriptive
connHandle	connection handle
pReq	Points to the command to be sent
taskID	ID of the notified task

```
bStatus_t GATT_DiscAllPrimaryServices( uint16_t connHandle, uint8_t taskId)
```

Discover all master services on the server.

parameters	descriptive
connHandle	connection handle
taskID	ID of the notified task

```
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 master service on the server. Since there may be more than one master service on the server, the discovered master service is identified by its UUID.

parameters	descriptive
connHandle	connection handle
pUUID	Pointer to the UUID of the server to look up
len	Length of the value
taskID	ID of the notified task

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

The client uses this function to look up this service on the server. The service looked up is identified by the service

parameters	descriptive
connHandle	connection handle
startHandle	starting handle
endHandle	end handle
taskID	ID of the notified task

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

When only the service handle range is known, the client can use this function to look up all feature declarations on

parameters	descriptive
connHandle	connection handle
startHandle	starting handle

endHandle	end handle
taskID	ID of the notified task

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

Clients can use this function to discover features on the server when the service handle range and feature UUID are

parameters	descriptive
connHandle	connection handle
pReq	Pointer to the request to be sent
taskID	ID of the notified task

```
bStatus_t GATT_DiscAllCharDescs( uint16_t connHandle, uint16_t startHandle, uint16_t endHandle, uint8_t
   taskId )
```

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

parameters	descriptive
connHandle	connection handle
startHandle	starting handle
endHandle	end handle
taskID	ID of the notified task

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

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

parameters	descriptive
connHandle	connection handle
pReq	Pointer to the request to be sent
taskID	ID of the notified task

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

This function can be used to read feature values from the server when the client only knows the UUID of the feature but not the handle of the feature.

parameters	descriptive
connHandle	connection handle
pReq	Pointer to the request to be sent

taskID	ID of the notified task

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

Read the server feature value, but the feature value is longer than the length that can be sent in a single read

parameters	descriptive
connHandle	connection handle
pReq	Pointer to the request to be sent
taskID	ID of the notified task

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

Read multiple feature values from the server.

parameters	descriptive
connHandle	connection handle
pReq	Pointer to the request to be sent
taskID	ID of the notified task

```
bStatus_t GATT_WriteNoRsp( uint16_t connHandle, attWriteReq_t *pReq )
```

When the client knows the feature handle it can write the feature to the server without confirming that the write was

parameters	descriptive
connHandle	connection handle
pReq	Pointer to the command to be sent

```
bStatus_t GATT_SignedWriteNoRsp(uint16_t connHandle, attWriteReq_t *pReq)
```

This function can be used to write a feature value to the server when the client knows the feature handle and the ATT confirms that it is not encrypted. Only if the Characteristic Properties authentication bit is enabled and bindings are established on both the server and the client.

parameters	descriptive
connHandle	connection handle
pReq	Pointer to the command to be sent

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

This function writes the feature value to the server when the client knows the feature handle. Only the first octet of the feature value can be written. This function returns□ whether the write process was successful.

parameters	descriptive
connHandle	connection handle
pReq	Pointer to the request to be sent
taskID	ID of the notified task

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

This function can be used when the client knows the feature value handle but the feature value length is greater than the length defined in the individual write request attribute protocol.

parameters	descriptive
connHandle	connection handle
pReq	Pointer to the request to be sent
taskID	ID of the notified task

8.4.2 return (to)回

- SUCCESS (0x00) the instruction was executed as expected.

- INVALIDPARAMETER (0x02) invalid connection handle or request field.

- MSG_BUFFER_NOT_AVAIL (0x04): HCI buffer not available. Please retry later.

- bleNotConnected (0x14) the device is not connected.

- blePending (0x17)

  - When returning回 to the client function, the server o r GATT sub-process is in progress with a pending response.

    -Returns回 acknowledgements from the client are pending while the server is functioning.

- bleTimeout (0x16) previous transaction timeout. ATT or GATT messages cannot be sent until reconnection is made.

-bleMemAllocError (0x13) a memory allocation error occurred

- bleLinkEncrypted (0x19) the link is encrypted. Do not send PDUs containing authentication signatures over encrypted links.

8.4.3 event

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

The following are common event names and the format of the event delivery message. Refer to CH58xBLE_LIB.h for details.

- ATT_ERROR_RSP:

```
typedef struct
{
         uint8_t reqOpcode; //! < Request that generated this error response
        uint16_t handle; //!    //! < Attribute handle that generated error response
        uint8_t errCode.         //! < Reason why the request has generated error resp
  onse
} 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 instruction with corresponding ATT event

ATT Response Event	GATT API Calls
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) could not read the attribute.

-ATT_ERR_WRITE_NOT_PERMITTED (0x03) unable to write attribute.

- ATT_ERR_INVALID_PDU (0x04) attribute PDU is invalid.

- ATT_ERR_INSUFFICIENT_AUTHEN (0x05) this attribute requires authentication to be read or w

-ATT_ERR_UNSUPPORTED_REQ (0x06) the attribute server did 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 ready to be written.

- ATT_ERR_ATTR_NOT_FOUND (0x0A) attribute could not be found within the given attribute handle range.

- ATT_ERR_ATTR_NOT_LONG (0x0B) could not read or write the attribute using a Read Blob request or a Prepare to Write request.

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

-ATT_ERR_INVALID_VALUE_SIZE (0x0D) attribute value length is not valid for this operation.

- ATT_ERR_UNLIKELY (0x0E) the requested attribute request encountered an unlikely error or failed to complete as requested.

-ATT_ERR_INSUFFICIENT_ENCRYPT (0x0F) this attribute requires encryption to read or write.

- ATT_ERR_UNSUPPORTED_GRP_TYPE (0x10) the attribute type is not a supported grouping attribute as defined by a higher level specification.

- ATT_ERR_INSUFFICIENT_RESOURCES (0x11) insufficient resources to complete the request.

8.5 GATTServApp API

8.5.1 directives

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

Initialize the client feature configuration table.

parameters	descriptive

connHandle	connection handle
charCfgTbl	Client Feature Configuration Table
return (to)回	not have

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

Reads the client's feature configuration.

parameters	descriptive
connHandle	connection handle
charCfgTbl	Client Feature Configuration Table
return (to)回	attribute value

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

Writes the feature configuration to the client.

parameters	descriptive
connHandle	connection handle
charCfgTbl	Client Feature Configuration Table
value	new value
return (to)回	SUCCESS FAILURE

```
bStatus_t GATTServApp_ProcessCCCWriteReq( uint16_t connHandle,
                                          gattAttribute_t *pAttr,
                                          uint8_t *pValue,
                                          uint16_t len,
                                          uint16_t offset,
                                          uint16_t validCfg );.
```

Processes client feature configuration write requests.

parameters	descriptive
connHandle	connection handle
pAttr	Pointers to properties
pvalue	Pointer to written data
len	data length
offset	Offset of the first octet of data written in
validCfg	Efficient Configuration
return (to)回	SUCCESS FAILURE

8.6 GAPBondMgr API

8.6.1 directives

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

Sets parameters for binding management.

parameters	descriptive
param	Configuration parameters
len	Write Length
pValue	Pointer to write data
return (to)回	SUCCESS INVALIDPARAMETER: invalid parameter

```
bStatus_t GAPBondMgr_GetParameter( uint16_t param, void *pValue )
```

Gets the parameters of the binding management.

parameters	descriptive
param	Configuration parameters
pValue	Points to the address where the data was read out
return (to)回	SUCCESS INVALIDPARAMETER: invalid parameter

```
bStatus_t GAPBondMgr_PasscodeRsp( uint16_t connectionHandle, uint8_t status, uint32_t passcode )
```

Responds to a password request.

parameters	descriptive
connectionHandle	connection handle
status	SUCCESS: Password available See SMP_PAIRING_FAILED_DEFINES for additional details.
passcode	Integer value password
return (to)回	SUCCESS: Binding record found and changed bleIncorrectMode: connection not found

8.6.2 Configuration parameters

Commonly used configuration parameters are shown in the following table, please refer to CH58xBLE.LIB.h for

Parameter ID	fill out or in (information on a form)	magnitude	descriptive
GAPBOND_PERI_PAIRI	readable	uint8	The pairing is done by default:

NG_MODE	writable		GAPBOND_PAIRING_MODE_WAIT_FOR_REQ
GAPBOND_PERI_DEFAU LT_PASSCODE		uint32	Default man-in-the-middle protection key, range: 0-999999, silent Think 0.
GAPBOND_PERI_MITM_ PROTECTION	readable writable	uint 8	Man-in-the-middle (MITM) protection. Default is 0. Turn off man-in-the-middle Protection.
GAPBOND_PERI_IO_CA PABILITIES	reada ble and writa ble	uint 8	I/O capability, die defaults to: GAPBOND_IO_CAP_DISPLAY_ONLY, i.e., the device is capable of only Reality.
GAPBOND_PERI_BONDI NG_ENABLED	readable writable	uint 8	If enabled, the binding is requested during the pairing process. The default is 0. Binding is not requested.

8.7 RF PHY API

8.7.1 directives

```
bStatus_t RF_RoleInit( void )
```

RF protocol stack initialization.

parameters	descriptive
return (to)回	SUCCESS: Initialization successful

```
bStatus_t RF_Config( rfConfig_t *pConfig )
```

RF Parameter Configuration.

parameters	descriptive
pConfig	Pointer to configuration parameters
return (to)回	SUCCESS

```
bStatus_t RF_Rx( uint8_t *txBuf, uint8_t txLen, uint8_t pktRxType, uint8_t pktTxType )
```

RF Accept Data Function: Configures the RF PHY to the accept state and needs to be reconfigured after receiving data.

parameters	descriptive
txBuf	Pointer to the data returned to回 after the RF receives the data in automatic mode.
txLen	Length (0-251) of data returned to回 after RF receives data in auto mode
pkRxType	Type of packet accepted (0xFF: all types of packets accepted)
pkTxType	Packet type of data that RF returns to回 after receiving data in auto mode
return (to)回	SUCCESS

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

RF Send Data function.

parameters	descriptive
txBuf	Pointer to RF send data
txLen	Data length of RF transmit data (0-251)
pkTxType	Type of packet sent
pkRxType	The data type of the data received after the RF sends the data in auto mode (0xFF: the data type of the data received by the (with type packet)
return (to)回	SUCCESS

```
bStatus_t RF_Shut( void )
```

Turn off RF to stop sending or receiving.

parameters	descriptive
return (to)回	SUCCESS

```
uint8_t RF_FrequencyHoppingTx( uint8_t resendCount )
```

RF Transmitter Enable Frequency Hopping

parameters	descriptive
resendCount	Maximum count for sending HOP_TX pdu (0: unlimited)
return (to)回	0: SUCCESS

```
uint8_t RF_FrequencyHoppingRx( uint32_t timeoutMS );
```

RF receiver on frequency hopping

parameters	descriptive
timeoutMS	Maximum time to wait to receive HOP_TX pdu (Time = n * 1ms, 0: unlimited)
return (to)回	0: SUCCESS 1: Failed 2: LLEMode error (needs to be in auto mode)

```
void RF_FrequencyHoppingShut( void )
```

Disable RF frequency hopping

8.7.2 Configuration parameters

The RF configuration parameter rfConfig_t is described below:

parameters	descriptive
LLEMode	LLE_MODE_BASIC: Basic mode, enters idle mode after sending or receiving is finished LLE_MODE_AUTO: Auto mode, automatically switches to receive mode after transmission is complete

	LLE_MODE_EX_CHANNEL: Switch to Frequency configuration band LLE_MODE_NON_RSSI: set the first byte of received data to packet type
Channel	RF communication channels (0-39)
Frequency	RF communication frequency (2400000KHz-2483500KHz) not recommended to be used more than 24 times Bit Flip with no more than 6 consecutive 0's or 1's
AccessAddress	RF Communication Address
CRCInit	CRC Initial value
RFStatusCB	RF Status回 Tuning Functions
ChannelMap	Channel map, each bit corresponds to a channel. A bit value of 1 means the channel is valid, and vice versa. Effective. Channels are incremented by bit and channel 0 corresponds to bit 0.
Resv	reservations
HeartPeriod	Heartbeat packet interval, integer multiple of 100ms
HopPeriod	Jump period (T=32n*RTC clock) default is 8
HopIndex	The frequency hopping channel interval value in the data channel selection algorithm, the default is 17
RxMaxlen	Maximum data length received in RF mode, default 251
RxMaxlen	Maximum data length to be transmitted in RF mode, default 251

8.7.3 回invoke a function

```
void RF_2G4StatusCallBack( uint8_t sta , uint8_t crc, uint8_t *rxBuf )
```

RF status回 call function, send or receive completion will enter this回 call function.

Note: You can't call the RF Receive or Send API directly in this function, you need to use the event method to

parameters	descriptive
sta	RF transceiver status
crc	Packet status checksum, each bit characterizes a different status: bit0: data CRC checksum error; bit1: Packet type error;
rxBuf	Pointer to received data
return (to)回	NULL

revised record

releases	timing	revision
V1.0	2021/3/22	Version Release
V1.1	2021/4/19	Add RF usage examples and API
V1.2	2021/5/28	Content errata, add RF description
V1.3	2021/7/3	1. Name Adjustment; 2. Modify the preamble and the description of the development platform; 3. Figure 3.1 Errata.
V1.4	2021/11/2	1. A new section 7.5.2.1 has been added to describe the RF frequency hopping function; 2. Added RF frequency hopping API description; 3. Optimize code display format.
V1.5	2022/5/6	1. Erratum: Section 7.2 Code citation error; 2. Erratum: Section 5.3.2 was incorrectly numbered; 3. Section 3.3 Refinement of TMOS Task Execution Structure.
V1.6	2022/8/19	1. Adjustment of the title of chapter IV; 2. Added application routine descriptions; 3. A new section 5.5.2 has been added to describe GATT services and protocols; 4. Added TMOS API description; 5. Optimize content descriptions; 6. Optimization Figure 4.2; 7. The sample code is synchronized with the routine; 8. Content errata.
V1.7	2022/9/30	1. Erratum: Section 8.7.3 crc description error 2. Header Adjustment
V1.8	2024/1/3	1. Errata: Figure 1.1 Errata 2. Erratum: Textual description

Imprint and Disclaimer