
WCH Bluetooth Low Energy Software Development Reference Manual

V1.7
September 30, 2022

Content

Content	1
Preface	1
1. Overview	2
1.1 Introduction	2
1.2 Basic Introduction to Bluetooth Low Energy Protocol Stack	2
2. Development Platform	4
2.1 Overview	4
2.2 Configuration	4
2.3 Software Overview	4
3. Task Management Operating System (TMOS)	5
3.1 Overview	5
3.2 Task Initialization	5
3.3 Task Events and Event Execution	5
3.4 Memory Management	7
3.5 TMOS Data Transfer	7
4. Brief Analysis of Application Routines	9
4.1 Overview	9
4.2 Project Preview	9
4.3 Start with main ()	10
4.4 Application Initialization	10
4.4.1 Bluetooth Low Energy Library Initialization	10
4.4.2 HAL Layer Initialization	10
4.4.3 Bluetooth Low Energy Slave Initialization	11
4.5 Event Processing	14
4.5.1 Timing Event	15
4.5.2 TMOS Message Transfer	15
4.6 Call Back	15
5. Bluetooth Low Energy Protocol Stack	16
5.1 Overview	16
5.2 Generic Access Profile (GAP)	16

5.2.1 Overview	16
5.2.2 GAP Abstraction Layer	18
5.2.3 GAP Layer Configuration.....	19
5.3 GAPRole Task.....	19
5.3.1 Peripheral Role.....	19
5.3.2 Central Role.....	21
5.4 GAP Bond Management.....	22
5.4.1 Close Pairing.....	23
5.4.2 Pair Directly but not Bind.....	23
5.4.3 Binding via MITM Pairing.....	24
5.5 Generic Attribute Profile (GATT)	24
5.5.1 GATT Characteristics and Properties	25
5.5.2 GATT Serve and Protocol.....	25
5.5.3 GATT Client Abstraction Layer.....	26
5.5.4 GATT Server Abstraction Layer.....	28
5.6 Logical Link Control and Adaptation Protocol.....	34
5.7 Host and Controller Interface.....	34
6. Create a BLE Application.....	36
6.1 Overview	36
6.2 Configure Bluetooth Protocol Stack.....	36
6.3 Define Bluetooth Low Energy Behavior.....	36
6.4 Define Application Tasks.....	36
6.5 Application Configuration File.....	36
6.6 Limit Application Processing during Bluetooth Low Energy Operation.....	36
6.7 Interrupt	36
7. Create a Simple RF Application	38
7.1 Overview	38
7.2 Configure Protocol Stack	38
7.3 Define Application Tasks.....	38
7.4 Application Configuration File.....	38
7.5 RF Communication	38
7.5.1 Basic Mode	38

7.5.2 Auto Mode	39
8. API	41
8.1 TMOS API	41
8.1.1 Command	41
8.2 GAP API	46
8.2.1 Command	46
8.2.2 Configure Parameter	46
8.2.3 Event	47
8.3 GAPRole API	49
8.3.1 GAPRole Common Role API	49
8.3.2 GAPRolePeripheral Role API	51
8.3.3 GAPRole Central Role API	53
8.4 GAPRole API	56
8.4.1 Command	56
8.4.2 Return	61
8.4.3 Event	61
8.4.4 GATT Instruction and Corresponding ATT Events	63
8.4.5 ATT_ERROR_RSP Error Code	64
8.5 GATTServApp API	64
8.5.1 Command	64
8.6 GAPBondMgr API	66
8.6.1 Command	66
8.6.2 Configuration Parameters	66
8.7 RF PHY API	67
8.7.1 Command	67
8.7.2 Configuration Parameters	69
8.7.3 Callback Function	69
Revision History	71
Imprint and Disclaimer	72

Preface

This manual provides a brief introduction to the software development of WCH Bluetooth Low Energy. Including software development platform, basic framework of software development and Bluetooth Low Energy protocol stack, etc. For the convenience of understanding, this manual uses the CH58x chip as an example. Software development of other Bluetooth Low Energy chips of our company can also refer to this manual.

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

1. Overview

1.1 Introduction

Bluetooth supports 2 wireless technologies since version 4.0:

- Bluetooth Basic Rate/Enhanced Data Rate (Often called BR/EDR Classic Bluetooth)
- Bluetooth Low Energy

The Bluetooth Low Energy protocol was created to transmit very small data packets at a time and therefore consumes significantly less power than classic Bluetooth.

Devices that can support classic Bluetooth and Bluetooth Low Energy are called dual-mode devices, such as mobile phones. Devices that only support Bluetooth Low Energy are called single-mode devices. These devices are primarily intended for low-power applications, such as those powered by coin cell batteries.

1.2 Basic Introduction to Bluetooth Low Energy Protocol Stack

The results of the Bluetooth Low Energy protocol stack are shown in Figure 1-1.

Figure 1-1

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

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

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

Physical layer is in the 2.4GHz channel and uses GFSK-Gauss frequency Shift Keying technology for modulation. There are three implementation solutions for the physical layer of BLE5.0, namely 1Mbps uncoded physical layer, 2Mbps uncoded physical layer and 1Mbps coded physical layer. Among them, the 1Mbps uncoded physical layer is compatible with the physical layer of the BLE4.0 series protocol, and the other two physical layers extend the communication rate and communication distance respectively.

LinkLayer controls the device to be in one of the five states of standby, advertising, scanning, initiating and connected. Around these states, BLE devices can perform operations such as broadcasting and connecting, and the

link layer defines the packet formats, timing specifications and interface protocols in various states.

Generic Access Profile is the external interface layer of the internal functions of the BLE device. It stipulates three aspects: GAP roles, models and procedures, and security issues. Mainly manages the broadcast, connection and device binding of Bluetooth devices.

Broadcaster - A device that is always broadcasting and cannot be connected

Observer - A device that can scan for broadcasting devices, but cannot initiate a connection

Slave - A broadcasting device that can be connected, can act 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

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

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

Attribute Protocol defines the concept of attribute entities, including UUID, handles and attribute values, and stipulates the operation methods and details of attributes such as reading, writing, and notification.

Generic Attribute Profile defines the service framework and protocol structure using ATT. The communication of application data between two devices is implemented through the GATT layer of the protocol stack.

GATT server - A device that provides data services to GATT clients

GATT client - A device that reads and writes application data from a GATT server

2. Development Platform

2.1 Overview

CH58x is a 32-bit RISC-V microcontroller integrating BLE wireless communication. The chip integrates a 2Mbps Bluetooth Low Energy communication module, two full-speed USB host and device controller transceivers, 2 SPI, RTC and other rich peripheral resources. This manual uses the CH58x development platform as an example. Other Bluetooth Low Energy chips of our company can also refer to this manual.

2.2 Configuration

CH58x is a true single-chip solution, controller, host, profile and application are all implemented on CH58x. See 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 configuration files:

- Peripheralrole
- Centralrole
- Broadcasterrole
- Observerrole

Some GATT configuration files and applications are also provided.

Please refer to the CH58xEVT software package for details.

3. Task Management Operating System (TMOS)

3.1 Overview

The Bluetooth Low Energy protocol stack and applications are based on TMOS. TMOS is a control loop through which the event execution method can be set. TMOS serves as the core of scheduling, and the BLE protocol stack, profile definition, and all applications are implemented around it. TMOS is not an operating system in the traditional sense, but a system resource management mechanism with the core of realizing multi-tasking.

For a task, the unique task ID, task initialization and executable events under the task are all indispensable.

3.2 Task Initialization

First of all, in order to ensure that TMOS keeps running, it is necessary to loop through `TMOS_SystemProcess()` at the end of `main()`.

And initialize the task need to call `tmTaskID TMOS_ProcessEventRegister(pTaskEventHandlerFneventCb)` function to register the event callback function into TMOS and generate a unique 8-bit task ID. different tasks are initialized with increasing task IDs, and the smaller the task ID, the higher the priority of the task. Stack tasks must have the highest priority.

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

3.3 Task Events and Event Execution

TMOS is scheduled through polling. The system clock is generally derived from RTC and the unit is 625μs. Users add customized events to the task list of TMOS by registering tasks, and then TMOS schedules and runs them. After the task initialization is completed, TMOS will poll task events in a loop. The event flags are stored in 16-bit variables, each of which corresponds to a unique event in the same task. If the flag bit is 1, it means that the event corresponding to the bit is running, and if it is 0, it means it is not running. Each Task user can customize up to 15 events. 0x8000 is the `SYS_EVENT_MSG` event reserved by the system, which is the system messaging event and 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 whether there are events that need to be executed according to the priority of the task. System tasks, such as automatic reception of responses after sending in 2.4G automatic mode and related transactions in Bluetooth, have the highest priority. After the system task execution is completed, if there is a user event, the corresponding callback function will be executed. When a cycle ends, if there is still time left, the system will enter idle or sleep mode.

Figure 3.1 TMOS task Management Diagram

In order to illustrate the general code format of TMOS processing events, take the HAL_TEST_EVENT event of the HAL layer as an example. Replacing HAL_TEST_EVENT here with other events is also applicable. If you want to define a TEST event in the HAL layer, you can add the event HAL_TEST_EVENT in the task callback function after the task initialization of the HAL layer is completed. Its basic format is as follows:

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

After the event is executed, the corresponding 16-bit event variable needs to be returned to clear the event, so as 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)` to execute the corresponding event immediately, and the event is 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 is that after setting the task ID and event flag of the task that you want to execute, you need to add a third parameter. The function is similar to

tmos_set_event, except that after setting the task ID and event flag of the task2 you want to execute, you also need to add a third parameter: The timeout of the event execution. That is, the event will be executed once after the timeout is reached. If you define the next time for the task to be executed in the event, you can loop through the execution of a certain event.

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

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

Figure 3.2 Timing task

3.4 Memory Management

TMOS uses a separate piece of memory, and users can customize the memory address and size. The task event management in TMOS all uses this memory. You can analyze the memory usage by turning on the Bluetooth binding function and encryption function. Since the protocol stack of Bluetooth Low Energy also uses this memory, it needs to be tested under the maximum expected working conditions.

3.5 TMOS Data Transfer

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

To send a piece of data, follow these steps:

1. Use the tmos_msg_allocate () function to apply for memory for the sent data. If the application is successful, the

memory address will be returned. If it fails, NULL will be returned.

2. Copy the data to memory.
3. Call the `tmos_msg_send()` function to send the data pointer to the specified task.

```

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

```

After the function is completed, `SYS_EVENT_MSG` is set to valid, at which point the system executes the `SYS_EVENT_MSG` event, and in practice retrieves the data by calling the `tmos_msg_receive()` function. After the data has been processed, it must be retrieved using the `tmos_msg_deallocate()` function to free the memory. Refer to the example program for details.

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

```

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

```

Query to the `KEY_CHANGE` event:

```

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

```

4. Brief Analysis of Application Routines

4.1 Overview

The Bluetooth Low Energy EVT routine includes a simple BLE project: Peripheral. By burning this project into the CH58x chip, a simple Bluetooth Low Energy slave device can be realized.

4.2 Project Preview

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

Figure 4.1 Project files

Files can be divided into the following categories:

1. APP – Source files and header files related to the application can be placed here, and the main function of the routine is also here.
2. HAL – This folder contains the source code and header files of the HAL layer, which is the interaction layer between the Bluetooth protocol stack and the chip hardware driver.
3. LIB –The protocol stack library file of Bluetooth Low Energy.
4. LD – Linker script.
5. Profile – This file contains the source code and header files of the GAP role profile, GAP security profile and GATT profile, and also contains the header files required by the GATT service. Please refer to Section 5 for details.
6. RVMSIS–Source code and header files accessed by the RISC-V core.
7. Startup–Startup file.
8. StdPeriphDriver - Includes the underlying driver files for the peripherals of the chip.
9. obj - Compiler-generated files, including map files and hex files.

4.3 Start with main ()

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

```

1. int main( void )
2. {
3.     #if (defined (DCDC_ENABLE)) && (DCDC_ENABLE == TRUE)
4.         PWR_DCDCfg( ENABLE );
5.     #endif
6.     SetSysClock( CLK_SOURCE_PLL_60MHz );    //Set system clock
7.     GPIOA_ModeCfg( GPIO_Pin_All, GPIO_ModeIN_PU );    //Configure IO port
8.     GPIOB_ModeCfg( GPIO_Pin_All, GPIO_ModeIN_PU );
9.     #ifdef DEBUG
10.        GPIOA_SetBits(bTXD1);    //Configure serial port
11.        GPIOA_ModeCfg(bTXD1, GPIO_ModeOut_PP_5mA);
12.        UART1_DefInit( );    //Initialize serial port
13.    #endif
14.    PRINT("%s\n",VER_LIB);
15.    CH58X_BLEInit( );    //Initialize Bluetooth library
16.    HAL_Init( );
17.    GAPRole_PeripheralInit( );
18.    Peripheral_Init( );
19.    while(1){
20.        TMOS_SystemProcess( );    //main loop
21.    }
22. }
```

4.4 Application Initialization

4.4.1 Bluetooth Low Energy Library Initialization

Bluetooth Low Energy library initialization function CH58X_BLEInit() configures the memory, clock, transmit power and other parameters of the library 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

Register HAL layer tasks and initialize hardware parameters, such as RTC clock, sleep wake-up, RF calibration, etc.

```

1. void HAL_Init()
```









```
2. {
3.     halTaskID = TMOS_ProcessEventRegister( HAL_ProcessEvent );
4.     HAL_TimeInit();
5. #if (defined HAL_SLEEP) && (HAL_SLEEP == TRUE)
6.     HAL_SleepInit();
7. #endif
8. #if (defined HAL_LED) && (HAL_LED == TRUE)
9.     HAL_LedInit( );
10. #endif
11. #if (defined HAL_KEY) && (HAL_KEY == TRUE)
12.     HAL_KeyInit( );
13. #endif
14. #if ( defined BLE_CALIBRATION_ENABLE ) && ( BLE_CALIBRATION_ENABLE == TRUE )
15.     tmos_start_task( halTaskID, HAL_REG_INIT_EVENT, MS1_TO_SYSTEM_TIME( BLE_CA
        LIBRATION_PERIOD ) );    // Add calibration tasks, single calibration takes less than 10ms
16. #endif
17.     tmos_start_task( halTaskID, HAL_TEST_EVENT, 1000 );    // Add a test task
18. }
```












4.4.3 Bluetooth Low Energy Slave Initialization

This process consists of two parts:

1. Initialization of the GAP role, this process is completed by the Bluetooth Low Energy library;
2. Initialization of the Bluetooth Low Energy slave application, including registration of slave tasks, parameter configuration (such as broadcast parameters, connection parameters, binding parameters, etc.), registration of GATT layer services, and registration of callback functions. See section 5.5.3.2 for details.

Figure 4.2 shows the complete attribute table of the routine Peripheral, which can be used as a reference during Bluetooth Low Energy communication. Please refer to Chapter 5 for details.

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	

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	

4.5 Event Processing

After the initialization is completed, the event is turned on (That is, a bit is set in the event), and the application task will process the event in `Peripheral_ProcessEvent`. The following subsections will describe the possible sources of the event.

4.5.1 Timing 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 `SBP_PERIODIC_EVT` after `SBP_PERIODIC_EVT` is processed (Default 5000ms). The periodic event occurs every 5 seconds and the `performPeriodicTask()` function is called to realize the function.

```
1. if(events & SBP_PERIODIC_EVT)
2. {
3.     // Restart timer
4.     if(SBP_PERIODIC_EVT_PERIOD)
5.     {
6.         tmos_start_task(Peripheral_TaskID, SBP_PERIODIC_EVT,
7.             SBP_PERIODIC_EVT_PERIOD);
8.     }
9.     // Perform periodic application task
10.    performPeriodicTask();
11.    return (events ^ SBP_PERIODIC_EVT);
12. }
```

This periodic event handling is just an example, but highlights how to perform custom actions within a periodic task. Before processing the periodic event, a new TMOS timer will be started to set the next periodic task.

4.5.2 TMOS Message Transfer

TMOS messages may originate from various layers of the BLE protocol stack. For information about this part, see 3.5 TMOS Data Transfer.

4.6 Call Back

Application code can be written either in event handling snippets or in callback functions such as `simpleProfileChangeCB()` and `peripheralStateNotificationCB()`. Communication between the stack and the application is achieved by callback functions, such as `simpleProfileChangeCB()` which notifies the application of a change in a feature value.

5. Bluetooth Low Energy Protocol Stack

5.1 Overview

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

5.2 Generic Access Profile (GAP)

5.2.1 Overview

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

Figure 5.1 GAP state

Among them: Standby: the idle state in which the Bluetooth Low Energy protocol stack is not enabled;

Advertiser: The device uses specific data to broadcast, and the broadcast can include the name, address and other data of the device. Broadcasting indicates that this device can be connected.

Scanner: After receiving the broadcast data, the scanning device sends a scan request packet to the broadcaster, and the broadcaster will return to scan the corresponding packet. The scanner will read the broadcaster's information and determine whether it can connect. This procedure describes the process of discovering a device.

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

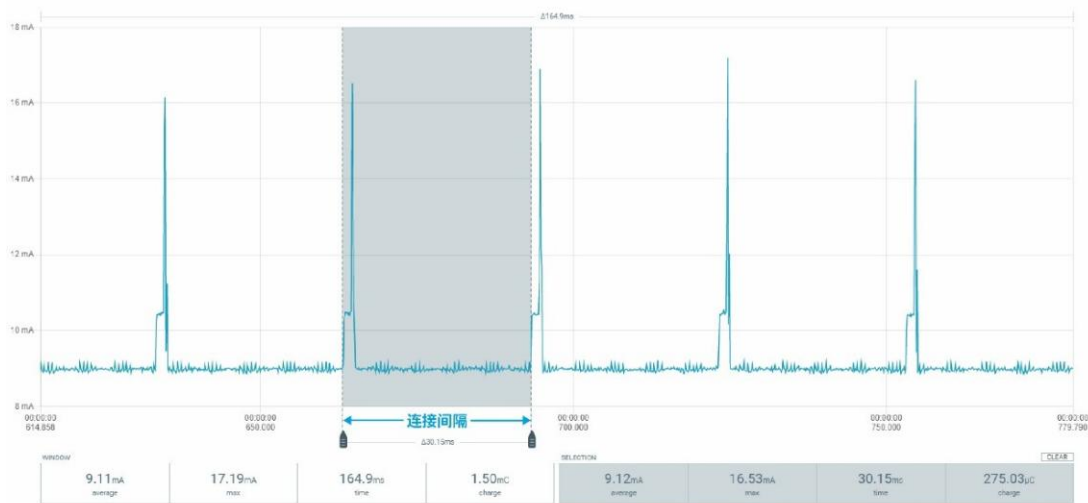
Master or Slave: If the device is an advertiser before connecting, it is a slave when connecting; if the device is an initiator before connecting, it is a master after connecting.

5.2.1.1 Connection Parameter

This section describes the connection parameters when establishing a connection. These connection parameters can be modified by both the master and the slave.

- Connection Interval – Bluetooth low energy uses a frequency hopping scheme, and the device sends and receives data on a specific channel at a specific time. A data transmission and reception between two devices becomes a connection event. The connection interval is the time between two connection events, and its time unit is 1.25ms. The connection interval ranges from 7.5ms to 4s.

Figure 5.2 Connection event and connection intervals



Different applications may require different connection intervals. Smaller connection intervals will reduce data response time and increase power consumption accordingly.

- **SlaveLatency** – This parameter allows the slave to skip some connection events. If the device has no data to send, the slave delay can skip the connection event and stop the radio during the connection event, thereby reducing power consumption. The value of the slave device delay indicates the maximum number of events that can be skipped, and its range is 0~499. However, it is necessary to ensure that the effective connection interval is less than 16s. For the effective connection interval, please refer to 5.2.1.2

Figure 5.3 Slave device delay

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

5.2.1.2 Valid Connection Interval

With slave latency enabled and no data transferred, the valid connection interval is the time between two connection

events. If the slave device delay is not enabled or its value is 0, the effective connection interval is the configured connection interval.

The calculation formula is as follows:

$$\text{Valid connection interval} = \text{Connection interval} \times (1 + \text{Slave device delay})$$

When

Connection interval: 80 (100ms)

Slave device delay: 4

Valid connection interval: $(100\text{ms}) \times (1 + 4) = 500\text{ms}$

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

5.2.1.3 Connection Parameter

Reasonable connection parameters help optimize the power consumption and performance of Bluetooth Low Energy.

The following summarizes the trade-offs in connection parameter settings:

Reducing the connection interval will:

- Increase the power consumption of the host and slave
- Increase the throughput between the two devices
- Reduce the time required for data to and from the two devices

Increasing the connection interval will:

- Reduce the power consumption of the host and slave
- Reduce the throughput between the two devices
- Increase the time required for data to and from the two devices

Reducing slave latency or setting it to 0 will:

- Increase the power consumption of the slave
- Reduce the time required for the master to send data to the slave

Increasing the slave device delay will:

- Reduce the power consumption of the slave when no data needs to be sent to the master
- Increase the time required 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 depending on the application. The slave can send a connection parameter update request to the master 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
- Slave delay
- Supervision timeout

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

5.2.1.5 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 by terminating the connection before the two devices disconnect.

5.2.2 GAP Abstraction Layer

Applications can implement responsive BLE functions, such as broadcast and connection, by calling the API

functions of the GAP layer.

Figure 5.4 GAP abstraction layer



5.2.3 GAP Layer Configuration

Most functions of the GAP layer are implemented in the library, and users can find the corresponding function declarations in CH58xBLE_LIB.h.

Section 8.1 defines the GAP API, which can be used to set and detect parameters such as broadcast interval, scan interval, etc. via `GAPRole_SetParameter()` and `GAPRole_GetParameter()`. The GAP layer configuration example is as follows:

```

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

5.3 GAPRole Task

As described in Section 4.4, GAPRole is a separate task (`GAPRole_PeripheralInit`), and most of the GAPRole code runs in the Bluetooth library, thus simplifying the application layer program. This task is started and configured by the application during initialization. And there is a callback, the application can register the callback function with the GAPRole task.

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

Broadcaster - Only broadcasts and cannot be connected

Observer - Only scans broadcasts but cannot establish connections

Peripheral - Can broadcast and can establish connections at the link layer as a slave.

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

The following describes the two roles of Peripheral and Central.

5.3.1 Peripheral Role

The general steps for initializing peripheral devices are as follows:

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

```

1. // Setup the GAP Peripheral Role Profile
2. {
3.     uint8_t initial_advertising_enable = TRUE;
4.     uint16_t desired_min_interval = DEFAULT_DESIRED_MIN_CONN_INTERVAL;
5.     uint16_t desired_max_interval = DEFAULT_DESIRED_MAX_CONN_INTERVAL;
6.
7. // Set the GAP Role Parameters
8.     GAPRole_SetParameter( GAPROLE_ADVERT_ENABLED, sizeof( uint8_t ),
&initial_advertising_enable );
9.     GAPRole_SetParameter( GAPROLE_SCAN_RSP_DATA, sizeof ( scanRspData ),
scanRspData );
10.    GAPRole_SetParameter( GAPROLE_ADVERT_DATA, sizeof( advertData ),
advertData );
11.    GAPRole_SetParameter( GAPROLE_MIN_CONN_INTERVAL, sizeof( uint16_t ),
&desired_min_interval );
12.    GAPRole_SetParameter( GAPROLE_MAX_CONN_INTERVAL, sizeof( uint16_t ),
&desired_max_interval );
13. }
14.
15. // Set the GAP Characteristics
16. GGS_SetParameter( GGS_DEVICE_NAME_ATT, GAP_DEVICE_NAME_LEN,
attDeviceName);
17.
18. // Set advertising interval
19. {
20. uint16_t advInt = DEFAULT_ADVERTISING_INTERVAL;
21.
22. GAP_SetParamValue( TGAP_DISC_ADV_INT_MIN, advInt );
23. GAP_SetParamValue( TGAP_DISC_ADV_INT_MAX, advInt );
24. }

```

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

```

1. if ( events & SBP_START_DEVICE_EVT ){
2. // Start the Device
3.     GAPRole_PeripheralStartDevice( Peripheral_TaskID,
&Peripheral_BondMgrCBs,&Peripheral_PeripheralCBs );

```



```

4.     return ( events ^ SBP_START_DEVICE_EVT );
5. }

```

3. Transmit GAPRole commands from the application layer

The application layer updates connection parameters.

```

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

```

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

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

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

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

```

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

```

5.3.2 Central Role

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

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

```

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

```

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

```

1. if ( events & START_DEVICE_EVT )
2. {

```

```

3.      // Start the Device
4.      GAPRole_CentralStartDevice( centralTaskId, &centralBondCB, &centralRoleCB
    );
5.      return ( events    START_DEVICE_EVT );
6.  }

```

3. Transmit GAPRole commands from the application layer

The application layer calls application functions and transmits GAP commands.

```

1.  GAPRole_CentralStartDiscovery( DEFAULT_DISCOVERY_MODE,
2.                                DEFAULT_DISCOVERY_ACTIVE_SCAN,
3.                                DEFAULT_DISCOVERY_WHITE_LIST );

```

The GAP layer sends commands to the Bluetooth protocol stack. The protocol stack receives the commands, performs scanning operations, and returns the corresponding status.

4. GAPRole task delivers events related to the protocol stack and GAP to the application layer.

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

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

```

1.  static void centralEventCB( gapRoleEvent_t *pEvent )
2.  {
3.      switch ( pEvent->gap.opcode )
4.      {
5.          ...
6.          case GAP_DEVICE_DISCOVERY_EVENT:
7.          {
8.              uint8_t i;
9.              // See if peer device has been discovered
10.             for ( i = 0; i < centralScanRes; i++ )
11.             {
12.                 if (tmos_memcmp( PeerAddrDef, centralDevList[i].addr, B_ADDR_LEN))
13.                     break;
14.             }
15.             ...

```

5.4 GAP Bond Management

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

Table 5.1 GAP binding management terminology

Term	Description
Pairing	The process of key exchange
Encryption	Data is encrypted or re-encrypted after pairing
Authentication	The matching process is done under the protection of a MITM (MITM: Manin the Middle).
Bonding	Store the encryption key in non-volatile memory for the next encryption sequence
Authorization	In addition to authentication, additional application-level key exchange
OOB	Keys are not exchanged wirelessly, but through other sources such as serial ports or NFC. This also provides MITM protection.
MITM	Man-in-the-middle protection. This prevents the key from being eavesdropped on the wireless transmission to break the encryption.
JustWorks	No MITM matching method.

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

1. Key pairing (including the following two methods).
 - A. JustWorks, sending keys wirelessly
 - B. MITM, sending the key through a man-in-the-middle
2. Encrypting the connection through the key.
3. Bind the key and store the key.
4. When connecting again, use the stored key to encrypt the connection.

5.4.1 Close Pairing

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

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

5.4.2 Pair Directly but not Bind

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

It should be noted that to enable the pairing function, you also need to configure the IO function of the device, that is, whether the device supports display output and keyboard input. If the device does not actually support keyboard input, but is configured to enter a password through the device, pairing cannot be established.

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

5.4.3 Binding via MITM Pairing

```
1. uint32_t passkey = 0; // passkey "000000"
2. uint8_t pairMode = GAPBOND_PAIRING_MODE_WAIT_FOR_REQ;
3. uint8_t mitm = TRUE;
4. uint8_t bonding = TRUE;
5. uint8_t ioCap = GAPBOND_IO_CAP_DISPLAY_ONLY;
6. GAPBondMgr_SetParameter( GAPBOND_PERI_DEFAULT_PASSCODE, sizeof ( uint32_t ),
   &passkey );
7. GAPBondMgr_SetParameter( GAPBOND_PERI_PAIRING_MODE, sizeof ( uint8_t ), &pairMode );
8. GAPBondMgr_SetParameter( GAPBOND_PERI_MITM_PROTECTION, sizeof ( uint8_t ), &mitm );
9. GAPBondMgr_SetParameter( GAPBOND_PERI_IO_CAPABILITIES, sizeof ( uint8_t ), &ioCap );
10. GAPBondMgr_SetParameter( GAPBOND_PERI_BONDING_ENABLED, sizeof ( uint8_t ), &bonding );
```

Use a middleman for pairing and binding, generating a key via a 6-digit password.

5.5 Generic Attribute Profile (GATT)

The GATT layer provides applications with data communication between two connected devices. Data is transferred and stored in the form of characteristics. In GATT, when two devices are connected, they will play one of two roles:

- GATT Server – This device provides GATT clients to read or write signature databases.
- GATT Client – This device reads and writes data from GATT servers.

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

Figure 5.5 GATT Server and client

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

5.5.1 GATT Characteristics and Properties

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 the attributes that need to be sent to the GATT server (notified), or sent to the GATT server with the expectation of a response (indicated).
- CharacteristicUserDescription: An ASCII string describing the value of the characteristic.

These attributes are stored in the GATT server's attribute table, and the following characteristics are associated with each attribute.

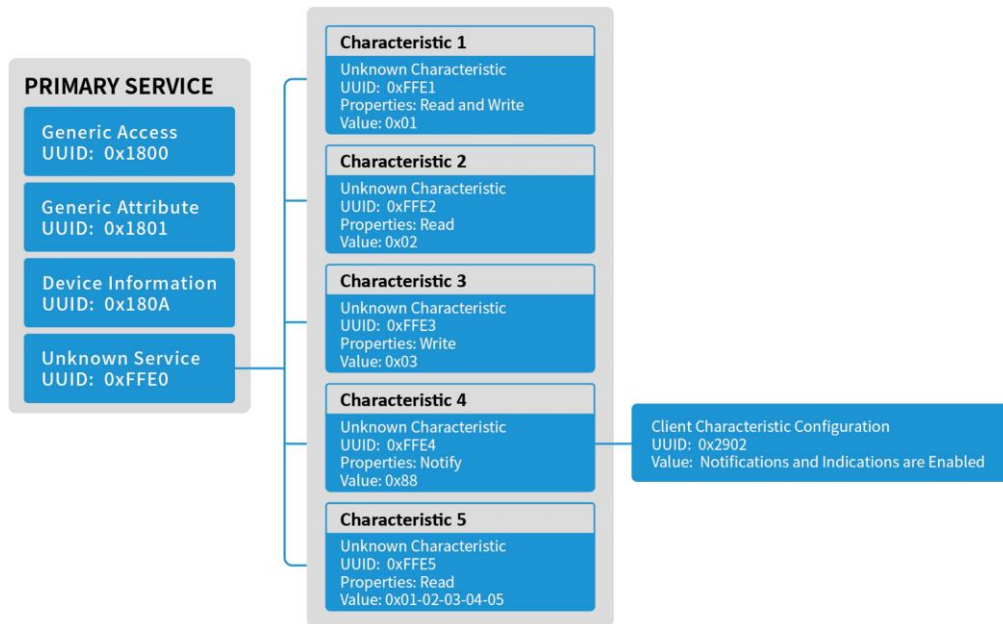
- Handle - The index of the attribute in the table, each attribute has a unique handle.
- Type - This attribute indicates what the attribute represents and is called a Universally Unique Identifier (UUID). Some UUIDs are defined by BluetoothSIG, others can be customized by the user.
- Permissions - Used to restrict how a GATT client can access the value of this property.
- pValue - A pointer to the value of the attribute, the length of which cannot be changed after initialization. The maximum size is 512 bytes.

5.5.2 GATT Serve and Protocol

GATT services are collections of features.

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

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

The following are some of the relevant attributes:

- 0x02: allow reading of feature values
- 0x04: allow writing feature values without response
- 0x08: permission to write feature values (With response)
- 0x10: permission for feature value notification (Without acknowledgement)
- 0x20: permission for feature value notification (With acknowledgement)

5.5.3 GATT Client Abstraction Layer

GATT clients do not have attribute tables because clients receive information rather than provide information. Most interfaces to the GATT layer come directly from applications.

Figure 5.6 GATT client abstraction layer

5.5.3.1 GATT Layer Application

This section describes how to use the GATT client directly in your application. The corresponding source code can be found in the routine Central.

1. Initialize the GATT client.

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

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

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

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

```
1. bStatus_t GATT_WriteCharValue( uint16_t connHandle, attWriteReq_t *pReq, uint8_t taskId )
```

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

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

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

The application layer task queries the GATT message:

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

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

```
1. static void centralProcessGATTMsg( gattMsgEvent_t *pMsg )
```

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

Clear the message after application processing is complete:

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

5.5.4 GATT Server Abstraction Layer

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

Figure 5.7 GATT Server abstraction layer

The specifications for using GATT are as follows:

1. Create a GATT Profile to configure the GATTServApp module.
2. Use the API interfaces 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 the attribute table of the application. Various configuration files use this module to add their characteristic values to the attribute table. Its functions include: finding specific attributes, reading the client's characteristic values, and modifying the client's characteristic values. Please refer to the API chapter for details.

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

Figure 5.8 Property table initialization

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

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

5.5.4.2 Configuration File Schema

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 Create Attribute Table

Each service must define a fixed-size attribute table for passing to the GATT layer.

In the Peripheral project, the definitions are as follows:

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

The format of each attribute is as follows:

```

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

```

Individual elements in attributes:

- type - The UUID associated with the attribute.

```

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

```

Where 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 - Configure whether the GATT client device can access the value of the attribute. The configurable permissions are as follows:
 - GATT_PERMIT_READ // Readable
 - GATT_PERMIT_WRITE // Writable
 - GATT_PERMIT_AUTHEN_READ // Requires authentication to read
 - GATT_PERMIT_AUTHEN_WRITE // Identity verification required
 - GATT_PERMIT_AUTHOR_READ // Requires authorization to read
 - GATT_PERMIT_ENCRYPT_READ // Need to encrypt reading
 - GATT_PERMIT_ENCRYPT_WRITE // Need to encrypt writing
- Handle - Handles assigned by GATTServApp. Handles are automatically assigned in order.
- pValue - Pointer to the property value. Its length cannot be changed after initialization. Maximum size 512 bytes.

Next create the attribute table in the Peripheral project:

First create the service attributes:

```

1. // Simple Profile Service
2. {
3.     { ATT_BT_UUID_SIZE, primaryServiceUUID }, /* type */

```

```

4.         GATT_PERMIT_READ,                /* permissions */
5.         0,                                /* handle */
6.         (uint8_t *)&simpleProfileService    /* pValue */
7.     },

```

This attribute is the primary service UUID (0x2800) 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 to 0xFFE0.

```

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

```

Then create the signature's declaration, value, user description, and client signature configuration, which are described in Section 5.5.1.

```

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

```

The type of Characteristic Declaration needs to be set to the value of the BluetoothSIG-defined characteristic UUID (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.

```

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

```

In the 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:

```

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

```

This type must be set to the Client Feature Configuration UUID defined by the Bluetooth 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.

```

1. 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 some user-defined services, such as SimpleProfile_AddService in the Peripheral project. AddService(), for example, these functions perform the following actions:

First you need to define the Client Characteristic Configuration (CCC).

```

1. static gattCharCfg_t simpleProfileChar4Config[4];

```

The CCC is then initialized.

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.

```

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

```

Finally the profile is registered through GATTServApp.

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

```

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

```

5.5.4.2.3 Register Application Callback Function

In the Peripheral project, the GATTProfile invokes an application callback whenever a GATT client writes a feature value. To use the callback function, you first need to set up the callback function at initialization time.

```

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

```

The callback function is as follows:

```

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

```

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

```

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

```

5.5.4.2.4 Read and Write Callback Functions

When the configuration file is read or written, a corresponding callback function is required, and its registration method is consistent with the application callback function. For details, please refer to the Peripheral project.

5.5.4.2.5 Obtaining and Setting Configuration Files

The configuration file contains read and write feature functions. Figure 5.9 describes the logic of the application setting configuration file parameters.

Figure 5.9 Get and set configuration file parameters

The application code is as follows:

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

5.6 Logical Link Control and Adaptation Protocol

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

5.7 Host and Controller Interface

HCI (Host Controller Interface), which connects the host and the controller, converts the host's operations into HCI instructions and passes them to the controller. There are four types of HCI supported by BLECoreSpec: UART, USB, SDIO, and 3-Wire UART. For a single Bluetooth chip with a full protocol stack, you only need to call the API interface function. At this time, HCI is a function call and callback; for products with only a controller, the main

control chip is used to operate the BLE chip, and the BLE chip is used as a plug-in. The chip is connected to the main control chip. At this time, the main control chip only needs to interact with the BLE chip through standard HCI commands (usually UART).

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

6. Create a BLE Application

6.1 Overview

After reading the previous chapters, you should understand how to implement a Bluetooth Low Energy application. This chapter explains how to get started writing Bluetooth Low Energy applications, along with some considerations.

6.2 Configure Bluetooth Protocol Stack

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

- Central
- Peripheral
- Broadcaster
- Observer

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

6.3 Define Bluetooth Low Energy Behavior

Use the API of the Bluetooth low energy protocol stack to define system behavior, such as adding configuration files, adding GATT database, configuring security mode, etc. See Chapter 5 for details.

6.4 Define Application Tasks

Make sure your application includes callbacks to the protocol stack and event handlers from TMOS. You can refer to adding other tasks introduced in Chapter 3.

6.5 Application Configuration File

Configure the DCDC enable, the RTC clock, the sleep function, the MAC address, the size of the RAM used by the Bluetooth Low Energy stack, and so on in config.h. See config.h for details.

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, stability, etc. You need to add a buffer to the wakeup time to improve stability.

6.6 Limit Application Processing during Bluetooth Low Energy Operation

Due to the time dependence of the Bluetooth Low Energy protocol, the controller must process each connection event or broadcast event before it arrives. If not processed in time, it will result in retransmission or disconnection. And TOMS is not multi-threaded, so when Bluetooth Low Energy has transactions, other tasks must be stopped for the controller to process. Therefore, make sure not to occupy a large number of events in the application. If complex processing is required, please refer to Section 3.3 to split it.

6.7 Interrupt

During the operation of Bluetooth Low Energy, the time needs to be calculated through the RTC timer, so during this period, do not disable global interrupts, and the time occupied by a single interrupt service program should not

be too long, otherwise interrupting the operation of Bluetooth Low Energy for a long time will cause Disconnect.

7. Create a Simple RF Application

7.1 Overview

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 RF applications do not establish the protocol of BLE.

7.2 Configure Protocol Stack

First initialize the Bluetooth library:

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

Then configure the role of this device as RF Role:

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

7.3 Define Application Tasks

Register RF tasks, initialize RF functions, and register RF callback functions:

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

7.4 Application Configuration File

Configure DCDC enable, RTC clock, sleep function, MAC address, RAM size used by the Bluetooth low energy protocol stack, etc. in the config.h file. See the config.h file for details.

It should be noted 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 and other factors. You need to add a buffer of time to the wake-up time to improve stability.

7.5 RF Communication

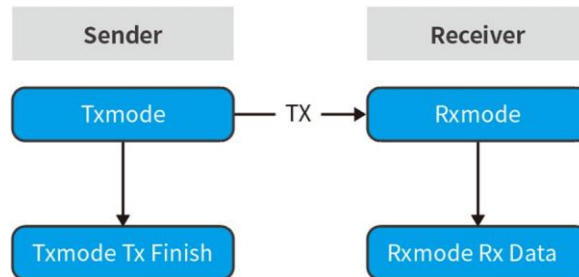
7.5.1 Basic Mode

In Basic mode, just keep the receiver in receiving mode, that is, call the RF_RX() function. However, it should be noted that after receiving data, the RF_RX() function needs to be called again to put the device in receiving mode

again, and do not directly call the RF transceiver function in the RF_2G4StatusCallBack() callback function, which may cause confusion in its status.

The communication diagram is as follows:

Figure 7.1 Basic mode communication schematic diagram



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

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

7.5.2 Auto Mode

Since the Basic mode is only a one-way transmission, the user has no way of knowing whether the communication was successful or not, thus giving rise to the Auto mode.

Auto mode is based on the Basic mode, and adds the mechanism of receiving response, that is, after receiving data, the receiver will send data to the sender to inform the sender that the data has been successfully received.

Auto mode adds a receive response mechanism to Basic mode, i.e., after receiving data, the receiver sends data to the sender to notify the sender that the data has been successfully received.

The communication diagram is as follows:

Figure 7.2 Auto mode communication diagram

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

7.5.2.1 Auto Hopping

The automatic frequency hopping solution designed based on RF Auto mode can effectively solve the interference

problem of 2.4GHz channel.

To use the frequency hopping function, you need to actively enable frequency hopping reception or frequency hopping transmission events:

```

1. // Enable frequency hopping transmission
2. if( events & SBP_RF_CHANNEL_HOP_TX_EVT ){
3.     PRINT("\n----- hop tx...\n");
4.     if( RF_FrequencyHoppingTx( 16 ) ){
5.         tmos_start_task( taskID , SBP_RF_CHANNEL_HOP_TX_EVT ,100 );
6.     }
7.     return events^SBP_RF_CHANNEL_HOP_TX_EVT;
8. }
9. // Enable frequency hopping reception
10. if( events & SBP_RF_CHANNEL_HOP_RX_EVT ){
11.     PRINT("hop rx...\n");
12.     if( RF_FrequencyHoppingRx( 200 ) )
13.     {
14.         tmos_start_task( taskID , SBP_RF_CHANNEL_HOP_RX_EVT ,400 );
15.     }
16.     else
17.     {
18.         RF_Rx( TX_DATA,10,0xFF,0xFF );
19.     }
20.     return events^SBP_RF_CHANNEL_HOP_RX_EVT;
21. }

```

After configuring the RF communication mode to automatic mode, the sender enables frequency hopping to send events:

```

1. tmos_set_event( taskID , SBP_RF_CHANNEL_HOP_TX_EVT );

```

Receive and send frequency hopping reception events:

```

1. tmos_set_event( taskID , SBP_RF_CHANNEL_HOP_RX_EVT );

```

The frequency hopping function can be realized.

It should be noted that if the receiver is already in receiving mode, it needs to turn off RF first (call the RF_Shut() function), and then turn on the frequency hopping reception event.

8. API

8.1 TMOS API

8.1.1 Command

```
1. bStatus_t TMOS_TimerInit( pfnGetSysClock fnGetClock )
```

TMOS clock initialization

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

```
1. tmosTaskID TMOS_ProcessEventRegister( pTaskEventHandIerFn eventCb )
```

Register event callback function, generally used to execute first when registering a task.

Parameter	Description
eventCb	TMOS task callback function
Return	Assigned ID value, 0xFF means invalid

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

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

Parameter	Description
taskID	tmos assigned task ID
event	Events in the task
Return	0: Success

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

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

Parameter	Description
taskID	tmos assigned task ID
event	Events in the task
time	Delay time

Return	0: Success
--------	------------

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

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

Parameter	Description
taskID	tmos assigned task ID
event	Events in the task
Return	0: Success

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

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

Parameter	Description
taskID	tmos assigned task ID
event	Events in the task
Return	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.

Parameter	Description
taskID	tmos assigned task ID
event	Events in the task
time	Delay time
Return	0: Success

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

Gets the number of ticks until the event expires.

Parameter	Description
taskID	tmos assigned task ID
event	Events in the task
Return	!0: The number of ticks until the event expires 0: Event not found

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

Returns the tmos system runtime in 625 μ s, e.g. 1600=1s.

Parameter	Description
Return	TMOS runtime

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

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

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

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

Parameter	Description
taskID	tmos assigned task ID
msg_ptr	Message pointer
Return	SUCCESS: Success INVALID_TASK: Task ID invalid INVALID_MSG_POINTER: Message pointer valid

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

Receive messages.

Parameter	Description
taskID	tmos assigned task ID
Return	Message received or no message to be received (NULL)

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

Allocate memory space for the message.

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

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

Release the memory space occupied by the message.

Parameter	Description
msg_ptr	Message pointer
Return	0: Success

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

Read data from NV.

Note: The read and write operations in the NV area should be called before the TMOS system is running as much as possible.

Parameter	Description
id	Valid NV project ID
len	Read the length of data
pBuf	Data pointer to read
Return	SUCCESS NV_OPER_FAILED: Failed

```
1. void TMOS_TimerIRQHandler( void )
```

TMOS timer interrupt function.

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

```
1. uint32_t tmos_rand( void )
```

Generate pseudo-random numbers.

Parameter	Description
Return	Pseudorandom number

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

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

Parameter	Description
src1	Memory block pointer
src2	Memory block pointer

len	The number of bytes to be compared
Return	1: Same 0: Different

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

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

Parameter	Description
Buf	Buffer address
val	Value
len	Data length
Return	1: Same 0: Different

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

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

Parameter	Description
pString	The string whose length is to be calculated
Return	length of 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.

Parameter	Description
pDst	Memory block to fill
Value	Value to be set
len	The number of characters to be set to this value
Return	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.

Parameter	Description
dst	Target array used to store copied content, type cast to void* pointer
src	Data source to copy, type cast to void* pointer
len	Number of bytes to be copied
Return	Pointer to target storage area dst

8.2 GAP API

8.2.1 Command

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

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

Parameter	Description
paramID	Parameter ID, refer to 8.1.2
paramValue	New parameter value
Return	SUCCESS or INVALIDPARAMETER (Invalid parameter ID)

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

Get GAP parameter value.

Parameter	Description
paramID	Parameter ID, refer to 8.1.2
Return	The value of the GAP parameter; if the parameter ID is invalid, return 0xFFFF.

8.2.2 Configure Parameter

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

Parameter ID	Description
TGAP_GEN_DISC_ADV_MIN	Broadcast duration of general broadcast mode, unit: 0.625ms (Default value: 0)
TGAP_LIM_ADV_TIMEOUT	The broadcast duration of the limited-time discoverable broadcast mode, unit: 1s (Default value: 180)
TGAP_DISC_ADV_INT_MIN	Minimum broadcast interval, unit: 0.625ms (Default value: 160)
TGAP_DISC_ADV_INT_MAX	Maximum broadcast interval, unit: 0.625ms (Default value: 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	Scan window, unit: 0.625ms (Default: 16)
TGAP_CONN_EST_SCAN_INT	Scanning interval to establish a connection, unit: 0.625ms (Default: 16)
TGAP_CONN_EST_SCAN_WIND	Scan window for establishing connection, unit: 0.625ms (Default: 16)
TGAP_CONN_EST_INT_MIN	Minimum connection interval for establishing a connection, unit: 1.25ms (Default: 80)

TGAP_CONN_EST_INT_MAX	The maximum connection interval for establishing a connection, unit: 1.25ms (Default: 80)
TGAP_CONN_EST_SUPERV_TIME OUT	Connection management timeout for establishing a connection, unit: 10ms (Default: 2000)
TGAP_CONN_EST_LATENCY	Slave delay for establishing connection (Default: 0)

8.2.3 Event

This section introduces events related to the GAP layer, and relevant statements can be found in the CH58xBLE_LIB.h file. Some of these events are delivered directly to the application, and some are handled by GAPRole and GAPBondMgr.

Regardless of which layer they are passed to, they will be passed as headered GAP_MSG_EVENT:

```
1. typedef struct
2. {
3.     tmos_event_hdr_t  hdr;           //!< GAP_MSG_EVENT and status
4.     uint8_t opcode;           //!< GAP type of command. Ref: @ref GAP
5.     MSG_EVENT_DEFINES
6. } gapEventHdr_t;
```

The following are common event names and the format of event delivery messages. Please refer to CH58xBLE_LIB.h for details.

➤ GAP_DEVICE_INIT_DONE_EVENT: This event is set when the device initialization is completed.

```
1. typedef struct
2. {
3.     tmos_event_hdr_t  hdr;           //!< GAP_MSG_EVENT and status
4.     uint8_t opcode;           //!< GAP_DEVICE_INIT_DONE_EVENT
5.     uint8_t devAddr[B_ADDR_LEN];    //!< Device's BD_ADDR
6.     uint16_t dataPktLen;           //!< HC_LE_Data_Packet_Length
7.     uint8_t numDataPkts;           //!< HC_Total_Num_LE_Data_Packets
8. } gapDeviceInitDoneEvent_t;
```

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

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

- **GAP_END_DISCOVERABLE_DONE_EVENT**: This event is set when the broadcast ends.

```

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

```

- **GAP_LINK_ESTABLISHED_EVENT**: This event is dispatched after the connection is established.

```

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

```

- **GAP_LINK_TERMINATED_EVENT**: This event is set after the connection is disconnected.

```

1. typedef struct
2. {
3.     tmos_event_hdr_t hdr;    //!< GAP_MSG_EVENT and status
4.     uint8_t opcode;          //!< GAP_LINK_TERMINATED_EVENT
5.     uint16_t connectionHandle; //!< connection Handle
6.     uint8_t reason;          //!< termination reason from LL
7.     uint8_t connRole;
8. } gapTerminateLinkEvent_t;

```

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

```

1. typedef struct

```

```

2. {
3.     tmos_event_hdr_t hdr;      //!< GAP_MSG_EVENT and status
4.     uint8_t opcode;           //!< GAP_LINK_PARAM_UPDATE_EVENT
5.     uint8_t status;           //!< bStatus_t
6.     uint16_t connectionHandle; //!< Connection handle of the update
7.     uint16_t connInterval;     //!< Requested connection interval
8.     uint16_t connLatency;      //!< Requested connection latency
9.     uint16_t connTimeout;      //!< Requested connection timeout
10. } gapLinkUpdateEvent_t;

```

- **GAP_DEVICE_INFO_EVENT:** The discovery device sets this event during device discovery.

```

1. typedef struct
2. {
3.     tmos_event_hdr_t hdr;      //!< GAP_MSG_EVENT and status
4.     uint8_t opcode;           //!< GAP_DEVICE_INFO_EVENT
5.     uint8_t eventType;         //!< Advertisement Type: @ref GAP_ADVERTISEMEN
6.     T_REPORT_TYPE_DEFINES
7.     uint8_t addrType;         //!< address type: @ref GAP_ADDR_TYPE_DEFINES
8.     uint8_t addr[B_ADDR_LEN]; //!< Address of the advertisement or SCAN_RSP
9.     int8_t rssi;              //!< Advertisement or SCAN_RSP RSSI
10.    uint8_t dataLen;           //!< Length (in bytes) of the data field (evtD
11.    ata)
12.    uint8_t *pEvtData;         //!< Data field of advertisement or SCAN_RSP
13. } gapDeviceInfoEvent_t;

```

8.3 GAPRole API

8.3.1 GAPRole Common Role API

8.3.1.1 Command

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

Set the GAP role parameters.

Parameter	Description
param	Configuration parameter ID, see Section 8.2.1.2 for details.
len	length of data written
pValue	Pointer to the set parameter value. This pointer depends on the parameter ID and will be cast to the appropriate data type.

Return	SUCCESS INVALIDPARAMETER: Invalid parameter bleInvalidRange: Invalid parameter length blePending: The last parameter update has not ended bleIncorrectMode: Mode error
--------	--

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

Get the GAP role parameters.

Parameter	Description
param	Configuration parameter ID, see Section 8.2.1.2 for details.
pValue	Pointer to the location where the parameters are obtained. This pointer depends on the parameter ID and will be cast to the appropriate data type.
Return	SUCCESS INVALIDPARAMETER: Invalid parameter

```
1. bStatus_t GAPRole_TerminateLink( uint16_t connHandle )
```

Disconnects the connection specified by the current connHandle.

Parameter	Description
connHandle	Connect handle
Return	SUCCESS bleIncorrectMode: Incorrect mode

```
1. bStatus_t GAPRole_ReadRssiCmd( uint16_t connHandle )
```

Reads the RSSI value for the current connHandle specified connection.

Parameter	Description
connHandle	Connect handle
Return	SUCCESS 0x02: No valid connection

8.3.1.2 Common Configurable Parameters

Parameter	Read/Write	Size	Description
GAPROLE_BD_ADDR	RO	uint8	Device address
GAPRPLE_ADVERT_ENABLE	RW	uint8	Enable or disable broadcast, enabled by default
GAPROLE_ADVERT_DATA	RW	≤240	Broadcast data, default is all 0.
GAPROLE_SCAN_RSP_DATA	RW	≤240	Scan response data, default is all 0
GAPROLE_ADV_EVENT_TYPE	RW	uint8	Broadcast type, non-directed broadcast can

			be connected by default
GAPROLE_MIN_CONN_INTER VAL	RW	uint16	Minimum connection interval, range: 1.5ms~4s, default 8.5ms.
GAPROLE_MAX_CONN_INTER VAL	RW	uint16	Maximum connection interval, range: 1.5ms~4s, default 8.5ms.

8.3.1.3 Callback Function

```

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

```

This function is a callback function that reads RSSI, and its pointer points to the application so that GAPRole can return events to the application. The delivery method is as follows:

```

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

```

8.3.2 GAPRolePeripheral Role API

8.3.2.1 Command

```

1. bStatus_t GAPRole_PeripheralInit( void )

```

Bluetooth slave GAPRole task initialization.

Parameter	Description
Return	SUCCESS bleInvalidRange: Parameter out of range

```

1. bStatus_t GAPRole_PeripheralStartDevice( uint8_t taskid, gapBondCBs_t *pCB, gapRolesCBs_t
    *pAppCallbacks )

```

Bluetooth slave device initialization.

Parameter	Description
taskid	task assigned tasks ID

pCB	Binding callback functions, including key callbacks and pairing status callbacks
pAppCallbacks	GAPRole callback function, including device status callback, RSSI callback, parameter update callback
Return	SUCCESS bleAlreadyInRequestedMode: The device has been initialized

```

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

```

Bluetooth slave connection parameters updated.

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

Parameter	Description
connHandle	Connect handle
minConnInterval	Minimum connection interval
maxConnInterval	Maximum connection interval
latency	Number of slave device delay events
connTimeout	Connection timed out
taskID	toms assigned task ID
Return	SUCCESS: Parameters uploaded successfully BleNotConnected: There is no connection so parameters cannot be updated. bleInvalidRange: Parameter error

8.3.2.2 Callback Function

```

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

```


Slave status callback function:

```

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

```

Among them, the status is divided into the following categories:

- GAPROLE_INIT // Waiting to start
- GAPROLE_STARTED // Initialization completed but not broadcast
- GAPROLE_ADVERTISING // Broadcasting
- GAPROLE_WAITING // The device started but did not broadcast. It is waiting to broadcast again.
- GAPROLE_CONNECTED // Connection Status
- GAPROLE_CONNECTED_ADV // Connected and broadcasting
- GAPROLE_ERROR // Invalid status, if this status indicates an error

Slave parameter update callback function:

```

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

```

This callback function is called successfully when parameters are updated.

8.3.3 GAPRole Central Role API

8.3.3.1 Command

```

1. bStatus_t GAPRole_CentralInit( void )

```

Host GAPRole task initialization.

Parameter	Description
Return	SUCCESS bleInvalidRange: Parameter out of range

```

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

```

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

Parameter	Description
taskId	tmos assigned task ID
pCB	Binding callback functions, including key callbacks and pairing status callbacks
pAppCallbacks	GAPRole callback function, including device status callback, RSSI callback, parameter update callback
Return	SUCCESS bleAlreadyInRequestedMode: The device has been started

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

Host scan parameter configuration.

Parameter	Description
mode	Scan mode is divided into: DEVDISC_MODE_NONDISCOVERABLE: No settings DEVDISC_MODE_GENERAL: Scan for universal discoverable devices DEVDISC_MODE_LIMITED: Scan limited discoverable devices DEVDISC_MODE_ALL: Scan all
activeScan	TRUE to enable scanning
whiteList	TRUE to scan only whitelisted devices
Return	SUCCESS

```
1. bStatus_t GAPRole_CentralCancelDiscovery( void )
```

The host stops scanning.

Parameter	Description
Return	SUCCESS bleInvalidTaskID: No tasks are scanning bleIncorrectMode: Not in scan mode

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

Connect with the peer device.

Parameter	Description
-----------	-------------

highDutyCycle	TURE Enable high duty cycle scanning
whiteList	TURE Use a whitelist
addrTypePeer	The address type of the peer device, including: ADDRTYPE_PUBLIC: BD_ADDR ADDRTYPE_STATIC: Static address ADDRTYPE_PRIVATE_NONRESOLVE: Unresolvable private address ADDRTYPE_PRIVATE_RESOLVE: Resolvable private address
peerAddr	Peer device address
Return	SUCCESS: Successful connection bleIncorrectMode: Invalid profile bleNotReady: Scanning in progress bleAlreadyInRequestedMode: Cannot be processed for the time being bleNoResources: Too many connections

8.3.3.2 Callback Function

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

```

1. typedef struct
2. {
3.     gapRolesRssiRead_t rssiCB; //!< RSSI callback.
4.     pfnGapCentralRoleEventCB_t eventCB; //!< Event callback.
5.     pfnHciDataLenChangeEvCB_t ChangCB; //!< Length Change Event Callback.
6. } gapCentralRoleCB_t; // gapCentralRoleCB_t

```

Host RSSI callback function:

```

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

```

This function reports RSSI to the application.

Host event callback function:

```

1. /**
2.  * Central Event Callback Function
3.  */

```

```

4.  typedef void ( *pfnGapCentralRoleEventCB_t ) ( gapRoleEvent_t *pEvent );
5.  //!< Pointer to event structure.

```

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

For callback events, please refer to Section 8.1.3.

MTU interaction callback function:

```

1.  typedef void (*pfnHciDataLenChangeEvCB_t)
2.  (
3.      uint16_t connHandle,
4.      uint16_t maxTxOctets,
5.      uint16_t maxRxOctets
6.  );

```

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

8.4 GAPRole API

8.4.1 Command

8.4.1.1 Slave Command

```

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

```

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

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

Parameter	Description
connHandle	Connect handle
pInd	Point to the command to send
authenticated	Whether an authenticated connection is required
taskId	tm0s assigned task ID

```

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

```

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

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

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

8.4.1.2 Host Command

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

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

Parameter	Description
connHandle	Connect handle
pReq	Point to the command to send
taskID	The ID of the notification task

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

Discover all master services on the server.

Parameter	Description
connHandle	Connect handle
taskID	The ID of the notification task

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

The client can discover the main service on the server through this function when only the UUID is known. Since there may be multiple main services on the server, the discovered main service is identified by a UUID.

Parameter	Description
connHandle	Connect handle
pUUID	Pointer to the UUID of the server to lookup
len	Length of value
taskID	The ID of the notification task

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

The client uses this function to find this service on the server. The service being looked up is identified by a service handle scope.

Parameter	Description
connHandle	Connect handle
startHandle	Start handle
endHandle	End handle
taskID	The ID of the notification task

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

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

Parameter	Description
connHandle	Connect handle
startHandle	Start handle
endHandle	End handle
taskID	The ID of the notification task

```
1. bStatus_t GATT_DiscCharsByUUID( uint16_t connHandle, attReadByTypeReq_t *pReq, uint8_t
    taskId )
```

Clients can use this function to discover characteristics on the server when the service handle range and characteristic UUID are known.

Parameter	Description
connHandle	Connect handle
pReq	Pointer to the request to send
taskID	The ID of the notification task

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

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

Parameter	Description
connHandle	Connect handle
startHandle	Start handle
endHandle	End handle

taskID	The ID of the notification task
--------	---------------------------------

```
1. bStatus_t GATT_ReadCharValue( uint16_t connHandle, attReadReq_t *pReq, uint8_t taskId )
```

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

Parameter	Description
connHandle	Connect handle
pReq	Pointer to the request to send
taskID	The ID of the notification task

```
1. bStatus_t GATT_ReadUsingCharUUID( uint16_t connHandle, attReadByTypeReq_t *pReq, uint8_t taskId )
```

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

Parameter	Description
connHandle	Connect handle
pReq	Pointer to the request to send
taskID	The ID of the notification task

```
1. bStatus_t GATT_ReadLongCharValue( uint16_t connHandle, attReadBlobReq_t *pReq, uint8_t taskId )
```

Read the server characteristic value, but the characteristic value is longer than can be sent in a single read response protocol.

Parameter	Description
connHandle	Connect handle
pReq	Pointer to the request to send
taskID	The ID of the notification task

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

Read multiple characteristic values from the server.

Parameter	Description
connHandle	Connect handle
pReq	Pointer to the request to send
taskID	The ID of the notification task

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

When the feature handle is known to the client, the feature can be written to the server without confirming whether the write was successful.

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

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

This function can be used to write the characteristic value to the server when the client knows the characteristic handle and the ATT confirmation is not encrypted. Only when the CharacteristicProperties authentication bit is enabled and both the server and client establish a binding.

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

```
1. bStatus_t GATT_WriteCharValue( uint16_t connHandle, attWriteReq_t *pReq, uint8_t  
    taskId )
```

This function writes the characteristic value to the server when the client knows the characteristic handle. Only the first eight bits of the characteristic value can be written. This function returns whether the writing process was successful.

Parameter	Description
connHandle	Connect handle
pReq	Pointer to the request to send
taskID	The ID of the notification task

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


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

Parameter	Description
connHandle	Connect handle
pReq	Pointer to the request to send
taskID	The ID of the notification task

8.4.2 Return

- SUCCESS (0x00): The instructions execute as expected.
- INVALIDPARAMETER (0x02): Invalid connection handle or request field.
- MSG_BUFFER_NOT_Avail (0x04): HCI buffer is not available. Please try again later.
- bleNotConnected (0x14): The device is not connected.
- blePending (0x17):
 - When returning to the client function, the server or GATT subprocess is in progress with a pending response.
 - When returning to the server function, a confirmation from the client is pending.
- bleTimeout (0x16): The previous transaction timed out. No ATT or GATT messages can be sent until reconnected.
- bleMemAllocError (0x13): A memory allocation error occurred
- bleLinkEncrypted (0x19): The link is encrypted. Do not send PDUs containing authentication signatures over encrypted links.

8.4.3 Event

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

The following are common event names and the format of event delivery messages. Please refer to CH58xBLE_LIB.h for details.

- ATT_ERROR_RSP:

```

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

```

- ATT_EXCHANGE_MTU_REQ:

```

1. typedef struct
2. {
3.     uint16_t clientRxMTU; //!< Client receive MTU size
4. } attExchangeMTUReq_t;

```

- ATT_EXCHANGE_MTU_RSP:

```

1. typedef struct

```

```

2. {
3.     uint16_t serverRxMTU; //!< Server receive MTU size
4. } attExchangeMTURsp_t;

```

➤ ATT_READ_REQ:

```

1. typedef struct
2. {
3.     uint16_t handle; //!< Handle of the attribute to be read (must be first
4.     field)
5. } attReadReq_t;

```

➤ ATT_READ_RSP:

```

1. typedef struct
2. {
3.     uint16_t len;    //!< Length of value
4.     uint8_t *pValue; //!< Value of the attribute with the handle given (0 to
5.     ATT_MTU_SIZE-1)
6. } attReadRsp_t;

```

➤ ATT_WRITE_REQ:

```

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

```

➤ ATT_WRITE_RSP:

➤ ATT_HANDLE_VALUE_NOTI:

```

1. typedef struct
2. {
3.     uint16_t handle; //!< Handle of the attribute that has been changed (mu
4.     st be first field)

```

```

5.     uint16_t len;    //!< Length of value
6.     uint8_t *pValue; //!< Current value of the attribute (0 to ATT_MTU_SIZE
7. -3)
8. } attHandleValueNoti_t;

```

➤ ATT_HANDLE_VALUE_IND:

```

1. typedef struct
2. {
3.     uint16_t handle; //!< Handle of the attribute that has been changed (mu
4. st be first field)
5.     uint16_t len;    //!< Length of value
6.     uint8_t *pValue; //!< Current value of the attribute (0 to ATT_MTU_SIZE
7. -3)
8. } attHandleValueInd_t;

```

➤ ATT_HANDLE_VALUE_CFM:

- Empty msg field

8.4.4 GATT Instruction and Corresponding ATT Events

ATT response event	GATT API calls
ATT_EXCHANGE_MTU_RSP	GATT_ExchangeMTU
ATT_FIND_INFO_RSP	GATT_DiscAllCharDescs 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 property handle value is not valid on this property server.
- ATT_ERR_READ_NOT_PERMITTED (0x02): Unable to read property.
- ATT_ERR_WRITE_NOT_PERMITTED (0x03): Unable to write property.
- ATT_ERR_INVALID_PDU (0x04): Attribute PDU is invalid.
- ATT_ERR_INSUFFICIENT_AUTHEN (0x05): This property requires authentication before it can be read or written.
- ATT_ERR_UNSUPPORTED_REQ (0x06): The property server does not support the request received from the property client.
- ATT_ERR_INVALID_OFFSET (0x07): The specified offset is beyond the end of the property.
- ATT_ERR_INSUFFICIENT_AUTHOR (0x08): This property requires authorization to be read or written.
- ATT_ERR_PREPARE_QUEUE_FULL (0x09): There are too many queues ready to be written to.
- ATT_ERR_ATTR_NOT_FOUND (0x0A): Property not found in the given property handle scope.
- ATT_ERR_ATTR_NOT_LONG (0x0B): The property cannot be read or written using the Read Blob request or Prepare to Write request.
- ATT_ERR_INSUFFICIENT_KEY_SIZE (0x0C): The encryption key size used to encrypt this link is insufficient.
- ATT_ERR_INVALID_VALUE_SIZE (0x0D): The attribute value length is not valid for this operation.
- ATT_ERR_UNLIKELY (0x0E): The requested property request encountered an unlikely error and failed to complete as required.
- ATT_ERR_INSUFFICIENT_ENCRYPT (0x0F): This property requires encryption to be read or written.
- ATT_ERR_UNSUPPORTED_GRP_TYPE (0x10): The attribute type is not a supported grouping attribute as defined by the higher-level specification.
- ATT_ERR_INSUFFICIENT_RESOURCES (0x11): Insufficient resources to complete the request.

8.5 GATTServApp API

8.5.1 Command

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

Initialize the client feature configuration table.

Parameter	Description
connHandle	Connect handle
charCfgTbl	Client feature configuration table
Return	None

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

Read the client's feature configuration.

Parameter	Description
connHandle	Connect handle
charCfgTbl	Client feature configuration table
Return	Attribute value

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

Writes feature configuration to the client.

Parameter	Description
connHandle	Connect handle
charCfgTbl	Client feature configuration table
value	New value
Return	SUCCESS FAILURE

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

Handles client feature configuration write requests.

Parameter	Description
connHandle	Connect handle
pAttr	Pointer to property
pvalue	Pointer to the data to be written
len	Data length
offset	Offset of the first eight bits of data written
validCfg	Valid configuration
Return	SUCCESS FAILURE

8.6 GAPBondMgr API

8.6.1 Command

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

Set binding management parameters.

Parameter	Description
param	Configure parameter
len	Write length
pValue	Pointer to the data to be written
Return	SUCCESS INVALIDPARAMETER: Invalid parameter

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

Get the parameters of binding management.

Parameter	Description
param	Configure parameter
pValue	Pointer to the data to be read
Return	SUCCESS INVALIDPARAMETER: Invalid parameter

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

Respond to password requests.

Parameter	Description
connectionHandle	Connect handle
status	SUCCESS: Password available For other details, see SMP_PAIRING_FAILED_DEFINES
passcode	Integer value password
Return	SUCCESS: Binding record found and changed bleIncorrectMode: No connection found

8.6.2 Configuration Parameters

Commonly used configuration parameters are shown in the table below. For detailed parameter IDs, please refer to CH58xBLE.LIB.h.

Parameter ID	Read/	Size	Description
--------------	-------	------	-------------

	write		
GAPBOND_PERI_PAIRING_MODE	RO/WO	uint8	Pairing method, default is: GAPBOND_PAIRING_MODE_WAIT_FOR_REQ
GAPBOND_PERI_DEFAULT_PASSCODE		uint32	Default MITM protection key, range: 0-999999, default is 0.
GAPBOND_PERI_MITM_PROTECTION	RO/WO	uint8	MITM protection. The default is 0, which turns off man-in-the-middle protection.
GAPBOND_PERI_I/O_CAPABILITIES	RO/WO	uint8	I/O capabilities, the default model is: GAPBOND_I/O_CAP_DISPLAY_ONLY, that is, the device can only be realistic.
GAPBOND_PERI_BONDING_ENABLED	RO/WO	uint8	If enabled, binding is requested during the pairing process. The default is 0, no binding is requested.

8.7 RF PHY API

8.7.1 Command

```
1. bStatus_t RF_RoleInit( void )
```

RF protocol stack initialization.

Parameter	Description
Return	SUCCESS: Initialization successful

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

RF parameter configuration

Parameter	Description
pConfig	Pointer to configuration parameter
Return	SUCCESS

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

RF receiving data function: Configure the RFPHY in the receiving state and need to reconfigure it after receiving the data.

Parameter	Description
txBuf	In automatic mode, pointer to the data returned by RF after receiving the data
txLen	In automatic mode, the length of the data returned by RF after receiving the data (0-251)

pkRxType	Accepted packet types (0xFF: accept all types of packets)
pkTxType	In automatic mode, the data packet type of the data returned by RF after receiving the data
Return	SUCCESS

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

RF transmit data function.

Parameter	Description
txBuf	Pointer to RF transmit data
txLen	Data length of RF transmitted data (0-251)
pkTxType	Type of packet transmitted
pkRxType	In automatic mode, the data type of the data received after RF sends data (0xFF: accepts all types of data packets)
Return	SUCCESS

```
1. bStatus_t RF_Shut( void )
```

Turn off RF and stop sending or receiving.

Parameter	Description
Return	SUCCESS

```
1. uint8_t RF_FrequencyHoppingTx( uint8_t resendCount )
```

RF transmitter turns on frequency hopping

Parameter	Description
resendCount	Maximum count for sending HOP_TX pdu (0: unlimited)
Return	0: SUCCESS

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

RF receiver turns on frequency hopping

Parameter	Description
timeoutMS	Maximum time to wait to receive HOP_TX pdu (Time = n * 1ms, 0: unlimited)
Return	0: SUCCESS 1: Failed 2: LLEMode error (Requires to be in automatic mode)

1. **void** RF_FrequencyHoppingShut(**void**)

Turn off RF frequency hopping function

8.7.2 Configuration Parameters

The RF configuration parameter rfConfig_t is described below:

Parameter	Description
LLEMode	LLE_MODE_BASIC: Basic mode, enter idle mode after sending or receiving LLE_MODE_AUTO: Auto mode, automatically switches to receiving mode after sending is completed LLE_MODE_EX_CHANNEL: Switch to Frequency configuration band LLE_MODE_NON_RSSI: Set the first byte of received data to the packet type
Channel	RF communication channel (0-39)
Frequency	RF communication frequency (2400000KHz-2483500KHz), it is recommended that no more than 24-bit flips and no more than 6 consecutive 0's or 1's be used.
AccessAddress	RF communication address
CRCInit	CRC initial value
RFStatusCB	RF status callback function
ChannelMap	Channel map, each bit corresponds to a channel. A bit value of 1 means the channel is valid, otherwise it is invalid. Channels are incremented by bits and channel 0 corresponds to bit 0
Resv	Reserved
HeartPeriod	Heartbeat packet interval, an integer multiple of 100ms
HopPeriod	Hopping period ($T=32n \times \text{RTC clock}$), 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 transmitted in RF mode, default 251

8.7.3 Callback Function

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

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

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

Parameter	Description
sta	RF transceiver status
crc	Data packet status check, each bit represents a different status: bit0: Data CRC check error; bit1: Wrong packet type;
rxBuf	Pointer to the received data

Return	NULL
--------	------

Revision History

Version	Date	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 preface and description of the development platform; 3. Corrigendum of Figure 3.1.
V1.4	2021/11/2	1. A new section 7.5.2.1 has been added to describe the RF frequency hopping (FH) functionality; 2. Added a new RF FH API description; 3. Optimize the code display format.
V1.5	2022/5/6	1. Erratum: Section 7.2 code citation error; 2. Erratum: Section 5.3.2 numbering error; 3. Section 3.3 refinement of TMOS task execution structure.
V1.6	2022/8/19	1. Adjust the title name of Chapter 4; 2. Add application routine description; 3. Add section 5.5.2 to describe GATT services and protocols; 4. Add TMOSAPI description; 5. Optimize content description; 6. Optimize diagrams 4.2; 7. Synchronization of sample code and routines; 8. Content correction.
v1.7	2022/9/30	1. Erratum: Section 8.7.3 crc description error 2. Header adjustment

Imprint and Disclaimer

The copyright of this manual belongs to Nanjing Qinheng Microelectronics Co., Ltd. (Copyright ©Nanjing Qinheng Microelectronics Co., Ltd. All Rights Reserved). Without the written permission of Nanjing Qinheng Microelectronics Co., Ltd., no one may use it for any purpose or in any form (Including but Improper use of any information in this product manual without limitation in whole or in part copying, leaking or distributing to any person).

Nanjing Qinheng Microelectronics Co., Ltd. has nothing to do with any unauthorized changes to the contents of this product manual.

The documentation provided by Nanjing Qinheng Microelectronics Co., Ltd. is only used as a reference for the use of related products and does not contain any guarantee for special purposes. Nanjing Qinheng Microelectronics Co., Ltd. reserves the right to change and upgrade this product manual and the products or software involved in the manual.

The reference manual may contain a small number of inadvertent errors. Errors found will be regularly corrected and updated in reprints to avoid such errors.