
N32WB031蓝牙芯片主机和从机通信过程应用笔记V1.0

简介

本文档介绍 N32WB031 系列 32 位 蓝牙芯片（以下简称 N32WB031）主从通信过程说明,包括单主机扫描过程、从机广播过程，以及主从机之间建立连接和进行数据收发过程。本文档目的在于让使用者能够快速熟悉主从机通信过程，以减少开发前期的准备时间，降低开发难度和出现错误操作的概率，最终达到快速开发量产。

主机下载 **rdtsc** 例程，从机下载 **rdtss** 例程，解释主机数据收发时对应的是 **rdtsc** 例程，解释从机数据收发时对应的程序是 **rdtss** 例程，建议同时打开主从机例程，对照笔记的解释进行数据传输过程分析和理解。

目 录

简介.....	1
目 录	2
1. 主机扫描建立.....	3
2. 从机广播建立.....	4
3. 主从机进行连接.....	5
3.1 通过设备名称进行连接.....	5
3.2 通过设备 MAC 地址进行连接.....	6
4. 主从机通信过程解析.....	8
5. 主机 HOST 通过蓝牙发送数据给从机 HOST.....	9
5.1 主机接收来自 host 的数据.....	10
5.2 主机下发数据给从机.....	11
5.3 从机接收主机下发的数据.....	13
5.4 从机把数据发送给从机 host.....	14
6. 从机 HOST 通过蓝牙发送数据给主机 HOST.....	16
6.1 从机接收来自 host 的数据.....	17
6.2 从机上发数据给主机.....	18
6.3 主机接收从机上发的数据.....	20
6.4 主机把数据发送给主机 host.....	21
注意事项	22
历史版本.....	23
声明.....	24

1. 主机扫描建立

蓝牙初始化

```

61
62  /**
63   * @brief main function
64   * @param
65   * @return
66   * @note Note
67   */
68  int main(void)
69  {
70      //for hold the SWD before sleep
71      delay_n_10us(200*1000);
72
73      NS_LOG_INIT();
74
75      app_ble_init();
76
77      NS_LOG_INFO(DEMO_STRING);
78
79      // periph init
80      LedInit(LED1_PORT, LED1_PIN); // power led
81      LedInit(LED2_PORT, LED2_PIN); //connection state
82      LedOn(LED1_PORT, LED1_PIN);
83      app_usart_dma_enable(ENABLE);
84      //init text
85      usart_tx_dma_send((uint8_t*)DEMO_STRING, sizeof(DEMO_STRING));
86
87      delay_n_10us(500);
88      //disable usart for enter sleep
89      app_usart_dma_enable(DISABLE);
90
91
92      while (1)
93      {
94          /*schedule all pending events*/
95          rwip_schedule();
96          ns_sleep();
97      }
98  }
99
100

```

进行主机扫描初始化，开启蓝牙扫描，主动扫描搜索目标从机蓝牙设备并发送连接请求进行蓝牙连接

```

259 void app_ble_init(void)
260 {
261
262     struct ns_stack_cfg_t app_handler = {0};
263     app_handler.ble_msg_handler = app_ble_msg_handler;
264     app_handler.user_msg_handler = app_user_msg_handler;
265     //initialization ble stack
266     ns_ble_stack_init(&app_handler);
267
268     app_ble_gap_params_init();
269     app_ble_sec_init();
270     app_ble_scan_init();
271     app_ble_prf_init();
272     //start scan
273     ns_ble_start_scan();
274 }
275

```

2. 从机广播建立

初始化蓝牙

```

70 int main(void)
71 {
72     //for hold the SWD before sleep
73     delay_n_10us(200*1000);
74
75     NS_LOG_INIT();
76
77     #if (CFG_APP_NS_IUS)
78     if(CURRENT_APP_START_ADDRESS == NS_APP1_START_ADDRESS) {
79         NS_LOG_INFO("application 1 start new ... \r\n");
80     }
81     else if(CURRENT_APP_START_ADDRESS == NS_APP2_START_ADDRESS) {
82         NS_LOG_INFO("application 2 start new ... \r\n");
83     }
84     #endif
85     app_ble_init();
86
87     NS_LOG_INFO(DEMO_STRING);
88
89     // periph init
90     LedInit(LED1_PORT, LED1_PIN); // power led
91     LedInit(LED2_PORT, LED2_PIN); //connection state
92     LedOn(LED1_PORT, LED1_PIN);
93     app_usart_dma_enable(ENABLE);
94     //init text
95     usart_tx_dma_send((uint8_t*)DEMO_STRING, sizeof(DEMO_STRING));
96
97     delay_n_10us(500);
98     //disable usart for enter sleep
99     app_usart_dma_enable(DISABLE);
100
101
102     while (1)
103     {
104         /*schedule all pending events*/
105         rwip_schedule();
106         ns_sleep();
107     }
108 }

```

进行从机广播初始化并开始广播，等待主机发送连接请求进行蓝牙连接

```

271 void app_ble_init(void)
272 {
273     struct ns_stack_cfg_t app_handler = {0};
274     app_handler.ble_msg_handler = app_ble_msg_handler;
275     app_handler.user_msg_handler = app_user_msg_handler;
276     //initialization ble stack
277     ns_ble_stack_init(&app_handler);
278
279     app_ble_gap_params_init();
280     app_ble_sec_init();
281     app_ble_adv_init();
282     app_ble_prf_init();
283     //start adv
284     ns_ble_adv_start();
285 }

```

3.主从机进行连接

主机可以通过扫描搜索指定的设备名称或者 MAC 地址的蓝牙从机设备，发送连接请求进行连接。

3.1 通过设备名称进行连接

以下代码配置选择设备的名称进行连接，当主机扫描到以“NS_RDTSS”为设备名的从机时就与之建立连接。主机下载 rdtsc 例程，如下配置目标连接参数，从机下载设备名称为“NS_RDTSS”的 rdtss 例程，当主从机上电后立即建立连接，连接速度很快。

主机扫描参数配置：app_ble_scan_init(void)

```

193  * @param
194  * @return
195  * @note
196  */
197 void app_ble_scan_init(void)
198 {
199     struct ns_scan_params_t init = {0};
200     /*static const uint8_t scan_target_addr[] = {"\x11\x11\x11\x11\x11\x11"};
201     //static const uint8_t target_name[] = {"NS_RDTSS"};
202     //static const uint16_t target_uuid = 0x1800;
203
204     init.type          = SCAN_PARAM_TYPE;
205     init.dup_filt_pol  = SCAN_PARAM_DUP_FILT_POL;
206     init.connect_enable = SCAN_PARAM_CONNECT_EN;
207     init.prop_active_enable = SCAN_PARAM_PROP_ACTIVE;
208     init.scan_intv     = SCAN_PARAM_INTV;
209     init.scan_wd       = SCAN_PARAM_WD;
210     init.duration     = SCAN_PARAM_DURATION;
211
212     init.filter_type   = SCAN_FILTER_BY_NAME;//SCAN_FILTER_BY_ADDRESS;
213     init.filter_data   = (uint8_t*)&target_name;
214
215     init_ble_scan_data_handler = app_ble_scan_data_handler;
216     init_ble_scan_state_handler = app_ble_scan_state_handler;
217
218     ns_ble_scan_init(&init);
219 }
220
221 void app_ble_sec_init(void)
222 {
223     struct ns_sec_init_t sec_init = {0};
224
225     sec_init.rand_pin_enable = false;
226     sec_init.pin_code = 123456;
227
228     sec_init.pairing_feat_auth = ( SEC_PARAM_BOND | (SEC_PARAM_MITH<<2) | (SEC_PARAM_LESC<<3) | (SEC_PARAM_KEYPRESS<<4) );
229     sec_init.pairing_feat_iocap = SEC_PARAM_IO_CAPABILITIES;
230     sec_init.pairing_feat_key_size = SEC_PARAM_KEY_SIZE;
231     sec_init.pairing_feat_oob = SEC_PARAM_OOB;
232     sec_init.pairing_feat_ikey_dist = SEC_PARAM_IKEY;
233     sec_init.pairing_feat_rkey_dist = SEC_PARAM_RKEY;
    
```

1、打开头文件

2、修改扫描的目标从机设备名称

3、修改扫描参数

从机设备名称配置：CUSTOM_DEVICE_NAME

```

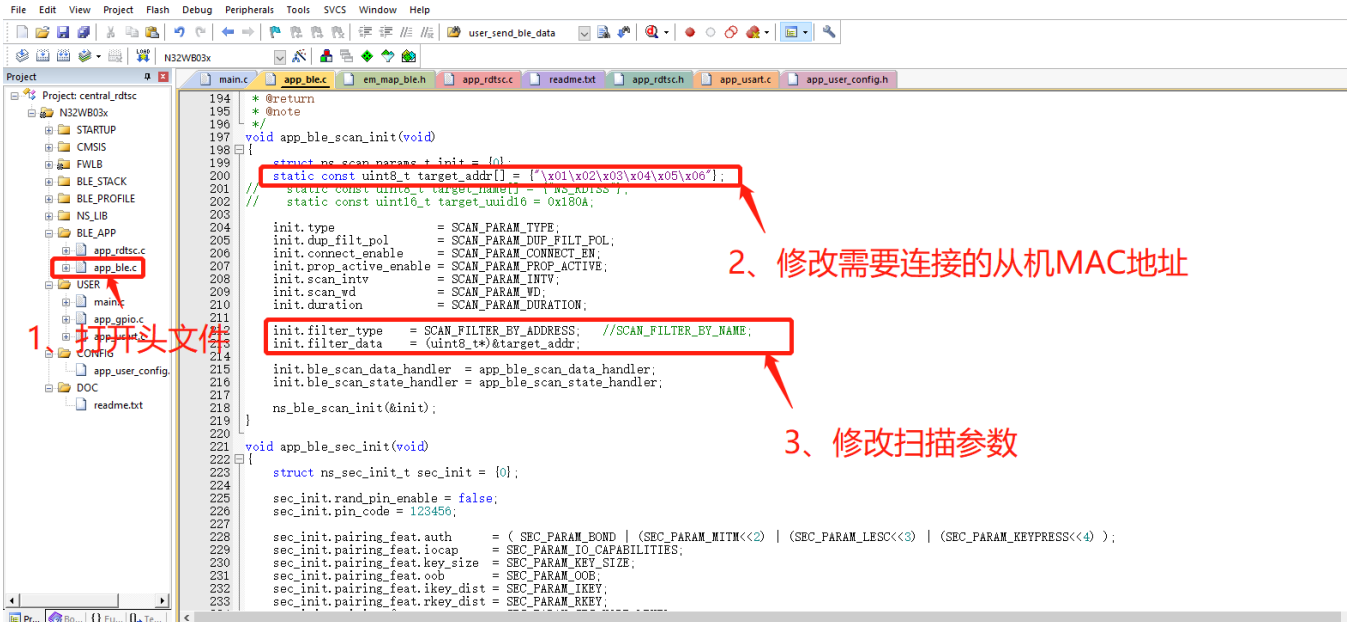
25  * EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
26  *
27  *
28  */
29  * @file app_user_config.h
30  * @author Nations Firmware Team
31  * @version v1.0.1
32  *
33  * @copyright Copyright (c) 2019, Nations Technologies Inc. All rights reserved.
34  */
35
36 #ifndef _APP_USER_CONFIG_H_
37 #define _APP_USER_CONFIG_H_
38
39 #include "ns_adv_data_def.h"
40
41 /* Device name */
42 #define CUSTOM_DEVICE_NAME "NS_RDTSS"
43
44 /* adv config */
45 #define CUSTOM_ADV_FAST_INTERVAL 100 /* Fast advertising interval (in units of 0.625 ms. This value corresponds to 100 ms.) */
46 #define CUSTOM_ADV_SLOW_INTERVAL 3200 /* Slow advertising interval (in units of 0.625 ms. This value corresponds to 2 seconds.) */
47
48 #define CUSTOM_ADV_FAST_DURATION 0/30 /* The advertising duration of fast advertising in units of 1 seconds. maximum is 655 seconds */
49 #define CUSTOM_ADV_SLOW_DURATION 180 /* The advertising duration of slow advertising in units of 1 seconds. maximum is 655 seconds */
50
51 // Advertise data
52 #define CUSTOM_USER_ADVERTISE_DATA \
53     "\x03" \
54     ADV_TYPE_SERVICE_DATA_16BIT_UUID \
55     ADV_UUID_DEVICE_INFORMATION_SERVICE \
56
57
58 #define CUSTOM_USER_ADVERTISE_DATA_LEN (sizeof(CUSTOM_USER_ADVERTISE_DATA)-1)
59
60 // Scan response data
61 #define CUSTOM_USER_ADV_SCAN_RSP_DATA \
62     "\x0a" \
63     ADV_TYPE_MANUFACTURER_SPECIFIC_DATA \
64     "\xff\xff\xff\xff"
65
    
```

1、打开头文件

2、修改从机设备名称

3.2 通过设备 MAC 地址进行连接

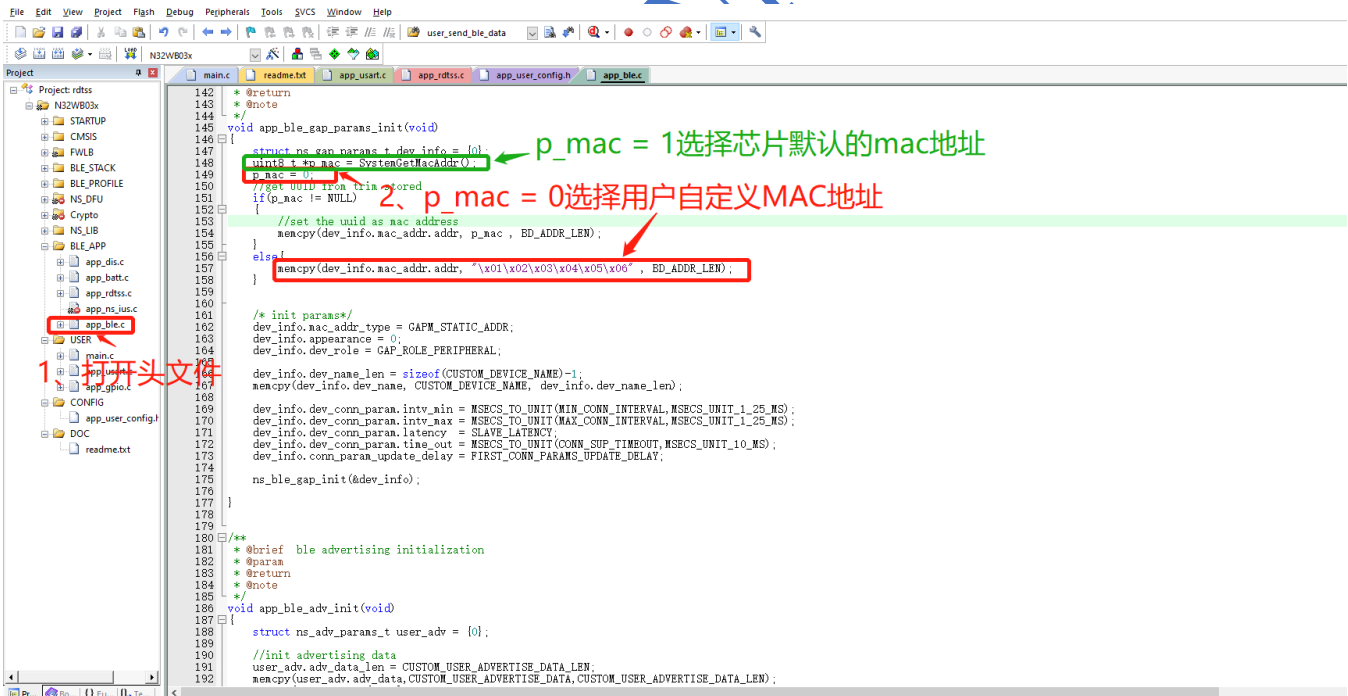
主机参数配置: app_ble_scan_init(void)



```

194  * @return
195  * @note
196  */
197 void app_ble_scan_init(void)
198 {
199     struct ns_scan_params_t init = {0};
200     static const uint8_t target_addr[] = {"\x01\x02\x03\x04\x05\x06"};
201     static const uint8_t target_name[] = "abc123456";
202     static const uint16_t target_uuid16 = 0x180A;
203
204     init.type = SCAN_PARAM_TYPE;
205     init.dup_filt_pol = SCAN_PARAM_DUP_FILT_POL;
206     init.connect_enable = SCAN_PARAM_CONNECT_EN;
207     init.prop_active_enable = SCAN_PARAM_PROP_ACTIVE;
208     init.scan_intv = SCAN_PARAM_INTV;
209     init.scan_yd = SCAN_PARAM_YD;
210     init.duration = SCAN_PARAM_DURATION;
211
212     init.filter_type = SCAN_FILTER_BY_ADDRESS; //SCAN_FILTER_BY_NAME;
213     init.filter_data = (uint8_t*)target_addr;
214
215     init.ble_scan_data_handler = app_ble_scan_data_handler;
216     init.ble_scan_state_handler = app_ble_scan_state_handler;
217
218     ns_ble_scan_init(&init);
219 }
220
221 void app_ble_sec_init(void)
222 {
223     struct ns_sec_init_t sec_init = {0};
224
225     sec_init.rand_pin_enable = false;
226     sec_init.pin_code = 123456;
227
228     sec_init.pairing_feat.auth = (SEC_PARAM_BOND | (SEC_PARAM_MITM<<2) | (SEC_PARAM_LESC<<3) | (SEC_PARAM_KEYPRESS<<4));
229     sec_init.pairing_feat.iocap = SEC_PARAM_IO_CAPABILITIES;
230     sec_init.pairing_feat.key_size = SEC_PARAM_KEY_SIZE;
231     sec_init.pairing_feat.oob = SEC_PARAM_OOB;
232     sec_init.pairing_feat.ikey_dist = SEC_PARAM_IKEY;
233     sec_init.pairing_feat.rkey_dist = SEC_PARAM_RKEY;
234 }
    
```

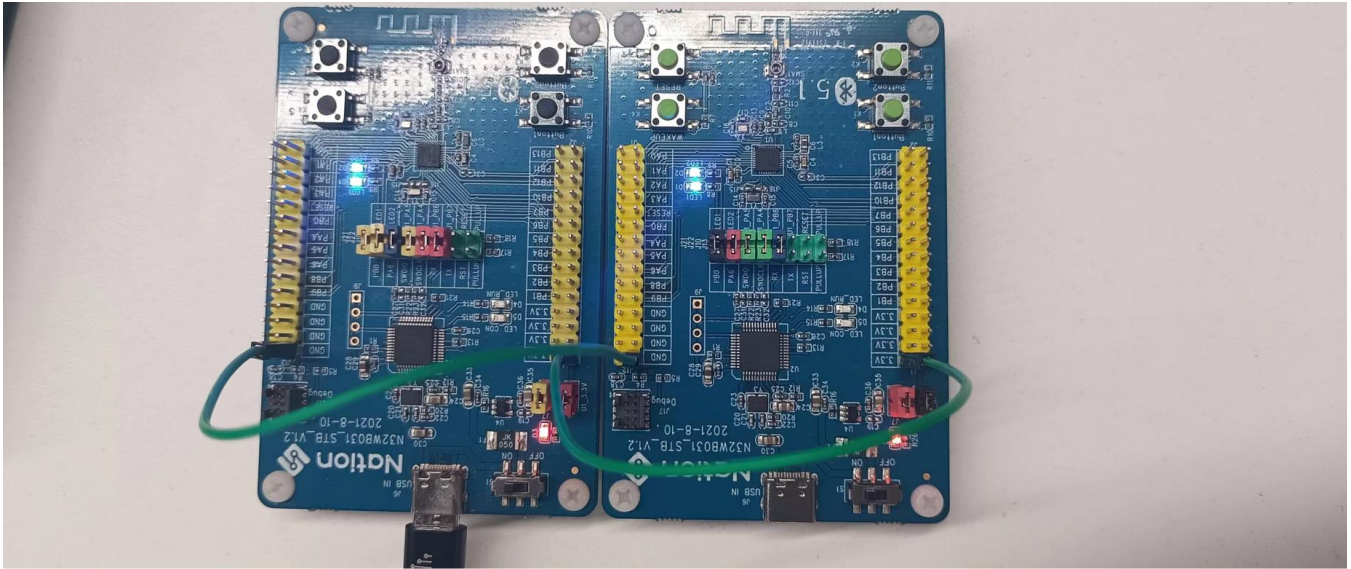
从机 MAC 地址配置: app_ble_gap_params_init(void)



```

142  * @return
143  * @note
144  */
145 void app_ble_gap_params_init(void)
146 {
147     struct ns_gap_params_t dev_info = {0};
148     uint8_t p_mac = SystemGetMacAddr(0);
149     p_mac = 1; //p_mac = 1选择芯片默认的mac地址
150     if (p_mac != NULL)
151     {
152         //set the uuid as mac address
153         memcpy(dev_info.mac_addr.addr, p_mac, BD_ADDR_LEN);
154     }
155     else
156     {
157         memcpy(dev_info.mac_addr.addr, "\x01\x02\x03\x04\x05\x06", BD_ADDR_LEN);
158     }
159
160     /* init parames */
161     dev_info.mac_addr_type = GAPM_STATIC_ADDR;
162     dev_info.appearance = 0;
163     dev_info.dev_role = GAP_ROLE_PERIPHERAL;
164
165     dev_info.dev_name_len = sizeof(CUSTOM_DEVICE_NAME)-1;
166     memcpy(dev_info.dev_name, CUSTOM_DEVICE_NAME, dev_info.dev_name_len);
167
168     dev_info.dev_comn_param.intv_min = MSECSTO_UNIT(MIN_CONN_INTERVAL, MSECSTO_UNIT_1_25_MS);
169     dev_info.dev_comn_param.intv_max = MSECSTO_UNIT(MAX_CONN_INTERVAL, MSECSTO_UNIT_1_25_MS);
170     dev_info.dev_comn_param.latency = SLAVE_LATENCY;
171     dev_info.dev_comn_param.time_out = MSECSTO_UNIT(CONN_SUP_TIMEOUT, MSECSTO_UNIT_10_MS);
172     dev_info.conn_param_update_delay = FIRST_CONN_PARAMS_UPDATE_DELAY;
173
174     ns_ble_gap_init(&dev_info);
175 }
176
177
178
179
180 /**
181  * @brief ble advertising initialization
182  * @param
183  * @return
184  * @note
185  */
186 void app_ble_adv_init(void)
187 {
188     struct ns_adv_params_t user_adv = {0};
189
190     //init advertising data
191     user_adv.adv_data_len = CUSTOM_USER_ADVERTISE_DATA_LEN;
192     memcpy(user_adv.adv_data, CUSTOM_USER_ADVERTISE_DATA, CUSTOM_USER_ADVERTISE_DATA_LEN);
193 }
    
```

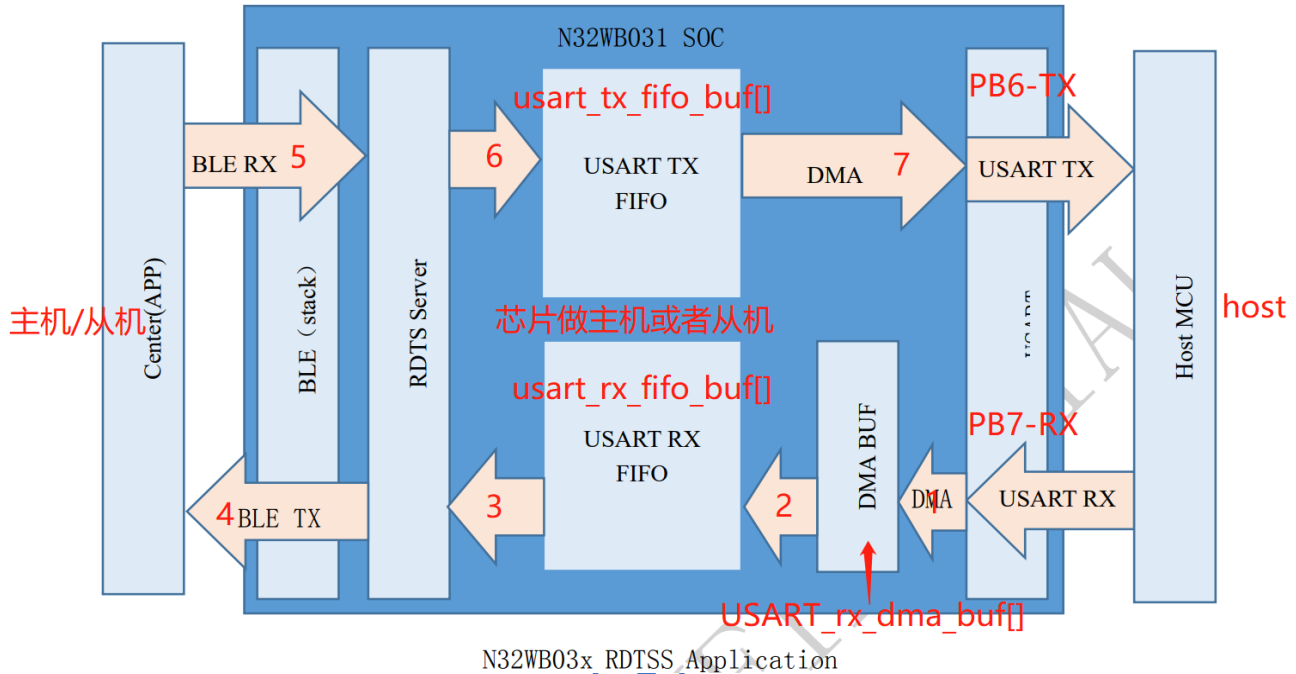
实际连接效果如下：芯片上电之后两颗蓝色的 LED 灯都亮，表示主从机蓝牙已经建立了连接



NATIONS CONFIDENTIAL

4. 主从机通信过程解析

数据传输示意图:



芯片做主机时，数据传输示意图中各序号调用的函数如下：

- 1: DmaCh1234_irq_handler()、 USART_irq_handler()
- 2: USART_rx_check_in_irq()、 app_USART_rx_data_fifo_enter()
- 3: rdtss_data_tx_cfm_handler()、 USART_forward_to_ble_loop()
- 4: user_send_ble_data()
- 5: rdtss_data_rx_ind_handler()
- 6: app_USART_tx_fifo_enter()
- 7: app_user_msg_handler()、 app_USART_tx_process()

芯片做从机时，数据传输示意图中各序号调用的函数如下：

- 1: DmaCh1234_irq_handler() 和 USART_irq_handler()
- 2: USART_rx_check_in_irq()、 app_USART_rx_data_fifo_enter()
- 3: rdtss_val_ntf_cfm_handler()、 USART_forward_to_ble_loop()
- 4: rdtss_send_notify()
- 5: rdtss_val_write_ind_handler()
- 6: app_USART_tx_fifo_enter()
- 7: app_user_msg_handler()、 app_USART_tx_process()

5. 主机 host 通过蓝牙发送数据给从机 host

数据传输过程示意图：



The screenshot shows two instances of the SSCOM V5.13.1 software. The left window, titled '从机host', shows a terminal window with the text 'data sent from client' and a '发送' (Send) button. The right window, titled '主机host', shows a terminal window with the text 'data sent from client' and a '发送' (Send) button. Red arrows indicate the data flow: from the '从机host' terminal to '从机rtdtss', from '主机rtdtsc' to the '主机host' terminal, and from '主机rtdtsc' to '从机rtdtss' via BLE. The text '主机host通过蓝牙主机发送数据给从机host的传输过程' is overlaid on the diagram.

5.1 主机接收来自 host 的数据

主机的主控 MCU 通过串口 1 的 PB6-TX 和 PB7-RX 把数据发送给主机。

主机调用 DmaCh1234_irq_handler() 和 usart_irq_handler() 中断函数进行数据接收

```

499 /**
500  * @brief dma irq handler
501  */
502 void DmaCh1234_irq_handler(void)
503 {
504     //TX
505     if(DMA_GetFlagStatus(DMA_FLAG_TC1, DMA))
506     {
507         //TX Transfer complete interrupt
508         usart_sending = false;
509         ke_ns8_send_basic(APP_UART_TX_EVT, TASK_APP, TASK_APP);
510         DMA_ClearFlag(DMA_FLAG_TC1, DMA);
511     }
512     //RX
513     if(DMA_GetFlagStatus(DMA_FLAG_TC2, DMA))
514     {
515         //RX Transfer complete interrupt
516         DMA_ClearFlag(DMA_FLAG_TC2, DMA); //当host连续发送大量的数据, 数据量超过DMA一次搬运的设置的数据量时USART_RX_DMA_SIZE, 可以触发DMA中断接收定长数据
517         usart_rx_check_in_irq0; //检查DMA已传输完成的数据
518     }
519     if(DMA_GetFlagStatus(DMA_FLAG_HT2, DMA))
520     {
521         //RX Half transfer interrupt
522         DMA_ClearFlag(DMA_FLAG_HT2, DMA); //检查DMA已传输完成的数据
523         usart_rx_check_in_irq0; //检查DMA已传输完成的数据
524     }
525 }
526 /**
527  * @brief usart irq handler
528  */
529 void usart_irq_handler(void)
530 {
531     uint8_t temp;
532     //usart idle interrupt
533     if(USART_GetFlagStatus(USARTx, USART_FLAG_IDLEF))
534     {
535         //read sts and data will clear rx idle interrupt
536         temp = USARTx->DR;
537         usart_rx_check_in_irq0; //当host一次性发送的数据没有达到DMA一次搬运的设置的数据量时USART_RX_DMA_SIZE, 可以触发空闲中断接收不定长数据
538         (void) temp;
539     }
540 }
541
542
543
544

```

1. 使用串口的DMA中断和空闲中断接收来自host的数据, 数据存入USART_rx_dma_buf[]中

检查处理接收到的数据: usart_rx_check_in_irq()

```

460
461
462 /**
463  * @brief check the dma buffer which has been received
464  */
465 void usart_rx_check_in_irq(void) //检查DMA已传输完成的数据, 并把数据从串口DMA接收数据缓冲区 USART_rx_dma_buf[]复制到串口数据接收缓冲区 usart_rx_fifo_buf[]
466 {
467     uint16_t rx_pos;
468     rx_pos = USART_RX_DMA_SIZE - DMA_GetCurrDataCounter(USARTx_Rx_DMA_Channel);
469     NS_LOG_DEBUG("RX_POS = %x\r\n", rx_pos);
470     NS_LOG_DEBUG("RX_OLD_POS = %x\r\n", rx_old_pos);
471     if(rx_pos < rx_old_pos)
472     {
473         app_usart_rx_data_fifo_enter(&USART_rx_dma_buf[rx_old_pos], (USART_RX_DMA_SIZE - rx_old_pos)); //把已从host接收到的数据转移到fifo (usart_rx_fifo_buf)
474     }
475     if(rx_pos > 0)
476     {
477         app_usart_rx_data_fifo_enter(&USART_rx_dma_buf[0], rx_pos); //把已从host接收到的数据转移到fifo (usart_rx_fifo_buf)
478     }
479     rx_old_pos = rx_pos;
480     else if(rx_pos > rx_old_pos)
481     {
482         app_usart_rx_data_fifo_enter(&USART_rx_dma_buf[rx_old_pos], (rx_pos - rx_old_pos)); //把已从host接收到的数据转移到fifo (usart_rx_fifo_buf)
483     }
484     rx_old_pos = rx_pos;
485     else if(rx_pos != rx_old_pos)
486     {
487         //error
488         rx_old_pos = rx_pos;
489     }
490 }
491
492
493
494
495
496
497

```

2. 检查DMA已传输完成的数据, 并把数据从串口DMA接收数据缓冲区 USART_rx_dma_buf[]复制到串口数据接收缓冲区 usart_rx_fifo_buf[]

主机在串口接收到数据后，会创建一个定时器任务 `ke_timer_set(RDTSC_DATA_TX_CFM, TASK_APP, 10)`，在串口接收数据断流 10ms 后，调用定时器任务对应的回调函数把数据通过蓝牙发送到从机。

复制数据到 `usart_rx_data_fifo []`: `app_usart_rx_data_fifo_enter()`。

```

288 * @brief usart rx data enter fifo and active ble send first package if not active yet
289
290 uint8_t app_usart_rx_data_fifo_enter(const uint8_t *p_data, uint16_t len) //把USART_rx_fifo_buf[]数据复制到usart_rx_fifo_buf[]
291 {
292     uint32_t in_len;
293     //store data in fifo
294     while(len)
295     {
296         if(usart_rx_fifo_in >= usart_rx_fifo_out)
297         {
298             in_len = USART_RX_FIFO_SIZE - usart_rx_fifo_in;
299             if(in_len > len)
300             {
301                 in_len = len;
302                 memcpy(usart_rx_fifo_buf + usart_rx_fifo_in, p_data, in_len);
303                 usart_rx_fifo_in += in_len;
304                 p_data += in_len;
305                 usart_rx_fifo_in = (usart_rx_fifo_in + in_len) % USART_RX_FIFO_SIZE;
306             }
307             else if(usart_rx_fifo_in < usart_rx_fifo_out)
308             {
309                 in_len = usart_rx_fifo_out - usart_rx_fifo_in - 1;
310                 if(in_len > len)
311                 {
312                     in_len = len;
313                     memcpy(usart_rx_fifo_buf + usart_rx_fifo_in, p_data, in_len);
314                     usart_rx_fifo_in += in_len;
315                     usart_rx_fifo_in = (usart_rx_fifo_in + in_len) % USART_RX_FIFO_SIZE;
316                 }
317                 //fifo full, drop the rest data
318                 if(len)
319                 {
320                     NS_LOG_WARNING("F_RX:%d,%d,%d\r\n", len, usart_rx_fifo_in, usart_rx_fifo_out);
321                 }
322                 break;
323             }
324         }
325     }
326     if(!ble_sending)
327     {
328         // active send ble event after usart streaming out
329         //在USART接收数据断流(10ms没有新数据)后，调用usart_forward_to_ble_loop()函数检查fifo(usart_rx_fifo_buf)和把数据通过rdtsc_send_notify函数发送到BLE
330         ke_timer_set(RDTSC_DATA_TX_CFM, TASK_APP, 10); //在每一包数据发送完成的回调函数rdtsc_val_ntf_cfm_handler中再次调用usart_forward_to_ble_loop()函数直到所有数据发送完成。
331     }
332     return len;
333 }
    
```

3、把USART_rx_dma_buf[]中的数据复制到 usart_rx_fifo_buf[]中，并且在串口接收数据断流10ms之后，在创建的定时器任务中把数据通过蓝牙发送给从机

5.2 主机下发数据给从机

在用户信息处理函数中添加数据发送函数的标志：RDTSC_DATA_TX_CFM

```

250
251 // Default State handlers definition 默认状态处理 //conformation: 构象、结构
252 const struct ke_msg_handler app_rdtsc_msg_handler_list[] =
253 {
254     [RDTSC_ENABLE_CFM, (ke_msg_func_t)rdtsc_enable_cfm_handler],
255     [RDTSC_DATA_TX_CFM, (ke_msg_func_t)rdtsc_data_tx_cfm_handler], //主机向从机发送数据回调函数
256     [RDTSC_DATA_RX_IND, (ke_msg_func_t)rdtsc_data_rx_ind_handler], //主机接收从机数据回调函数
257 };
258
259 const struct app_subtask_handlers app_rdtsc_handlers = APP_HANDLERS(app_rdtsc);
260
261 #endif //BLE_RDTSC_CLIENT
    
```

主机向从机发送数据回调函数：rdtsc_data_tx_cfm_handler()

```

197 * @param[in] msgid Id of the message received.
198 * @param[in] param Pointer to the parameters of the message.
199 * @param[in] dest_id ID of the receiving task instance (TASK_APP).
200 * @param[in] src_id ID of the sending task instance.
201
202 * @return If the message was consumed or not.
203 *****
204 int rdtsc_data_tx_cfm_handler(ke_msg_id_t const msgid, //在USART接收数据断流(10ms没有新数据)后
205                             struct rdtsc_data_tx_cfm const *param,
206                             ke_task_id_t const dest_id,
207                             ke_task_id_t const src_id)
208 {
209     NS_LOG_DEBUG("%s\r\n", __func__);
210     usart_forward_to_ble_loop(); //调用usart_forward_to_ble_loop()函数检查fifo(usart_rx_fifo_buf)和把数据通过rdtsc_send_notify函数发送到BLE
211     return (KE_MSG_CONSUMED);
212 }
    
```

4、发送数据回调函数，调用 usart_forward_to_ble_loop()函数把数据发送给从机，并且每发送完一包数据都再次调用此函数发送数据直到数据发送完成

循环调用该函数直到把所有的数据发送给从机: `usart_forward_to_ble_loop()`

```

240 }
241 /**
242  * @brief forward usart rx data too ble notify
243  */
244 void usart_forward_to_ble_loop(void)
245 {
246     NS_LOG_DEBUG("usart_rx_fifo_out = %x\r\n", usart_rx_fifo_out);
247     NS_LOG_DEBUG("usart_rx_fifo_in = %x\r\n", usart_rx_fifo_in);
248
249     uint32_t in_temp;
250     uint16_t ble_send_len;
251
252     in_temp = usart_rx_fifo_in;
253     if(usart_rx_fifo_out < in_temp)
254     {
255         ble_send_len = in_temp-usart_rx_fifo_out;
256     }
257     else if(usart_rx_fifo_out > in_temp){
258         ble_send_len = USART_RX_FIFO_SIZE-usart_rx_fifo_out;
259     }
260     else if(usart_rx_fifo_out == in_temp){
261         // fifo empty, stop send loop
262         ble_sending = false;
263         return;
264     }
265     if(ble_send_len > ble_att_mtu)
266     {
267         ble_send_len = ble_att_mtu;
268     }
269     ble_sending = true;
270     if (BLE_RDTSC_CLIENT)
271     {
272         NS_LOG_DEBUG("ble_send_len = %x\r\n", ble_send_len);
273         user_send_ble_data(usart_rx_fifo_buf(usart_rx_fifo_out), ble_send_len); //主机发送数据给蓝牙从机
274     }
275     usart_rx_fifo_out = (usart_rx_fifo_out+ble_send_len)%USART_RX_FIFO_SIZE;
276 }
277
278
279
280
281
282
283
284

```

发送数据给从机: `user_send_ble_data()`

```

151
152
153 void user_send_ble_data(const uint8_t *data, uint16_t length)
154 {
155     NS_LOG_DEBUG("%s\r\n", __func__);
156     struct rdtsc_data_tx_req * req = KE_MSG_ALLOC_DYN(RDTSC_DATA_TX_REQ,
157                                                       prf_get_task_from_id(TASK_ID_RDTSC),
158                                                       TASK_APP,
159                                                       rdtsc_data_tx_req,
160                                                       length);
161
162     req->length = length;
163     memcpy(&req->data[0], data, length);
164     ke_msg_send(req);
165 }
166
167
168 /**

```

5.3 从机接收主机下发的数据

从机接收蓝牙数据回调函数: rdtss_val_write_ind_handler()

```

167 static int rdtss_val_write_ind_handler(ke_msg_id_t const msgid, //接收蓝牙主机数据回调函数
168                                     struct rdtss_val_write_ind const *ind_value,
169                                     ke_task_id_t const dest_id,
170                                     ke_task_id_t const src_id)
171 {
172     NS_LOG_DEBUG("%s,write handle = %x,length = %x\r\n",__func__, ind_value->handle, ind_value->length);
173     for(uint16_t i=0; i<ind_value->length; i++)
174     {
175         NS_LOG_DEBUG("%x ", ind_value->value[i]);
176     }
177     NS_LOG_DEBUG("\r\n");
178     uint16_t handle = ind_value->handle;
179     uint16_t length = ind_value->length;
180     switch (handle)
181     {
182     case RDTSS_IDX_NTF_CFG:
183         if(length == 2)
184         {
185             uint16_t cfg_value = ind_value->value[0] + ind_value->value[1];
186             if(cfg_value == PRF_CLI_START_MTF)
187                 //enabled notify
188             }
189             else if(cfg_value == PRF_CLI_STOP_MTFIND)
190             {
191             }
192             break;
193     case RDTSS_IDX_WRITE_VAL:
194         //把接收到的数据存入usart_tx_fifo_buf[]中
195         app_usart_tx_fifo_enter(ind_value->value, ind_value->length);
196         break;
197     default:
198         break;
199     }
200     return (KE_MSG_CONSUMED);
201 }
202
203 /**
204  * @brief Functions
205  * @param
206  */
    
```

1、从机接收到主机通过蓝牙发送过来的数据后，会进入此蓝牙数据接收回调函数，把接收到的数据存入usart_tx_fifo_buf[]中

把蓝牙接收到的数据复制到 usart_tx_fifo_buf[]: app_usart_tx_fifo_enter(), 并创建发送数据定时器任务发送数据。

```

376 /**
377  * @brief usart tx fifo enter and active dma send out
378  * @param
379  * @return
380  */
381 uint8_t app_usart_tx_fifo_enter(const uint8_t *p_data, uint16_t len)
382 {
383     uint32_t in_len, out_temp;
384     //store data in fifo
385     NS_LOG_DEBUG("%d,%d\r\n", len, usart_tx_fifo_in, usart_tx_fifo_out);
386     out_temp = usart_tx_fifo_out;
387     while(len)
388     {
389         if(usart_tx_fifo_in >= out_temp)
390         {
391             in_len = USART_TX_FIFO_SIZE-usart_tx_fifo_in;
392             if(in_len > len)
393             {
394                 in_len = len;
395                 memcpy(usart_tx_fifo_buf[usart_tx_fifo_in], p_data, in_len);
396                 len -= in_len;
397                 p_data += in_len;
398                 usart_tx_fifo_in = (usart_tx_fifo_in + in_len)%USART_TX_FIFO_SIZE;
399             }
400             else if(usart_tx_fifo_in < out_temp)
401             {
402                 in_len = out_temp-usart_tx_fifo_in-1;
403                 if(in_len > len)
404                 {
405                     in_len = len;
406                     memcpy(usart_tx_fifo_buf[usart_tx_fifo_in], p_data, in_len);
407                     len -= in_len;
408                     usart_tx_fifo_in = (usart_tx_fifo_in + in_len)%USART_TX_FIFO_SIZE;
409                 }
410                 //fifo full, drop the rest data
411                 if(len)
412                 {
413                     NS_LOG_WARNING("F:%d,%d\r\n", len, usart_tx_fifo_in, out_temp);
414                 }
415                 break;
416             }
417         }
418     }
419     // ble streaming out timer, active usart send after it.
420     ke_timer_set(APP_CUSTS_TEST_EVT, TASK_APP, 50); //蓝牙数据断流50ms之后激活串口发送数据给host
421     return len;
422 }
423
424 /**
425  * @brief usart tx data from fifo
426  */
    
```

2、把蓝牙接收到的数据复制带usart_tx_fifo_buf[]中，并且激活串口发送数据给从机的host

5.4 从机把数据发送给从机 host

从机通过串口 1 即 PB6-TX 和 PB7-RX 把数据发送给从机的 host

创建一个用户自定义的定时器任务：ke_timer_set(APP_CUSTS_TEST_EVT, TASK_APP, 50)

用户事件标志：APP_CUSTS_TEST_EVT

用户信息处理函数：app_user_msg_handler()

```

269  * @param
270  * @return
271  * @note
272  */
273 void app_ble_init(void)
274 {
275     struct ns_stack_cfg_t app_handler = {0};
276     app_handler.ble_msg_handler = app_ble_msg_handler;
277     app_handler.user_msg_handler = app_user_msg_handler;
278     //start adv
279     ns_ble_stack_init(&app_handler);
280
281     app_ble_gap_params_init();
282     app_ble_sec_init();
283     app_ble_adv_init();
284     app_ble_prf_init();
285     //start adv
286     ns_ble_adv_start();
287
288 }
    
```

在蓝牙接收数据断流 50ms 之后，调用用户信息处理回调函数：app_user_msg_handler()

```

60 /* Private functions ----- */
61 void app_ble_connected(void);
62 void app_ble_disconnected(void);
63
64 /**
65  * @brief user message handler
66  * @param
67  * @return
68  * @note
69  */
70 void app_user_msg_handler(ke_msg_id_t const msgid, void const *p_param)
71 {
72
73     switch (msgid)
74     {
75     case APP_CUSTS_TEST_EVT:
76         app_usart_tx_process();
77         break;
78     case :
79         break;
80     default:
81         break;
82     }
83
84 }
85
86
87
88
    
```

1、从机在蓝牙接收完数据之后，会自动调用此回调函数把蓝牙接收到的数据通过串口发送到其host

把数据通过串口发送给其 host: app_usart_tx_process()

```
428  
429 /**  
430  * @brief usart tx data from fifo  
431  */  
432 void app_usart_tx_process(void)  
433 {  
434     uint32_t in_temp, len;  
435     uint8_t *p_data;  
436  
437     in_temp = usart_tx_fifo_in;  
438     if(usart_tx_fifo_out < in_temp)  
439     {  
440         len = in_temp-usart_tx_fifo_out;  
441         p_data = &usart_tx_fifo_buf[usart_tx_fifo_out];  
442         if(usart_tx_dma_send(p_data, len) == true)  
443         {  
444             usart_tx_fifo_out = in_temp;  
445         }  
446     }  
447     else if(usart_tx_fifo_out > in_temp)  
448     {  
449         len = USART_TX_FIFO_SIZE-usart_tx_fifo_out;  
450         p_data = &usart_tx_fifo_buf[usart_tx_fifo_out];  
451         if(usart_tx_dma_send(p_data, len) == true)  
452         {  
453             usart_tx_fifo_out = 0;  
454         }  
455     }  
456 }  
457  
458  
459  
460
```

2、使用串口DMA发送数据给其host

NATION'S CONFIDENTIAL

6. 从机 host 通过蓝牙发送数据给主机 host

数据传输过程示意图：



从机host通过主从机蓝牙发送数据给主机host数据传输过程

6.1 从机接收来自 host 的数据

从机主控 MCU 即从机 host 通过串口 1 的 PB6-TX 和 PB7-RX 把数据发送给从机。

从机通过 DmaCh1234_irq_handler() 和 usart_irq_handler() 中断接收数据。

```

490 /**
491  * @brief dma irq handler
492  */
493 void DmaCh1234_irq_handler(void)
494 {
495     //TX
496     if(DMA_GetFlagStatus(DMA_FLAG_TC1, DMA))
497     {
498         //TX Transfer complete interrupt
499         usart_sending = false;
500         ke_msg_send_basic(APP_CUSTS_TEST_EVT, TASK_APP, TASK_APP);
501
502         DMA_ClearFlag(DMA_FLAG_TC1, DMA);
503
504     }
505     //RX
506     if(DMA_GetFlagStatus(DMA_FLAG_TC2, DMA))
507     {
508         //RX Transfer complete interrupt
509         DMA_ClearFlag(DMA_FLAG_TC2, DMA);
510         usart_rx_check_in_irq();
511     }
512     if(DMA_GetFlagStatus(DMA_FLAG_HT2, DMA))
513     {
514         //RX Half transfer interrupt
515         DMA_ClearFlag(DMA_FLAG_HT2, DMA);
516         usart_rx_check_in_irq();
517     }
518 }
519
520 /**
521  * @brief usart irq handler
522  */
523 void usart_irq_handler(void)
524 {
525     USART_t temp;
526     //usart idle interrupt
527     if(USART_GetFlagStatus(USARTx, USART_FLAG_IDLEF))
528     {
529         //read sts and data will clear rx idle interrupt
530         temp = USARTx->DAT;
531         usart_rx_check_in_irq();
532         (void)temp;
533     }
534 }

```

1、从机使用串口DMA中断和空闲中断接收来自从机的host的数据

检查处理接收到的数据: usart_rx_check_in_irq()

```

458 /**
459  * @brief check the dma buffer which has been received
460  */
461 void usart_rx_check_in_irq(void)
462 {
463     uint16_t rx_pos;
464
465     rx_pos = USART_RX_DMA_SIZE - DMA_GetCurrDataCounter(USARTx_RX_DMA_Channel);
466     if(rx_pos < rx_old_pos)
467     {
468         app_usart_rx_data_fifo_enter(&USART_rx_dma_buf[rx_old_pos], (USART_RX_DMA_SIZE - rx_old_pos));
469     }
470     if(rx_pos > 0)
471     {
472         app_usart_rx_data_fifo_enter(&USART_rx_dma_buf[0], rx_pos);
473     }
474     rx_old_pos = rx_pos;
475 }
476 else if(rx_pos > rx_old_pos)
477 {
478     app_usart_rx_data_fifo_enter(&USART_rx_dma_buf[rx_old_pos], (rx_pos - rx_old_pos));
479     rx_old_pos = rx_pos;
480 }
481 else if(rx_pos != rx_old_pos)
482 {
483     //error
484     rx_old_pos = rx_pos;
485 }
486 }
487
488

```

2、检查串口DMA接收的数据, 并把数据从 USART rx dma buf[]复制到usart_rx_fifo_buf[]中

6.2 从机上发数据给主机

把 DMA 接收到的数据复制到 usart_rx_data_fifo 缓冲区：app_usart_rx_data_fifo_enter(), 并创建发送数据定时器任务发送数据。

```

284 0/**
285  * @brief usart rx data enter fifo and active ble send first package if not active yet
286  *
287  * @param[in] p_data: usart rx data
288  * @param[in] len: usart rx data length
289  *
290  * @return: usart rx data length
291  */
292 void usart_rx_data_fifo_enter(const uint8_t *p_data, uint16_t len)
293 {
294     uint32_t in_len;
295     //store data in fifo
296     while(len)
297     {
298         if(usart_rx_fifo_in >= usart_rx_fifo_out)
299         {
300             in_len = USART_RX_FIFO_SIZE - usart_rx_fifo_in;
301             if(in_len > len)
302             {
303                 in_len = len;
304                 memcpy(&usart_rx_fifo_buf[usart_rx_fifo_in], p_data, in_len);
305                 p_data += in_len;
306                 usart_rx_fifo_in = (usart_rx_fifo_in + in_len) % USART_RX_FIFO_SIZE;
307             }
308             else if(usart_rx_fifo_in < usart_rx_fifo_out)
309             {
310                 in_len = usart_rx_fifo_out - usart_rx_fifo_in - 1;
311                 if(in_len > len)
312                 {
313                     in_len = len;
314                     memcpy(&usart_rx_fifo_buf[usart_rx_fifo_in], p_data, in_len);
315                     len -= in_len;
316                     usart_rx_fifo_in = (usart_rx_fifo_in + in_len) % USART_RX_FIFO_SIZE;
317                 }
318                 //fifo full, drop the rest data
319                 if(len)
320                 {
321                     NS_LOG_WARNING("F:%d, %d, %d\r\n", len, usart_rx_fifo_in, usart_rx_fifo_out);
322                     break;
323                 }
324             }
325         }
326         if(ble_sending)
327         {
328             ke_timer_set(RDTSS_VAL_NTF_CFM, TASK_APP, 10);
329         }
330     }
331     return len;
332 }
    
```

3、把数据复制到usart_rx_fifo_buf[]中，并在串口接收数据断流10ms之后通过BLE把数据发送到蓝牙主机

从机发送数据事件标志：RDTSS_VAL_NTF_CFM

```

246 req->notification = true;
247 req->handle = RDTSS_IDX_NTF_VAL; //通知特征值，从机该特征值向主机发送数据
248 req->length = length;
249 memcpy(&req->value[0], data, length);
250
251 ke_msg_send(req);
252
253
254
255
256 // Default State handlers definition //默认状态处理函数定义，达到一定条件就调用对应的函数
257 const struct ke_msg_handler app_rdtss_msg_handler_list[] =
258 {
259     [RDTSS_VALUE_REQ_IND, (ke_msg_func_t)rdtss_value_req_ind_handler],
260     [RDTSS_VAL_WRITE_IND, (ke_msg_func_t)rdtss_val_write_ind_handler], //从机接收主机蓝牙数据回调函数
261     [RDTSS_VAL_NTF_CFM, (ke_msg_func_t)rdtss_val_ntf_cfm_handler], //从机向主机发送数据回调函数
262 };
263
264 const struct app_subtask_handlers app_rdtss_handlers = APP_HANDLERS(app_rdtss);
265
266
267
    
```

从机发送数据回调函数：rdtss_val_ntf_cfm_handler()

```

208 }
209
210 return (KE_MSG_CONSUMED);
211
212
213 0/**
214  * @brief Functions
215  * @param
216  * @return
217  * @note Note
218  */
219
220 static int rdtss_val_ntf_cfm_handler(ke_msg_id_t const msg_id, //在每一包数据发送完成的回调函数，再次调用usart_forward_to_ble_loop函数直到所有数据发送完成。
221 struct rdtss_val_ntf_cfm const *cfm_value,
222 ke_task_id_t const dst_id,
223 ke_task_id_t const src_id)
224 {
225     NS_LOG_DEBUG("%s, ntf cfm handle = %x, status = %x\r\n", __func__, cfm_value->handle, cfm_value->status);
226     usart_forward_to_ble_loop0; //把来自host的数据上传到蓝牙主机
227     return (KE_MSG_CONSUMED);
228 }
229
230
    
```

4、从机蓝牙发送数据回调函数，在每一包数据发送完成之后再次调用此函数发送数据，直到数据全部发送

循环调用该函数直到把所有的数据发送给主机： usart_forward_to_ble_loop()

```

249 /**
250  * @brief forward usart rx data too ble notify
251  */
252 void usart_forward_to_ble_loop(void)
253 {
254     uint32_t in_temp;
255     uint16_t ble_send_len;
256
257     in_temp = usart_rx_fifo_in;
258     if(usart_rx_fifo_out < in_temp)
259     {
260         ble_send_len = in_temp-usart_rx_fifo_out;
261     }
262     else if(usart_rx_fifo_out > in_temp){
263         ble_send_len = USART_RX_FIFO_SIZE-usart_rx_fifo_out;
264     }
265     else if(usart_rx_fifo_out == in_temp){
266         // fifo empty, stop send loop
267         ble_sending = false;
268         return;
269     }
270     if(ble_send_len > ble_att_mtu)
271     {
272         ble_send_len = ble_att_mtu;
273     }
274     ble_sending = true;
275     #if (BLE_RDTSS_SERVER)
276     rdtss_send_notify(&usart_rx_fifo_buf[usart_rx_fifo_out], ble_send_len);
277     #endif
278     usart_rx_fifo_out = (usart_rx_fifo_out+ble_send_len)%USART_RX_FIFO_SIZE;
279
280     return;
281 }
    
```

把数据发送给主机： rdtss_send_notify()

```

235 * @note
236 //
237 void rdtss_send_notify(uint8_t *data, uint16_t length)
238 {
239     struct rdtss_val_ntf_ind_req *req = KE_MSG_ALLOC_DYN(RDTSS_VAL_NTF_REQ,
240                                                         pri_get_task_from_id(TASK_ID_RDTSS),
241                                                         TASK_APP,
242                                                         rdtss_val_ntf_ind_req,
243                                                         length);
244
245     req->conidx = app_env.conidx;
246     req->notification = true;
247     req->handle = RDTSS_IDX_NTF_VAL; //通知特征值，从机该特征值向主机发送数据
248     req->length = length;
249     memcpy(&req->value[0], data, length);
250
251     ke_msg_send(req);
252 }
    
```

NATIONS

6.3 主机接收从机上发的数据

主机接收蓝牙数据回调函数：rdtsc_data_rx_ind_handler()

```

226 * @param[in] src_id ID of the sending task instance.
227 *
228 * @return If the message was consumed or not.
229 *****
230
231 uint rdtsc_data_rx_ind_handler(ke_msg_id_t const msg_id,
232 struct rdtsc_data_rx_ind const *param,
233 ke_task_id_t const dest_id,
234 ke_task_id_t const src_id)
235 {
236     NS_LOG_DEBUG("%s\r\n", __func__);
237     NS_LOG_INFO("BLE Data was received\r\n");
238     NS_LOG_DEBUG("length = %x\r\n", param->length);
239     NS_LOG_DEBUG("handle = %x\r\n", param->handle);
240     NS_LOG_DEBUG("type = %x\r\n", param->type);
241     NS_LOG_DEBUG("data ");
242     for(uint8_t i=0; i<param->length; i++)
243     {
244         NS_LOG_DEBUG(" %x", param->data[i]);
245     }
246     NS_LOG_DEBUG("\r\n");
247     app_usart_tx_fifo_enter(param->data,param->length);
248
249     return (KE_MSG_CONSUMED);
250 }
251
252
253 // Default State handlers definition 默认状态处理 //conformation: 构象、结构
254 const struct ke_msg_handler app_rdtsc_msg_handler_list[] =
255 {
256     (RDTSCE_ENABLE_CFM, (ke_msg_func_t)rdtsc_enable_cfm_handler),
257     (RDTSCE_DATA_TX_CFM, (ke_msg_func_t)rdtsc_data_tx_cfm_handler), //主机向从机发送数据回调函数
258     (RDTSCE_DATA_RX_IND, (ke_msg_func_t)rdtsc_data_rx_ind_handler), //主机接收从机数据回调函数
259 };
260
261 const struct app_subtask_handlers app_rdtsc_handlers = APP_HANDLERS(app_rdtsc);
262
263
264
265 #endif //BLE_RDTSCE_CLIENT
    
```

1、当主机接收到来自蓝牙从机的数据时，就会自动调用此回调函数进行蓝牙数据的接收，并且把数据存入 usart_tx_fifo_buf[]中

把接收到的数据复制到 usart_tx_fifo_buf[]中：app_usart_tx_fifo_enter(), 并创建定时器任务发送数据。

```

382 * @brief usart tx fifo enter and active dma send out
383 *
384 * @param[in] p_data: usart tx data
385 * @param[in] len: usart tx data length
386 *
387 * @return: usart tx data length
388 *
389 * @note: usart tx fifo is full, drop the rest data
390 *
391 * @note: usart tx fifo is full, drop the rest data
392 *
393 * @note: usart tx fifo is full, drop the rest data
394 *
395 * @note: usart tx fifo is full, drop the rest data
396 *
397 * @note: usart tx fifo is full, drop the rest data
398 *
399 * @note: usart tx fifo is full, drop the rest data
400 *
401 * @note: usart tx fifo is full, drop the rest data
402 *
403 * @note: usart tx fifo is full, drop the rest data
404 *
405 * @note: usart tx fifo is full, drop the rest data
406 *
407 * @note: usart tx fifo is full, drop the rest data
408 *
409 * @note: usart tx fifo is full, drop the rest data
410 *
411 * @note: usart tx fifo is full, drop the rest data
412 *
413 * @note: usart tx fifo is full, drop the rest data
414 *
415 * @note: usart tx fifo is full, drop the rest data
416 *
417 * @note: usart tx fifo is full, drop the rest data
418 *
419 * @note: usart tx fifo is full, drop the rest data
420 *
421 * @note: usart tx fifo is full, drop the rest data
422 *
423 * @note: usart tx fifo is full, drop the rest data
424 *
425 * @note: usart tx fifo is full, drop the rest data
426 *
427 * @note: usart tx fifo is full, drop the rest data
428 *
429 * @note: usart tx fifo is full, drop the rest data
430 *
431 * @note: usart tx fifo is full, drop the rest data
432 *
433 * @note: usart tx fifo is full, drop the rest data
434 *
435 * @note: usart tx fifo is full, drop the rest data
436 *
437 * @note: usart tx fifo is full, drop the rest data
438 *
439 * @note: usart tx fifo is full, drop the rest data
440 *
441 * @note: usart tx fifo is full, drop the rest data
442 *
443 * @note: usart tx fifo is full, drop the rest data
444 *
445 * @note: usart tx fifo is full, drop the rest data
446 *
447 * @note: usart tx fifo is full, drop the rest data
448 *
449 * @note: usart tx fifo is full, drop the rest data
450 *
451 * @note: usart tx fifo is full, drop the rest data
452 *
453 * @note: usart tx fifo is full, drop the rest data
454 *
455 * @note: usart tx fifo is full, drop the rest data
456 *
457 * @note: usart tx fifo is full, drop the rest data
458 *
459 * @note: usart tx fifo is full, drop the rest data
460 *
461 * @note: usart tx fifo is full, drop the rest data
462 *
463 * @note: usart tx fifo is full, drop the rest data
464 *
465 * @note: usart tx fifo is full, drop the rest data
466 *
467 * @note: usart tx fifo is full, drop the rest data
468 *
469 * @note: usart tx fifo is full, drop the rest data
470 *
471 * @note: usart tx fifo is full, drop the rest data
472 *
473 * @note: usart tx fifo is full, drop the rest data
    
```

2、把蓝牙接收到的数据复制到usart_tx_fifo_buf[]中，并且在蓝牙接收数据断流50ms之后，激活串口向主机host发送数据

6.4 主机把数据发送给主机 host

创建一个事件处理定时器，定时处理指定的事件：ke_timer_set(APP_CUSTS_TEST_EVT, TASK_APP, 50)

用户事件标志：APP_CUSTS_TEST_EVT

用户信息处理函数：app_user_msg_handler()

```

257  * @note
258  */
259  void app_ble_init(void)
260  {
261
262      struct ns_stack_cfg_t app_handler = {0};
263      app_handler.ble_msg_handler = app_ble_msg_handler;
264      app_handler.user_msg_handler = app_user_msg_handler;
265      //initialization ble stack
266      ns_ble_stack_init(&app_handler);
267
268      app_ble_gap_params_init();
269      app_ble_sec_init();
270      app_ble_scan_init();
271      app_ble_prf_init();
272      //start scan
273      ns_ble_start_scan();
274  }
275
276  /**
277   * @brief ble connected
278   * @param
279   * @return
280   * @note
    
```

初始化用户信息处理函数

在蓝牙接收数据断流 50ms 之后，调用用户信息处理回调函数：app_user_msg_handler()

```

61  * @param
62  * @return
63  * @note
64  */
65  void app_user_msg_handler(ke_msg_id_t const msgid, void const *p_param)
66  {
67
68      switch (msgid)
69      {
70          case APP_UART_TX_EVT:
71              app_usart_tx_process();
72              break;
73          case :
74              break;
75          default:
76              break;
77      }
78  }
79
80  }
81
    
```

1、主机在蓝牙接收完数据之后，会自动调用此函数把蓝牙接收到的数据通过串口发送到其host

把数据通过串口发送给其主机：app_usart_tx_process()

```

433  */
434  void app_usart_tx_process(void)
435  {
436      uint32_t in_temp, len;
437      uint8_t *p_data;
438
439      in_temp = usart_tx_fifo_in;
440      if(usart_tx_fifo_out < in_temp)
441      {
442          len = in_temp-usart_tx_fifo_out;
443          p_data = @usart_tx_fifo_buf[in_temp-usart_tx_fifo_out];
444          if(usart_tx_dma_send(p_data, len) == true)
445          {
446              usart_tx_fifo_out = in_temp;
447          }
448      }
449      else if(usart_tx_fifo_out > in_temp)
450      {
451          len = USART_TX_FIFO_SIZE-usart_tx_fifo_out;
452          p_data = @usart_tx_fifo_buf[in_temp-usart_tx_fifo_out];
453          if(usart_tx_dma_send(p_data, len) == true)
454          {
455              usart_tx_fifo_out = 0;
456          }
457      }
458  }
459
460  }
461
    
```

2、把数据通过串口DMA传输到主机的host

注意事项

主从机收发数据接口：

主机 rdtsc 通过蓝牙下发数据给从机 rdtss 的接口：user_send_ble_data()，主机调用该函数给从机发数据；

主机 rdtsc 接收从机 rdtss 上发的蓝牙数据接口：rdtsc_data_rx_ind_handler()，主机在此函数中接收从机数据；

从机 rdtss 通过蓝牙上发数据给主机 rdtsc 的接口：rdtss_send_notify()，从机调用该函数给主机发数据；

从机 rdtss 接收主机 rdtsc 下发的蓝牙数据接口：rdtss_val_write_ind_handler()，从机在此函数中接收主机数据。

NATION'S CONFIDENTIAL

历史版本

版本	日期	备注
V1.0	2022.10.17	新建文档

NATION'S CONFIDENTIAL

声明

国民技术股份有限公司（以下简称国民技术）保有在不事先通知而修改这份文档的权利。国民技术认为提供的信息是准确可信的。尽管这样，国民技术对文档中可能出现的错误不承担任何责任。在购买前请联系国民技术获取该器件说明的最新版本。对于使用该器件引起的专利纠纷及第三方侵权国民技术不承担任何责任。另外，国民技术的产品不建议应用于生命相关的设备和系统，在使用该器件中因为设备或系统运转失灵而导致的损失国民技术不承担任何责任。国民技术对本手册拥有版权等知识产权，受法律保护。未经国民技术许可，任何单位及个人不得以任何方式或理由对本手册进行使用、复制、修改、抄录、传播等。

NATIONS CONFIDENTIAL