LWNS 开发指南

更新日期：2021 年 12 月 15 日
文档版本：V06

LWNS 开发指南

LWNS 协议栈，全称为 Lightweight Wireless Networking Stack，为 LLNs 和 WSN 等场景下使用的轻量无线组网协议栈。可广泛应用于智能家居、智能照明、智能工厂、智能商超等领域。

具有广播、组播、单播、可靠单播，可靠单播文件传输、网络泛洪、mesh 等功能。

1. 协议栈移植接口函数讲解

本章节将讲解如何将 LWNS 库移植到 CH573 上。用户可以根据讲解，自行修改相关函数代码，实现不同的功能。

本库只是 net 层的协议栈方案， mac 层协议用户可以自己根据实际情况，进行编写。沁恒微电子提供参考方案，用户可以使用提供的模板程序，也可以自己参考进行编写：

模板工程中的 **lwns_adapter_no_mac.c** 提供了不使用 mac 层协议的模板，无需使用邻居表过滤数据。

模板工程中的 **lwns_adapter_csma_mac.c** 提供了模仿 IEEE 802.15.4 的 csma mac 层协议，发送遇到冲突就执行退避，必须使用邻居表管理，过滤接收到的数据。

模板工程中的 **lwns_adapter_blemesh_mac.c** 提供了模仿蓝牙 mesh 的 mac 层协议，在三个 channel 内进行交替发送和接收，必须使用邻居表管理，过滤接收到的数据。

这三个 mac 层协议各有优劣点，用户结合使用场景，自行修改宏定义进行配置，一般情况下推荐使用 **lwns_adapter_blemesh_mac.c**。

在更改工程中使用的 mac 层协议时，需要注意三个头文件的宏定义 LWNS_USE_CSMA_MAC、LWNS_USE_BLEMESH_MAC 和 LWNS_USE_NO_MAC 只可以有一个为 1，其他两个都必须为 0，否则编译不会通过。

注意：头文件中除了 HTIMER_CLOCK_SECONDS 宏定义可以根据情况修改，其他宏定义都不可以修改，否则会出现不可预料的 BUG。本库所有函数的时间参数都是以 HTIMER_CLOCK_SECONDS 为基准进行计算。

通过学习本协议栈，可以对无线传输有初步认识，你可以结合本协议栈提供的思路，自行编写实现部分功能的无线协议栈。

使用沁恒微电子提供的 CH573/CH579 程序模板的话，可以不看本章节。从第二章开始学习如何使用 LWNS 库函数。

1.1 发送函数接口

在结构体 lwns_fuc_interface_t 的定义中，有一个函数接口，为：

BOOL (*lwns_phy_output)(uint8_t *dataptr, uint8_t len);

dataptr 为发送数据缓冲区的指针，len 为需要发送的数据长度，单位为 byte。即需要发送的数据就是从 dataptr 读取的 len 个字节。dataptr 数据区后**预留有 2 字节空间**供用户添加校验等数据，用户可直接使用 **dataptr[len+1]=x;** 进行赋值，在接收中校验。

发送成功返回 TRUE，失败返回 FALSE。发送失败就不会调用发送完成回调函数。

LWNS 库每次发送的长度最少为头文件 WCH_LWNS_LIB.h 中定义的 **LWNS_PHY_OUTPUT_MIN**

_SIZE，目前为 9 个字节。这 9 个字节包括发送者地址 6 字节，mac 层序号 1 字节，端口号 1 字节，包头信息长度 1 字节。所以在接收函数中需要注意判断数据长度。

发送函数最好是异步进行，即发送接口函数中，通过信号量、发送邮件等操作，将数据发送出来，在外部进行发送处理。例如 CH573，提供的模拟 csma mac 的程序模板中：

CH573 的发送功能由 RF_Tx 函数实现，而且由于 CH573 为蓝牙芯片，采用 802.15.1 协议，所以本身不具备 csma/ca 功能，所以不可以采用立刻发送的方式。

采用**模拟 csma/ca 防冲突**的发送方式：每次发送前将随机等待，如果在等待期间有接收到数据包，则取消本次发送。在下一次周期性发送检测中，重新置位发送。同时为了避免一直被延迟，当延迟次数过多时，不再等待，立刻发送。

每次发送将多次重发数据，保证接收方可以接收到数据，同时接收方将通过邻居表来过滤接收到的来自一个邻居节点的多次重发数据，再将过滤后的数据交由协议栈下一层处理。

```
static BOOL ble_phy_output(u8 * dataptr, uint8_t len) {
  u8 *pMsg, i;
  struct csma_mac_phy_manage_struct *p;
  for (i = 0; i < LWNS_MAC_SEND_PACKET_MAX_NUM; i++) {
    if (csma_phy_manage_list[i].data == NULL) {
      break; //寻找到了一个空的结构体可以使用。
    } else {
      if (i == (LWNS_MAC_SEND_PACKET_MAX_NUM - 1)) {
        PRINTF("send failed!\n"); //列表满了，发送失败，直接返回。
        return FALSE;
      }
    }
  }
#if LWNS_ENCRYPT_ENABLE
  pMsg = tmos_msg_allocate((((len + 1 + 15) & 0xf0) + 1 + 1)); //校验位1位加上后再进行16字节对齐，存储发送长度+1，真实数据长度+1
#else
  pMsg = tmos_msg_allocate(len + 1); //申请内存空间存储消息，存储发送长度+1
#endif
  if (pMsg != NULL) {//成功申请
    p = csma_phy_manage_list_head;
    if (p != NULL) {
      while (p->next != NULL) {//寻找发送链表的终点
        p = p->next;
      }
    }
#if LWNS_ENCRYPT_ENABLE
    //lwns buffer内部预留有两字节，用户可直接使用dataptr[len]进行赋值两字节内容
    dataptr[len] = dataptr[len - 1] ^ len;//校验字节仅取最后一个字节和长度进行异或运算，首字节相同port是一样的，可能有影响。和校验比较浪费时间，所以不采用
    pMsg[1] = len;//真实数据长度占一字节，不加密，用来接收做第一步校验
    pMsg[0] = lwns_msg_encrypt(dataptr, pMsg + 2, len + 1) + 1;//获取数据加密后的长度，也就是需要发送出去的字节数，真实数据长度不加密
#else
    pMsg[0] = len;
    tmos_memcpy(pMsg + 1, dataptr, len);
#endif
    if (csma_phy_manage_list_head != NULL) {
      p->next = &csma_phy_manage_list[i];     //链表添加尾结点
    } else {
      csma_phy_manage_list_head = &csma_phy_manage_list[i];     //链表为空，则节点作为头结点
    }
    csma_phy_manage_list[i].data = pMsg;     //绑定消息
    csma_phy_manage_list[i].next = NULL;
    return TRUE;
  } else {
    PRINTF("send failed!\n");//无法申请到内存，则无法发送
  }
  return FALSE;
}
```

启用模板中的加密功能需要将 LWNS_ENCRYPT_ENABLE 宏定义置为 1 即可。秘钥等设置都在 lwns_sec.c 中。

然后在 tmos 给 LWNS 设置的周期性检测任务中，检测 htimer 更新以后，再检测到了列表中有需要发送的数据，就会进行发送准备工作：随机延迟，置位发送状态：

```
if (events & LWNS_PHY_PERIOD_EVT) {
  lwns_htimer_update(); //htimer的更新需要和mac的phy管理放一起，保持一致。
  if ((csma_phy_manage_list_head != NULL)) { //有需要发送的包
    if((ble_phy_manage_state == BLE_PHY_MANAGE_STATE_FREE) || (ble_phy_manage_state == BLE_PHY_MANAGE_STATE_RECEIVED)){//当前不在发送过程中
      if (ble_phy_manage_state == BLE_PHY_MANAGE_STATE_RECEIVED) { //当前周期发送碰撞，延迟发送
        ble_phy_wait_cnt++; //记录发送延迟次数
      } else {//BLE_PHY_MANAGE_STATE_FREE
        ble_phy_send_cnt = 0;//清除计数
        ble_phy_wait_cnt = 0;//清除计数
      }
      ble_phy_manage_state = BLE_PHY_MANAGE_STATE_WAIT_SEND; //设置为等待发送状态
      if (ble_phy_wait_cnt >= LWNS_MAC_SEND_DELAY_MAX_TIMES) { //发送被取消次数，延迟过多，不再随机等待，立刻开始发送
        tmos_set_event(lwns_phyoutput_taskid, LWNS_PHY_OUTPUT_EVT);
        PRINTF("too many delay\n");
      } else {
        u8 rand_delay;
        rand_delay = tmos_rand() % LWNS_MAC_SEND_DELAY_MAX_625US + BLE_PHY_ONE_PACKET_MAX_625US; //随机延迟，防止冲突，随机延迟等待周期里收到
        tmos_start_task(lwns_phyoutput_taskid, LWNS_PHY_OUTPUT_EVT, rand_delay);
        PRINTF("rand send:%d\n", rand_delay);
      }
    }
  }
  tmos_start_task(lwns_phyoutput_taskid, LWNS_PHY_PERIOD_EVT, MS1_TO_SYSTEM_TIME(LWNS_MAC_PERIOD_MS)); //每个周期毫秒检测一次有误数据要发送，
  return (events ^ LWNS_PHY_PERIOD_EVT);
}
```

当随机延迟结束后，在随机延迟期间没有收到其他邻居节点的数据包，开始发送数据包：

```
if (events & LWNS_PHY_OUTPUT_EVT) {//发送任务，竞争发送成功
  if (ble_phy_manage_state == BLE_PHY_MANAGE_STATE_WAIT_SEND){
    ble_phy_manage_state = BLE_PHY_MANAGE_STATE_SENDING; //改为发送中状态，竞争包发送完毕，需要等待一下接收方做好准备工作
    tmos_clear_event(lwns_adapter_taskid, LWNS_PHY_RX_OPEN_EVT);//停止可能已经置位的、可能会打开接收的任务
  }
  RF_Shut();
  RF_Tx((u8 *) (csma_phy_manage_list_head->data + 1),
              csma_phy_manage_list_head->data[0], USER_RF_RX_TX_TYPE,
              USER_RF_RX_TX_TYPE);
  tmos_start_task(lwns_phyoutput_taskid, LWNS_PHY_OUTPUT_FINISH_EVT, MS1_TO_SYSTEM_TIME(LWNS_PHY_OUTPUT_TIMEOUT_MS));//开始发送超时计数
  return (events ^ LWNS_PHY_OUTPUT_EVT);
}
if (events & LWNS_PHY_OUTPUT_FINISH_EVT) {//发送完成任务
  ble_phy_send_cnt++;//发送计数
  if (ble_phy_send_cnt < LWNS_MAC_TRANSMIT_TIMES) {//发送没结束
    tmos_set_event(lwns_phyoutput_taskid, LWNS_PHY_OUTPUT_EVT);//发送次数没结束，继续发送
  } else {//发送流程结束
    ble_phy_manage_state = BLE_PHY_MANAGE_STATE_FREE;   //清除状态
    RF_Shut();
    RF_Rx(NULL, 0, USER_RF_RX_TX_TYPE, USER_RF_RX_TX_TYPE); //重新打开接收
    tmos_msg_deallocate(csma_phy_manage_list_head->data); //释放内存
    csma_phy_manage_list_head->data = NULL; //恢复默认参数
    csma_phy_manage_list_head = csma_phy_manage_list_head->next; //链表pop，去除掉首元素
  }
  return (events ^ LWNS_PHY_OUTPUT_FINISH_EVT);
}
```

发送完成还需要在 CH573 的 rf 发送回调中置位发送完成任务，以便接下来的处理。

```
case TX_MODE_TX_FINISH:
case TX_MODE_TX_FAIL:
  tmos_stop_task(lwns_phyoutput_taskid, LWNS_PHY_OUTPUT_FINISH_EVT);//停止超时计数
  tmos_set_event(lwns_phyoutput_taskid, LWNS_PHY_OUTPUT_FINISH_EVT);//进入发送完成处理
  break;
```

最后还需要将该函数指针赋予给 lwns_fuc_interface_t 定义出的结构体中的 lwns_phy_output 成员变量即可。

其他两种 mac 协议用户可以自行学习查看。

1.2 接收函数移植

库中接收数据的函数分为两部分：

1. void lwns_input(uint8_t *rxBuf, uint8_t len);
2. void lwns_dataHandler(void);

lwns_input 函数功能为将数据拷贝到内部缓冲区。

lwns_dataHandler 函数功能为调用协议栈处理缓冲区数据。

用户在移植时需要在将芯片收到的数据通过调用上述两个函数进行输入到内部处理。

例如在 CH573 中，函数接收为 rf 初始化时的回调函数，为了避免长时间占用回调函数，在回调函数中先调用通过 tmos_send_msg 把数据作为消息发送出来：

```
case RX_MODE_RX_DATA: {
  if (crc == 1) {...
  } else if (crc == 2) {...
  } else {
    u8 *pMsg;
#if LWNS_ENCRYPT_ENABLE//是否启用消息加密，采用aes128，为硬件实现
    if (((rxBuf[1] % 16) == 1) && (rxBuf[1] >= 17) && (rxBuf[1] > rxBuf[2])) {//对齐后数据区最少16个字节，加上真实数据长度一字节
      //长度校验通过，所以rxBuf[1] - 1必为16的倍数
      pMsg = tmos_msg_allocate(rxBuf[1]);//申请内存空间，真实数据长度不需要解密
      if (pMsg != NULL) {
        //发送接收到的数据到接收进程中
        lwns_msg_decrypt(rxBuf + 3, pMsg + 1, rxBuf[1] - 1);
        //解密成功
        if((rxBuf[2] ^ pMsg[rxBuf[2]]) == pMsg[rxBuf[2]+1]){
          //异或校验通过
          pMsg[0] = rxBuf[2];
          PRINTF("send rx msg\n");
          tmos_msg_send(lwns_adapter_taskid, pMsg);
        } else {
          //异或校验失败
          PRINTF("verify rx msg err\n");
          tmos_msg_deallocate(pMsg);
        }
      } else {
        //申请内存失败，无法发送接收到的数据
        PRINTF("send rx msg failed\n");
      }
    } else {
      PRINTF("bad len\n");//包长度不对
    }
#else...
    //当接收到一个数据包时，将本时间段内的还未开始的发送任务停止，等待下一时间段发送，已经开始的发送任务不暂停，模仿csma/ca，进行防碰撞相关检测。
    if (ch573_phy_manage_state == CH573_PHY_MANAGE_STATE_WAIT_SEND) { //等待发送状态中收到数据包
      PRINTF("send delay\n");
      ch573_phy_manage_state = CH573_PHY_MANAGE_STATE_RECEIVED; //等待发送周期内收到了数据包状态
      tmos_stop_task(lwns_phyoutput_taskid, LWNS_PHY_OUTPUT_EVT); //竞争发送权限失败，放弃自己的发送任务。
      tmos_clear_event(lwns_phyoutput_taskid, LWNS_PHY_OUTPUT_EVT);//放弃自己的发送任务
    }
  }
  tmos_set_event(lwns_adapter_taskid, LWNS_PHY_RX_OPEN_EVT);
  break;
}
```

因为协议栈内部当前数据处理和发送都是采用的同一个缓冲区，所以编写程序时应避免在 lwns_input 和 lwns_dataHandler 函数调用间隔期间发送数据，否则会导致数据丢失。所以 CH573 任务数据接收处理编写如下：

```
static uint16 lwns_adapter_ProcessEvent(uint8 task_id, uint16 events) {
  if (events & LWNS_PHY_RX_OPEN_EVT) {
    RF_Shut();
    RF_Rx(NULL, 0, USER_RF_RX_TX_TYPE, USER_RF_RX_TX_TYPE); //重新打开接收
    return (events ^ LWNS_PHY_RX_OPEN_EVT);
  }
  if (events & SYS_EVENT_MSG) {
    uint8 *pMsg;
    if ((pMsg = tmos_msg_receive(lwns_adapter_taskid)) != NULL) {
      // Release the TMOS message,tmos_msg_allocate
      lwns_input(pMsg + 1, pMsg[0]); //将数据存入协议栈缓冲区
      tmos_msg_deallocate(pMsg); //先释放内存，在数据处理前释放，防止数据处理中需要发送数据，而内存不够。
      lwns_dataHandler(); //调用协议栈处理数据函数
    }
    // return unprocessed events
    return (events ^ SYS_EVENT_MSG);
  }
  // Discard unknown events
  return 0;
}
```

在任务接收处理中，调用 lwns_input 函数将数据拷贝到库内部缓冲区，先释放掉占用 tmos 的消息缓存，然后调用 lwns_dataHandler 处理数据。先释放再处理数据，就可以防止数据处理中需要发送数据，而申请内存时内存不足情况的发送。

1.3 memcpy 接口

该接口提供内存拷贝函数给库内部使用，不可为空，必须进行设置。将该函数指针赋予给 lwns_fuc_interface_t 定义的结构体中的 lwns_memcpy 成员变量即可。

1.4 memcmp 接口

该接口提供内存拷贝函数给库内部使用，不可为空，必须进行设置。将该函数指针赋予给 lwns_fuc_interface_t 定义的结构体中的 lwns_memcmp 成员变量即可。

函数接口需要实现的功能是两块内存长度数据比较相同返回 TRUE，不同返回 FALSE，和 string.h 中的 memcmp 不同，用户移植时需要注意。

1.5 memset 接口

该接口提供内存拷贝函数给库内部使用，不可为空，必须进行设置。将该函数指针赋予给 lwns_fuc_interface_t 定义的结构体中的 lwns_memset 成员变量即可。

1.6 rand 接口

该接口需要用户提供一个产生随机数的函数。应用于库函数内部的随机时间发送。不可设置为空。将该函数指针赋予给 lwns_fuc_interface_t 定义的结构体中的 lwns_rand 成员变量即可。

1.7 htimer 移植

htimer 为 heart timer 的简称，即为 lwns 内部超时重发、自动重发等计算时间的心跳时钟。

htimer 移植只需要用户实现一个定时器，在定时器函数中调用定时器处理函数：void lwns_htimer_update(void); 即可。

根据提供的定时器时钟周期，修改 WCH_LWNS_LIB.h 中的 HTIMER_SECOND_NUM 的宏定义即可。

例如：在采用模拟 csma 的防碰撞 mac 协议的工程中，HTIMER_SECOND_NUM 的宏定义为 50，则 htimer 一秒钟需要运行 50 次。即，htimer 心跳为（1000ms/50）= 20 毫秒。用户需要自行实现一个 20ms 的周期性定时器即可。

当使用 mac 层协议和邻居表管理重发等操作时，htimer 心跳周期应大于等于 mac 层的发送周期为佳，目前 CH573 中使用 20ms 作为一个心跳周期和 mac 发送周期，所以将 HTIMER_SECOND_NUM 定义为（1000/20）=50。

同时也将 void lwns_htimer_update(void);函数放入周期性发送检测任务中同时处理：

```
if (events & LWNS_PHY_PERIOD_EVT) {
  lwns_htimer_update(); //htimer的更新需要和mac的phy管理放一起，保持一致。
  if (phy_manage_list_ptr != NULL) { //有需要发送的包
    if ((phy_manage_list_ptr->data != NULL)) { //数据包参数合法
```

1.8 数据大小

LWNS 发送支持的数据大小，在 WCH_LWNS_LIB.h 定义为：

```
//can not be modified
#define LWNS_DATA_SIZE       200
#define LWNS_HEADER_SIZE      36
```

即数据大小最大为 200 个字节。

用户在发送数据时，必须注意不可以超过最大数据长度，即 200 个字节。多余的数据会自动删除，协议栈底层不会自动分包。

接收数据的时候，最大长度为包头长度和数据长度相加，即为 236 个字节，当调用库接收函数时，如果数据长度超过 236 个字节，超出部分会被删除，不会存入协议栈缓冲区。

1.9 数据缓冲区 qbuf 定义

qbuf 缓冲区作用为自动重发缓冲、等待延后发送缓冲、netflood 转发临时缓冲，mesh 临时缓冲等所用。

用户根据自己使用模块的数量来定义 qbuf 缓冲区数量，模块在发送过程中缓冲区不足可能会导致丢包等现象。

```
//can not be modified
#define LWNS_QBUF_LIST_U8_SIZE   273
typedef struct _lwns_qbuf_ptr_struct {
   uint8_t data[LWNS_QBUF_LIST_U8_SIZE];
} lwns_qbuf_list_t; //for user manual allocate
```

由于数据缓冲区(qbuf)使用内存挺大的，而且没有变动的需求，所以直接定义为数组，将数组指针传递给 lwns 内部使用。

每个功能模块占用的 qbuf 单位数量不同：

mesh 最多使用 3 个 qbuf 单位。

ruc 和 rucft 固定使用 1 个 qbuf 单位。

netflood、uninetflood 和 multinetflood 最多使用 2 个 qbuf 单位。

broadcast、multicast 和 unicast 不需要使用 qbuf 单位。

一个 qbuf 单位的内存占用内存空间可查看在 WCH_LWNS_LIB.h 中定义的 LWNS_QBUF_UINT8_T_SIZE，通过计算，即可得出在使用中一个 qbuf 单位占用 LWNS_QBUF_UINT8_T_SIZE 个 1 字节内存空间。用户可以通过头文件中提供的 lwns_qbuf_list_t 快速定义一个内存空间，注意必须要四字节对齐：

```
//for lwns_packet_buffer save
//由于使用内存挺大的，而且一般情况下没有一直变动的需求，所以不使用动态内存，直接将数组指针传递给LWNS内部使用
#define QBUF_MANUAL_NUM  4
__attribute__((aligned(4)))  static lwns_qbuf_list_t qbuf_memp[QBUF_MANUAL_NUM];
```

1.10 邻居表内存空间定义

邻居表为必须提供的内存空间，用来过滤邻居节点重发的数据包。邻居表数量应为网络中一个节点最大可以收到数据包的其他节点的数量，一般设置为 8，具体需要看部署情况。

邻居表结构体定义可见 WCH_LWNS_LIB.h 中定义的 lwns_neighbor_info 结构体。用户可以通过头文件中提供的 lwns_neighbor_list_t 快速定义一个内存空间,供给库内部使用，注意必须要四字节对齐：

```
//for neighbor manage
//邻居表内存空间申请
#define NEIGHBOR_MANUAL_NUM  LWNS_NEIGHBOR_MAX_NUM
__attribute__((aligned(4)))  static lwns_neighbor_list_t neighbor_memp[NEIGHBOR_MANUAL_NUM];
```

1.11 路由表内存空间定义

路由表也需要提供内存空间进行使用。mesh 在使用前必须初始化路由表，不然 mesh 初始化不会成功。因为 mesh 模块在初始化的时候就会检查路由表，如果路由表非法，就会返回打开 mesh 失败。

路由表结构体定义可见 WCH_LWNS_LIB.h 中定义的 lwns_route_entry 结构体。一个路由表内存管理单位会占用 LWNS_ROUTE_ENTRY_UINT8_T_SIZE 个 1 字节。用户可以通过头文件中提供的 lwns_route_entry_data_t 快速定义一个内存空间，注意必须要四字节对齐：

```
//for lwns_route_entry manage
//路由表内存空间申请
#define ROUTE_ENTRY_MANUAL_NUM  32
#if ROUTE_ENTRY_MANUAL_NUM
__attribute__((aligned(4)))  static lwns_route_entry_data_t route_entry_memp[ROUTE_ENTRY_MANUAL_NUM];
#endif
```

1.12 lwns_lib_init

```
/*********************************************************************
 * @fn      lwns_lib_init
 *
 * @brief   Init lwns lib.
 *
 * input parameters
 *
 * @param   fuc - fuction interface of lwns lib. should define by lwns_fuc_interface_t.
 * @param   cfg - init config params of lwns lib. should define by lwns_config_t.
 *
 * output parameters
 *
 * @param   None.
 *
 * @return  0-success. error defined @ ERR_LWNS_LIB_INIT.
 */
extern int lwns_lib_init(void *fuc, void* cfg);
```

lwns 库的初始化就是由此函数进行。传递的第一个参数就是上述几小节讲述的函数接口，用户应使用头文件中提供的 lwns_fuc_interface_t 结构体定义一个结构体全局变量，并且将用户自行编写的接口函数地址在结构体变量初始化赋值进去：

```
//lwns必用的函数接口
static lwns_fuc_interface_t ch573_lwns_fuc_interface = {
    .lwns_phy_output = ch573_phy_output,
    .lwns_rand       = tmos_rand,
    .lwns_memcpy     = tmos_memcpy,
    .lwns_memcmp     = tmos_memcmp,
    .lwns_memset     = tmos_memset,
    .new_neighbor_callback = ch573_new_neighbor_callback,
};
```

函数接口中除了 new_neighbor_callback 回调函数接口，其他接口都是必须的，不然会出现 hardfault 错误。而 new_neighbor_callback 不使用的话，可以设置为 NULL。该函数为新增一个邻居信息时调用的回调函数。

初始化函数传递的第二个参数为一些内存空间和参数配置，用户应使用头文件中提供的 **lwns_config_t** 结构体定义一个结构体，并且赋值。该结构体无需定义为全局变量，采用临时变量。示例：

```
void lwns_init(void) {
    uint8_t s;
    lwns_config_t cfg;
    cfg.lwns_lib_name = (u8*) VER_LWNS_FILE; //验证函数库名称，防止版本出错
    cfg.qbuf_num = QBUF_MANUAL_NUM; //必须分配，至少1个内存单位，根据你程序中使用的端口数对应模块使用的qbuf单位来定义。
    cfg.qbuf_ptr = qbuf_memp; //mesh最多使用3个qbuf单位，(uni)netflood最多使用2个，其他模块都使用1个。
    cfg.routetable_num = ROUTE_ENTRY_MANUAL_NUM; //如果需要使用mesh，必须分配路由表内存空间。不然会进入hardfault。
#if ROUTE_ENTRY_MANUAL_NUM
    cfg.routetable_ptr = route_entry_memp; //如果需要使用mesh，必须分配。不然会进入hardfault。
#else
    cfg.routetable_ptr = NULL; //如果需要使用mesh，必须分配。不然会进入hardfault。
#endif
    cfg.neighbor_num = NEIGHBOR_MANUAL_NUM; //邻居表数量，必须分配
    cfg.neighbor_list_ptr = neighbor_memp; //邻居表内存空间
    cfg.neighbor_mod = LWNS_NEIGHBOR_AUTO_ADD_STATE_RECALL_ADDALL; //邻居表初始化默认管理模式为接收所有包，添加所有邻居并且过滤重复包的模式
#if LWNS_ADDR_USE_BLE_MAC
    GetMACAddress(cfg.addr.u8); //蓝牙硬件的mac地址
#else
//自行定义的地址
        uint8 MacAddr[6] = {0,0,0,0,0,1};
        tmos_memcpy(cfg.addr.u8, MacAddr, LWNS_ADDR_SIZE);
#endif
    s = lwns_lib_init(&ch573_lwns_fuc_interface, &cfg); //lwns库底层初始化
    if (s) {
        PRINTF("%s init err:%d\n", VER_LWNS_FILE, s);
    } else {
        PRINTF("%s init ok\n", VER_LWNS_FILE);
    }
```

在示例中，定义了一个 cfg 结构体。

cfg 结构体中，lwns_lib_name 指针型成员变量的值需要赋值为头文件中提供的 **VER_LWNS_FILE** 的指针，用户不可以修改这个宏定义，不然初始化不会成功。

cfg 结构体中，qbuf_num 为分配给协议栈使用的 qbuf 缓存单位数量，qbuf_ptr 为缓存区内存空间头指针。qbuf 相关定义和需求可见本章第 9 节。用户可根据自身需求，分配大小，最少为 1 个 qbuf 单位。

cfg 结构体中，routetable_num 为路由表数目。即一个节点自身缓存的路由表数量。对于普通节点来说，可能只会定时往中心主节点发送数据，所以仅需一个路由表即可。但是对于中心主节点，网络中存在多少个节点会给他发送数据包，路由表就需要多少个。路由表相关信息见本章第 11 节。

cfg 结构体中，neighbor_num 为分配给协议栈使用的邻居表缓存单位数量，neighbor_list_ptr 为缓存区内存空间头指针。neighbor 相关定义和需求可见本章第 10 节。用户可根据自身需求，分配大小，最少为 1 个邻居。

cfg 结构体中，neighbor_mod 为邻居表默认管理方式。具体功能和定义请见第三章第 2 节。

cfg 结构体中，addr 为需要设置的 lwns 地址。使用 **lwns_addr_t** 定义的 union 联合体，占用大小为 6 个字节。可以直接采用 mac 地址作为节点地址，也可以使用自定义地址。

返回值含义定义与枚举型变量 **ERR_LWNS_LIB_INIT** 中，为 0 则无错误，其他值，则初

始化失败。

2. lwns 常用函数讲解

在使用 lwns 库的时候，有一些必须要使用和了解的函数。这些函数都是必须先了解才方便后续的学习。

2.1 lwns_addr_set

```
/*********************************************************************
 * @fn      lwns_addr_set
 *
 * @brief   set lwns_addr with f.
 *
 * input parameters
 *
 * @param   f - pointer to lwns_addr need to set.
 *
 * output parameters
 *
 * @param   None.
 *
 * @return  None.
 *
 */
extern void lwns_addr_set(lwns_addr_t *f);
```

本库地址固定为 6 字节。**注意：不可以修改**！用户自行修改头文件中 **LWNS_ADDR_SIZE** 是无效的。

像 CH573 等蓝牙芯片，可以使用蓝牙芯片的 mac 地址。

用户可以使用自定义的地址，但是地址不可以设置为全 0，这是非法地址，函数会对参数进行检查，如果为全 0 地址，则不会设置库地址，直接返回。

如果在库初始化成功后，在使用中，有变动自身地址的需求，可以通过此函数设置修改自身的地址。

2.2 lwns_buffer_load_data

```
/*********************************************************************
 * @fn      lwns_buffer_load_data
 *
 * @brief   load data to lwns_buffer.
 *
 * input parameters
 *
 * @param   from - the data pointer need to be load to lwns_buffer.
 * @param   len - the length need to be load to lwns_buffer.
 *
 * output parameters
 *
 * @param   None.
 *
 * @return  the length of data be load to lwns_buffer actually.
 */
extern int lwns_buffer_load_data(const void *from, uint16_t len);
```

该函数为发送数据时，将需要发送的数据包载入协议栈内部缓冲区。第一个参数为数据头指针，第二个参数为数据的长度。

在本库中，所有数据的发送，都是需要先调用此函数，将数据载入缓冲区，然后再调用

各个模块的 send 函数进行发送。

2.3 lwns_buffer_save_data

```
/***************************************************************************
 * @fn        lwns_buffer_save_data
 *
 * @brief     save data from lwns_buffer to another buffer.
 *
 * input parameters
 *
 * @param     to - the data pointer need to be saved to from lwns_buffer.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    the length of data be saved actually.
 */
extern int lwns_buffer_save_data(void *to);
```

该函数的功能为在模块的接收回调中，读取协议栈内部缓冲区解析后除去包头信息的数据到用户自己的缓冲区的函数。

返回值为实际存储到用户自己的缓冲区的长度。

2.4 lwns_buffer_dataptr

```
/***************************************************************************
 * @fn        lwns_buffer_dataptr
 *
 * @brief     get lwns_buffer data pointer in lwns_buffer.
 *
 * input parameters
 *
 * @param     None.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    data pointer in buffer
 */
extern void *lwns_buffer_dataptr(void);
```

该函数的功能为在接收回调中，获取内部缓冲区中，除去包头信息的数据区域的起始指针。

用户不需要对数据进行其他操作的情况下，可以直接用此指针来解析数据，选择需要保存的数据保存下来。

该函数和 lwns_buffer_save_data 的区别在于，本函数没有数据拷贝的过程，直接给出缓冲区内的除去包头信息后的数据头指针。

2.5 lwns_buffer_datalen

```
/*****************************************************************************
 * @fn        lwns_buffer_datalen
 *
 * @brief     get data length in lwns_buffer.
 *
 * input parameters
 *
 * @param     None.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    data length in buffer
 */
extern uint16_t lwns_buffer_datalen(void);
```

该函数的功能为接收回调中，读取缓冲区中除去包头信息后数据的长度。返回值为数据区的长度。

2.6 lwns_buffer_clear

```
/*****************************************************************************
 * @fn        lwns_buffer_clear
 *
 * @brief     clear lwns_buffer data in lwns_buffer.
 *
 * input parameters
 *
 * @param     None.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    None.
 */
extern void lwns_buffer_clear(void);
```

清空 lwns 协议栈内部缓冲区的参数和数据。

2.7 lwns_buffer_set_datalen

```
/*********************************************************************************
 * @fn        lwns_buffer_set_datalen
 *
 * @brief     set datalen of lwns_buffer data in lwns_buffer.can not over LWNS_DATA_SIZE.
 *
 * input parameters
 *
 * @param     len - the length of buffer data.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    None.
 */
extern void lwns_buffer_set_datalen(uint8_t len);
```

设置缓冲区内数据长度，用户可先使用 lwns_buffer_clear 先清空数据区，然后再使用 lwns_buffer_dataptr 获取数据缓冲区指针。用该指针直接往缓冲区内填写数据。填写完数据后，需要使用本函数设置填写的数据长度。即完成和 lwns_buffer_load_data 函数一样的功能。

3. 邻居表管理

邻居表为 mac 层协议的管理实现。

因为无线信号会在一定区域内被区域内所有节点接收,所以区域内节点发送会产生冲突,导致数据出错，通信失败率高。

为了保证数据收发的成功率，每次都将数据包多次发送，再通过包序号和发送者地址判断是否为发送的重复包，mac 层将利用邻居表对数据包进行过滤，再传往协议栈下一层进行处理。

在本协议栈中，两个节点之间必须有打开相同的端口号，并且物理接口上可以通信，才会成为邻居，否则协议栈会将其接收到的数据包忽略。

3.1 lwns_neighbor_info 结构体

```
struct lwns_neighbor_info
{
  struct lwns_neighbor_info *next;
  lwns_addr_t sender;
  uint8_t seqno;
  uint8_t time;
};
```

邻居表，采用链表方式进行管理，所以会有 next 指针。

邻居表中存有邻居节点号地址，包序号，和时间参数。

每次节点发送数据包都会改变自己的 seqno，邻居表会将包序号相同的数据包全部拦截下来，保证底层协议的通讯不会重复。

3.2 lwns_neighbor_mode_set

```
/*********************************************************************************
 * @fn      lwns_neighbor_mode_set
 *
 * @brief   if set to LWNS_NEIGHBOR_AUTO_ADD_STATE_REC_TBONLY,this node will only receive messages from
 *          the neighbors in neighbor table memory,
 *          if a new neighbor send a packet to it, it will drop it.
 *          if set to LWNS_NEIGHBOR_AUTO_ADD_STATE_RECALL_ADDALL,this node will receive messages from all the neighbors,
 *          if a new neighbor send a packet to it, it will receive it,add it to neighbor table.
 *          then call new_neighbor_callback in lwns_fuc_interface_t.
 *          if set to LWNS_NEIGHBOR_AUTO_ADD_STATE_RECALL_NOTADD,this node will receive messages from all the neighbors,
 *          but not add it to neighbor table.so it will not pass duplicate packets.
 *
 * input parameters
 *
 * @param    status,uint8_t type,value is defined in LWNS_NEIGHBOR_AUTO_ADD_STATE_t.
 *
 * output parameters
 *
 * @param    None.
 *
 * @return   None.
 */
extern void lwns_neighbor_mode_set(uint8_t mode);
```

该函数设置邻居表管理方式，方便客户根据不同的情况进行更改。

该函数需要一个参数，为邻居表管理方式，定义如下：

```
typedef enum {
    LWNS_NEIGHBOR_AUTO_ADD_STATE_REC_TBONLY = 0,
    LWNS_NEIGHBOR_AUTO_ADD_STATE_RECALL_ADDALL,
    LWNS_NEIGHBOR_AUTO_ADD_STATE_RECALL_NOTADD,
} LWNS_NEIGHBOR_AUTO_ADD_STATE_t;
```

图中一共有三种管理方式：

LWNS_NEIGHBOR_AUTO_ADD_STATE_REC_TBONLY：

该方式为邻居表将只接收邻居表内现有的邻居节点的数据包,其他节点的数据包都会被过滤掉。适合某些组网操作完成后，将网内设备添加到邻居信息内后，再启用。那么来自其他不在邻居表内节点的数据包，都不会被传递到协议栈中处理。

LWNS_NEIGHBOR_AUTO_ADD_STATE_RECALL_ADDALL：

该方式为邻居表将会接收所有邻居节点的数据包，并将其添加到邻居表中，如果邻居表满了，就自动删除一个邻居表，再添加进去。

LWNS_NEIGHBOR_AUTO_ADD_STATE_RECALL_NOTADD：

该方式为邻居表将会接收所有邻居节点的数据包，并不将信息添加到邻居表中，所以也不会自动过滤重发的数据包，不推荐使用。即使发送接口只发一次，也还是推荐使用邻居表管理。

3.3 lwns_neighbor_lookup

```
/*********************************************************************************************
 * @fn      lwns_neighbor_lookup
 *
 * @brief   look up a neighbor info from neighbor table.
 *
 * input parameters
 *
 * @param   sender - the sender addr
 *
 * output parameters
 *
 * @param   None.
 *
 * @return  NULL if not find, other is the pointer to the neighbor info.
 */
extern struct lwns_neighbor_info *lwns_neighbor_lookup(const lwns_addr_t *sender);
```

该函数功能为从邻居表内查找一个邻居的邻居表信息。找到就会返回该邻居表信息结构体指针。

3.4 lwns_neighbor_add

```
/**************************************************************************
 * @fn        lwns_neighbor_add
 *
 * @brief     add a neighbor info to neighbor table.
 *
 * input parameters
 *
 * @param     sender - the sender addr.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    0 if success,other failed.
 */
extern int lwns_neighbor_add(const lwns_addr_t *sender);
```

该函数功能为添加一个邻居节点地址到邻居表内，添加后默认的 seqno 为 0。

3.5 lwns_neighbor_flush_all

```
/**************************************************************************
 * @fn        lwns_neighbor_flush_all
 *
 * @brief     remove all neighbors in neighbor table.
 *
 * input parameters
 *
 * @param     None.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    None.
 */
extern void lwns_neighbor_flush_all(void);
```

该函数功能为删除邻居表内所有的邻居信息。

3.6 lwns_neighbor_remove

```
/*****************************************************************************
 * @fn        lwns_neighbor_remove
 *
 * @brief     remove a neighbor in neighbor table.
 *
 * input parameters
 *
 * @param     e - lwns_neighbor_info pointer
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    None.
 */
extern void lwns_neighbor_remove(struct lwns_neighbor_info *e);
```

该函数功能为从邻居表内移除一个邻居信息。移除后邻居表将无法查询到该邻居信息。

3.7 lwns_neighbor_num

```
/*****************************************************************************
 * @fn        lwns_neighbor_num
 *
 * @brief     get neighbors number in memory.
 *
 * input parameters
 *
 * @param     None.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    neighbors number in memory.
 */
extern int lwns_neighbor_num(void);
```

该函数功能为获取邻居表内存中存储的邻居信息数量。

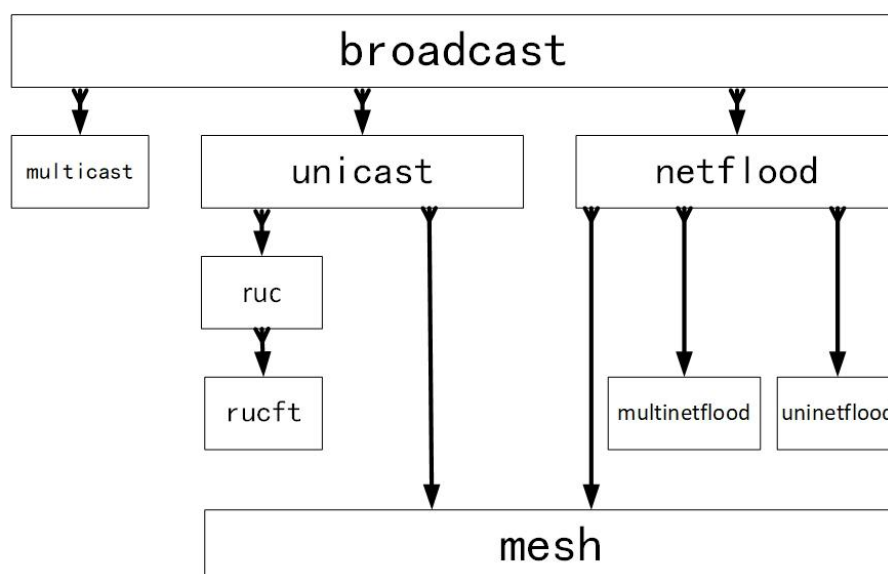
3.8 lwns_neighbor_get

```
/******************************************************************************
 * @fn        lwns_neighbor_get
 *
 * @brief     get a neighbor info from neighbor table.
 *
 * input parameters
 *
 * @param     num - which one from first in neighbor table.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    NULL if not find, other is the pointer to the neighbor info.
 */
extern struct lwns_neighbor_info *lwns_neighbor_get(uint16_t num);
```

该函数的功能为获取邻居表内第几个邻居信息的函数,如果参数 num 超出了目前邻居表内的邻居信息数量，那就会返回 NULL，否则如果找到，就会返回指定的邻居信息指针。

4. 广播

broadcast 为广播的意思，发送数据中包括了自身的地址，接收方通过解析可以得到发送方的地址。broadcast 是一切模块的根源，所有模块最后都是将包头信息和数据包封装后由 broadcast 模块将数据发送出去。



使用广播需要用 lwns_broadcast_controller 定义一块内存空间。

```
//can not be modified
#define LWNS_BROADCAST_CONTROLLER_U32_SIZE   4
typedef struct _lwns_broadcast_controller_struct {
  uint32_t data[LWNS_BROADCAST_CONTROLLER_U32_SIZE];
} lwns_broadcast_controller;
```

4.1 lwns_broadcast_callbacks

```
struct lwns_broadcast_callbacks {
  void (*recv)(lwns_controller_ptr ptr, const lwns_addr_t *sender);
  void (*sent)(lwns_controller_ptr ptr);
};
```

在本结构体中，有两个回调函数成员，一个是 sent，一个是 recv。

sent 回调函数会在调用发送后，立刻调用。表示发送完成回调函数。

recv 回调函数会在收到一个本端口的 broadcast 消息后调用，表示收到数据，和 abc 的接收回调函数不同，broadcast 的接收回调函数有一个 sender 的地址指针，用户可以从该指针读出发送方的地址。

两个回调函数的参数中都有 lwns_controller_ptr 指针,就是本次收到的数据所属的控制句柄，也就是定义出的内存空间的地址。用户可以通过 get_lwns_object_port 将该指针作为参数，读出当前数据所属的数据端口。

4.2 lwns_broadcast_init

```
/*********************************************************************************
 * @fn      lwns_broadcast_init
 *
 * @brief   open a port for broadcast.
 *
 * input parameters
 *
 * @param   h - lwns_controller_ptr,the pointer of a lwns_broadcast_controller
 * @param   port_num   -value:(1-255)  - port to recognize data
 * @param   callbacks defined with lwns_broadcast_callbacks
 *
 * output parameters
 *
 * @param   None.
 *
 * @return  return 1 if success,0 if failed.
 */
extern int lwns_broadcast_init(lwns_controller_ptr h, uint8_t port_num,
    const struct lwns_broadcast_callbacks *u);
```

所有类别的 lwns_controller 要想进行发送或接收，都必须经过相应功能的 init 函数进行注册初始化，不然必然出错。

在 lwns_broadcast_init 中，需要三个参数：

第一个参数为 lwns_controller_ptr 控制结构体指针，他不可以是 NULL，必须是一块可用内存的头指针，即 lwns_broadcast_controller 定义的内存空间地址。

第二个参数为 lwns_port 端口号，端口号为标识数据包到底属于哪一个控制结构体，是在数据包解析过程中的必要参数，所以不同结构体中，不可以使用相同的端口号，不然后打开的控制结构体会覆盖掉上次打开的结构体。

第三个参数为用户自己编写的回调函数结构体指针。可以设置为 NULL，但是如果用户想要进行数据的解析等操作，需要在接收回调函数中，进行数据的接收等操作。所以一般情况下，都会编写相应的回调函数。

4.3 lwns_broadcast_send

```
/*******************************************************************************
 * @fn        lwns_broadcast_send
 *
 * @brief     send a broadcst message through lwns_controller_ptr h.
 *
 * input parameters
 *
 * @param     h - lwns_controller_ptr,the pointer of a lwns_broadcast_controller,
 *            which must be initialized by lwns_broadcast_init.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    return 1 if success,0 if failed.
 */
extern int lwns_broadcast_send(lwns_controller_ptr h);
```

当用户需要发送一条广播消息时，就需要调用该函数。参数就是你当前需要发送数据的控制结构体。

调用之前需要先将需要发送的数据存到数据缓冲区，因为在协议栈内部，包的接收和发送处理都是采用的同一块内存缓冲区，所以我们需要使用函数将需要发送的数据载入内部缓冲区中：lwns_buffer_load_data。

然后再调用此函数，就会立刻在控制结构体参数的端口号上发送一条广播消息，发送方和接收方必须都打开同样端口号，才可以收到消息。

4.4 lwns_broadcast_close

```
/*******************************************************************************
 * @fn        lwns_broadcast_close
 *
 * @brief     close a port for broadcast.
 *
 * input parameters
 *
 * @param     h - lwns_controller_ptr,the pointer of a lwns_broadcast_controller,
 *            which must be initialized by lwns_broadcast_init.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    None.
 */
extern void lwns_broadcast_close(lwns_controller_ptr h);
```

当用户不需要再接收某个端口的消息时，就可以调用 close 函数将该端口关闭。关闭后，rf 接收到的数据解析的端口号匹配不到相应的控制结构体，所以就会被丢弃。

4.5 example

本章例程为 lwns_broadcast_example.c。

使用广播模块编写程序必须要做的事：

i. 使用 lwns_broadcast_controller 定义一块内存空间

```
static lwns_broadcast_controller broadcast;//广播控制结构体
```

ii. 编写相关回调函数

broadcast 模块有两个回调函数：recv 和 sent，并且需要把函数指针赋予回调函数结构体中。

接收回调函数：

```
static void broadcast_recv(lwns_controller_ptr ptr,
    const lwns_addr_t* sender) {
  uint8_t len;
  len = lwns_buffer_datalen(); //获取当前缓冲区接收到的数据长度
  if (len == 10) {
    lwns_buffer_save_data(RX_DATA); //接收数据到用户数据区域
    PRINTF("broadcast %d rec from %02x %02x %02x %02x %02x %02x\n",get_lwns_object_port(ptr),
        sender->v8[0], sender->v8[1], sender->v8[2], sender->v8[3],
        sender->v8[4], sender->v8[5]);//打印出本次消息的发送者地址
    PRINTF("data:");
    for (uint8_t i = 0; i < len; i++) {
      PRINTF("%02x ", RX_DATA[i]);//打印数据
    }
    PRINTF("\n");

  } else {
    PRINTF("data len err\n");
  }
}
```

发送完成回调函数：

```
static void broadcast_sent(lwns_controller_ptr ptr) {
  PRINTF("broadcast %d sent\n",get_lwns_object_port(ptr));//打印发送完成信息
}
```

声明回调函数结构体：

```
static const struct lwns_broadcast_callbacks broadcast_call = {
    broadcast_recv, broadcast_sent };//声明广播回调函数结构体，注册回调函数
```

准备工作都做好后,即可开始初始化相关工作,通过广播模块的 init 函数进行初始化：

```
void lwns_broadcast_process_init(void) {
  broadcast_taskID = TMOS_ProcessEventRegister(lwns_broadcast_ProcessEvent);
  lwns_broadcast_init(&broadcast, 136, &broadcast_call); //打开一个端口号为136的广播端口，注册回调函数
  tmos_start_task(broadcast_taskID, BROADCAST_EXAMPLE_TX_PERIOD_EVT,
      MS1_TO_SYSTEM_TIME(1000));//开始周期性广播任务
}
```

在 tmos 中的 BROADCAST_EXAMPLE_TX_PERIOD_EVT 任务处理中，进行周期性的发送：

```c
if (events & BROADCAST_EXAMPLE_TX_PERIOD_EVT) {
  uint8 temp;
  temp = TX_DATA[0];
  for (uint8 i = 0; i < 9; i++) {
    TX_DATA[i] = TX_DATA[i + 1];
  }
  TX_DATA[9] = temp;
  lwns_buffer_load_data(TX_DATA, sizeof(TX_DATA));//载入需要发送的数据到缓冲区
  lwns_broadcast_send(&broadcast);//广播发送数据
  tmos_start_task(broadcast_taskID, BROADCAST_EXAMPLE_TX_PERIOD_EVT,
      MS1_TO_SYSTEM_TIME(1000));//周期性发送

  return events ^ BROADCAST_EXAMPLE_TX_PERIOD_EVT;
}
```

将程序编译后烧录到两块板子里，实验现象可以通过串口观察，可以发现现象如下：

```
[09:36:45.486]收←◆broadcast 136 sent

[09:36:46.018]收←◆broadcast 136 rec from ab df 38
data:38 39 30 31 32 33 34 35 36 37

[09:36:46.486]收←◆broadcast 136 sent

[09:36:47.019]收←◆broadcast 136 rec from ab df 38
data:39 30 31 32 33 34 35 36 37 38

[09:36:47.486]收←◆broadcast 136 sent

[09:36:48.018]收←◆broadcast 136 rec from ab df 38
[09:36:48.039]收←◆e4 c2 84
data:30 31 32 33 34 35 36 37 38 39

[09:36:48.486]收←◆broadcast 136 sent

[09:36:49.018]收←◆broadcast 136 rec from ab df 38
data:31 32 33 34 35 36 37 38 39 30

[09:36:49.486]收←◆broadcast 136 sent

[09:36:50.017]收←◆broadcast 136 rec from ab df 38
data:32 33 34 35 36 37 38 39 30 31

[09:36:50.486]收←◆broadcast 136 sent
```

```
[09:36:45.488]收←◆broadcast 136 rec from d9 37 3c e4 c2 84
data:31 32 33 34 35 36 37 38 39 30

[09:36:46.016]收←◆broadcast 136 sent

[09:36:46.487]收←◆broadcast 136 rec from d9 37 3c e4 c2 84
data:32 33 34 35 36 37 38 39 30 31

[09:36:47.016]收←◆broadcast 136 sent

[09:36:47.490]收←◆broadcast 136 rec from d9 37 3c e4 c2 84
data:33 34 35 36 37 38 39 30 31 32

[09:36:48.016]收←◆broadcast 136 sent

[09:36:48.488]收←◆broadcast 136 rec from d9 37 3c e4 c2 84
data:34 35 36 37 38 39 30 31 32 33

[09:36:49.016]收←◆broadcast 136 sent

[09:36:49.488]收←◆broadcast 136 rec from d9 37 3c e4 c2 84
data:35 36 37 38 39 30 31 32 33 34

[09:36:50.016]收←◆broadcast 136 sent

[09:36:50.488]收←◆broadcast 136 rec from d9 37 3c e4 c2 84
data:36 37 38 39 30 31 32 33 34 35
```

a 节点发送数据包后，调用了自身的发送完成回调，打印了 broadcast 136 sent，136 就是当前的端口号，通过 get_lwns_object_port 获取到的端口号，b 节点就接收到了来自 a 节点的数据包，调用了编写的 broadcast_recv 回调函数，打印出了接收到的数据和发送方的地址。

b 节点和 a 节点现象相同。

5. 组播

multicast 为组播的意思，组播为发送数据到一个目标组，发送数据中包括了需要发往的订阅组地址。订阅组了相同地址的节点才会进入接收回调处理数据。

使用组播需要用 lwns_multicast_controller 定义一块内存空间。

```
//can not be modified
#define LWNS_MULTICAST_CONTROLLER_U32_SIZE   6
typedef struct _lwns_multicast_controller_struct {
    uint32_t data[LWNS_MULTICAST_CONTROLLER_U32_SIZE];
} lwns_multicast_controller;
```

5.1 lwns_multicast_callbacks

```
struct lwns_multicast_callbacks {
    void (*recv)(lwns_controller_ptr ptr,uint16_t subaddr, const lwns_addr_t *sender);
    void (*sent)(lwns_controller_ptr ptr);
};
```

multicast 有两个回调函数：

recv 为接收回调函数，其中 subaddr 为本次接收到的订阅组地址，只有是自己订阅组的地址才会接收，sender 为本次网络泛洪发起人的地址。

sent 函数为发送后调用，表明完成本次发送。

5.2 lwns_multicast_init

```
/*********************************************************************************
 * @fn        lwns_multicast_init
 *
 * @brief     open a port for multinetflood.
 *
 * input parameters
 *
 * @param     h - lwns_controller_ptr,the pointer of a lwns_multicast_controller.
 * @param     port_num   -value:(1-255)   - port to recognize data
 * @param     subaddr    - the array pointer of subscribe addresses.
 * @param     sub_num    - the number of subscribe addresses.
 * @param     callbacks defined with lwns_multicast_callbacks
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    return 1 if success,0 if failed.
 */
extern int lwns_multicast_init(lwns_controller_ptr h, uint8_t port_num,
    uint16_t *subaddr ,uint8_t sub_num, const struct lwns_multicast_callbacks *u);
```

所有类别的 lwns_controller 要想进行发送或接收，都必须经过相应功能的 init 函数进行注册初始化，不然必然出错。

在 lwns_multicast_init 中，需要五个参数：

第一个参数为 lwns_controller_ptr 控制结构体指针，他不可以是 NULL，必须是一块可用内存的头指针，即 lwns_multicast_controller 定义的内存空间地址。

第二个参数为 lwns_port 端口号，端口号为标识数据包到底属于哪一个控制结构体，是在数据包解析过程中的必要参数，所以不同结构体中，不可以使用相同的端口号，不然后打开的控制结构体会覆盖掉上次打开的结构体。

第三个参数 subaddr 为订阅组的地址起始指针。

第四个参数 sub_num 为订阅组的地址数量。

第五个参数为用户自己编写的回调函数结构体指针。可以设置为 NULL，但是如果用户想要进行数据的解析等操作，需要在接收回调函数中，进行数据的接收等操作。所以一般情况下，都会编写相应的回调函数。

5.3 lwns_multicast_send

```
/*********************************************************************
 * @fn      lwns_multicast_send
 *
 * @brief   send a multicast message to a subaddr. used lwns_controller_ptr h.
 *
 * input parameters
 *
 * @param   h - lwns_controller_ptr,the pointer of a lwns_multicast_controller,
 *          which must be initialized by lwns_multicast_init.
 * @param   subaddr - the dst subaddr.
 *
 * output parameters
 *
 * @param   None.
 *
 * @return  return 1 if success,0 if failed.
 */
extern int lwns_multicast_send(lwns_controller_ptr h, uint16_t subaddr);
```

当用户需要发送一条组播消息时，就需要调用该函数。参数一就是你当前需要发送数据的控制结构体，参数二为需要发送到的目的组播地址。

调用之前需要先将需要发送的数据存到数据缓冲区，因为在协议栈内部，包的接收和发送处理都是采用的同一块内存缓冲区，所以我们需要使用函数将需要发送的数据载入内部缓冲区中：lwns_buffer_load_data。

然后再调用此函数，就会立刻在控制结构体参数的端口号上发送一条组播消息，发送方和接收方必须都打开同样端口号，才可以收到消息。

5.4 lwns_multicast_close

```
/******************************************************************************
 * @fn          lwns_multicast_close
 *
 * @brief       close a port for multicast.
 *
 * input parameters
 *
 * @param       h - lwns_controller_ptr,the pointer of a lwns_multicast_controller,
 *              which must be initialized by lwns_multicast_init.
 *
 * output parameters
 *
 * @param       None.
 *
 * @return      None.
 */
extern void lwns_multicast_close(lwns_controller_ptr h);
```

当用户不需要再接收某个端口的消息时，就可以调用 close 函数将该端口关闭。关闭后，rf 接收到的数据解析的端口号匹配不到相应的控制结构体，所以就会被丢弃。

5.5 example

本章例程为 lwns_multicast_example.c。

使用广播模块编写程序必须要做的事：

iii. 使用 lwns_multicast_controller 定义一块内存空间

```
static lwns_multicast_controller multicast;//声明组播控制结构体
```

iv. 编写相关回调函数

multicast 模块有两个回调函数：recv 和 sent，并且需要把函数指针赋予回调函数结构体中。

接收回调函数：

```c
static void multicast_recv(lwns_controller_ptr ptr, uint16_t subaddr, const lwns_addr_t *sender){
  uint8_t len;
  len = lwns_buffer_datalen(); //获取当前缓冲区接收到的数据长度
  if (len == 10) {
    lwns_buffer_save_data(RX_DATA); //接收数据到用户数据区域
    PRINTF("multicast %d rec from %02x %02x %02x %02x %02x %02x\n",
        get_lwns_object_port(ptr),
        sender->v8[0], sender->v8[1], sender->v8[2], sender->v8[3],
        sender->v8[4], sender->v8[5]);//from为接收到的数据的发送方地址
    PRINTF("subaddr:%d data:",subaddr);
    for (uint8_t i = 0; i < len; i++) {
      PRINTF("%02x ", RX_DATA[i]);
    }
    PRINTF("\n");
  } else {
    PRINTF("data len err\n");
  }
}
```

发送完成回调函数：

```c
static void multicast_sent(lwns_controller_ptr ptr) {
  PRINTF("multicast %d sent\n",get_lwns_object_port(ptr));
}
```

声明回调函数结构体：

```c
static const struct lwns_multicast_callbacks multicast_callbacks =
{multicast_recv,multicast_sent};//注册回调函数
```

准备工作都做好后，即可开始初始化相关工作，通过广播模块的 init 函数进行初始化：

```c
void lwns_multicast_process_init(void) {
  multicast_taskID = TMOS_ProcessEventRegister(lwns_multicast_ProcessEvent);
  lwns_multicast_init(&multicast,
      136,//打开一个端口号为136的组播
      subaddrs,//订阅地址数组指针
      SUBADDR_NUM,//订阅地址数量
      &multicast_callbacks
      );//返回0代表打开失败。返回1打开成功。
  tmos_start_task(multicast_taskID, MULTICAST_EXAMPLE_TX_PERIOD_EVT,
      MS1_TO_SYSTEM_TIME(1000));
}
```

本次实验采用两块开发板，其中一块开发板的程序的订阅组地址改为：

```c
static u8 subaddrs_index = 0;//发送订阅地址序号
#define SUBADDR_NUM    3//订阅地址数量
static uint16 subaddrs[SUBADDR_NUM]={1,20,3};//订阅地址数组
```

另一块开发板的程序的订阅组地址改为：

```c
static u8 subaddrs_index = 0;//发送订阅地址序号
#define SUBADDR_NUM    3//订阅地址数量
static uint16 subaddrs[SUBADDR_NUM]={1,2,3};//订阅地址数组
```

在 tmos 中的 MULTICAST_EXAMPLE_TX_PERIOD_EVT 任务处理中，进行周期性的发送：

```c
if (events & MULTICAST_EXAMPLE_TX_PERIOD_EVT) {//周期性在不同的组播地址上发送组播消息
    uint8 temp;
    temp = TX_DATA[0];
    for (uint8 i = 0; i < 9; i++) {
        TX_DATA[i] = TX_DATA[i + 1];
    }
    TX_DATA[9] = temp;
    lwns_buffer_load_data(TX_DATA, sizeof(TX_DATA));//载入需要发送的数据到缓冲区
    if(subaddrs_index >= SUBADDR_NUM){
        subaddrs_index = 0;
    }
    lwns_multicast_send(&multicast,subaddrs[subaddrs_index]);//组播发送数据给指定节点
    subaddrs_index++;
    tmos_start_task(multicast_taskID, MULTICAST_EXAMPLE_TX_PERIOD_EVT,
        MS1_TO_SYSTEM_TIME(1000));//周期性发送
    return events ^ MULTICAST_EXAMPLE_TX_PERIOD_EVT;
}
```

将程序编译后烧录到两块板子里，实验现象可以通过串口观察，可以发现现象如下：

```
[16:15:04.155]收←◆multicast 136 sent
rand send:16
rand send:1

[16:15:04.451]收←◆send rx msg
subnum:3
send rx msg

[16:15:05.167]收←◆multicast 136 sent
rand send:5
rand send:12

[16:15:05.448]收←◆send rx msg
subnum:3
sub idx:2,num:3
h:36,b:11
multicast 136 rec from d9 37 3c e4 c2 84
subaddr:3 data:38 39 30 31 32 33 34 35 36 37
send rx msg

[16:15:06.156]收←◆multicast 136 sent
rand send:12
rand send:15

[16:15:06.452]收←◆send rx msg
subnum:3
sub idx:0,num:1
h:36,b:11
multicast 136 rec from d9 37 3c e4 c2 84
subaddr:1 data:39 30 31 32 33 34 35 36 37 38
send rx msg
```

```
[16:15:04.170]收←◆send rx msg
subnum:3
sub idx:2,num:3
h:36,b:11
multicast 136 rec from a3 df 38 e4 c2 84
subaddr:3 data:35 36 37 38 39 30 31 32 33 34
send rx msg

[16:15:04.451]收←◆multicast 136 sent
rand send:3
rand send:10

[16:15:05.167]收←◆send rx msg
subnum:3
sub idx:0,num:1
h:36,b:11
multicast 136 rec from a3 df 38 e4 c2 84
subaddr:1 data:36 37 38 39 30 31 32 33 34 35
send rx msg

[16:15:05.448]收←◆multicast 136 sent
rand send:12
rand send:8

[16:15:06.172]收←◆send rx msg
subnum:3
send rx msg

[16:15:06.437]收←◆multicast 136 sent
rand send:2
rand send:6
```

a 节点发送数据包后，调用了自身的发送完成回调，打印了 multicast 136 sent，136 就是当前的端口号，通过 get_lwns_object_port 获取到端口号。b 节点接收到了来自 a 节点的数据包，并不都会进入 multicast_recv 回调函数，而是 a 发送的订阅组地址也是 b 的订阅组地址时，b 节点才会调用接收回调函数。

b 节点和 a 节点现象相同。

6. 单播

unicast 为单播的意思，意思是向指定节点发送信息。

无线信号并不像有线信号,可以有路由,在指定线路发送信息。无线信号所发出的信息,无论其他节点想不想接收，只要在接收范围内都会接收到。发送者也不会知道指定的接收者在哪个具体位置。所以对于无线信号来说,单播只是接收者是否过滤掉发送者的数据包而已。

最后得出结论，即发送方在发送数据时，在数据中存有接收方的地址，接收方识别是自身地址的数据包才会处理数据，调用接收回调函数。

使用单播需要用 lwns_unicast_controller 定义一块内存空间。

```
//can not be modified
#define LWNS_UNICAST_CONTROLLER_U32_SIZE   5
typedef struct _lwns_unicast_controller_struct {
  uint32_t data[LWNS_UNICAST_CONTROLLER_U32_SIZE];
} lwns_unicast_controller;
```

6.1 lwns_unicast_callbacks

```
struct lwns_unicast_callbacks {
  void (*recv)(lwns_controller_ptr ptr, const lwns_addr_t *sender);
  void (*sent)(lwns_controller_ptr ptr);
};
```

在本结构体中，有两个回调函数成员，一个是 sent，一个是 recv。

sent 回调函数会在调用发送后，立刻调用。表示发送完成回调函数。

recv 回调函数会在收到一个本端口并且是发送给自己的 unicast 消息后调用，表示收到数据，和 broadcast 的接收回调函数一样，都有一个 sender 的地址指针，用户可以从该指针读出发送方的地址。

两个回调函数的参数中都有 lwns_controller_ptr 指针。用户可以通过 get_lwns_object_port 读出当前数据所属的数据端口。

6.2 lwns_unicast_init

```
/*******************************************************************************
 * @fn        lwns_unicast_init
 *
 * @brief     open a port for unicast.
 *
 * input parameters
 *
 * @param     h - lwns_controller_ptr,the pointer of a lwns_unicast_controller.
 * @param     port_num    -value:(1-255)   - port to recognize data
 * @param     callbacks defined with lwns_unicast_callbacks
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    return 1 if success,0 if failed.
 */
extern int lwns_unicast_init(lwns_controller_ptr h, uint8_t port_num,
    const struct lwns_unicast_callbacks *u);
```

在 lwns_unicast_init 中，需要三个参数：

第一个参数为 lwns_controller_ptr 控制结构体指针，他不可以是 NULL，必须是一块可用内存的头指针，即 lwns_unicast_controller 定义的内存空间地址。

第二个参数为端口号，端口号为标识数据包到底属于哪一个控制结构体，是在数据包解析过程中的必要参数，所以不同结构体中，不可以使用相同的端口号，不然后打开的控制结构体会覆盖掉上次打开的结构体。

第三个参数为用户自己编写的回调函数结构体指针。

6.3 lwns_unicast_send

```
/*********************************************************************************
 * @fn        lwns_unicast_send
 *
 * @brief     send a unicast message through lwns_controller_ptr h.
 *
 * input parameters
 *
 * @param     h - lwns_controller_ptr,the pointer of a lwns_unicast_controller,
 *            which must be initialized by lwns_unicast_init.
 * @param     receiver - the addr of the dest node.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    return 1 if success,0 if failed.
 */
extern int lwns_unicast_send(lwns_controller_ptr h,
    const lwns_addr_t *receiver);
```

当用户需要给指定节点发送一条单播消息时，就需要调用该函数。

参数一为当前需要发送数据的控制结构体。

参数二为目标节点的地址。

调用此函数就会立刻在控制结构体参数的端口号上发送一条单播消息，发送方和接收方必须都打开同样端口号，才可以收到消息。

接收方会对数据中地址和自身地址进行匹配，决定是否接收。

6.4 lwns_unicast_close

```
/*********************************************************************************
 * @fn        lwns_unicast_close
 *
 * @brief     close a port for unicast.
 *
 * input parameters
 *
 * @param     h - lwns_controller_ptr,the pointer of a lwns_unicast_controller,
 *            which must be initialized by lwns_unicast_init.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    None.
 */
extern void lwns_unicast_close(lwns_controller_ptr h);
```

当用户不需要再接收某个端口的消息时,就可以调用 close 函数将该端口关闭。关闭后,rf 接收到的数据解析的端口号匹配不到相应的控制结构体, 所以就会被丢弃。

6.5 example

本章例程为 lwns_unicast_example.c。

使用单播模块编写程序必须要做的事:

i. 使用 lwns_unicast_controller 定义一块内存空间

```
static lwns_unicast_controller unicast;//声明单播控制结构体
```

ii. 编写相关回调函数

unicast 模块有两个回调函数: recv 和 sent, 并且需要把函数指针赋予回调函数结构体中。

接收回调函数:

```
static void unicast_recv(lwns_controller_ptr ptr, const lwns_addr_t *sender){
  uint8_t len;
  len = lwns_buffer_datalen(); //获取当前缓冲区接收到的数据长度
  if (len == 10) {
    lwns_buffer_save_data(RX_DATA); //接收数据到用户数据区域
    PRINTF("unicast %d rec from %02x %02x %02x %02x %02x %02x\n",
        get_lwns_object_port(ptr),
        sender->v8[0], sender->v8[1], sender->v8[2], sender->v8[3],
        sender->v8[4], sender->v8[5]);//sender为接收到的数据的发送方地址
    PRINTF("data:");
    for (uint8_t i = 0; i < len; i++) {
      PRINTF("%02x ", RX_DATA[i]);
    }
    PRINTF("\n");
  } else {
    PRINTF("data len err\n");
  }
}
```

发送完成回调函数:

```
static void unicast_sent(lwns_controller_ptr ptr) {
  PRINTF("unicast %d sent\n",get_lwns_object_port(ptr));
}
```

声明回调函数结构体:

```
static const struct lwns_unicast_callbacks unicast_callbacks =
{unicast_recv,unicast_sent};//注册回调函数
```

准备工作都做好后,即可开始初始化相关工作,通过各个模块的 init 函数进行初始化:

```c
void lwns_unicast_process_init(void) {
  unicast_taskID = TMOS_ProcessEventRegister(lwns_unicast_ProcessEvent);
  lwns_unicast_init(&unicast,
      136,//打开一个端口号为136的单播
      &unicast_callbacks
      );//返回0代表打开失败。返回1打开成功。
  tmos_start_task(unicast_taskID, UNICAST_EXAMPLE_TX_PERIOD_EVT,
      MS1_TO_SYSTEM_TIME(1000));
}
```

在 tmos 中的 UNICAST_EXAMPLE_TX_PERIOD_EVT 任务处理中，进行周期性的发送：

```c
if (events & UNICAST_EXAMPLE_TX_PERIOD_EVT) {
  uint8 temp;
  temp = TX_DATA[0];
  for (uint8 i = 0; i < 9; i++) {
    TX_DATA[i] = TX_DATA[i + 1];
  }
  TX_DATA[9] = temp;
  lwns_buffer_load_data(TX_DATA, sizeof(TX_DATA));//载入需要发送的数据到缓冲区
  lwns_unicast_send(&unicast,&dst_addr);//单播发送数据给指定节点
  tmos_start_task(unicast_taskID, UNICAST_EXAMPLE_TX_PERIOD_EVT,
      MS1_TO_SYSTEM_TIME(1000));//周期性发送
  return events ^ UNICAST_EXAMPLE_TX_PERIOD_EVT;
}
```

第一次实验，将程序中的 dst_addr 修改：

```c
//static lwns_addr_t dst_addr = { { 0xab, 0xdf, 0x38, 0xe4, 0xc2, 0x84 } };/
static lwns_addr_t dst_addr = { { 0xd9, 0x37, 0x3c, 0xe4, 0xc2, 0x84 } };
```

将 a 节点程序中的 dst_addr 修改为 b 节点的地址，将 b 节点程序中的 dst_addr 修改为 a 节点的地址。分别编译后烧录到两块板子里，实验现象可以通过串口观察，可以发现现象如下：

```
[09:54:53.809]收←◆unicast 136 sent

[09:54:54.315]收←◆unicast 136 rec from ab df 38 e4 c2 84
data:38 39 30 31 32 33 34 35 36 37

[09:54:54.802]收←◆unicast 136 sent

[09:54:55.314]收←◆unicast 136 rec from ab df 38 e4 c2 84
data:39 30 31 32 33 34 35 36 37 38

[09:54:55.802]收←◆unicast 136 sent

[09:54:56.315]收←◆unicast 136 rec from ab df 38 e4 c2 84
data:30 31 32 33 34 35 36 37 38 39

[09:54:56.802]收←◆unicast 136 sent

[09:54:57.314]收←◆unicast 136 rec from ab df 38 e4 c2 84
data:31 32 33 34 35 36 37 38 39 30

[09:54:57.801]收←◆unicast 136 sent

[09:54:58.314]收←◆unicast 136 rec from ab df 38 e4 c2 84
data:32 33 34 35 36 37 38 39 30 31

[09:54:58.801]收←◆unicast 136 sent

[09:54:59.315]收←◆unicast 136 rec from ab df 38 e4 c2 84
data:33 34 35 36 37 38 39 30 31 32
```

```
[09:54:53.805]收←◆unicast 136 rec from d9 37 3c e4 c2 84
data:37 38 39 30 31 32 33 34 35 36

[09:54:54.313]收←◆unicast 136 sent

[09:54:54.804]收←◆unicast 136 rec from d9 37 3c e4 c2 84
data:38 39 30 31 32 33 34 35 36 37

[09:54:55.312]收←◆unicast 136 sent

[09:54:55.804]收←◆unicast 136 rec from d9 37 3c e4 c2 84
data:39 30 31 32 33 34 35 36 37 38

[09:54:56.313]收←◆unicast 136 sent

[09:54:56.804]收←◆unicast 136 rec from d9 37 3c e4 c2 84
data:30 31 32 33 34 35 36 37 38 39

[09:54:57.312]收←◆unicast 136 sent

[09:54:57.803]收←◆unicast 136 rec from d9 37 3c e4 c2 84
data:31 32 33 34 35 36 37 38 39 30

[09:54:58.312]收←◆unicast 136 sent

[09:54:58.803]收←◆unicast 136 rec from d9 37 3c e4 c2 84
data:32 33 34 35 36 37 38 39 30 31

[09:54:59.313]收←◆unicast 136 sent
```

a 节点发送数据包后，调用了自身的发送完成回调，打印了 unicast 136 sent，136 就是当前的端口号，通过 get_lwns_object_port 获取，b 节点就接收到了来自 a 节点的数据包，调用了编写的 unicast_recv 回调函数，打印出了接收到的数据和发送方的地址。

b 节点和 a 节点现象相同。

第二次实验，将程序中的 dst_addr 修改，将 a 节点的目标地址修改为其他非 a 和 b 节点的地址。b 节点的目标地址依旧为 a 节点的地址。分别编译后烧录到两块板子里，实验现象可以通过串口观察，可以发现现象如下：

```
[09:57:46.949]收←◆unicast 136 sent

[09:57:47.282]收←◆unicast 136 rec from ab df 38 e4 c2 84
data:33 34 35 36 37 38 39 30 31 32

[09:57:47.948]收←◆unicast 136 sent

[09:57:48.282]收←◆unicast 136 rec from ab df 38 e4 c2 84
data:34 35 36 37 38 39 30 31 32 33

[09:57:48.948]收←◆unicast 136 sent

[09:57:49.282]收←◆unicast 136 rec from ab df 38 e4 c2 84
data:35 36 37 38 39 30 31 32 33 34

[09:57:49.949]收←◆unicast 136 sent

[09:57:50.281]收←◆unicast 136 rec from ab df 38 e4 c2 84
data:36 37 38 39 30 31 32 33 34 35

[09:57:50.948]收←◆unicast 136 sent

[09:57:51.282]收←◆unicast 136 rec from ab df 38 e4 c2 84
data:37 38 39 30 31 32 33 34 35 36

[09:57:51.948]收←◆unicast 136 sent
```

```
[09:57:39.281]收←◆unicast 136 sent

[09:57:40.280]收←◆unicast 136 sent

[09:57:41.280]收←◆unicast 136 sent

[09:57:42.281]收←◆unicast 136 sent

[09:57:43.281]收←◆unicast 136 sent

[09:57:44.280]收←◆unicast 136 sent

[09:57:45.280]收←◆unicast 136 sent

[09:57:46.281]收←◆unicast 136 sent

[09:57:47.280]收←◆unicast 136 sent

[09:57:48.280]收←◆unicast 136 sent

[09:57:49.280]收←◆unicast 136 sent

[09:57:50.280]收←◆unicast 136 sent

[09:57:51.280]收←◆unicast 136 sent
```

通过现象发现，a 节点程序的目标地址修改为不是 b 节点的地址后，b 节点再也不会进入到 unicast 接收回调中。所以 unicast 接收函数必须是接收到自己地址的数据进入接收回调函数。

7. 可靠单播

ruc 是 reliable unicast 的简称，为可靠单播。基于 unicast 之上做了自动回复 ack 处理。如果指定时间内未回复 ack，则会自动重发。直到超过设定的最大发送次数。ruc 通过确认收到和重发机制来保证相邻节点收到了数据包。

使用可靠单播需要用 lwns_ruc_controller 定义一块内存空间。

```
//can not be modified
#define LWNS_RUC_CONTROLLER_U32_SIZE   25
typedef struct _lwns_ruc_controller_struct {
  uint32_t data[LWNS_RUC_CONTROLLER_U32_SIZE];
} lwns_ruc_controller;
```

7.1 lwns_ruc_callbacks

```
struct lwns_ruc_callbacks {
  void (*recv)(lwns_controller_ptr ptr, const lwns_addr_t *sender);
  void (*sent)(lwns_controller_ptr ptr, const lwns_addr_t *to,
      uint8_t retransmissions);
  void (*timedout)(lwns_controller_ptr ptr, const lwns_addr_t *to);
};
```

在本结构体中，有两个回调函数成员，一个是 sent，一个是 recv。

ruc 的 sent 回调函数不像 unicast 等协议会在调用发送后立刻调用，而是在目标节点回复 ack 以后调用 sent。表示发送完成回调函数，第二个参数为目标地址，第三个参数为第几次发送才收到了 ack。

recv 回调函数会在收到一个本端口并且是发送给自己的 ruc 消息后调用，表示收到数据，和 broadcast 的接收回调函数一样，都有一个 sender 的地址指针，用户可以从该指针读出发送方的地址，

在一些情况下，可能主机收不到从机发送的 **ack**，这个时候包会重发，对于从机而言可能会收到**两次一样的包**，所以包会标有**序号**。序号范围为 0-127，库内部会自动排除和上次发送相同序号的数据包。每个 ruc 控制结构体内部会存放 4 个发送的节点信息和节点序号。所以在编程中要注意同一时间段，不要用一个 ruc 模块和大于 4 个节点同时通信。当接收超过 4 个节点的信息时，会自动删除最旧的信息。

发送方重发指定次数后，还是没有收到目标节点回复的 ack，就会调用 timedout 回调函数。

三个回调函数的参数中都有 lwns_controller_ptr 指针，就是收到数据所属的控制句柄。用户可以通过 get_lwns_object_port 读出当前数据所属的数据端口。

7.2 lwns_ruc_init

```
/*****************************************************************************
 * @fn        lwns_ruc_init
 *
 * @brief     open a port for reliable unicast.
 *
 * input parameters
 *
 * @param     h - lwns_controller_ptr,the pointer of a lwns_ruc_controller.
 * @param     port_num   -value:(1-255)  - port to recognize data
 * @param     retransmit_time  if dest node not give ack,we will retry to send message after this retransmit_time time.
 * @param     callbacks defined with lwns_ruc_callbacks
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    return 1 if success,0 if failed.
 */
extern int lwns_ruc_init(lwns_controller_ptr h, uint8_t port_num,
    lwns_clock_time_t retransmit_time,
    const struct lwns_ruc_callbacks *u);
```

在 lwns_ruc_init 中，需要四个参数：

第一个参数为 lwns_controller_ptr 控制结构体指针，他不可以是 NULL，必须是一块可用内存的头指针，即 lwns_ruc_controller 定义的内存空间地址。

第二个参数为端口号，端口号为标识数据包到底属于哪一个控制结构体，是在数据包解析过程中的必要参数，所以不同结构体中，不可以使用相同的端口号，不然打开的控制结构体会覆盖掉上次打开的结构体。

第三个参数为等待回复 ack 的时间，即如果在该时间内没有收到 ack，就会进行数据重发，直到达到 ruc_send 函数中设置的重发次数后就会停止发送，进入 timedout 回调函数中。该值以 HTIMER_CLOCK_SECONDS 为基准，即参数值为 HTIMER_CLOCK_SECONDS，重试时间为 1 秒。

第四个参数为用户自己编写的回调函数结构体指针。

7.3 lwns_ruc_send

```
/*****************************************************************************
 * @fn        lwns_ruc_send
 *
 * @brief     send a reliable unicast message used lwns_controller_ptr h.
 *
 * input parameters
 *
 * @param     h - lwns_controller_ptr,the pointer of a lwns_ruc_controller,
 *            which must be initialized by lwns_ruc_init.
 * @param     receiver - the addr of the dest node.
 * @param     max_retransmissions - if the dest not give ack,we will retry for this set value times.the legal value is 1-255;
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    return 1 if success,0 if failed.
 */
extern int lwns_ruc_send(lwns_controller_ptr h,
    const lwns_addr_t *receiver, uint8_t max_retransmissions);
```

当用户需要给指定节点发送一条可靠单播消息时，就需要调用该函数进行发送。

参数一就是你当前需要发送数据的控制结构体。

参数二就是目标节点的地址。

参数三就是最大重发尝试次数，重发达到此值就会停止重发。进入 timedout 回调函数中。

调用此函数就会立刻在控制结构体参数的端口号上发送一条单播消息,发送方和接收方必须都打开同样端口号，才可以收到消息。

7.4 lwns_ruc_close

```
/*****************************************************************************
 * @fn        lwns_ruc_close
 *
 * @brief     close a port for reliable unicast.
 *
 * input parameters
 *
 * @param     h - lwns_controller_ptr,the pointer of a lwns_ruc_controller,
 *            which must be initialized by lwns_ruc_init.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    None.
 */
extern void lwns_ruc_close(lwns_controller_ptr h);
```

当用户不需要再接收某个端口的消息时,就可以调用 close 函数将该端口关闭。关闭后,rf 接收到的数据解析的端口号匹配不到相应的控制结构体，所以就会被丢弃。

7.5 lwns_ruc_is_busy

```
/*****************************************************************************
 * @fn        lwns_ruc_is_busy
 *
 * @brief     get status of this ruc lwns_controller_ptr.
 *
 * input parameters
 *
 * @param     h - lwns_controller_ptr,the pointer of a lwns_ruc_controller,
 *            which must be initialized by lwns_ruc_init.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    return 1 if busy,0 if free.
 */
extern uint8_t lwns_ruc_is_busy(lwns_controller_ptr h);
```

该函数可以获取当前可靠单播控制结构体的状态，是否在发送上一条消息。因为忙碌状态下再次调用 ruc_send 是不会发送数据包的。

7.6 lwns_ruc_clean_info

```
/*****************************************************************************
 * @fn        lwns_ruc_clean_info
 *
 * @brief     clean senders and seqno in lwns_ruc_controller.
 *
 * input parameters
 *
 * @param     h - lwns_controller_ptr,the pointer of a lwns_ruc_controller or lwns_rucft_controller,
 *            which must be initialized by lwns_ruc_init/lwns_rucft_init.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    return 1 if busy,0 if free.
 */
extern void lwns_ruc_clean_info(lwns_controller_ptr h);
```

该函数为清除 ruc 控制结构体中存有的发送者和序号。也可以用于清除 rucft 中的信息。用户可以在空闲的时候清除。

7.7 lwns_ruc_remove_an_info

```
/*******************************************************************************
 * @fn          lwns_ruc_remove_an_info
 *
 * @brief       remove a sender info in lwns_ruc_controller.
 *
 * input parameters
 *
 * @param       h - lwns_controller_ptr,the pointer of a lwns_ruc_controller or lwns_rucft_controller,
 *              which must be initialized by lwns_ruc_init/lwns_rucft_init.
 * @param       sender - the addr of the sender node.
 *
 * output parameters
 *
 * @param       None.
 *
 * @return      None.
 */
extern void lwns_ruc_remove_an_info(lwns_controller_ptr h, lwns_addr_t* sender);
```

该函数为清除 ruc 控制结构体中存有的指定发送者和序号。也可以用于清除 rucft 中的信息。用户可以在空闲的时候清除。

7.8 example

本章例程为 lwns_ruc_example.c。

ruc 是建立在 unicast 之上带有重发机制的协议，所以本次实验只关注重发特性。

使用可靠单播模块编写程序必须要做的事：

i. 使用 lwns_ruc_controller 定义一块内存空间

```
static lwns_ruc_controller ruc;//声明可靠单播控制结构体
```

ii. 编写相关回调函数

ruc 模块有两个回调函数：recv 和 sent，并且需要把函数指针赋予回调函数结构体中。

接收回调函数：

```
static void recv_ruc(lwns_controller_ptr ptr,
      const lwns_addr_t* sender) {
  //ruc中接收到发送给自己的数据后，才会调用该回调
  uint8_t len;
  len = lwns_buffer_datalen(); //获取当前缓冲区接收到的数据长度
  if (len == 10) {
    lwns_buffer_save_data(RX_DATA); //接收数据到用户数据区域
    PRINTF("ruc %d rec %02x %02x %02x %02x %02x %02x\r\n",get_lwns_object_port(ptr), sender->v8[0],
        sender->v8[1], sender->v8[2], sender->v8[3], sender->v8[4], sender->v8[5]);
    PRINTF("data:");
    for (uint8_t i = 0; i < len; i++) {
      PRINTF("%02x ", RX_DATA[i]);
    }
    PRINTF("\n");
  } else {
    PRINTF("data len err\n");
  }
}
```

发送完成回调函数：

```
static void sent_ruc(lwns_controller_ptr ptr,
      const lwns_addr_t* to, uint8_t retransmissions) {
  //ruc中发送成功，并且收到目标节点的ack回复后，才会调用该回调
  PRINTF("ruc %d sent %d\r\n",get_lwns_object_port(ptr),retransmissions);
  tmos_start_task(ruc_taskID, RUC_EXAMPLE_TX_PERIOD_EVT,
          MS1_TO_SYSTEM_TIME(1000));//更新任务时间，发送并收到回复后，1秒钟后再发送
}
```

超时回调函数：

```
static void timedout_ruc(lwns_controller_ptr ptr,
      const lwns_addr_t* to) {
  //ruc中，再重发次数超过最大重发次数后，会调用该回调。
  PRINTF("ruc %d timedout\n",get_lwns_object_port(ptr));
  tmos_start_task(ruc_taskID, RUC_EXAMPLE_TX_PERIOD_EVT,
          MS1_TO_SYSTEM_TIME(1000));
}
```

声明回调函数结构体：

```
static const struct lwns_ruc_callbacks ruc_callbacks = {
      recv_ruc, sent_ruc, timedout_ruc };//声明可靠单播回调结构体
```

准备工作都做好后,即可开始初始化相关工作,通过 ruc 模块的 init 函数进行初始化:

```
void lwns_ruc_process_init(void) {
  lwns_ruc_init(&ruc,
      144,//打开一个端口号为144的可靠单播
      HTIMER_SECOND_NUM,//等待ack时间间隔，没收到就会重发
      &ruc_callbacks);//返回0代表打开失败。返回1打开成功。
  ruc_taskID = TMOS_ProcessEventRegister(lwns_ruc_ProcessEvent);
  tmos_start_task(ruc_taskID, RUC_EXAMPLE_TX_PERIOD_EVT,
          MS1_TO_SYSTEM_TIME(1000));
}
```

在 tmos 中的 RUC_EXAMPLE_TX_PERIOD_EVT 任务处理中，进行周期性的发送：

```
if (events & RUC_EXAMPLE_TX_PERIOD_EVT) {
  uint8 temp;
  temp = TX_DATA[0];
  for (uint8 i = 0; i < 9; i++) {
    TX_DATA[i] = TX_DATA[i + 1];
  }
  TX_DATA[9] = temp;
  lwns_buffer_load_data(TX_DATA, sizeof(TX_DATA));//载入需要发送的数据到缓冲区
  lwns_ruc_send(&ruc,
      &dst_addr,//可靠单播目标地址
      4//最大重发次数
      );//可靠单播发送函数：发送参数，目标地址，最大重发次数，默认一秒钟重发一次
  return events ^ RUC_EXAMPLE_TX_PERIOD_EVT;
}
```

第一次实验，将程序中的 dst_addr 修改：

```
//static lwns_addr_t dst_addr = { { 0xab, 0xdf, 0x38, 0xe4, 0xc2, 0x84 } };/
static lwns_addr_t dst_addr = { { 0xd9, 0x37, 0x3c, 0xe4, 0xc2, 0x84 } };
```

将 a 节点程序中的 dst_addr 修改为 b 节点的地址，将 b 节点程序中的 dst_addr 修改为 a 节点的地址。分别编译后烧录到两块板子里，实验现象可以通过串口观察，可以发现现象如下：

```
[10:17:44.455]收←◆ruc 144 sent 1

[10:17:45.029]收←◆ruc 144 rec ab df 38 e4 c2 84
data seq 7:38 39 30 31 32 33 34 35 36 37

[10:17:45.462]收←◆ruc 144 sent 1

[10:17:46.037]收←◆ruc 144 rec ab df 38 e4 c2 84
data seq 8:39 30 31 32 33 34 35 36 37 38

[10:17:46.468]收←◆ruc 144 sent 1

[10:17:47.042]收←◆ruc 144 rec ab df 38 e4 c2 84
data seq 9:30 31 32 33 34 35 36 37 38 39

[10:17:47.476]收←◆ruc 144 sent 1

[10:17:48.050]收←◆ruc 144 rec ab df 38 e4 c2 84
data seq 10:31 32 33 34 35 36 37 38 39 30

[10:17:48.483]收←◆ruc 144 sent 1

[10:17:49.057]收←◆ruc 144 rec ab df 38 e4 c2 84
data seq 11:32 33 34 35 36 37 38 39 30 31

[10:17:49.489]收←◆ruc 144 sent 1
```

```
[10:17:44.450]收←◆ruc 144 rec d9 37 3c e4 c2 84
data seq 5:36 37 38 39 30 31 32 33 34 35

[10:17:45.034]收←◆ruc 144 sent 1

[10:17:45.456]收←◆ruc 144 rec d9 37 3c e4 c2 84
data seq 6:37 38 39 30 31 32 33 34 35 36

[10:17:46.041]收←◆ruc 144 sent 1

[10:17:46.463]收←◆ruc 144 rec d9 37 3c e4 c2 84
data seq 7:38 39 30 31 32 33 34 35 36 37

[10:17:47.049]收←◆ruc 144 sent 1

[10:17:47.471]收←◆ruc 144 rec d9 37 3c e4 c2 84
data seq 8:39 30 31 32 33 34 35 36 37 38

[10:17:48.055]收←◆ruc 144 sent 1

[10:17:48.478]收←◆ruc 144 rec d9 37 3c e4 c2 84
data seq 9:30 31 32 33 34 35 36 37 38 39

[10:17:49.062]收←◆ruc 144 sent 1

[10:17:49.484]收←◆ruc 144 rec d9 37 3c e4 c2 84
data seq 10:31 32 33 34 35 36 37 38 39 30
```

a 节点发送数据包后，调用了自身的发送完成回调，打印了 ruc 144 sent 1，144 就是当前的端口号，通过 get_lwns_object_port 获取，1 表示是第几次发送就收到了 ack，b 节点就接收到了来自 a 节点的数据包，调用了编写的 ruc_recv 回调函数，打印出了接收到的数据和发送方的地址。

b 节点和 a 节点现象相同。

第二次实验，只打开 a 节点，b 节点关闭。观察 a 节点的现象如下：

```
[10:20:10.277]收←◆ruc 144 timedout
[10:20:15.277]收←◆ruc 144 timedout
[10:20:20.277]收←◆ruc 144 timedout
[10:20:25.277]收←◆ruc 144 timedout
[10:20:30.277]收←◆ruc 144 timedout
[10:20:35.277]收←◆ruc 144 timedout
[10:20:40.277]收←◆ruc 144 timedout
[10:20:45.277]收←◆ruc 144 timedout
[10:20:50.277]收←◆ruc 144 timedout
```

通过现象发现，a 节点收不到 b 节点发送的 ack 后，a 节点再也不会进入到 ruc 发送回调中，只会在尝试数次以后，调用 timedout 回调函数。

8. 可靠单播文件传输

rucft 为 reliable unicast file transfer 的简称。即可靠单播文件传输。每次传输的字节数为：

```
#define LWNS_RUCFT_DATASIZE 192 //rucft each packet size
```

即每个数据包都是 192 个字节。再发送完所有数据后，最后一个数据包会发送一个空包。使用可靠单播文件传输需要用 lwns_rucft_controller 定义一块内存空间。

```
//can not be modified
#define LWNS_RUCFT_CONTROLLER_U32_SIZE   35
typedef struct _lwns_rucft_controller_struct {
  uint32_t data[LWNS_RUCFT_CONTROLLER_U32_SIZE];
} lwns_rucft_controller;
```

8.1 lwns_rucft_callbacks

```
struct lwns_rucft_callbacks {
  void (*write_file)(lwns_controller_ptr ptr, const lwns_addr_t *sender,
       int offset, int flag, char *data, int len);
  int (*read_file)(lwns_controller_ptr ptr, int offset, char *to,
       int maxsize);
  void (*timedout)(lwns_controller_ptr ptr);
};
```

write_file 即为接收到数据后调用的 recv 函数，和 ruc 不同，因 rucft 在一次传输中，只会有一个主机，只有接收到同一主机的消息后才会调用 write_file 接收回调函数。或者超时后，清空了存有的主机参数。

在 write_file 回调函数中，flag 有如下三个值：

```
enum {
  LWNS_RUCFT_FLAG_NONE = 0,
  LWNS_RUCFT_FLAG_NEWFILE,
  LWNS_RUCFT_FLAG_END,
};
```

LWNS_RUCFT_FLAG_NONE 表示接收到的是正在传输的中间包。

LWNS_RUCFT_FLAG_END 表示接收到的是本次传输的最后一个数据包。

LWNS_RUCFT_FLAG_NEWFILE 表示本次传输接收到的第一个包。

ptr 即为当前的控制结构体。

from 为发送方的地址。

offset 为本次传输的偏移量。

data 为本次接收到的数据的头指针。

len 为本次接收到的数据长度。

read_file 回调函数在每次在准备发送信息时都会调用，即第一次发送会调用，剩下都需要收到目标节点回复的 ack 以后才会调用此回调函数。它的主要功能是清空包缓冲，然后把包缓冲数据段头部指针做为参数传给用户自定义回调函数 read_chunk 并执行。用户需要根据返回的长度设置包缓冲数据段长度，并将数据存储到 to 参数指向的内存空间中。

timedout 回调函数和 ruc 中的一样，都是在指定时间不断重发尝试，直到到达最大尝试次数，就会进入超时回调函数中。

8.2 lwns_rucft_init

```
/**********************************************************************************
 * @fn      lwns_rucft_init
 *
 * @brief   open a port for reliable unicast file transfer.
 *
 * input parameters
 *
 * @param   h - lwns_controller_ptr,the pointer of a lwns_rucft_controller.
 * @param   port_num   -value:(1-255)   - port to recognize data
 * @param   retransmit_time  if dest node not give ack,we will retry to send message after this retransmit_time time.
 * @param   max_retransmissions - if the dest not give ack,we will retry for this set value times.
 * @param   callbacks defined with lwns_rucft_callbacks.
 *
 * output parameters
 *
 * @param   None.
 *
 * @return  return 1 if success,0 if failed.
 */
extern int lwns_rucft_init(lwns_controller_ptr h, uint8_t port_num,
    lwns_clock_time_t retransmit_time, uint8_t max_retransmissions,
    const struct lwns_rucft_callbacks *u);
```

rucft 是建立在 ruc 之上的文件传输，所以参数的意义可以参考 ruc 的参数。

第三个参数 retransmit_time 和 ruc_init 中的 retransmit_time 功能一致。

第四个参数 maxRetry 和 ruc_send 中的 max_retransmitions 功能一致。

第五个参数即为用户自行编写的回调函数。

8.3 lwns_rucft_send

```
/**********************************************************************************
 * @fn      lwns_rucft_send
 *
 * @brief   start a reliable unicast file transfer to dest addr:receiver,used lwns_controller_ptr h.
 *
 * input parameters
 *
 * @param   h - lwns_controller_ptr,the pointer of a lwns_rucft_controller,
 *          which must be initialized by lwns_rucft_init.
 * @param   receiver - the addr of the dest node.
 *
 * output parameters
 *
 * @param   None.
 *
 * @return  return 1 if success,0 if failed.
 */
extern int lwns_rucft_send(lwns_controller_ptr h,
    const lwns_addr_t *receiver);
```

第一个参数为控制结构体。

第二个参数为接收方地址。

调用此函数即可开始一次文件传输的发送。

因为用户需要在 read_file 回调函数中实现文件读取即可。
具体如何操作可以见本节的 example。

8.4 lwns_rucft_close

```
/*****************************************************************************
 * @fn      lwns_rucft_close
 *
 * @brief   close a port for reliable unicast file transfer.
 *
 * input parameters
 *
 * @param   h - lwns_controller_ptr,the pointer of a lwns_rucft_controller,
 *              which must be initialized by lwns_rucft_init.
 *
 * output parameters
 *
 * @param   None.
 *
 * @return  None.
 */
extern void lwns_rucft_close(lwns_controller_ptr h);
```

当用户不需要再接收某个端口的消息时,就可以调用 close 函数将该端口关闭。关闭后,rf 接收到的数据解析的端口号匹配不到相应的控制结构体，所以就会被丢弃。

8.5 example

本章例程为 lwns_rucft_example.c。
使用可靠单播文件传输模块编写程序必须要做的事：

i. 使用 lwns_rucft_controller 定义一块内存空间

```
static lwns_rucft_controller rucft; //声明rucft控制结构体
```

ii. 编写相关回调函数
rucft 模块有三个回调函数：write_file、read_file 和 timedout，并且需要把函数指针赋予回调函数结构体中。可以理解为 rucft 的 write_file 就是 ruc 的 recv，rucft 的 read_file 就是 ruc 的 sent，rucft 的 timedout 就是 ruc 的 timedout。
接收回调函数：

```
static void write_file(lwns_controller_ptr ptr, const lwns_addr_t *sender,
      int offset, int flag, char *data, int datalen) {
  //sender为发送方的地址
  //如果需要接收不同的文件，需要在此函数中做好接口
  if (datalen > 0) {    //声明个缓冲从data里取数据打印
    PRINTF("r:%c\n", *data);
  }
  if (flag == LWNS_RUCFT_FLAG_END) {
    PRINTF("re\n");
    //本次文件传输的最后一个包
  } else if (flag == LWNS_RUCFT_FLAG_NONE) {
    PRINTF("ru\n");
    //本次文件传输正常的包
  } else if (flag == LWNS_RUCFT_FLAG_NEWFILE) {
    PRINTF("rn\n");
    //本次文件传输的第一个包
  }
}
```

发送数据准备回调函数：

```
static int read_file(lwns_controller_ptr ptr, int offset, char *to,
      int maxsize) {
  //to为需要保存数据过去的指针
  //如果需要发送不同的文件，需要在此函数中做好接口
  int size = maxsize;
  if (offset >= FILESIZE) {
    //上次已经发完,本次是最后确认
    PRINTF("Send done\n");
    tmos_start_task(rucft_taskID, RUCFT_EXAMPLE_TX_PERIOD_EVT,
        MS1_TO_SYSTEM_TIME(5000)); //5秒钟后继续发送测试
    return 0;
  } else if (offset + maxsize >= FILESIZE) {
    size = FILESIZE - offset;

  }
  //把本次需要发送的内容压进包缓冲
  tmos_memcpy(to, strp + offset, size);
  return size;
}
```

超时回调函数：

```
static void timedout_rucft(lwns_controller_ptr ptr) {
  //rucft中，发送方再重发次数超过最大重发次数后，会调用该回调。
  //接收方超时没接收到下一个包也会调用
  PRINTF("rucft %d timedout\r\n", get_lwns_object_port(ptr));
}
```

声明回调函数结构体：

```
const static struct lwns_rucft_callbacks rucft_callbacks = { write_file,
      read_file, timedout_rucft };
```

准备工作都做好后，即可开始初始化相关工作，通过 rucft 模块的 init 函数进行初始

化：

```
void lwns_rucft_process_init(void) {
    lwns_addr_t MacAddr;
    rucft_taskID = TMOS_ProcessEventRegister(lwns_rucft_ProcessEvent);
    lwns_rucft_init(&rucft, 137, //端口号
        HTIMER_SECOND_NUM / 10, //等待目标节点ack时间
        5, //最大重发次数，与ruc中的ruc_send的重发次数功能一样
        &rucft_callbacks//回调函数
        ); //返回0代表打开失败。返回1打开成功。
    int i;
    for (i = 0; i < FILESIZE; i++) {   //LWNS_RUCFT_DATASIZE个LWNSNK_RUCFT_DATASIZE个b，等等，初始化需要发送的数据
      strsend[i] = 'a' + i / LWNS_RUCFT_DATASIZE;
    }
    strp = strsend;
    GetMACAddress(MacAddr.u8);
    if (lwns_addr_cmp(&MacAddr, &dst_addr)) {

    } else {
      tmos_start_task(rucft_taskID, RUCFT_EXAMPLE_TX_PERIOD_EVT,
          MS1_TO_SYSTEM_TIME(1000));
    }
}
```

在 tmos 中的 RUCFT_EXAMPLE_TX_PERIOD_EVT 任务处理中，进行发送：

```
if (events & RUCFT_EXAMPLE_TX_PERIOD_EVT) {
   PRINTF("send\n");
   lwns_rucft_send(&rucft, &dst_addr); //开始发送至目标节点，用户启用发送时要配置好回调函数中的数据包读取
   return events ^ RUCFT_EXAMPLE_TX_PERIOD_EVT;
}
```

将程序中的 dst_addr 修改：

```
//static lwns_addr_t dst_addr = { { 0x66, 0xdf, 0x38, 0xe4, 0xc2, 0x84 } };
static lwns_addr_t dst_addr = { { 0xd9, 0x37, 0x3c, 0xe4, 0xc2, 0x84 } };
```

将 a 节点程序中的 dst_addr 修改为 b 节点的地址，b 节点的 dst_addr 也是 b 节点的地址，程序会判断是否为自身地址来决定是否开启发送。因为大数据量传输不适合双向互发。分别编译后烧录到两块板子里，实验现象可以通过串口观察，可以发现现象如下：

```
ru
r:h
ru
r:i
ru
r:j
ru
r:k
ru
r:l
ru
r:m
ru
r:n
ru
r:o
ru
r:p
ru
r:q
ru
r:r
ru

[14:51:53.372]收←◆r:s
ru
r:t
ru
r:u
ru
re
```

```
[14:51:53.274]收←◆send
[14:51:53.340]收←◆crc error
[14:51:53.380]收←◆Send done
```

a 节点开始本次文件发送后，给 b 节点发送数据包的各个部分，a 节点收到 b 节点发送的 ack 后，就会通过 read_file 回调函数读取下一次需要发送的数据，而 b 节点每次接收到 a 节点的消息都会调用 write_file 回调函数。

直到本次所有的数据包发送完成。

9. 网络泛洪

网络泛洪 netflood 要求接收到信息的节点以广播方式转发数据包。

例如，源节点希望发送一段数据给目标节点。源节点首先通过网络将数据传送给它的每个邻居节点，每个邻居节点再将数据传送给各自的除发送数据来的节点之外的其他。如此继续下去，直到设定的 hops 为 0 为止。为了减少网络中重发数据包的数量，引入 polite gossip 机制，设定 dups 值来管理转发条件。

netflood 每次发送的消息中都会存有包序号，包序号无需用户干预，由内部自动加 1 实现。包序号是判断一个 netflood 数据包是否为旧数据包的参数。只有判定为新的数据包才会被 netflood 进行接收处理。否则数据包被丢弃。

使用 netflood 需要用 lwns_netflood_controller 定义一块内存空间。

```
//can not be modified
#define LWNS_NETFLOOD_CONTROLLER_U32_SIZE   31
typedef struct _lwns_netflood_controller_struct {
  uint32_t data[LWNS_NETFLOOD_CONTROLLER_U32_SIZE];
} lwns_netflood_controller;
```

9.1 lwns_netflood_callbacks

```
struct lwns_netflood_callbacks {
  int (*recv)(lwns_controller_ptr ptr, const lwns_addr_t *from,
       const lwns_addr_t *originator, uint8_t hops);
  void (*sent)(lwns_controller_ptr ptr);
  void (*dropped)(lwns_controller_ptr ptr);
};
```

netflood 网络泛洪模块有三个回调函数：

recv 为接收回调函数，其中的参数 from 为转发者的地址，originator 为本次网络泛洪发起人的地址，hops 为当前是第几次转发。该回调函数具有返回值，返回 1，节点会继续转发 netflood，返回 0，节点不会继续转发。

sent 函数为发送后调用，表明完成本次发送。

dropped 回调函数为等待期间，接收到了指定数量的数据包，放弃此次发送后会调用的函数。

9.2 lwns_netflood_init

```
/*****************************************************************************
 * @fn        lwns_netflood_init
 *
 * @brief     open a port for netflood.
 *
 * input parameters
 *
 * @param     h - lwns_controller_ptr,the pointer of a lwns_netflood_controller.
 * @param     port_num    -value:(1-255)   - port to recognize data
 * @param     queue_time - wait queue_time/2  with a random time to receive message.
 * @param     dups - cannot be 0,if this value is set to over 1,this controller will wait to receive the same message over this value,and then drop the message.
 * @param     hops - the message max transmit life time.when another node send this message,this value will -1 until it is 0.
 * @param     drop_old_packet - FALSE/TRUE,TURE will drop the old packet if we prepare to send a new packet.FALSE will send old message immediately.
 * @param     old_phase_value - this value is for self-healing when network error appeared,
 *                if seqno is too many than old_phase_value smaller than in memory,
 *                it will still handle this message.
 * @param     callbacks defined with lwns_netflood_callbacks
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    return 1 if success,0 if failed.
 */
extern int lwns_netflood_init(lwns_controller_ptr h, uint8_t port_num,
    lwns_clock_time_t queue_time, uint8_t dups, uint8_t hops, uint8_t drop_old_packet,
    uint8_t old_phase_value, const struct lwns_netflood_callbacks *u);
```

netflood_init 一共采用了 7 个参数：

queue_time 为周期，netflood 模块会等待二分之一周期加上二分之一周期内的随机延迟，一旦在等待期间，接收到数据包符合条件，就会计数加一，直到达到 dups 数值放弃发送，或者时间到期开始发送。如果 queue_time 为 0 或者 1，则立刻发送，不会等待接收进行比较。该值以 HTIMER_CLOCK_SECONDS 为基准。一定要注意的是用户应用层发送数据的周期必须比此 queue_time 大，否则会导致网络堵塞等不良后果。

dups 很重要，收到数据包立刻就转发，会导致网络拥堵。所以引入 dups 概念，当节点收到来自不同邻居节点的达到 dups 数量的数据包，就会放弃自己数据包的发送。对于有些场景部署下的时候，收到一个包就不发，可能导致某些边界节点无法收到数据，所以 dups 值的设定需要根据现场情况而定。

hops 表示当前的 netflood 控制结构体下的消息最多被转发几个层级。

drop_old_packet，因为有时 netflood 不是立刻发送的，他会将数据包存下来，进行等待接收，等待期间，如果收到了一个新数据包的发送请求后，旧数据包如何处理，就按照此值进行丢弃或者立刻发送的操作。FALSE 为立刻发送旧数据包，TRUE 为丢弃旧数据包。。

old_phase_value 是网络恢复参数，不可以小于 2，不可以大于 254。

网络泛洪判断一个数据包是否为旧数据包是依靠发起者地址和发起者数据包序号进行判断的。序号为 0 为特殊情况，当节点内存中存有的序号不为 0，又收到了一个序号为 0 的数据包，那么都会进行数据的接收处理，适合节点掉线后，重新入网后，发起更改网内信息的情况使用。发起一个序号为 0 的数据包后，必须等待本次泛洪结束后，才可以进行下一次发送，不然会导致重复接收数据包。等待时间为 queue_time * hops。

网络泛洪控制结构体中存有一字节 seqno 序号，seqno 每次发送后都将加 1。但是除了初始化后的首次发送序号为 0，其他时候不会再为 0。除非调用 lwns_netflood_seqno_set 将序号设置为 0。

比如 old_phase_value 设置为 254。当一个节点的序号重新从 255 回到 1 重新开始的时候，原内存中为 255，新数据包序号为 1，差值为 254，大于等于了设置的 old_phase_value，数据包就会认被为是新的数据包，然后数据包就会进接收回调函数处理。

比如 old_phase_value 设置为 2。如果此时，收到的数据包序号为 3，原内存中为 1，

则差值为 2。即使比内存中的大，但是差值不小于设置的 old_phase_value。不会被认为是新的数据包。所以该值应设置的较大些，也可以定时清理内存中存有的序号数据。

netflood 控制结构体中存储有除自身以外 8 个节点的 seq,超出 8 个会删除最旧的节点信息。所以同一时刻内应避免网络中同时存在超过 8 个设备发起的泛洪消息。如果一定需要 8 个以上，可以定义多个 netflood 控制结构体，注册不同的 PORT 和同一个回调函数。

9.3 lwns_netflood_send

```
/*****************************************************************************
 * @fn        lwns_netflood_send
 *
 * @brief     start a netflood,used lwns_controller_ptr h.
 *
 * input parameters
 *
 * @param     h - lwns_controller_ptr,the pointer of a lwns_netflood_controller,
 *            which must be initialized by lwns_netflood_init.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    return 1 if success,0 if failed.
 */
extern int lwns_netflood_send(lwns_controller_ptr h);
```

调用此函数会立刻开始一次网络泛洪发送。

nerflood 模块在发送时，内部会存储有本机的 seq，会自动添加到包头内部。无需用户干预。保险起见，用户最好在自己的数据区，也添加一个包序号，自行校验。

9.4 lwns_netflood_close

```
/*****************************************************************************
 * @fn        lwns_netflood_close
 *
 * @brief     close a port for netflood.
 *
 * input parameters
 *
 * @param     h - lwns_controller_ptr,the pointer of a lwns_netflood_controller,
 *            which must be initialized by lwns_netflood_init.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    None.
 */
extern void lwns_netflood_close(lwns_controller_ptr h);
```

当用户不需要再接收某个端口的消息时,就可以调用 close 函数将该端口关闭。关闭后,

rf 接收到的数据解析的端口号匹配不到相应的控制结构体，所以就会被丢弃。

9.5 lwns_netflood_clean_info

```
/**********************************************************************************
 * @fn      lwns_netflood_clean_info
 *
 * @brief   clean senders and seqno in lwns_netflood_controller.
 *
 * input parameters
 *
 * @param   h - lwns_controller_ptr,the pointer of a lwns_netflood_controller/lwns_uninetflood_controller/lwns_multinetflood_controller,
 *              which must be initialized by these controller.
 *
 * output parameters
 *
 * @param   None.
 *
 * @return  return 1 if busy,0 if free.
 */
extern void lwns_netflood_clean_info(lwns_controller_ptr h);
```

该函数作用为清除掉 netflood 控制结构体中存放的其他 netflood 节点信息。参数可以是 netflood、uninetflood、multinetflood 和 mesh 的控制结构体指针。

9.6 lwns_netflood_remove_an_info

```
/**********************************************************************************
 * @fn      lwns_netflood_remove_an_info
 *
 * @brief   remove a sender info in lwns_netflood_controller.
 *
 * input parameters
 *
 * @param   h - lwns_controller_ptr,the pointer of a lwns_netflood_controller/lwns_uninetflood_controller/lwns_multinetflood_controller,
 *              which must be initialized by these controller.
 * @param   sender - the addr of the sender node.
 *
 * output parameters
 *
 * @param   None.
 *
 * @return  None.
 */
extern void lwns_netflood_remove_an_info(lwns_controller_ptr h, lwns_addr_t* sender);
```

该函数为清除 netflood 控制结构体中存有的指定发送者和序号。参数可以是 netflood、uninetflood、multinetflood 和 mesh 的控制结构体指针。

9.7 lwns_netflood_seqno_set

```
/*****************************************************************************
 * @fn        lwns_netflood_seqno_set
 *
 * @brief     set seqno of the lwns_netflood_controller.
 *
 * input parameters
 *
 * @param     h - lwns_controller_ptr,the pointer of a lwns_netflood_controller/
 *            lwns_uninetflood_controller/lwns_multinetflood_controller/lwns_mesh_controller,
 *            which must be initialized by these controller init function.
 * @param     seqno   -value:(0-255)   - seqno need to set.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    return 1 if busy,0 if free.
 */
extern void lwns_netflood_seqno_set(lwns_controller_ptr h,uint8_t seqno);
```

该函数作用为设置 netflood 控制结构体中存放的本节点的发送序号信息。参数可以是 netflood、uninetflood 和 multinetflood 的控制结构体指针

9.8 example

本章例程为 lwns_netflood_example.c，本次将采用三个节点进行实验。

使用 netflood 模块编写程序必须要做的事：

i. 使用 lwns_netflood_controller 定义一块内存空间

```
static lwns_netflood_controller netflood;//网络泛洪控制结构体
```

ii. 编写相关回调函数

netflood 模块有三个回调函数：recv、sent 和 dropped，并且需要把函数指针赋予回调函数结构体中。

接收回调函数：

```
static int netflood_recv(lwns_controller_ptr ptr,
     const lwns_addr_t *from,
     const lwns_addr_t *originator, uint8_t hops) {
  uint8_t len;
  len = lwns_buffer_datalen(); //获取当前缓冲区接收到的数据长度
  PRINTF("netflood %d rec %02x %02x %02x %02x %02x %02x,hops=%d\r\n",get_lwns_object_port(ptr),
      from->v8[0], from->v8[1], from->v8[2], from->v8[3], from->v8[4],
      from->v8[5], hops);//打印转发者，即为收到的本次转发数据是谁转发的。
  PRINTF("netflood orec %02x %02x %02x %02x %02x %02x,hops=%d\r\n",
      originator->v8[0], originator->v8[1], originator->v8[2],
      originator->v8[3], originator->v8[4], originator->v8[5],
      hops);//打印出信息发起者，即为发起本次网络泛洪的节点地址。
  lwns_buffer_save_data(RX_DATA); //接收数据到用户数据区域
  PRINTF("data:");
  for (uint8_t i = 0; i < len; i++) {
    PRINTF("%02x ", RX_DATA[i]);
  }
  PRINTF("\n");
  return 1;//返回1，则本节点将继续发送netflood，转发数据
  //return 0;//返回0，则本节点不再继续netflood，直接终止
}
```

发送完成回调函数：

```
static void netflood_sent(lwns_controller_ptr ptr) {
  PRINTF("netflood %d sent\n",get_lwns_object_port(ptr));
}
```

数据包丢弃回调函数：

```
static void netflood_dropped(lwns_controller_ptr ptr) {
  PRINTF("netflood %d dropped\n",get_lwns_object_port(ptr));
}
```

声明回调函数结构体：

```
static const struct lwns_netflood_callbacks callbacks = { netflood_recv,
    netflood_sent, netflood_dropped };
```

准备工作都做好后，即可开始初始化相关工作，通过 netflood 模块的 init 函数进行初始化：

```
void lwns_netflood_process_init(void) {
  netflood_taskID = TMOS_ProcessEventRegister(lwns_netflood_ProcessEvent);
  for(uint8_t i = 0; i < LWNS_DATA_SIZE ;i++){
    TX_DATA[i]=i;
  }
  lwns_netflood_init(&netflood,
      137,//打开一个端口号为137的泛洪结构体
      HTIMER_SECOND_NUM*1,//等待转发时间
      1,//在等待期间，接收到几次同样的数据包就取消本数据包的发送
      3,//最大转发层级
      FALSE,//在等待转发过程中，收到了新的需要转发的数据包，旧数据包是立刻发送出去还是丢弃，FALSE为立刻发送，TRUE为丢弃。
      50,//网络恢复参数，该值定义了一个差距，如果包序号比内存内保存的数据包序号小的值大于此值，则会认为网络故障恢复，继续接收该数据包。
      //同时，该值也决定了判定为新数据包的差值，即来自同一个节点的新数据包的序号不可以比内存中的大过多，即比此值还大。
      //例如，内存中保存的为10，新数据包序号为60，差值为50，大于等于此时设置的50，所以将不会被认为为新的数据包，被丢弃。
      //只有序号为59，差值为49，小于该值，才会被接收。
      &callbacks);//返回0代表打开失败。返回1打开成功。
#if 1
  tmos_start_task(netflood_taskID, NETFLOOD_EXAMPLE_TX_PERIOD_EVT,
      MS1_TO_SYSTEM_TIME(1000));
#endif
}
```

在 tmos 中的 NETFLOOD_EXAMPLE_TX_PERIOD_EVT 任务处理中，进行周期性的发送：

```
if (events & NETFLOOD_EXAMPLE_TX_PERIOD_EVT) {
  PRINTF("send\n");
  rf_network_buffer_load_data(TX_DATA, sizeof(TX_DATA)); //载入需要发送的数据到缓冲区
  rf_network_netflood_send(&netflood); //发送网络泛洪数据包
  tmos_start_task(netflood_taskID, NETFLOOD_EXAMPLE_TX_PERIOD_EVT,
      MS1_TO_SYSTEM_TIME(1000)); //10s发送一次
  return (events ^ NETFLOOD_EXAMPLE_TX_PERIOD_EVT);
}
```

将一个节点的 netflood 发送打开，其他两个节点关闭。即：

```
#if 0
  tmos_start_task(netflood_taskID, NETFLOOD_EXAMPLE_TX_PERIOD_EVT,
      MS1_TO_SYSTEM_TIME(1000));
#endif
```

将程序编译后烧录进三个节点中，观察串口数据如下：

```
[14:32:26.798]收←◆netflood 137 rec 66 df 38 e4 c2
netflood orec 66 df 38 e4 c2 84, seqno=0, hops=0
data:30 31 32 33 34 35 36 37 38 39

[14:32:26.941]收←◆netflood 137 dropped

[14:32:36.799]收←◆netflood 137 rec 66 df 38 e4 c2
netflood orec 66 df 38 e4 c2 84, seqno=1, hops=0
data:30 31 32 33 34 35 36 37 38 39

[14:32:36.936]收←◆netflood 137 dropped

[14:32:46.799]收←◆netflood 137 rec 66 df 38 e4 c2
netflood orec 66 df 38 e4 c2 84, seqno=2, hops=0
data:30 31 32 33 34 35 36 37 38 39

[14:32:46.946]收←◆netflood 137 dropped

[14:32:56.799]收←◆netflood 137 rec 66 df 38 e4 c2
netflood orec 66 df 38 e4 c2 84, seqno=3, hops=0
data:30 31 32 33 34 35 36 37 38 39

[14:32:56.931]收←◆netflood 137 dropped

[14:33:06.799]收←◆netflood 137 rec 66 df 38 e4 c2
netflood orec 66 df 38 e4 c2 84, seqno=4, hops=0
data:30 31 32 33 34 35 36 37 38 39

[14:33:06.907]收←◆netflood 137 dropped
```

```
[14:32:26.798]收←◆netflood 137 rec 66 df 38 e4 c
netflood orec 66 df 38 e4 c2 84, seqno=0, hops=0
data:30 31 32 33 34 35 36 37 38 39

[14:32:26.940]收←◆netflood 137 sent

[14:32:36.799]收←◆netflood 137 rec 66 df 38 e4 c
netflood orec 66 df 38 e4 c2 84, seqno=1, hops=0
data:30 31 32 33 34 35 36 37 38 39

[14:32:36.935]收←◆netflood 137 sent

[14:32:46.799]收←◆netflood 137 rec 66 df 38 e4 c
netflood orec 66 df 38 e4 c2 84, seqno=2, hops=0
data:30 31 32 33 34 35 36 37 38 39

[14:32:46.944]收←◆netflood 137 sent

[14:32:56.799]收←◆netflood 137 rec 66 df 38 e4 c
netflood orec 66 df 38 e4 c2 84, seqno=3, hops=0
data:30 31 32 33 34 35 36 37 38 39

[14:32:56.930]收←◆netflood 137 sent

[14:33:06.799]收←◆netflood 137 rec 66 df 38 e4 c
netflood orec 66 df 38 e4 c2 84, seqno=4, hops=0
data:30 31 32 33 34 35 36 37 38 39

[14:33:06.905]收←◆netflood 137 sent
```

```
[14:32:26.797]收←◆netflood 137 sent

[14:32:36.797]收←◆netflood 137 sent

[14:32:46.797]收←◆netflood 137 sent

[14:32:56.797]收←◆netflood 137 sent

[14:33:06.797]收←◆netflood 137 sent
```

通过观察可以发现，源节点发送数据后，其他节点收到后会等待，一旦在等待中收到了指定数量的数据包，就会放弃发送，调用 dropped 回调函数。并且 netflood 模块会判断当前数据包是否为之前收到的数据包，判断依据就是发送时候的指示序号数是否小于内部存有的序号数。

10. 单播网络泛洪

uninetflood 为单播网络泛洪，基于 netflood 实现，区别在于指定了接收者，接收者收到数据包后，不再继续转发。

使用 uninetflood 需要用 lwns_uninetflood_controller 定义一块内存空间。

```
//can not be modified
#define LWNS_UNINETFLOOD_CONTROLLER_U32_SIZE   32
typedef struct _lwns_uninetflood_controller_struct {
  uint32_t data[LWNS_UNINETFLOOD_CONTROLLER_U32_SIZE];
} lwns_uninetflood_controller;
```

10.1 lwns_uninetflood_callbacks

```
struct lwns_uninetflood_callbacks {
  void (*recv)(lwns_controller_ptr ptr, const lwns_addr_t *sender, uint8_t hops);
  void (*sent)(lwns_controller_ptr ptr);
};
```

uninetflood 网络泛洪模块有两个回调函数：

recv 为接收回调函数，其中的参数 sender 为本次网络泛洪发起人的地址，hops 为当前数据包是第几次转发。

sent 函数为发送后调用，表明完成本次发送。

10.2 lwns_uninetflood_init

```
/******************************************************************************
 * @fn        lwns_uninetflood_init
 *
 * @brief     open a port for uninetflood.
 *
 * input parameters
 *
 * @param     h - lwns_controller_ptr,the pointer of a lwns_uninetflood_controller.
 * @param     port_num    -value:(1-255)   - port to recognize data
 * @param     queue_time - wait queue_time/2  with a random time to receive message.
 * @param     dups - cannot be 0,if this value is set to over 1,
 *            this controller will wait to receive the same message over this value,
 *            and then drop the message.
 * @param     hops - the message max transmit life time.when another node send this message,this value will -1 until it is 0.
 * @param     drop_old_packet - FALSE/TRUE,TURE will drop the old packet if we prepare to send a new packet.FALSE will send old message immediately.
 * @param     old_phase_value - this value is for self-healing when network error appeared,
 *                if seqno is too many than old_phase_value smaller than in memory,
 *                it will still handle this message.
 * @param     flood_choice - decide whether it will forword a received packet which is not for itself.
 * @param     callbacks defined with lwns_uninetflood_callbacks
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    return 1 if success,0 if failed.
 */
extern int lwns_uninetflood_init(lwns_controller_ptr h, uint8_t port_num,
    lwns_clock_time_t queue_time, uint8_t dups, uint8_t hops, uint8_t drop_old_packet,uint8_t old_phase_value,
    uint8_t flood_choice, const struct lwns_uninetflood_callbacks *u);
```

uninetflood_init 一共采用了 9 个参数：

其他参数都和 netflood_init 中参数含义一样。

只有第七个参数 flood_choice，该值决定了 uninetflood 模块会不会对非发往本机的

数据包进行转发。netflood 是否转发是在接收回调函数中，通过返回 0 或 1 决定的。而 uninetflood 是通过初始化时，设置此值决定的，TRUE 就会转发非本机数据包，FALSE 就不会转发。

10.3 lwns_uninetflood_send

```
/*****************************************************************************
 * @fn        lwns_uninetflood_send
 *
 * @brief     start a uninetflood,send message to dst. used lwns_controller_ptr h.
 *
 * input parameters
 *
 * @param     h - lwns_controller_ptr,the pointer of a lwns_uninetflood_controller,
 *            which must be initialized by lwns_uninetflood_init.
 * @param     dst - the addr of the dest node.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    return 1 if success,0 if failed.
 */
extern int lwns_uninetflood_send(lwns_controller_ptr h, lwns_addr_t *dst);
```

调用此函数会立刻开始一次给参数二目标节点的网络泛洪发送消息。

uninetflood 模块在发送时，内部会存储有本机的 seq，会自动添加到包头内部。无需用户干预。保险起见，用户最好在自己的数据区，也添加一个包序号，自行校验。

10.4 lwns_uninetflood_close

```
/*****************************************************************************
 * @fn        lwns_uninetflood_close
 *
 * @brief     close a port for uninetflood.
 *
 * input parameters
 *
 * @param     h - lwns_controller_ptr,the pointer of a lwns_uninetflood_controller,
 *            which must be initialized by lwns_uninetflood_init.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    None.
 */
extern void lwns_uninetflood_close(lwns_controller_ptr h);
```

当用户不需要再接收某个端口的消息时,就可以调用 close 函数将该端口关闭。关闭后,rf 接收到的数据解析的端口号匹配不到相应的控制结构体，所以就会被丢弃。

10.5 example

本章例程为 lwns_uninetflood_example.c，本次将采用三个节点进行实验。

使用 uninetflood 模块编写程序必须要做的事：

iii. 使用 lwns_uninetflood_controller 定义一块内存空间

```c
static lwns_uninetflood_controller uninetflood;//单播网络泛洪控制结构体
```

iv. 编写相关回调函数

uninetflood 模块有两个回调函数：recv 和 sent，并且需要把函数指针赋予回调函数结构体中。

接收回调函数：

```c
static void uninetflood_recv(lwns_controller_ptr ptr,const lwns_addr_t *sender, uint8_t hops){
  uint8_t len;
  len = lwns_buffer_datalen(); //获取当前缓冲区接收到的数据长度
  lwns_buffer_save_data(RX_DATA); //接收数据到用户数据区域
  PRINTF("uninetflood %d rec from %02x %02x %02x %02x %02x %02x\n",
      get_lwns_object_port(ptr),
      sender->v8[0], sender->v8[1], sender->v8[2], sender->v8[3],
      sender->v8[4], sender->v8[5]);//from为接收到的数据的发送方地址
  PRINTF("data:");
  for (uint8_t i = 0; i < len; i++) {
    PRINTF("%02x ", RX_DATA[i]);//打印出数据
  }
  PRINTF("\n");
}
```

发送完成回调函数：

```c
static void uninetflood_sent(lwns_controller_ptr ptr) {
  PRINTF("uninetflood %d sent\n",get_lwns_object_port(ptr));
}
```

声明回调函数结构体：

```c
static const struct lwns_uninetflood_callbacks uninetflood_callbacks =
{uninetflood_recv,uninetflood_sent};//注册单播网络泛洪回调函数
```

准备工作都做好后，即可开始初始化相关工作，通过 uninetflood 模块的 init 函数进行初始化：

```c
void lwns_uninetflood_process_init(void) {
  uninetflood_taskID = TMOS_ProcessEventRegister(lwns_uninetflood_ProcessEvent);
  for(uint8_t i = 0; i < LWNS_DATA_SIZE ;i++){
    TX_DATA[i]=i;
  }
  lwns_uninetflood_init(&uninetflood,
        137,//打开一个端口号为137的单播网络泛洪结构体
        HTIMER_SECOND_NUM*2,//最大等待转发时间
        1,//在等待期间，接收到几次同样的数据包就取消本数据包的发送
        3,//最大转发层级
        FALSE,//在等待转发过程中，收到了新的需要转发的数据包，旧数据包是立刻发送出去还是丢弃，FALSE为立刻发送，TRUE为丢弃。
        50,//网络恢复参数，该值定义了一个差距，如果包序号比内存内保存的数据包序号小的值大于此值，则会认为网络故障恢复，继续接收该数据包。
        //同时，该值也决定了判定为新数据包的差值，即来自同一个节点的新数据包的序号不可以比内存中的大过多，即比此值还大。
        //例如，内存中保存的为10，新数据包序号为60，差值为50，大于等于此时设置的50，所以将不会被认为为新的数据包，被丢弃。
        //只有序号为59，差值为49，小于该值，才会被接收。
        TRUE,//本机是否转发目标非本机的数据包，类似于蓝牙mesh是否启用relay功能。
        &uninetflood_callbacks
        );//返回0代表打开失败。返回1打开成功。
  tmos_start_task(uninetflood_taskID, UNINETFLOOD_EXAMPLE_TX_PERIOD_EVT,
      MS1_TO_SYSTEM_TIME(1000));
}
```

在 tmos 中的 UNINETFLOOD_EXAMPLE_TX_PERIOD_EVT 任务处理中，进行周期性的发送：

```c
if (events & UNINETFLOOD_EXAMPLE_TX_PERIOD_EVT) {
  uint8 temp;
  temp = TX_DATA[0];
  for (uint8 i = 0; i < 9; i++) {
    TX_DATA[i] = TX_DATA[i + 1];
  }
  TX_DATA[9] = temp;
  lwns_buffer_load_data(TX_DATA, sizeof(TX_DATA));//载入需要发送的数据到缓冲区
  lwns_uninetflood_send(&uninetflood,&dst_addr);//单播网络泛洪发送数据给目标地址
  tmos_start_task(uninetflood_taskID, UNINETFLOOD_EXAMPLE_TX_PERIOD_EVT,
      MS1_TO_SYSTEM_TIME(1000));//周期性发送
  return events ^ UNINETFLOOD_EXAMPLE_TX_PERIOD_EVT;
}
```

将一个节点的 uninetflood 发送打开，其他两个节点关闭。即：

```c
            );//返回0代表打开失败。返回1打开成功。
//    tmos_start_task(uninetflood_taskID, UNINETFLOOD_EXAMPLE_TX_PERIOD_EVT,
//            MS1_TO_SYSTEM_TIME(1000));
}
```

需要发起 uninetflood 发送的节点把这句话取消注释。

将会发送 uninetflood 数据包的节点中，程序的目标节点地址改为另外两个节点中的一个。

将程序编译后烧录进三个节点中，观察串口数据如下：

```
[13:38:52.809]收←◆send rx msg
uninetflood 137 rec from d9 37 3c e4 c2 84
data:38 39 30 31 32 33 34 35 36 37
send rx msg

[13:38:53.245]收←◆send rx msg
send rx msg

[13:38:53.807]收←◆send rx msg
uninetflood 137 rec from d9 37 3c e4 c2 84
data:39 30 31 32 33 34 35 36 37 38
send rx msg

[13:38:54.045]收←◆send rx msg
send rx msg

[13:38:54.808]收←◆send rx msg
uninetflood 137 rec from d9 37 3c e4 c2 84
data:30 31 32 33 34 35 36 37 38 39
send rx msg

[13:38:55.125]收←◆send rx msg
send rx msg

[13:38:55.808]收←◆send rx msg
uninetflood 137 rec from d9 37 3c e4 c2 84
data:31 32 33 34 35 36 37 38 39 30
send rx msg

[13:38:56.046]收←◆crc error
send rx msg
```

```
rand send:5

[13:38:53.245]收←◆send rx msg
send rx msg

[13:38:53.784]收←◆uninetflood 137 sent
rand send:2

[13:38:54.046]收←◆send rx msg
send rx msg

[13:38:54.784]收←◆uninetflood 137 sent
rand send:4

[13:38:55.125]收←◆send rx msg
send rx msg

[13:38:55.784]收←◆uninetflood 137 sent
rand send:4

[13:38:56.046]收←◆send rx msg
send rx msg

[13:38:56.784]收←◆uninetflood 137 sent
rand send:6

[13:38:57.127]收←◆send rx msg
send rx msg

[13:38:57.784]收←◆uninetflood 137 sent
rand send:4
```

```
send rx msg

[13:38:53.241]收←◆rand send:4

[13:38:53.807]收←◆send rx msg
send rx msg

[13:38:54.040]收←◆rand send:5

[13:38:54.808]收←◆send rx msg
send rx msg

[13:38:55.120]收←◆rand send:4

[13:38:55.808]收←◆send rx msg
send rx msg

[13:38:56.041]收←◆rand send:5

[13:38:56.809]收←◆send rx msg
send rx msg

[13:38:57.121]收←◆rand send:8

[13:38:57.808]收←◆send rx msg
send rx msg

[13:38:58.101]收←◆rand send:3

[13:38:58.792]收←◆send rx msg
send rx msg
```

只有接收到目标地址是自己的 uninetflood 数据包，才会调用 recv 回调函数。不是自己的数据包，不会调用 recv 回调函数。

uninetflood 将不会再有 dropped 回调函数处理，它只关注与源节点的发送和目标节点的接收。

11. 组播网络泛洪

multinetflood 为组播网络泛洪，基于 netflood 实现，区别在于指定了接收组，接收组内成员收到数据包后，将会进入接收回调函数。

使用 multinetflood 需要用 lwns_multinetflood_controller 定义一块内存空间。

```
//can not be modified
#define LWNS_MULTINETFLOOD_CONTROLLER_U32_SIZE   33
typedef struct _lwns_multinetflood_controller_struct {
  uint32_t data[LWNS_MULTINETFLOOD_CONTROLLER_U32_SIZE];
} lwns_multinetflood_controller;
```

11.1 lwns_multinetflood_callbacks

```
struct lwns_multinetflood_callbacks {
  void (*recv)(lwns_controller_ptr ptr,uint16_t subaddr, const lwns_addr_t *sender, uint8_t hops);
  void (*sent)(lwns_controller_ptr ptr);
};
```

multinetflood 网络泛洪模块有两个回调函数：

recv 为接收回调函数，其中 subaddr 为本次接收到的订阅组地址，只有是自己订阅组的地址才会接收,sender 为本次网络泛洪发起人的地址,hops 为当前数据包是第几次转发。

sent 函数为发送后调用，表明完成本次发送。

11.2 lwns_multinetflood_init

```
/*********************************************************************************
 * @fn        lwns_multinetflood_init
 *
 * @brief     open a port for multinetflood.
 *
 * input parameters
 *
 * @param      h - lwns_controller_ptr,the pointer of a lwns_uninetflood_controller.
 * @param      port_num   -value:(1-255)   - port to recognize data
 * @param      queue_time - wait queue_time/2  with a random time to receive message.
 * @param      dups - cannot be 0,if this value is set to over 1,
 *             this controller will wait to receive the same message over this value,
 *             and then drop the message.
 * @param      hops - the message max transmit life time.when another node send this message,this value will -1 until it is 0.
 * @param      drop_old_packet - FALSE/TRUE,TURE will drop the old packet if we prepare to send a new packet.FALSE will send old message immediately.
 * @param      old_phase_value - this value is for self-healing when network error appeared,
 *                 if seqno is too many than old_phase_value smaller than in memory,
 *                 it will still handle this message.
 * @param      flood_choice - decide whether it will forword a received packet which is not for itself.
 * @param      subaddr    - the array pointer of subscribe addresses.
 * @param      sub_num     - the number of subscribe addresses.
 * @param      callbacks defined with lwns_multinetflood_callbacks
 *
 * output parameters
 *
 * @param      None.
 *
 * @return     return 1 if success,0 if failed.
 */
extern int lwns_multinetflood_init(lwns_controller_ptr h,
     uint8_t port_num, lwns_clock_time_t queue_time,
     uint8_t dups, uint8_t hops,uint8_t drop_old_packet,uint8_t old_phase_value,uint8_t flood_choice,
     uint16_t *subaddr ,uint8_t sub_num,
     const struct lwns_multinetflood_callbacks *u);
```

multinetflood_init 一共采用了 11 个参数：

很多参数都和 uninetflood_init 中参数含义一样。

flood_choice，该值和 uninetflood 模块中的 flood_choice 不同，在 multinetflood 中，该值决定了 multinetflood 会不会对数据包进行转发。无论是不是自己组内的数据包。TRUE 就会转发数据包，FALSE 就不会转发

第九个参数 subaddr 为 multinetflood 订阅组的地址起始指针，第十个参数 sub_num 为订阅组的地址数量，只有收到了指定订阅组地址的数据包才会触发接收回调函数。

11.3 lwns_multinetflood_send

```
/***************************************************************************
 * @fn      lwns_multinetflood_send
 *
 * @brief    start a multinetflood,send message to a subaddr. used lwns_controller_ptr h.
 *
 * input parameters
 *
 * @param     h - lwns_controller_ptr,the pointer of a lwns_uninetflood_controller,
 *           which must be initialized by lwns_uninetflood_init.
 * @param     subaddr - the dst subaddr.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    return 1 if success,0 if failed.
 */
extern int lwns_multinetflood_send(lwns_controller_ptr h, uint16_t subaddr);
```

调用此函数会立刻开始一次在参数二订阅组地址上的网络泛洪发送消息。

multinetflood 模块在发送时，内部会存储有本机的 seq，会自动添加到包头内部。无需用户干预。保险起见，用户最好在自己的数据区，也添加一个包序号，自行校验。

11.4 lwns_multinetflood_close

```
/******************************************************************************
 * @fn      lwns_multinetflood_close
 *
 * @brief   close a port for multinetflood.
 *
 * input parameters
 *
 * @param   h - lwns_controller_ptr,the pointer of a lwns_multinetflood_controller,
 *          which must be initialized by lwns_multinetflood_init.
 *
 * output parameters
 *
 * @param   None.
 *
 * @return  None.
 */
extern void lwns_multinetflood_close(lwns_controller_ptr h);
```

当用户不需要再接收某个端口的消息时,就可以调用 close 函数将该端口关闭。关闭后,rf 接收到的数据解析的端口号匹配不到相应的控制结构体，所以就会被丢弃。

11.5 example

本章例程为 lwns_multinetflood_example.c，本次将采用三个节点进行实验。

使用 multinetflood 模块编写程序必须要做的事：

v. 使用 lwns_multinetflood_controller 定义一块内存空间

```
static lwns_multinetflood_controller multinetflood;//声明组播网络泛洪控制结构体
```

vi. 编写相关回调函数

multinetflood 模块有两个回调函数：recv 和 sent，并且需要把函数指针赋予回调函数结构体中。

接收回调函数：

```
static void multinetflood_recv(lwns_controller_ptr ptr,uint16_t subaddr,const lwns_addr_t *sender, uint8_t hops){
  uint8_t len;
  len = lwns_buffer_datalen(); //获取当前缓冲区接收到的数据长度
  lwns_buffer_save_data(RX_DATA); //接收数据到用户数据区域
  PRINTF("multinetflood %d rec from %02x %02x %02x %02x %02x %02x\n",
      get_lwns_object_port(ptr),
      sender->v8[0], sender->v8[1], sender->v8[2], sender->v8[3],
      sender->v8[4], sender->v8[5]);//from为接收到的数据的发送方地址
  PRINTF("subaddr:%d,data:",subaddr);
  for (uint8_t i = 0; i < len; i++) {
    PRINTF("%02x ", RX_DATA[i]);//打印出数据
  }
  PRINTF("\n");
}
```

发送完成回调函数：

```
static void multinetflood_sent(lwns_controller_ptr ptr) {
  PRINTF("multinetflood %d sent\n",get_lwns_object_port(ptr));
}
```

声明回调函数结构体：

```
static const struct lwns_multinetflood_callbacks multinetflood_callbacks =
{multinetflood_recv,multinetflood_sent};//注册组播网络泛洪回调函数
```

准备工作都做好后，即可开始初始化相关工作，通过 multinetflood 模块的 init 函数进行初始化：

```
void lwns_multinetflood_process_init(void) {
  multinetflood_taskID = TMOS_ProcessEventRegister(lwns_multinetflood_ProcessEvent);
  for(uint8_t i = 0; i < LWNS_DATA_SIZE ;i++){
    TX_DATA[i]=i;
  }
  lwns_multinetflood_init(&multinetflood,
        137,//打开一个端口号为137的组播网络泛洪结构体
        HTIMER_SECOND_NUM,//最大等待转发时间
        1,//在等待期间，接收到几次同样的数据包就取消本数据包的发送
        3,//最大转发层级
        FALSE,//在等待转发过程中，收到了新的需要转发的数据包，旧数据包是立刻发送出去还是丢弃，FALSE为立刻发送，TRUE为丢弃。
        50,//网络恢复参数，该值定义了一个差距，如果包序号比内存内保存的数据包序号小的值大于此值，则会认为网络故障恢复，继续接收该数据包。
        //同时，该值也决定了判定为新数据包的差值，即来自同一个节点的新数据包的序号不可以比内存中的大过多，即比此值还大。
        //例如，内存中保存的为10，新数据包序号为60，差值为50，大于等于此时设置的50，所以将不会被认为为新的数据包，被丢弃。
        //只有序号为59，差值为49，小于该值，才会被接收。
        TRUE,//本机是否转发目标非本机的数据包，类似于蓝牙mesh是否启用relay功能。
        subaddrs,//订阅的地址数组指针
        SUBADDR_NUM,//订阅地址数量
        &multinetflood_callbacks
        );//返回0代表打开失败。返回1打开成功。
#if 1
  tmos_start_task(multinetflood_taskID, MULTINETFLOOD_EXAMPLE_TX_PERIOD_EVT,
      MS1_TO_SYSTEM_TIME(1000));
#endif
}
```

在 tmos 中的 MULTINETFLOOD_EXAMPLE_TX_PERIOD_EVT 任务处理中,进行周期性的发送：

```
if (events & MULTINETFLOOD_EXAMPLE_TX_PERIOD_EVT) {
  uint8 temp;
  temp = TX_DATA[0];
  for (uint8 i = 0; i < 9; i++) {
    TX_DATA[i] = TX_DATA[i + 1];
  }
  TX_DATA[9] = temp;
  lwns_buffer_load_data(TX_DATA, sizeof(TX_DATA));//载入需要发送的数据到缓冲区
  if(subaddrs_index >= SUBADDR_NUM){
    subaddrs_index = 0;
  }
  lwns_multinetflood_send(&multinetflood,subaddrs[subaddrs_index]);//组播网络泛洪发送数据到订阅地址
  subaddrs_index++;

  tmos_start_task(multinetflood_taskID, MULTINETFLOOD_EXAMPLE_TX_PERIOD_EVT,
      MS1_TO_SYSTEM_TIME(1000));//周期性发送
  return events ^ MULTINETFLOOD_EXAMPLE_TX_PERIOD_EVT;
}
```

将一个节点的 multinetflood 发送打开，另一节点关闭。即：

```
#if 0
  tmos_start_task(multinetflood_taskID, MULTINETFLOOD_EXAMPLE_TX_PERIOD_EVT,
      MS1_TO_SYSTEM_TIME(1000));
#endif
```

需要发起 multinetflood 发送的节点把这句话#if 改为 1，不需要的改为 0。

将发送的订阅组地址列表修改，发送 multinetflood 的节点：

```
static u8 subaddrs_index = 0;//发送订阅地址序号
#define SUBADDR_NUM    3//订阅地址数量
static uint16 subaddrs[SUBADDR_NUM]={1,2,3};//订阅地址数组
```

另一个节点的三个订阅组地址改为只有一个和发送方的订阅组地址相同：

```
static u8 subaddrs_index = 0;//发送订阅地址序号
#define SUBADDR_NUM    3//订阅地址数量
static uint16 subaddrs[SUBADDR_NUM]={1,20,30};//订阅地址数组
```

将程序编译后烧录进两个节点中，观察串口数据如下：

```
cd ce cf d0 d1
send rx msg

[17:51:04.163]收←◆rand send:3
rand send:6

[17:51:04.490]收←◆send rx msg
send rx msg

[17:51:05.336]收←◆rand send:6
rand send:10

[17:51:05.485]收←◆send rx msg
send rx msg

[17:51:06.394]收←◆rand send:4
rand send:8

[17:51:06.485]收←◆send rx msg
multinetflood 137 rec from a3 df 38 e4 c2 84
subaddr:1,data:00 01 02 03 04 05 06 07 08 09
25 26 27 28 29 2a 2b 2c 2d 2e 2f 30 31 32 33
4f 50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d
79 7a 7b 7c 7d 7e 7f 80 81 82 83 84 85 86 87
a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af b0 b1
cd ce cf d0 d1
send rx msg

[17:51:07.163]收←◆rand send:9
rand send:2

[17:51:07.488]收←◆send rx msg
send rx msg
```

```
[17:51:03.446]收←◆send rx msg
send rx msg
multinetflood 137 sent
rand send:16
rand send:5

[17:51:04.178]收←◆crc error
send rx msg

[17:51:04.475]收←◆multinetflood 137 sent
rand send:16
rand send:6

[17:51:05.346]收←◆send rx msg
crc error

[17:51:05.479]收←◆multinetflood 137 sent
rand send:3
rand send:9

[17:51:06.409]收←◆send rx msg

[17:51:06.480]收←◆multinetflood 137 sent
rand send:1
rand send:6

[17:51:07.163]收←◆crc error
send rx msg

[17:51:07.479]收←◆multinetflood 137 sent
rand send:1
rand send:9
```

发送方周期性的在自己订阅组的三个地址中发送网络泛洪数据包，接收方只有接收到订阅组地址是自己订阅组的地址里的数据包，才会调用 recv 回调函数。

multinetflood 将不会再有 dropped 回调函数处理，它只关注与源节点的发送和订阅组了指定地址的节点的接收。

12. 路由管理

route 为 mesh 必备的模块，为 mesh 提供路由管理功能。mesh 发送数据包时会从路由表中查询发送给目标节点的下一跳路由。然后将数据包发往下一跳即可。

mesh 节点收到了其他节点发往另一节点的数据转发请求，也会根据自身是否具有路由功能，去查询路由表，再将数据转发到下一跳节点。

12.1 lwns_route_entry 结构体

```
//the structure of route table
struct lwns_route_entry {
   struct lwns_route_entry *next;
   lwns_addr_t dest;
   lwns_addr_t nexthop;
   uint8_t cost;
   uint8_t time;
};
```

路由表，采用链表方式进行管理。

路由表条目中，只会存有前往目标节点（dest）的下一跳节点地址（nexthop）、路径消耗（cost），还有就是路由条目的生存时间（time）。

mesh 模块查询路由表，得出目标节点的下一跳地址，将数据通过 unicast 发送至下一跳节点，下一跳节点通过查询路由表继续发送往下一跳，如此循环往复，直到到达目的地址。

12.2 lwns_route_init

```
/*********************************************************************************
 * @fn      lwns_route_init
 *
 * @brief    init lwns_route_management.
 *
 * input parameters
 *
 * @param     route_entry_duplicate - decide route table can save one or more route entry to a dest.
 * @param     max_life_time - 0-255,max_life_time is check period times,if max_life_time = 0,
 *              it will not enable route auto clean.route will be always alive.
 *              you can manage route table by yourself.
 * @param     periodic_time - a route table entry real life time is (max_life_time*periodic_time)
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    None.
 */
extern void lwns_route_init(uint8_t route_entry_duplicate, uint8_t max_life_time,
    lwns_clock_time_t periodic_time);
```

路由表初始化函数，有三个参数。

第一个参数为是否保存前往一个节点的多条路由，FALSE 为不保存，TRUE 为保存，如果保存，那么使用 route_lookup 将会查询所有的路由表，找到一个跳数最少的最佳途径，会比较浪费时间。当节点路由表条目很多的时候，推荐设置为 false，提高运行效率。

第二个参数为一个路由表条目最大检测次数。

第三个参数为检测一次的周期时间。

每隔 periodic_time 时间，都会检测一次路由表。每次检测路由表都会将路由表中 time 值加 1，当 time 值超过设置的最大检测次数时，就会自动删除该路由条目。

当第一个参数 max_life_time 为 0 时，会停止路由表自动检测，所有路由条目不会自动删除，只能靠用户手动进行管理。

使用中，可以中途通过此函数进行设置，打开或者关闭自动删除路由条目的功能。

12.3 lwns_route_add

```
/*****************************************************************************
 * @fn      lwns_route_add
 *
 * @brief   add a route entry to route table.
 *
 * input parameters
 *
 * @param   dest - the dest addr
 * @param   nexthop - nexthop addr to the dest addr
 * @param   cost  - the hops from src to dest.
 *
 * output parameters
 *
 * @param   None.
 *
 * @return  0 is success.
 */
extern int lwns_route_add(const lwns_addr_t *dest,
    const lwns_addr_t *nexthop, uint8_t cost);
```

给路由表添加一个条目。

如果链表已满则删除链表尾部条目以腾出空间，新添加的条目放在链表头部。

如果 route_init 中 route_entry_duplicate 设置为 TRUE，会保存重复的目标节点路由表，使用本函数则不会删除原来的路由条目，数条路由条目共存。如果设置为 false，使用本函数则会删除原来的路由条目，保存新的路由条目。

第一个参数就是此次添加路由条目的目标地址。

第二个参数就是此次前往第一个参数目标地址的下一跳目标地址。mesh 消息将转发至 nexthop 节点，由 nexthop 进行处理。

第三个参数为 cost，为本节点发往目标节点的跳数，不是目标节点发往本节点的跳数，请注意不要理解错误。

12.4 lwns_route_lookup

```
/*****************************************************************************
 * @fn      lwns_route_lookup
 *
 * @brief   look up a route entry from route table.
 *
 * input parameters
 *
 * @param   dest - the dest addr
 *
 * output parameters
 *
 * @param   None.
 *
 * @return  NULL if not find, other is the pointer to the entry.
 */
extern struct lwns_route_entry *lwns_route_lookup(
    const lwns_addr_t *dest);
```

如果 route_init 中，route_entry_duplicate 设置为 TRUE，则路由表可以保存前往一个节点的多条路径。那么使用本函数将会遍历所有的路由条目，寻找到达 dest 花费最少的那条路径。

如果有其他的 cost 更高的路由条目,没有被使用的话,只能由周期检测将其自动删除,或者用户手动删除。

如果 route_entry_duplicate 设置为 FALSE，则路由表里只会存在到 dest 节点的一条路径。route_lookup 不会遍历路由表，找到一个路由条目即返回。

12.5 lwns_route_refresh

```
/*****************************************************************************
 * @fn      lwns_route_refresh
 *
 * @brief   refresh the lifetime of a route entry in route table.
 *
 * input parameters
 *
 * @param   e - lwns_route_entry pointer
 *
 * output parameters
 *
 * @param   None.
 *
 * @return  None.
 */
extern void lwns_route_refresh(struct lwns_route_entry *e);
```

从前面的 lwns_route_init 函数我们知道，一个条目在指定时间之后就会被扔掉，为让某条目不被扔掉，可使用此方法将 time 清 0。

mesh 模块收到消息会自动刷新路由条目，所以该函数是用户有特定需求下使用。

12.6 lwns_route_remove

```
/*******************************************************************************
 * @fn       lwns_route_remove
 *
 * @brief    remove a route entry in route table.
 *
 * input parameters
 *
 * @param    e - lwns_route_entry pointer
 *
 * output parameters
 *
 * @param    None.
 *
 * @return   None.
 */
extern void lwns_route_remove(struct lwns_route_entry *e);
```

使用该函数可以从路由表内存中移除一个指定的路由表条目。

12.7 lwns_route_flush_all

```
/*******************************************************************************
 * @fn       lwns_route_flush_all
 *
 * @brief    remove all route entry in route table.
 *
 * input parameters
 *
 * @param    None.
 *
 * output parameters
 *
 * @param    None.
 *
 * @return   None.
 */
extern void lwns_route_flush_all(void);
```

该函数会清除内存中所有的路由表条目。

12.8 lwns_route_num

```
/*****************************************************************************
 * @fn        lwns_route_num
 *
 * @brief     get number of a route entry in route table.
 *
 * input parameters
 *
 * @param     None.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    the number of a route entry in route table.
 */
extern int lwns_route_num(void);
```

该函数可以获得当前路由表内存中存在多少个路由表条目。

12.9 example

此处用例在 mesh 用例中一同讲解。

13. mesh 组网

在无线 Mesh 网络中，任何无线设备节点都可作为路由器，网络中的每个节点都能发送和接收信号，每个节点都能与一个或多个对等节点进行直接通信。

和蓝牙 mesh 不同，蓝牙 mesh 采用类似本库提供的 netflood 网络泛洪机制实现的无路由转发式的 mesh 网络，本库是用 netflood 发现路由，建立路由表，然后通过单播数据转发，实现 mesh 网络。两者之间有着本质区别。用户也可以利用本库的 uninetflood 模仿蓝牙 mesh 发送数据包。

本库采用了 6 字节地址，提供的 mesh 网络理论可以支持 0xffffffffffff-1 个节点。但是受限于设备内存等情况，一般来说如果需要有足够的内存给路由表使用，一个路由表列表条目占用 21 字节，内存可以放下多少路由表，即支持多少个节点组网。用户网络中的路由节点内存必须要足够大，即可最大限度的实现更多设备组网。

使用 mesh 需要用 lwns_mesh_controller 定义一块内存空间。

```
//can not be modified
#define LWNS_MESH_CONTROLLER_U32_SIZE   55
typedef struct _lwns_mesh_controller_struct {
  uint32_t data[LWNS_MESH_CONTROLLER_U32_SIZE];
} lwns_mesh_controller;
```

13.1 lwns_mesh_callbacks

```
//mesh functions must be used after you initialize route by lwns_route_init.
struct lwns_mesh_callbacks {
  void (*recv)(lwns_controller_ptr ptr, const lwns_addr_t *sender,
      uint8_t hops);
  void (*sent)(lwns_controller_ptr ptr);
  void (*timedout)(lwns_controller_ptr ptr);
};
```

mesh 模块有三个回调函数：

recv 是接收回调函数，表示收到了一条 mesh 消息。Recv 回调函数中第二个参数为发送方节点地址，第三个参数 hops 为当前发送过来数据一共经历了几次转发到达本节点。

sent 函数为本节点成功找到前往目标节点的下一跳地址，并将数据包发送至下一跳后，调用的回调函数。表示发送成功。

timedout 回调函数为超时回调，和 ruc 等超时不一样，当本次发送无法查询到路由条目时，本节点就会发起对目标节点的路由请求，当设定的时间内，没有收到目标节点的路由回复后，就会触发超时函数。

13.2 lwns_mesh_init

```
/******************************************************************************
 * @fn      lwns_mesh_init
 *
 * @brief   open three ports for mesh.
 *
 * input parameters
 *
 * @param   h - lwns_controller_ptr,the pointer of a lwns_mesh_controller.
 * @param   port_nums  -value:(1-255)  - port to recognize data,this controller will open 3 ports.so if you type in 128,then 128,129,130 port will be all opened.
 * @param   route_discovery_hoptime - wait route_discovery_hoptime/2  with a random time to receive message.
 * @param   dups - cannot be 0,if this value is set to over 1,this controller will wait to receive the same message over this value,and then drop the message.
 * @param   hops - the message max transmit life time.when another node send this message,this value will -1 until it is 0.
 * @param   drop_old_packet - FALSE/TRUE,TURE will drop the old packet if we prepare to send a new packet.FALSE will send old message immediately.
 * @param   old_phase_value - this value is for self-healing when network error appeared,
 *              if seqno is too many than old_phase_value smaller than in memory,
 *              it will still handle this message.
 * @param   route_enable - decide whether this node can be a route or not of this lwns_controller_ptr.
 * @param   route_loop_add_enable - decide whether add route_entry with a sender of a message received or not.
 * @param   mesh_over_time - this is to set find route timeout time.
 * @param   callbacks defined with lwns_mesh_callbacks
 *
 * output parameters
 *
 * @param   None.
 *
 * @return  return 1 if success,0 if failed.
 */
extern int lwns_mesh_init(lwns_controller_ptr h, uint8_t port_nums,
    lwns_clock_time_t route_discovery_hoptime, uint8_t dups, uint8_t hops,uint8_t rt_drop_old_packet,
    uint8_t old_phase_value, uint8_t self_route_enable,uint8_t route_loop_add_enable,
    lwns_clock_time_t mesh_over_time,
    const struct lwns_mesh_callbacks *callbacks);
```

mesh 模块使用了 netflood 和 unicast，netflood 用来寻找路由，建立路由关系，而 unicast 用来传递转发数据包。

mesh_init 中，需要 10 个参数：

第三个参数为路由等待时间，和 netflood_init 中的 queue_time 功能相同。

第四个参数为 dups，和 netflood_init 中的 dups 功能相同。该值为寻找路由时使用的 netflood 的参数。

第五个参数为 hops，和 netflood_init 中的 hops 功能相同，在 mesh 中理解为寻找目标节点的路由途径最多经过几次转发。该值为寻找路由时使用的 netflood 的参数。

第六个参数为 drop_old_packet，和 netflood_init 中的 drop_old_packet 一个作用。该值为寻找路由时使用的 netflood 的参数。

第七个参数为 old_phase_value，和 netflood_init 中的 old_phase_value 一个作用。该值为寻找路由时使用的 netflood 的参数。

第八个参数为 route_enable，表示本节点是否开启路由功能，有的节点不具备路由功能，就不会响应其他节点的路由请求，也就是不会转发基于 netflood 的路由请求数据包。只会处理和自身相关的路由请求数据包。

第九个参数为 route_loop_add_enable，决定是否添加路由回路，比如收到了一个来自 a 节点的 mesh 数据包，本机是否需要存储前往 a 节点的路由表。FALSE 不存，TRUE 就会存。

第十个参数为 mesh_over_time，该值为寻找路由的最大时间，如果超过这个时间还没有收到路由回复，则触发超时函数，进入 mesh 模块的超时回调函数中。该值应大于 **route_discovery_hoptime * (hops+1) * 2** ！否则可能会无法添加路由。

第十一个参数为 callbacks，即为 mesh 的回调函数。

13.3 lwns_mesh_send

```
/******************************************************************************
 * @fn        lwns_mesh_send
 *
 * @brief     send a mesh message used lwns_controller_ptr h.
 *
 * input parameters
 *
 * @param     h - lwns_controller_ptr,the pointer of a lwns_mesh_controller,
 *            which must be initialized by lwns_mesh_init.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    return 1 if success,0 if failed.
 */
extern int lwns_mesh_send(lwns_controller_ptr h,
    const lwns_addr_t *dest);
```

调用此函数会开始发送一个 mesh 数据包，首先函数内部会去查询路由表，寻找前往 dest 的最短路径。

如果找到了前往 dest 的路径，就会开始本次发送，将数据包转发到路由表中的 nexthop 节点。

如果没有找到，就会开始寻找路由，即调用 netflood 模块，去寻找路由。收到目标节点的路由回复后，就认为找到了路由。将会自动重发之前的数据包。

如果指定时间内，收不到目标节点的路由回复，就会调用超时回调函数。

13.4 lwns_mesh_close

```
/******************************************************************************
 * @fn        lwns_mesh_close
 *
 * @brief     close ports for mesh.
 *
 * input parameters
 *
 * @param     h - lwns_controller_ptr,the pointer of a lwns_mesh_controller,
 *            which must be initialized by lwns_mesh_init.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    None.
 */
extern void lwns_mesh_close(lwns_controller_ptr h);
```

当用户不需要再接收某个端口的消息时,就可以调用 close 函数将该端口关闭。关闭后,rf 接收到的数据解析的端口号匹配不到相应的控制结构体，所以就会被丢弃。

13.5 example

本章例程为 lwns_mesh_example.c，本次将采用三个以上的节点进行实验，最多 13 个节点。

测试逻辑是，0 号节点为 mesh 中心节点，0 号节点按顺序向其他节点发送数据，其他节点收到数据后将数据发回 0 号节点。其他 12 个节点分为 3 组，每组节点的第 1 号节点都具有路由功能，其他节点不具有路由功能。

例如将 MESH_TEST_SELF_ADDR_IDX 更改为 1，则当前节点初始化后，节点地址变为{0,0,0,0,1,1}。即为 1 号节点。

用户测试自行配置，修改以下的宏定义，更改当前测试的参数和功能：

```
#define MESH_TEST_ROUTE_AUTO   1 //设置是否自动管理路由表
#define MESH_TEST_SELF_ADDR_IDX  0//当前测试中节点地址，在device_addr数组中
#define MESH_TEST_ADDR_MAX_IDX  2//当前测试中，有多少个节点
```

使用 mesh 模块编写程序必须要做的事：

i. 使用 lwns_mesh_controller 定义一块内存空间

```
static lwns_mesh_controller lwns_mesh_test;
```

ii. 编写相关回调函数

mesh 模块有三个回调函数：recv、sent 和 timedout，并且需要把函数指针赋予回调函数结构体中。

接收回调函数：

```
static void lwns_mesh_recv(lwns_controller_ptr ptr,
    const lwns_addr_t *sender, uint8_t hops) {
  PRINTF("mesh %d received from %02x.%02x.%02x.%02x.%02x.%02x: %s (%d)\n",
      get_lwns_object_port(ptr), sender->v8[0], sender->v8[1],
      sender->v8[2], sender->v8[3], sender->v8[4], sender->v8[5],
      (char *)lwns_buffer_dataptr(), lwns_buffer_datalen());
  if (MESH_TEST_SELF_ADDR_IDX != 0) {
    //不为0，证明是从机，收到后回复主机
    lwns_buffer_save_data(TX_DATA);
    tmos_set_event(lwns_mesh_test_taskID, MESH_EXAMPLE_TX_NODE_EVT);
  }
}
```

发送完成回调函数：

```
static void lwns_mesh_sent(lwns_controller_ptr ptr) {
  PRINTF("mesh %d packet sent\n", get_lwns_object_port(ptr));
}
```

超时回调函数：

```
static void lwns_mesh_timedout(lwns_controller_ptr ptr) {
  //寻找路由超时才会进入该回调，如果已经有路由，但是下一跳节点掉线，不会进入该回调。由lwns_route_init(TRUE, 60, HTIMER_SECOND_NUM);来管理掉线时间。
  PRINTF("mesh %d packet timedout\n", get_lwns_object_port(ptr));
}
```

声明回调函数结构体：

```
static const struct lwns_mesh_callbacks callbacks = { lwns_mesh_recv,
        lwns_mesh_sent, lwns_mesh_timedout };
```

准备工作都做好后，通过 mesh 模块的 init 函数进行初始化：

第一次实验采用自动管理路由表：

```
void lwns_mesh_process_init(void) {
  uint8_t route_enable = FALSE;
  lwns_addr_set(
      (lwns_addr_t*) (&device_addr[MESH_TEST_SELF_ADDR_IDX])); //改变lwns内部addr
  if (device_addr[MESH_TEST_SELF_ADDR_IDX].v8[5] == 1) {
    //每一组第一个节点打开路由功能，其他节点不打开路由功能
    route_enable = TRUE;
  }
  lwns_mesh_test_taskID = TMOS_ProcessEventRegister(lwns_mesh_test_ProcessEvent);
#if MESH_TEST_ROUTE_AUTO
  lwns_route_init(TRUE, 60, HTIMER_SECOND_NUM);
#else
  lwns_route_init(TRUE, 0, HTIMER_SECOND_NUM);//disable auto clean route entry
#endif /*MESH_TEST_ROUTE_AUTO*/
  lwns_mesh_init(&lwns_mesh_test, 132, //打开一个端口号132的mesh网络，同时占用后其他的两个端口用作寻找路由，即133，134也同时打开了。
      HTIMER_SECOND_NUM / 2, //netflood的QUEUETIME功能
      1, //netflood的dups功能
      2, //最大跳跃次数层级为5级，即数据包最多可以经过5次转发，超出就丢弃数据包
      FALSE,//在路由请求的转发过程中，收到了新的需要转发的数据包，旧数据包是立刻发送出去还是丢弃，FALSE为立刻发送，TRUE为丢弃。
      50,//网络恢复参数，该值定义了一个差距，如果包序号比内存内保存的数据包序号小的值大于此值，则会认为网络故障恢复，继续接收该数据包。
      //同时，该值也决定了判定为新数据包的差值，即来自同一个节点的新数据包的序号不可以比内存中的大过多，即比此值还大。
      //例如，内存中保存的为10，新数据包序号为60，差值为50，大于等于此时设置的50，所以将不会被认为为新的数据包，被丢弃。
      //只有序号为59，差值为49，小于该值，才会被接收。
      route_enable, //是否使能本机的路由功能，如果为false，不会响应其他节点的路由请求。
      TRUE,//决定是否添加路由回路，比如收到了一个来自a节点的mesh数据包，本机是否需要存储前往a节点的路由表。FALSE不存，TRUE存。
      HTIMER_SECOND_NUM * 2, //路由超时时间，超过时间，停止寻找路由，进入timeout回调，该值应大于 route_discovery_hoptime * (hops+1) * 2
      &callbacks);//没有初始化路由表的话，会返回0，代表打开失败。返回1打开成功。
  if (MESH_TEST_SELF_ADDR_IDX == 0) {//如果是主机，则开始周期性轮训其他节点。
    mesh_test_send_dst = 1;
    tmos_start_task(lwns_mesh_test_taskID, MESH_EXAMPLE_TX_PERIOD_EVT,
            MS1_TO_SYSTEM_TIME(200));
#if MESH_TEST_ROUTE_AUTO
    PRINTF("Auto route\n");
#else...
  }
}
```

mesh 测试的用例比较复杂。初始化设置的东西很多。

首先改变协议栈内部的地址，用 mesh 测试的地址。

然后打开每组一号节点的路由功能。

接下来初始化 mesh 参数。

最后如果是 0 号节点，就会开始周期性对其他节点进行轮询任务。

在 tmos 中的 MESH_EXAMPLE_TX_PERIOD_EVT 任务处理中，进行周期性的发送：

```
if (events & MESH_EXAMPLE_TX_PERIOD_EVT) {//主机周期性轮询从机任务
  uint8_t temp;
  struct lwns_route_entry *rt;
  temp = TX_DATA[0];
  for (uint8_t i = 0; i < 9; i++) {
    TX_DATA[i] = TX_DATA[i + 1];//移位发送数据，以便观察效果
  }
  TX_DATA[9] = temp;
  lwns_buffer_load_data(TX_DATA, sizeof(TX_DATA)); //载入需要发送的数据到缓冲区
  rt = lwns_route_lookup(&device_addr[mesh_test_send_dst]);//在路由表中寻找下一跳路由
  if (rt != NULL) {
    //打印下一跳路由信息
    PRINTF("dst:%d,forwarding to %02x.%02x.%02x.%02x.%02x.%02x\n",
        mesh_test_send_dst, rt->nexthop.v8[0], rt->nexthop.v8[1],
        rt->nexthop.v8[2], rt->nexthop.v8[3], rt->nexthop.v8[4],
        rt->nexthop.v8[5]);
  }else{
     PRINTF("no route to dst:%d\n",mesh_test_send_dst);
  }
  lwns_mesh_send(&lwns_mesh_test, &device_addr[mesh_test_send_dst]);//发送mesh消息
  mesh_test_send_dst++;//轮询目标节点改为下一个
  if (mesh_test_send_dst > MESH_TEST_ADDR_MAX_IDX) {//循环往复轮询
    mesh_test_send_dst = 1;
  }
  tmos_start_task(lwns_mesh_test_taskID, MESH_EXAMPLE_TX_PERIOD_EVT,
      MS1_TO_SYSTEM_TIME(2500));//周期性轮询
  return (events ^ MESH_EXAMPLE_TX_PERIOD_EVT);
}
```

首次发送时，路由表内并没有路由条目，所以会发起寻找路由的请求。

找到路由路径后，就会重新开始发送数据。

forwarding to 后面跟着的就是前往目标节点的下一跳地址。

其他节点收到来自 0 号主机节点的数据包后，就会给其回复自己收到的数据：

```
if (events & MESH_EXAMPLE_TX_NODE_EVT) {//节点给主机发送数据的任务
  struct lwns_route_entry *rt;
  lwns_buffer_load_data(TX_DATA, sizeof(TX_DATA)); //载入需要发送的数据到缓冲区
  rt = lwns_route_lookup(&device_addr[0]);//在路由表中寻找下一跳路由
  if (rt != NULL) {
    //打印出下一跳路由信息
    PRINTF("src:%d,forwarding to %02x.%02x.%02x.%02x.%02x.%02x\n",
        MESH_TEST_SELF_ADDR_IDX, rt->nexthop.v8[0],
        rt->nexthop.v8[1], rt->nexthop.v8[2], rt->nexthop.v8[3],
        rt->nexthop.v8[4], rt->nexthop.v8[5]);
  }
  lwns_mesh_send(&lwns_mesh_test, &device_addr[0]);//调用mesh发送函数，进行数据的发送
  return (events ^ MESH_EXAMPLE_TX_NODE_EVT);
}
```

将程序编译后烧录进三个节点中，观察 0 号节点串口数据如下：

```
[15:23:03.001]收←◆dst:1,forwarding to 00.00.00.00.01.01
mesh 132 packet sent

[15:23:03.028]收←◆mesh 132 received from 00.00.00.00.01.01: 1234567890 (10)

[15:23:07.001]收←◆dst:2,forwarding to 00.00.00.00.01.01
mesh 132 packet sent

[15:23:07.044]收←◆mesh 132 received from 00.00.00.00.01.02: 2345678901 (10)

[15:23:11.001]收←◆dst:1,forwarding to 00.00.00.00.01.01
mesh 132 packet sent

[15:23:11.028]收←◆mesh 132 received from 00.00.00.00.01.01: 3456789012 (10)

[15:23:15.001]收←◆dst:2,forwarding to 00.00.00.00.01.01
mesh 132 packet sent

[15:23:15.044]收←◆mesh 132 received from 00.00.00.00.01.02: 4567890123 (10)

[15:23:19.000]收←◆dst:1,forwarding to 00.00.00.00.01.01
mesh 132 packet sent
mesh 132 received from 00.00.00.00.01.01: 5678901234 (10)
```

观察 1 号节点数据如下：

```
[15:23:03.006]收←◆mesh 132 received from 00.00.00.00.00.01: 1234567890 (10)
src:1,forwarding to 00.00.00.00.00.01
mesh 132 packet sent

[15:23:11.006]收←◆mesh 132 received from 00.00.00.00.00.01: 3456789012 (10)
src:1,forwarding to 00.00.00.00.00.01
mesh 132 packet sent

[15:23:19.006]收←◆mesh 132 received from 00.00.00.00.00.01: 5678901234 (10)
src:1,forwarding to 00.00.00.00.00.01
mesh 132 packet sent
```

观察 2 号节点数据如下：

```
[15:23:07.012]收←◆mesh 132 received from 00.00.00.00.00.01: 2345678901 (10)
src:2,forwarding to 00.00.00.00.01.01
mesh 132 packet sent

[15:23:15.013]收←◆mesh 132 received from 00.00.00.00.00.01: 4567890123 (10)
src:2,forwarding to 00.00.00.00.01.01
mesh 132 packet sent
```

通过对比三个节点的数据，观察可以发现：

0 号节点查询路由表，发现发往 1 号节点的下一跳就是 1 号节点，所以将数据包发送至 1 号节点。 1 号节点收到来自号节点发送的数据包，就会将该路径加入路由表。

0 号节点查询路由表，发现发往 2 号节点的下一跳是 1 号节点，所以会将数据包发送至 1 号节点，由 1 号节点转发至 2 号节点。2 号节点收到来自 1 号节点的 0 号节点发送的数据包，就会将该路径加入路由表。

1 号节点和 2 号节点收到来自 0 号节点的数据包后，都会将数据包原样发送至 0 号节点，同样，1 号节点会查询发现前往 0 号节点的下一跳就是 0 号节点，就将数据发送至 0 号节点。2 号节点查询路由表发送前往 0 号节点的下一跳为 1 号节点。就会将数据发送至 1 号节点，让 1 号节点将数据发送至 0 号节点。

第二次实验采用手动管理路由表：

将自动路由表管理的宏定义改为 0。看一下手动管理路由后，需要添加的路由条目：

```
  if (MESH_TEST_SELF_ADDR_IDX == 0) {
    tmos_start_task(lwns_mesh_test_taskID, MESH_EXAMPLE_TX_PERIOD_EVT,
            MS1_TO_SYSTEM_TIME(200));
#if MESH_TEST_ROUTE_AUTO
#else
    u8 i ;
    for( i = 1;i<5;i++){
      lwns_route_add(&device_addr[i],&device_addr[1],2);//手动管理，添加路由条目
    }
    for( i = 5;i<9;i++){
      lwns_route_add(&device_addr[i],&device_addr[5],2);//手动管理，添加路由条目
    }
    for( i = 9;i<13;i++){
     lwns_route_add(&device_addr[i],&device_addr[9],2);//手动管理，添加路由条目
    }
  } else if (MESH_TEST_SELF_ADDR_IDX == 1){
    lwns_route_add(&device_addr[4],&device_addr[4],1);//手动管理，添加路由条目
    lwns_route_add(&device_addr[3],&device_addr[3],1);//手动管理，添加路由条目
    lwns_route_add(&device_addr[2],&device_addr[2],1);//手动管理，添加路由条目
    lwns_route_add(&device_addr[0],&device_addr[0],1);//手动管理，添加路由条目
  }else if (MESH_TEST_SELF_ADDR_IDX == 2){
    lwns_route_add(&device_addr[0],&device_addr[1],2);//手动管理，添加路由条目
  }else if  (MESH_TEST_SELF_ADDR_IDX == 3){
    lwns_route_add(&device_addr[0],&device_addr[1],2);//手动管理，添加路由条目
  }else if  (MESH_TEST_SELF_ADDR_IDX == 4){
    lwns_route_add(&device_addr[0],&device_addr[1],2);//手动管理，添加路由条目
  } else if (MESH_TEST_SELF_ADDR_IDX == 5){
    lwns_route_add(&device_addr[6],&device_addr[6],1);//手动管理，添加路由条目
    lwns_route_add(&device_addr[7],&device_addr[7],1);//手动管理，添加路由条目
    lwns_route_add(&device_addr[8],&device_addr[8],1);//手动管理，添加路由条目
    lwns_route_add(&device_addr[0],&device_addr[0],1);//手动管理，添加路由条目
  }else if (MESH_TEST_SELF_ADDR_IDX == 6){
    lwns_route_add(&device_addr[0],&device_addr[5],2);//手动管理，添加路由条目
  }else if  (MESH_TEST_SELF_ADDR_IDX == 7){
    lwns_route_add(&device_addr[0],&device_addr[5],2);//手动管理，添加路由条目
  }else if  (MESH_TEST_SELF_ADDR_IDX == 8){
    lwns_route_add(&device_addr[0],&device_addr[5],2);//手动管理，添加路由条目
  } else if (MESH_TEST_SELF_ADDR_IDX == 9){
   lwns_route_add(&device_addr[10],&device_addr[10],1);//手动管理，添加路由条目
   lwns_route_add(&device_addr[11],&device_addr[11],1);//手动管理，添加路由条目
   lwns_route_add(&device_addr[12],&device_addr[12],1);//手动管理，添加路由条目
   lwns_route_add(&device_addr[0],&device_addr[0],1);//手动管理，添加路由条目
 }else if (MESH_TEST_SELF_ADDR_IDX == 10){
   lwns_route_add(&device_addr[0],&device_addr[9],2);//手动管理，添加路由条目
```

例如，节点 0 到节点 1 的路由表：

dst 为节点 1，nexthop 为节点 1，只转发一次，所以 cost 为 1。

节点 0 到节点 2 的路由表：

dst 为节点 2，nexthop 为节点 1，只转发两次，所以 cost 为 2。

将程序分别编译后烧录，观察现象，就会发现和自动管理路由表的现象是差不多的。

只不过省略了路由寻找的过程。

路由表是双向的，所以手动添加的时候也需要双向添加。

同时手动管理时也别忘了将自动路由表检查函数禁用：

```
#if MESH_TEST_ROUTE_AUTO
  lwns_route_init(TRUE, 60, HTIMER_SECOND_NUM);
#else
  lwns_route_init(TRUE, 0, HTIMER_SECOND_NUM);//disable auto clean route entry
#endif /*MESH_TEST_ROUTE_AUTO*/
```

14. 其他常用函数讲解

本章将讲解一些其他将会使用到的函数。

14.1 get_lwns_object_port

```
/**************************************************************************
 * @fn       get_lwns_object_port
 *
 * @brief    get port num of a lwns_controller_ptr.
 *
 * input parameters
 *
 * @param    controller  - defined with lwns_controller_ptr.
 *
 * output parameters
 *
 * @param    None.
 *
 * @return   uint8_t,return port number of the lwns_controller_ptr.
 */
extern uint8_t get_lwns_object_port(lwns_controller_ptr controller);
```

此函数可以在接收或者发送回调中，读取到当前收到数据所属的端口号。防止用户没有保存指针或者忘记端口号。

例如，用户将同一个回调函数给好几个不同端口号的控制结构体做回调函数，那么在接收数据的过程中，就可以通过此函数，通过回调函数里面的控制结构体指针，读取当前的端口号。

14.2 lwns_addr_cmp

```
/**************************************************************************
 * @fn       lwns_addr_cmp
 *
 * @brief    compare two address，return 1 if they are same.use memcmp ,the size is 6.
 *
 * input parameters
 *
 * @param    src1 - pointer to src1 data
 * @param    src2 - pointer to src2 data
 *
 * output parameters
 *
 * @param    None.
 *
 * @return   1 - same.
 *           0 - not same.
 */
extern int lwns_addr_cmp(const void *src1, const void *src2);
```

该函数用于比对两个地址是否相同，相同返回 TRUE，不相同返回 FALSE。

14.3 get_lwns_addr

```
/*********************************************************************************
 * @fn        get_lwns_addr
 *
 * @brief      get lwns_addr,and save to t.
 *
 * input parameters
 *
 * @param      t - pointer to lwns_addr
 *
 * output parameters
 *
 * @param      None.
 *
 * @return     None.
 *
 */
extern void get_lwns_addr(lwns_addr_t *t);
```

该函数用于获取协议栈内部的节点地址，保存到 t 指向的内存空间中。

14.4 lwns_htimer_flush_all

```
/*********************************************************************************
 * @fn        lwns_htimer_flush_all
 *
 * @brief      remove all htimers in htimer list.
 *
 * input parameters
 *
 * @param      None.
 *
 * output parameters
 *
 * @param      None.
 *
 * @return     None.
 */
extern void lwns_htimer_flush_all(void);
```

该函数会清空所有 htimer 内部自动超时计数的定时器。

14.5 lwns_lib_deInit

```
/*****************************************************************************
 * @fn        lwns_lib_deInit
 *
 * @brief     deInit lwns lib,then you cannot use lwns_lib,if you use,maybe it will enter hardfault mode.
 *
 * input parameters
 *
 * @param     None.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    None.
 */
extern void lwns_lib_deInit(void);
```

该函数用于重置库。重置以后，将无法再调用库内函数，否则可能会进入 hardfault。

14.6 lwns_controller_lookup

```
/*****************************************************************************
 * @fn        lwns_controller_lookup
 *
 * @brief     use port_num to find an opened lwns_controller_ptr .
 *
 * input parameters
 *
 * @param     port_num  - uint8_t.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    lwns_controller_ptr,return the lwns_controller_ptr of this port number.
 */
extern lwns_controller_ptr lwns_controller_lookup(uint8_t port_num);
```

该函数可以通过一个端口号找到其相应的控制句柄,但是这个端口号必须经过相应模块的 open 函数和控制句柄初始化。

14.7 lwns_controller_pop

```
/*****************************************************************************
 * @fn        lwns_controller_pop
 *
 * @brief     pop an opened lwns_controller_ptr from lwns_lib,
 *            the lwns_controller_ptr will remove from lwns_lib,
 *            so it will not handle data for this returned lwns_controller_ptr.
 *
 * input parameters
 *
 * @param     None.
 *
 * output parameters
 *
 * @param     None.
 *
 * @return    NULL if no opened in lib.other is an opened lwns_controller_ptr.
 */
extern lwns_controller_ptr lwns_controller_pop(void);
```

该函数将从库内部已经打开的端口组成的链表中弹出一个端口的控制结构体指针。弹出后，被弹出的控制结构体指针就相当于被执行了关闭操作，无法继续发送和接收数据包。