姓名 王奇 班级 2016211309 学号 2016211370

一、 实验内容与实验环境描述

1、实验任务与实验内容

利用所学数据链路层原理，设计一个滑动窗口协议，在仿真环境下编程实现有噪音 信道环境下两站点之间无差错双工通信。信道模型为 8000bps 全双工卫星信道，信道传 播时延 270 毫秒，信道误码率为 10-5，信道提供字节流传输服务，网络层分组长度固 定为 256 字节。 通过该实验，进一步巩固和深刻理解数据链路层误码检测的 CRC 校验技术，以及 滑动窗口的工作机理。滑动窗口机制的两个主要目标：(1) 实现有噪音信道环境下的无 差错传输; (2) 充分利用传输信道的带宽。在程序能够稳定运行并成功实现第一个目标之 后，运行程序并检查在信道没有误码和存在误码两种情况下的信道利用率。为实现第二 个目标，提高滑动窗口协议信道利用率，需要根据信道实际情况合理地为协议配置工作 参数，包括滑动窗口的大小和重传定时器时限以及 ACK 搭载定时器的时限。这些参数 的设计，需要充分理解滑动窗口协议的工作原理并利用所学的理论知识，经过认真的推 算，计算出优取值，并通过程序的运行进行验证。 2. 实验环境 Windows 8.1 环境下的 PC 机，使用 Visual Studio 2013 集成化开发环境。

2、实验环境

window10环境下的pc机，使用Visual Studio集成开发环境

二、软件设计

1、数据结构

a)帧

```
struct FRAME { //帧的内容
    seq_nr kind; //帧的种类，分为Ack，Nak 和 Data 三种
    seq_nr ack; ////捎带确认ack
    seq_nr seq; //本帧的序号
    seq_nr info[PKT_LEN]; //数据
};
```

b)宏定义

```
#define DATA_TIMER  5000 //Data帧超时时间
#define ACK_TIMER 280 //Ack帧超时时间

#define MAX_SEQ 31 //帧的序号空间，应当是2^n-1
#define NR_BUFS 16 //发送窗口、接收窗口大小，NR_BUFS=(MAX_SEQ+1)/2
```

全局变量

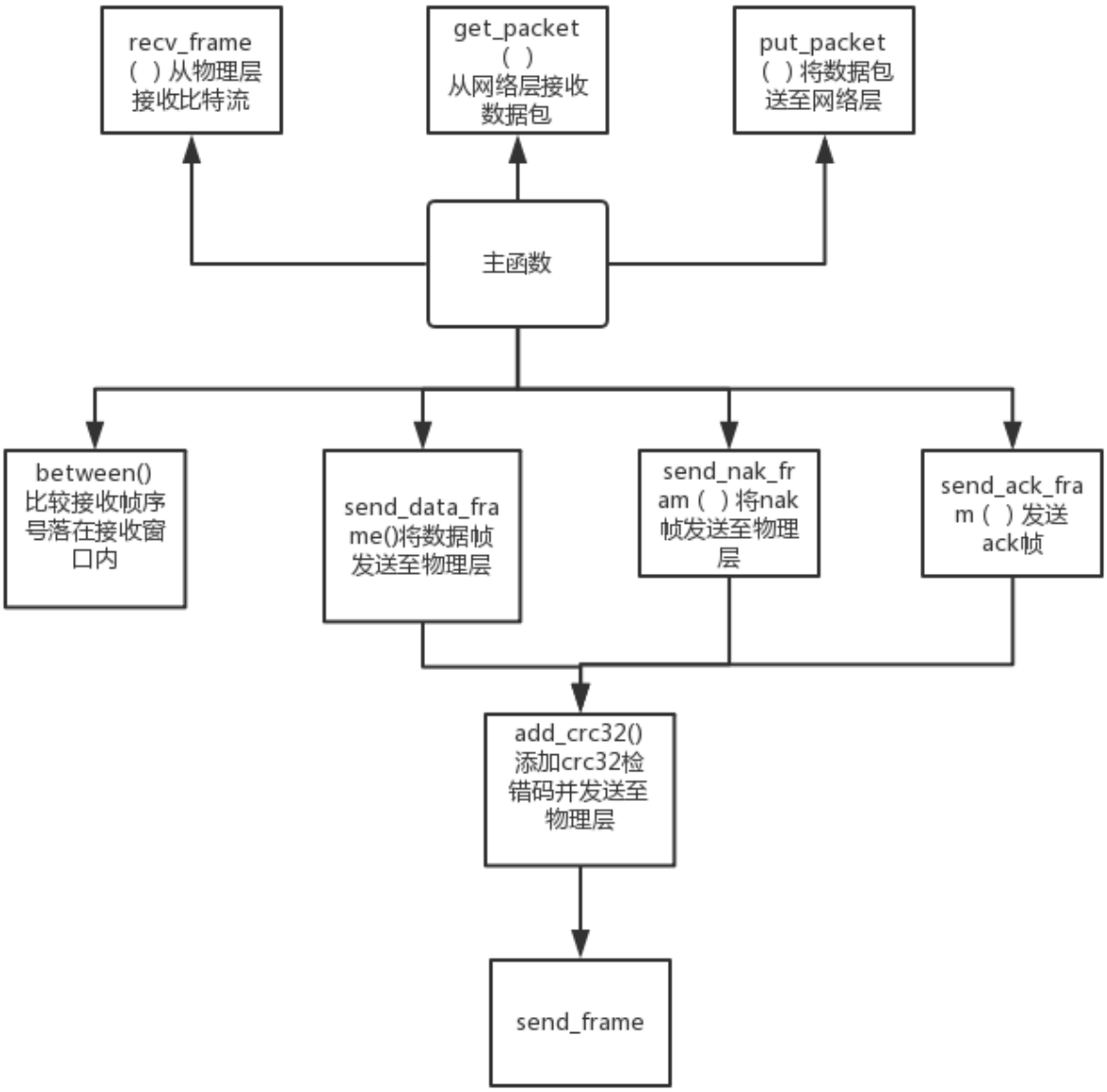
```
static unsigned char next_frame_to_send = 0, frame_expected = 0, ack_expected = 0;
//next_frame_to_send:将要发送的帧序号，发送方窗口的上界；frame_expected:期望收到的帧序号，接收方窗口的下界；ack_expected:期望收到的ack帧序号，发送方窗口的下界
static unsigned char out_buffer[NR_BUFS][PKT_LEN], in_buffer[NR_BUFS][PKT_LEN], nbuffered;
//对应输出缓存、输入缓存，以及目前输出缓存的个数
static unsigned char too_far = NR_BUFS; //接收方窗口的下界
static int phy_ready = 0; //物理层是否ready
bool arrived[NR_BUFS]; //接收方输入缓存的窗口
bool no_nak = true; //是否发送了NAK，true则不再重复发送
```

c)主函数变量

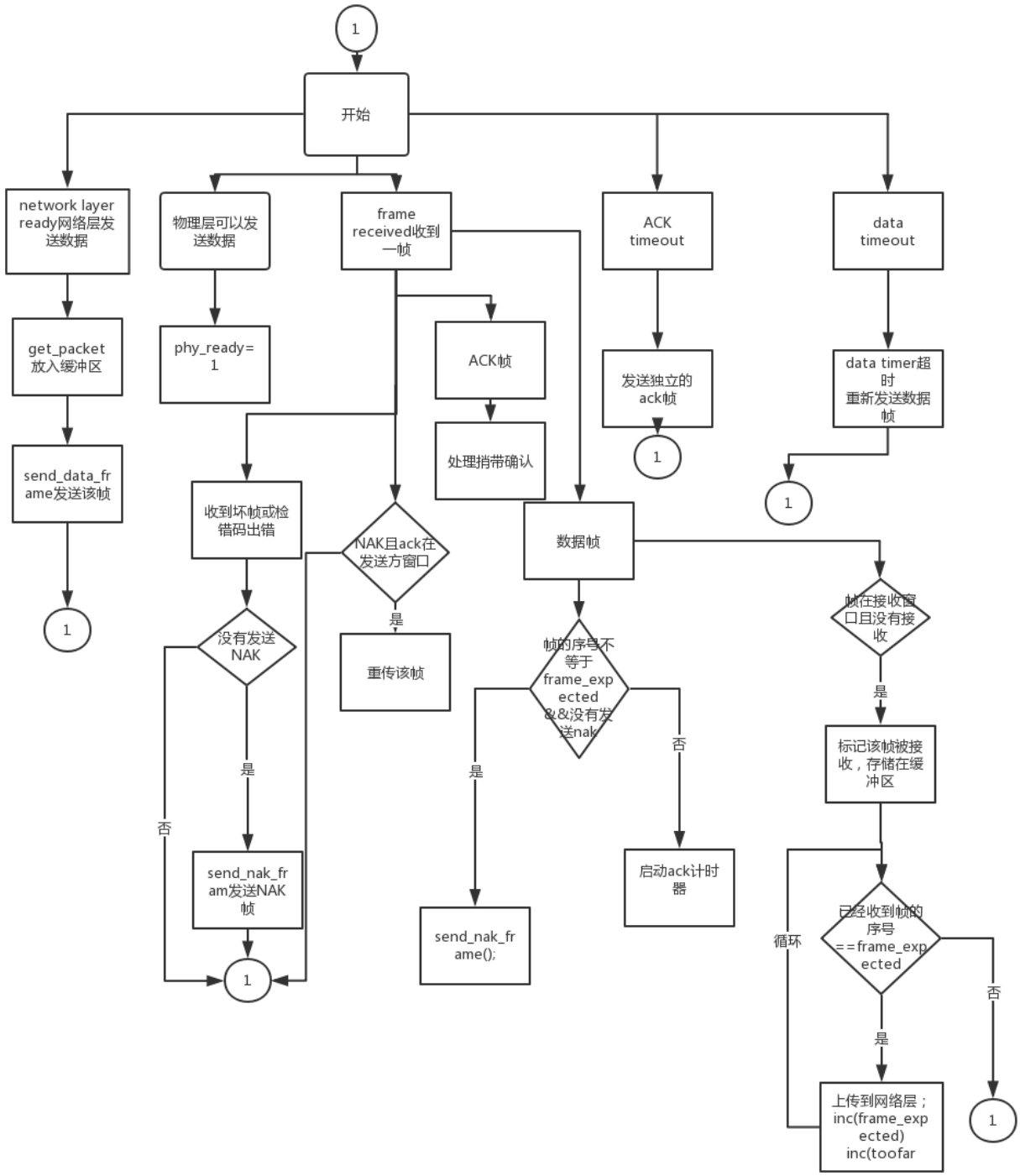
```
nbuffered = 0; //输出帧的总数，初始没有输出帧被缓存
int i; //arrived[]和into buffer[]的序号;
int event, oldest_frame;//事件和最初识没有确认的帧序号
struct FRAME f; //定义帧
int len = 0; //收到 data 的长度
```

2、模块结构

```
static bool between(int a, int b, int c)
/*如果 b 在 a 和 c 组成的窗口之间，则返回 true，否则返回 false 参数：a，b，c 是要进行比较的数字序号。 */

static void addcrc32(unsigned char *frame, int len) /*将 frame添加crc32检错码 一起发送到物理层 参数：unsigned char * frame：要发送到物理层的帧。int len：帧的长度。 */

static void send_data_frame(unsigned char frame_nr) /*将输出缓存打包成 frame 后调用 put_frame 发送到物理层，同时开始计时 data_timer 参数：unsigned char * frame：要发送到物理层的帧。 */

static void send_ack_frame(void) /*打包 ack 确认帧并发送到物理层 */

static void send_nak_frame(void) /*打包 nak 确认帧并发送到确认帧 */
```



3、算法流程图



算法流程

当接收到一个事件时，有5种情况：

1）网络层有数据发送->存储到发送窗口缓冲区，发送到物理层；

2)物理层可以发送数据->phy_ready=1;

3)收到帧->判断帧的种类->

3.1)坏帧->没有发送NAK时发送NAK帧

3.2）NAK帧->ACK序号在发送窗口内重传该帧

3.3）数据帧

3.3.1)不能被接收->发送了NAK则启动ACK计时器，没发送则发送NAK；

3.3.2）在接收窗口内没有被接受过->标记该帧已经接收，存储在缓冲区->当帧的序号==frame_expected 时上传到网络层

3.4）ACK帧，处理捎带确认，确认ac_expected序号到接收到的ack序号

4）ACK超时重发

5）data计时器超时重发

三、实验结果分析

(1)实现了选择重传协议，能够在有误码信道环境中实现无差错传输；

(2)健壮性良好，实际运行1个小时，说明可以长时间运行；

(3)协议参数的选取： 经多次实验测得当滑动窗口的大小MAX_SEQ为31，缓冲区的大小NR_BUFS为 16，重传定时器data_timer为5000ms,ACK定时器ack_timer为300ms时效率较高。

> 信道参数:速率 8000bps，传播时延 270ms;

当数据包从物理层发出时其长度为:256字节[1]+3字节[^控制信息]+4字节[^校验字长]=263字节。 发送一帧用时：$t_{trans}=\frac{263*8}{8000}$= 263ms。而数据链路层发送该帧并收到 ACK的短时间是：$T_{总}=t_{trans}+2*t_{prog}+t_{trans}$= 263 + 270 + 300 +270 = 1103ms。所以 NR_BUFS>=$\frac{t\ 总}{T\ trans}$=1103÷263=4.3，此时MAX_SEQ=15而NR_BUFS=8。 值得注意的是这只是无误码情况下得出的结果。当有误码时用时会增大，因而 和NR_BUFS应当相应增大。而重传时限也相应增大为2000ms。

(4)理论分析

a)1. 无误码的情况 在数据帧中，每一帧263字节，帧头3个字节，帧尾4个字节，所以信道利用率为: $\frac{256}{263}\times$ 100%=97.3%

b)有误码的情况 这里假设重传操作及时，重传的数据帧的回馈，ACK 帧可以100%正确传输。 当误码率为10^{-5},即发送10^5个比特，出现一个错误，则平均发送$\frac{100000}{263\times8}$=47.53≈48个帧会出现一个错误。 此时信道利用率为：$\frac{48}{48+1}\times$ 97.3%=95.3%

当误码率为10^{-4}，即发送10^4个比特，出现一个错误，则平均发送$\frac{10000}{263\times8}$=4.75≈5个帧就会出现一个错误。 此时信道利用率为：$\frac{5}{6}\times$ 97.3% = 81.0%

(5)实验结果分析

序号	命令	说明	运行时间(秒)	效率(%)		备注
				A	B	
1	datalink au datalink bu	无误码信道数据传输	1976.687	52.79	96.87	
2	datalink a datalink b	站点 A 分组层平缓方式发出数据，站点 B 周期性交替“发送 100 秒，停发 100 秒”	2164.340	52.79	94.59	
3	datalink afu datalink bfu	无误码信道，站点 A 和站点 B 的分组层都洪水式产生分组	2860.435	96.97	96.97	
4	datalink af datalink bf	站点 A/B 的分组层都洪水式产生分组	2013.335	94.73	94.70	
5	datalink af –ber 1e-4 datalink bf –ber 1e-4	站点 A/B 的分组层都洪水式产生分组，线路误码率设为 10^{-4}	1876.377	57.40	56.18	

表 1 性能测试记录表

相较于理论计算来说实际实验得到的效率与之相差 1%，由于存在对帧的处理时间 以及受到硬件的影响，此误差可以忽略。而在误码率为的 10-4情况下误差较大，鉴于 设计的程序是在无差错环境下测试而设置的滑动窗口大小以及超时时长，需要对原先 代码中的参数进行调整才能获得较高的效率。

(6)存在问题：信道利用率没有达到最大值

四、研究和探索的问题

1、 CRC 校验能力

本次实验使用的 32 位的 CRC 校验码。在理论上，可以检测出：所有的奇数个个错 误， 所有双比特错误，所有小于等于 32 位的突发错误，但是检测不出大于 32 位的突发错误。。因此如果出现 CRC32 不能检测出的错误，至少需要出现 33 位突发错误。

2、 get_ms()和 log_printf 的实现

C 语言的 `time.h` 当中提供了一些关于时间操作的函数可以实现 get_ms()函数。可以 利用的函数有 clock()函数原型为： `clock_t clock()` 该函数返回程序开始执行后占用的处理 器时间，如果无法获得占用时间则返回-1。因为我们计时的起点并不是程序开始之时， 而是开始通信之时，所以需要一个静态变量 `start_time` 来记录通信起始的时间。然后在 每次调用 `get_ms()` 后，获取当前的时间 `current_time` 。然后再返回 `start_time-current_time` 即可。 `printf` 是用变长参数实现的。prinf 的函数原型为 `printf(char * fmt,…)` ,后面…即表 示变长参数。通过对 fmt 字符串进行解析，将 fmt 字符串转化成终的字符串，然后通 过系统调用输出到屏幕上。

3、 软件测试方面的问题

设置多种测试方案的目的在于测试程序能否在正常环境下运行，通过多种测试方案 来模拟现实中的多种情况。

4、对等协议实体之间的流量控制

在选择重传协议中，当 `nbuffered < NR_BUFS && phy_ready` 才能 `enable_network_layer();` 即当前发送方缓冲区中帧的数目小于最大值，网络层才能将数据传给数据链路层，从而实现流量控制；

五、实验总结和心得体会

实验总结：本次实验在课下学习理解实验书用了大约2小时，实际上机撰写代码并调试的时间 为 7小时。由于直接从老师的项目中开始编辑，因此在编程环境上没有出现问题，而编程 过程中没有发现 C 语言方面的问题。协议参数用了很长时间才测试出较为理想的结果，而 通过本次实验的代码编写对数据链路层各功能的实现有了形象的概念，对选择重传协议以 及滑动窗口的概念有了更加深刻的了解。

心得体会：通过对整个实验过程的亲身经历，对实验有了更加全面而细致的了解，也深切理解到真 正的协议制定及参数调整其不易之处。 理论学习和实际运用有很大差距，有一些bug理论学习弥补不了

六、源程序清单

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

#include "protocol.h"
#include "datalink.h"

#define DATA_TIMER  5000 //Data帧超时时间
#define ACK_TIMER 280 //Ack帧超时时间

#define MAX_SEQ 31 //帧的序号空间，应当是2^n-1
#define NR_BUFS 16 //发送窗口、接收窗口大小，NR_BUFS=(MAX_SEQ+1)/2
#define inc(k) if(k < MAX_SEQ)k++;else k=0 //计算k+1

typedef  unsigned char seq_nr;
struct FRAME { //帧的内容
    seq_nr kind; //帧的种类，分为Ack，Nak 和 Data 三种
    seq_nr ack; ////捎带确认ack
    seq_nr seq; //本帧的序号
    seq_nr info[PKT_LEN]; //数据,存储大小256
};

static unsigned char next_frame_to_send = 0, frame_expected = 0, ack_expected = 0;
//next_frame_to_send:将要发送的帧序号，发送方窗口的上界；frame_expected:期望收到的帧序号，接收方窗口的下界；ack_expected:期望收到的ack帧序号，发送方窗口的下界
static unsigned char out_buffer[NR_BUFS][PKT_LEN], in_buffer[NR_BUFS][PKT_LEN], nbuffered;
//对应输出缓存、输入缓存，以及目前输出缓存的个数
static unsigned char too_far = NR_BUFS; //接收方窗口的下界
static int phy_ready = 0; //物理层是否ready
bool arrived[NR_BUFS]; //接收方输入缓存的窗口，
bool no_nak = true; //是否发送了NAK，true则不再重复发送

static bool between(int a, int b, int c){ //如果b在a和c组成的窗口之间，则返回true，否则返回false
    return((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c <a));
}


static void add_crc32(unsigned char *frame, int len){
```

```
    //将frame做CRC校验后附检错码一起发送到物理层
    *(unsigned int *)(frame + len) = crc32(frame, len);
    send_frame(frame, len + 4);// 发送到物理层，4字节检错
    phy_ready = 0; //将数据发送到物理层后，将物理层置为忙碌状态
}

static void send_data_frame(unsigned char frame_nr){//发送数据帧
    //将输出缓存打包成frame后调用add_crc32发送到物理层，同时开始计时data_timer
    struct FRAME s;

    s.kind = FRAME_DATA;
    s.seq = frame_nr;
    s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
    memcpy(s.info, out_buffer[frame_nr % NR_BUFS], PKT_LEN); //将out_buffer里的缓存复制到帧frame的info域中

    dbg_frame("Send DATA %d %d, ID %d\n", s.seq, s.ack, *(short *)s.info);

    add_crc32((unsigned char *)&s, 3 + PKT_LEN); //将打包好的帧发送到物理层，3字节控制信息

    start_timer(frame_nr % NR_BUFS, DATA_TIMER);
    stop_ack_timer();
}

static void send_ack_frame(void){
    //将ack封装成帧发送到物理层
    struct FRAME s;

    s.kind = FRAME_ACK;
    s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);

    dbg_frame("Send ACK  %d\n", s.ack);

    add_crc32((unsigned char *)&s, 2);
    stop_ack_timer();
}

static void send_nak_frame(void){
    //将ack封装成帧发送到物理层
    struct FRAME s;

    s.kind = FRAME_NAK;
    s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);

    no_nak = false;

    dbg_frame("Send NAK with ACK %d\n", s.ack);

    add_crc32((unsigned char *)&s, 2);//将打包好的帧发送到物理层
    stop_ack_timer();
}


int main(int argc, char **argv)
```

```
{
    //初始化
    nbuffered = 0; //初始没有输出帧被缓存
    int i;//arrived[]和into buffer[]的序号;
    for (i = 0; i < NR_BUFS; i++) //接收窗口初始化
        arrived[i] = false;
    int event, oldest_frame;//事件和最初识没有确认的帧序号
    struct FRAME f;
    int len = 0; //收到data的长度

    protocol_init(argc, argv); //协议初始化

    disable_network_layer();

    for (;;)
    {
        event = wait_for_event(&oldest_frame);

        switch (event) //对不同事件进行不同处理
        {
            case NETWORK_LAYER_READY: //当网络层ready时
                get_packet(out_buffer[next_frame_to_send % NR_BUFS]); //从网络层获取一帧放入输出缓存中
                nbuffered++; //缓存序号+1
                send_data_frame(next_frame_to_send);//发送该帧
                inc(next_frame_to_send); //将将发送方窗口上限+1
                break;

            case PHYSICAL_LAYER_READY: //当物理层ready时，将phy_ready置为1以便之后enable网络层
                phy_ready = 1;
                break;

            case FRAME_RECEIVED: //当物理层收到一帧时
                len = recv_frame((unsigned char *)&f, sizeof f); //从物理层获取一帧
                if (len < 5 || crc32((unsigned char *)&f, len) != 0) { //如果接收坏帧或CRC校验失败，则返回nak帧
                    dbg_event("**** Receiver Error, Bad CRC Checksum\n");
                    if (no_nak)
                        send_nak_frame();
                    break;
                }
                if (f.kind == FRAME_ACK) //如果是ack帧的话，由于所有帧都含有ack帧，因此统一处理
                    dbg_frame("Recv ACK  %d\n", f.ack);

                if (f.kind == FRAME_DATA) //如果是数据帧
                {
                    if ((f.seq != frame_expected) && no_nak) //如果收到的是不需要的帧则返回nak
                        send_nak_frame();
                    else
                        start_ack_timer(ACK_TIMER);
                    if (between(frame_expected, f.seq, too_far) && (arrived[f.seq%NR_BUFS] == false))

                    //如果收到的帧在接收方窗口内且这一帧未被接收过
```

```
139                     {
140                         dbg_frame("Recv DATA %d %d, ID %d\n",f.seq, f.ack, *(short
    *)f.info);
141                         arrived[f.seq%NR_BUFS] = true; //标记该帧为已接受
142                         memcpy(in_buffer[f.seq % NR_BUFS], f.info,PKT_LEN); //将接收到的帧的
    data域复制至输入缓存中
143                         while (arrived[frame_expected % NR_BUFS])//当收到接收方窗口下界的一帧
    时，对这一帧以及之后收到的帧进行处理
144                         {
145                             put_packet(in_buffer[frame_expected %NR_BUFS], len - 7); //将输
    入缓存送至网络层
146                             no_nak = true;
147                             arrived[frame_expected % NR_BUFS] = false;
148                             inc(frame_expected); //接收方窗口下界模 (MAX_SEQ+1)加1
149                             inc(too_far);//接收方窗口上界模 (MAX_SEQ+1)加1
150                             start_ack_timer(ACK_TIMER); //如果ack_timer超时则发送ack帧
151                         }
152                     }
153                 }
154 
155                 if ((f.kind == FRAME_NAK) && between(ack_expected, (f.ack + 1) % (MAX_SEQ +
    1), next_frame_to_send))
156                 //如果收到的是nak帧且ack的下一帧在发送方窗口里，则发送ack的下一帧
157                 {
158                     send_data_frame((f.ack + 1) % (MAX_SEQ + 1));
159                     dbg_frame("Recv NAK with ACK %d\n", f.ack);
160                 }
161 
162                 while (between(ack_expected, f.ack, next_frame_to_send))
163                 //累计确认，只要ack在发送方窗口内就不断地将窗口前移直至未确认的一帧
164                 {
165                     nbuffered--;
166                     stop_timer(ack_expected%NR_BUFS);
167                     inc(ack_expected);
168                 }
169                 break;
170 
171             case ACK_TIMEOUT://ack_timer超时时发送独立的ack帧
172                 dbg_event("---- DATA %d timeout\n", oldest_frame);
173                 send_ack_frame();
174                 break;
175 
176             case DATA_TIMEOUT: //某帧的data_timer超时说明未收到接收方的ack帧，则重新发送该帧
177                 dbg_event("---- DATA %d timeout\n", oldest_frame);
178                 if (!between(ack_expected, oldest_frame, next_frame_to_send))
179                     oldest_frame = oldest_frame + NR_BUFS;
180                 send_data_frame(oldest_frame);
181                 break;
182         }
183 
184         if (nbuffered < NR_BUFS && phy_ready) //如果物理层ready且目前缓存未满，则使网络层ready
185             enable_network_layer();
186         else
```

```
187             disable_network_layer();
188     }
189 }
```

1. ↩