计算机网络实验指导书

北京邮电大学 计算机学院

2018.11

目　　录

实验一：数据链路层滑动窗口协议的设计与实现

1. 实验类别

程序设计型

2. 实验内容和实验目的

利用所学数据链路层原理，自己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，计算出最优取值，并通过程序的运行进行验证。

通过该实验提高同学的编程能力和实践动手能力，体验协议软件在设计上各种问题和调试难度，设计在运行期可跟踪分析协议工作过程的协议软件，巩固和深刻理解理论知识并利用这些知识对系统进行优化，对实际系统中的协议分层和协议软件的设计与实现有基本的认识。

3. 实验学时

9 学时。

4. 实验组人数

1~3 人，共同完成协议的设计和理论分析，程序编码与调试，功能测试，协议工作参数的优化，研讨提出的相关问题，撰写实验报告。

5. 实验设备环境

Windows 环境 PC 机，Microsoft Visual Studio 2013+集成化开发环境。

6. 教学要点与学习难点

课堂教学和教材中给出了滑动窗口协议的基本原理，并给出了多个示意性程序，尤其是“回退 N 步（Go-Back-N）”协议和“选择重传”协议。这些示意性伪代码程序主要用于描述协议的基本工作过程并阐述滑动窗口的基本原理，为了突出主题还省略了许多处理细节，不能实际运行。一个网络协议的具体实现程序可能会作为一个操作系统支撑下的独立进程，或者，作为操作系统内核中的中断服务程序或驱动程序。协议的具体实现会受到协议软件所处的操作系统支撑环境或者内核编程模式的限制。本次实验所提供的编程环境用 Windows 中的一个进程仿真链路层的一个站点，程序的设计受限于编程环境所提供的功能以及实验题目所设定的具体问题和目标。因此，不可能完全照搬课堂教学中的示意性程序进行简单的原理验证，必须认真考虑具体问题和具体软件设计环境，但这些示意性程序有重要的参考作用。

实验题目给出了物理层信道模型和分组层数据的大小，链路层协议的设计有很大的自由度。由组内同学共同讨论完成，包括帧控制字段的设计，滑动窗口的过程控制。从易到难，可选的协议类型为“不搭载 ACK 的 Go-Back-N 协议”，“使用搭载 ACK 技术的 Go-Back-N 协议”，“选择重传协议”，要求必须是全双工通信协议。组内同学根据自身条件和不同协议类型的难度系数选做其中一种或多种。

教材中的示意性程序未对滑动窗口尺寸和重传定时器时限的详细设计进行充分说明。在实验过程中需要利用基本原理，明确协议工作参数取值的变化对协议工作正确性和信道利用率的利弊影响，根据具体信道情况和协议软件的实现方案，为协议参数设置最优值，以追求高线路利用率。指导教师提供一个可执行样例程序，将所实现的协议软件所达到的性能与样例程序比较，分析自己所实现软件的优势或缺陷。从理论上推导出线路利用率的极限，依此作为基本依据衡量所实现软件的性能优劣。找出所设计软件达到的线路利用率与理想性能之间的差距，并给出改进算法或者未能达到理想性能的原因。

整个程序的编程工作量不大，但是考虑到对滑动窗口协议的工作原理的理解可能不够深入和细致，考虑到协议软件调试的困难程度，以及完成实验的同学对 C 语言的运用和操控能力，软件调试中遇到问题后解决问题的能力，以及代码中逻辑的复杂程度，对完成整个实验应给予足够重视。

7. 实验步骤

7.1 熟悉编程环境

认真复习滑动窗口相关理论知识的内容，深入理解设计滑动窗口协议的目的和滑动窗口协议的基本工作过程。

安装好 VS2013 或兼容的更高版本的 C 语言编程环境，运行指导教师提供的样例程序，明确实现目标，了解信道工作参数的设置方法，网络层分组序列产生器的模式选择方法。根据所提供的资料熟悉编程环境，了解程序的主体运行框架，系统提供的子程序功能，与物理层和与网络层程序接口的方法，定时器的设置方法。其中，与物理层程序的接口方法与教材中的示意性程序差别较大，应特别注意。

7.2 协议设计和程序总体设计

设计好要实现的滑动窗口协议，定义帧字段，规划程序的总体结构，相关子程序的设置。

7.3 编码和调试

将所设计的协议编码实现并上机调试通过，实现数据链路层两个站点之间的通信。

7.4 软件测试和性能评价

在无误码信道环境下运行测试：理论上推导出无误码信道环境下的最佳信道利用率，设置仿真软件中的信道工作于无误码模式，观察和记录信道利用率，并与理想值进行比较。

有误码信道环境下的无差错传输：检查软件能否在误码信道环境下实现无差错传输，并进行调试。

测试阶段根据具体信道模型，通过细致的理论分析，合理调整协议工作参数和程序实现方式，追求有误码信道环境下更高的信道利用率，并对程序进行合理优化。

7.5 实验报告及程序验收

研讨第 10 节“可研究与探索的问题”提出的问题，总结实验过程中遇到的问题和解决方法，按要求撰写实验报告，并接受实验指导教师面对面现场点评和质疑。

8. 编程环境

由实验指导教师提供协议软件设计的基本程序库，利用仿真环境下所提供的物理层服务和定时器机制为网络层提供服务。

8.1 程序的总体结构

设数据链路层通信的两个站点分别为 A 和 B，仿真环境利用 Windows 环境下的 TCP 协议和 Socket 客户端/服务器机制构建两个站点之间的通信。编译、链接之后生成的可执行程序(.exe 文件)为字符界面命令行程序（不是图形界面程序）。可执行程序文件仅有一份，设为 datalink.exe，在 Windows 的两个命令行窗口中使用不同的命令行参数启动两个进程，分别仿真站点 A 和站点 B。例如：

启动站点 A 进程的命令：datalink a

启动站点 B 进程的命令：datalink b

其中，站点 A 以 TCP 服务器的身份运行，默认监听 TCP 端口 59144；站点 B 以 TCP 客户端身份运行，默认连接 127.0.0.1 的 TCP 端口 59144。端口号 59144，必要的时候可以更改为其他端口，参见 8.12 中的相关说明。在程序运行时应注意设置 Windows 的防火墙，解除对所用 TCP 端口的访问阻止。

可执行程序中包括物理层，数据链路层和网络层三部分内容，其中，物理层和网络层程序由程序库提供，数据链路层程序由同学自行完成，最终生成一个可执行文件。程序的总体结构与协议的分层结构相对应，如图 1 所示。

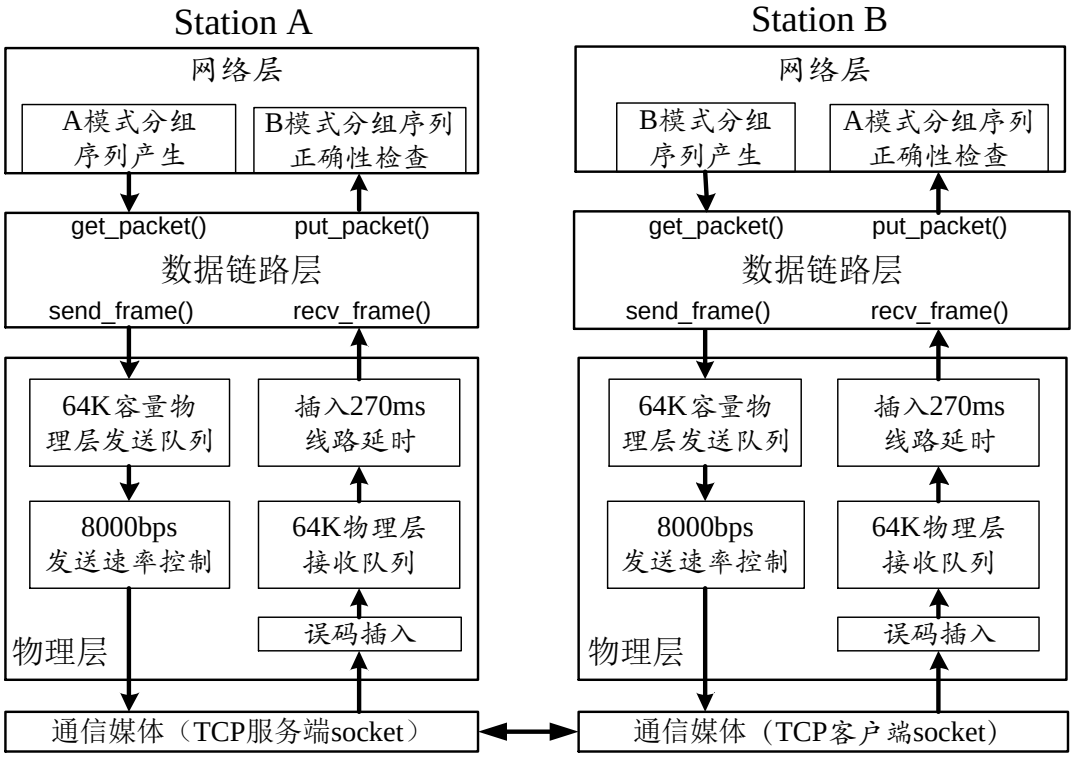


图 1　程序总体结构

物理层：为数据链路层提供的服务为 8000bps，270ms 传播延时，10^{-5} 误码率的字节流传输通道。为了仿真实现上述服务质量的信道，利用在同一台计算机上 TCP socket 完成两个站点之间的通信。由于同一台计算机上 TCP 通信传播时延短、传播速度快、没有误码，物理层仿真程序在发送端利用“令牌桶”算法限制发送速率以仿真 8000bps 线路；在接收端误码插入模块利用一个伪随机数“随机地”篡改从 TCP 收到的数据，使得所接收到的每个比特出现差错的概率为 10^{-5}；接收到的数据缓冲后延时 270ms 才

提交给数据链路层程序，以仿真信道的传播时延特性。为了简化程序，省略了“成帧”功能，数据链路层利用接口函数 send_frame()和 recv_frame()发送和接收一帧。

数据链路层：程序由同学自己设计完成，通过物理层提供的帧发送和接收函数，利用物理层提供的服务。通过 get_packet()函数从网络层得到一个分组；当数据链路层成功接收到一个分组后，通过 put_packet()函数提交给网络层。

网络层：利用数据链路层提供的“可靠的分组传输”服务，在站点 A 与站点 B 之间交换长度固定为 256 字节的数据分组。网络层把产生的分组交付数据链路层，并接受数据链路层提交来的数据分组。

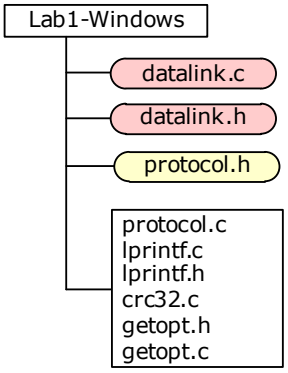
协议工作的正确性验证：如果数据链路层程序能够正确工作，站点 A 通过数据链路层发送的分组流，经过有误码的信道传输后，站点 B 就能够以同样的顺序收到同样内容的分组流。验证数据链路层程序的正确性的一种方法为：在站点 A 记录发送的分组流，在站点 B 记录接收到的分组流，比较这两份记录是否相同。为了简化这项工作，这里的网络层程序实现了一个“分组序列发生器”。“分组序列发生器”根据一个伪随机数公式，按照一种固定的模式产生分组流。在接收方用同样的公式计算一次，比对收到分组的每个字节值是否与约定的模式完全相同，这样就可以验证数据链路层是否实现了承诺的“无差错分组流传输”服务。一旦检查出数据链路层出错，程序打印错误信息后立刻中止运行（参见 8.13）。站点 A 到站点 B 方向和站点 B 到站点 A 方向分别使用不同的伪随机数公式，在全双工通信条件下双向同时进行分组序列的产生和验证。程序库提供了多种选项，在启动站点进程的命令中用不同的命令行参数选项可以控制“分组序列发生器”的分组产生模式，8.12“命令行选项”中有详细的说明。本次实验中库程序提供的“分组层”仅仅用来对数据链路层功能的验证，未实现一个实际网络中分组层实体所具有的路由和转发等功能。

按照协议分层的原则，网络层分组作为数据链路层的数据，数据链路层不应当解读分组中数据内容。为了便于同学调试数据链路层程序，“分组序列发生器”生成的每个分组的前两个字节放置了一个两字节整数作为“分组 ID”，例如：在调试数据帧重传功能时，可以打印出这个分组 ID 以分辨是哪个网络层分组数据在重传。设指针 p 的定义为 unsigned char *p，并且 p 已指向分组的首字节，那么，打印整数 *(short *)p 就可以得到分组 ID。站点 A 产生的分组的分组 ID 取值为 10000~19999，站点 B 产生的分组的分组 ID 取值为 20000~29999，分组 ID 值递增，递增到最大值后回卷到最小值。

8.2 Windows 环境下编译及运行方法

实验前必须事先安装好 Visual Studio 2013 或兼容的更高版本。

将自己设计的 C 语言程序 datalink.c 与提供的四个文件 protocol.c，lprintf.c， crc32.c，getopt.c，组成一个工程文件，编译链接生成可执行文件。



protocol.h：库函数中包括的函数原型以及相关的宏定义，调用库函数的 C 语言源程序应当#include 此文件。protocol.c 是实现这些库函数的源代码。

datalink.c：应由同学完成的数据链路层程序文件。C 语言源程序书写格式不要太凌乱，请参阅“附录一 源程序书写格式要求”。

crc32.c：查表法求 32 位 CRC 校验和。

lprintf.c：日志输出函数 lprintf 的源码。

getopt.c：提供库函数 getopt_long()(在 Linux 中是标准 C 函数)，用于分析命令行参数。

产生的日志文件为 datalink-A.log 和 datalink-B.log。

目录 Example 下的可执行文件有三个，分别是 datalink.exe，gobackn.exe，selective.exe。

goobackn.exe：使用搭载 ACK 技术的 Go-Back-N 协议的一种参考实现，可以直接运行以了解本次实验应达到的目的。

selective.exe：使用选择重传协议的一种参考实现。

编辑 datalink.c，输入自己的程序，然后，编译链接，生成可执行文件 datalink.exe。

程序运行需要启动两个进程:

打开一个命令行窗口，启动站点 A 的进程，输入命令： datalink –d3 A

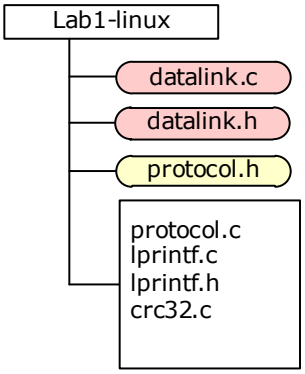
再打开一个命令行窗口，启动站点 B 的进程，输入命令：datalink –d3 B

那么，协议程序就开始运行。默认情况下，协议程序会持续运行 25 天。中止运行，按 Ctrl-C 键。在运行过程中，将生成“日志”(log)文件。默认情况下，日志文件的放在与可执行文件相同的目录（注意：有可能执行文件所处目录与当前目录不是同一个目录）。日志文件命名为可执行文件名加-A.log 或者-B.log 后缀。例如：可执行文件名为 datalink.exe，那么，日志文件的文件名为 datalink-A.log 和 datalink-B.log，分别对应站点 A 和站点 B 的日志文件。关于日志信息的输出，参见 8.4“日志”。

由于协议程序的工作需要毫秒数量级的实时操作，因此，在程序运行时务必关闭掉其他导致 CPU 负载加重和内存占用的任务，例如：多媒体播放器等，否则，由于操作系统调度问题可能会导致协议程序的运行产生阻塞或者停顿。严重时，库程序会给出警告信息。

8.3 Linux 环境下编译及运行方法

Linux 环境可以选用 Ubuntu 环境下的 GCC 编译器。提供开发包 Lab1-linux.tar，通过执行 tar xvf Lab1-linux.tar 命令后得到右图所示的文件。



Makefile：make 文件。

使用 linux 的文本编辑工具编辑 datalink.c 文件，编辑完成后执行 make 命令，就可以生成可执行文件 datalink。

打开终端窗口，启动站点 A 的进程，输入命令： ./datalink a

打开另一终端窗口，启动站点 B 的进程，输入命令：./datalink b

那么，协议程序就开始运行。

8.4 日志

protocol.h 头文件中定义的函数原型：

extern void lprintf(char *fmt, ...);

调试滑动窗口协议的程序时，常常需要用 printf 函数输出一条信息。例如：为了程序的调试，接收到了一个长度为 len 的完整的帧后，输出一条信息：

printf(”Received a frame, %d bytes\n”, len);

设该语句的输出为：

Received a frame, 248 bytes

这种传统的输出有两点不足。其一，类似这样的输出信息很多，在运行程序时，输出信息很快滚动过当前屏幕窗口；其二，一般来说，程序员不仅仅需要输出一条这样的信息，而且很关心执行这条语句时的时间坐标，用以判断超时重传等动作的正确性。因此，在程序库中维持一个时间坐标，时间坐标的 0 点设定为协议开始工作的时刻，时间刻度的精度为毫秒数量级，函数 get_ms()可以获得当前的时间坐标值，单位为毫秒。为了达到前述目的，程序库提供了 printf 的一种改进版本 lprintf，执行语句：

lprintf(”Received a frame, %d bytes\n”, len);

23.176 Received a frame, 248 bytes

lprintf 在输出信息之前首先输出当前的时间坐标，前面追加的 23.176 为执行这条语句时的当前时间

坐标 23.176 秒，其余的输出内容与 printf 完全相同，输出在当前屏幕窗口中。除此之外，使用 lprintf 在当前屏幕窗口中输出的信息被同时保存在一个日志文件中，以便于事后检查，关于日志文件的放置位置和文件名，8.2 已经给出了描述。

日志文件是程序员调用 lprintf 在当前屏幕窗口所输出信息的磁盘文件副本。

系统提供了 dbg_frame，dbg_event，dbg_warning 三个函数，它们最终调用 lprintf。但是，可以通过命令行参数的--debug 选项或者-d 选项调整输出。设置这三个函数的目的是简化程序中的 if(…)lprintf(…)。

8.5 协议运行环境的初始化

protocol.h 头文件中定义的函数原型：

```
void protocol_init(int argc, char **argv);
```

文件 datalink.c 中的主程序 main()必须首先调用此函数对运行环境初始化。该函数的两个参数必须传递 main()函数的两个同名参数。这样做的目的是从命令行参数中获取站点名及某些选项以提供一种配置系统参数的手段。这些选项包括重新指定日志文件，指定 TCP 端口号，设定误码率，等等。当命令行中重新指定了新的参数值，默认值就不再起作用。

protocol_init()建立两个站点之间的 TCP 连接，并且设定时间坐标的参考 0 点，通信的两个站点的时间坐标 0 点在建立 TCP 连接时被设置成相同的参考时间点。

作为站点 A，启动命令 GoBackN A，函数 protocol_init()的输出样例：

```
================================================
                        Station A
------------------------------------------------------------------------
Protocol.lib, version 4.0
Channel: 8000 bps, 270 ms propagation delay, bit error rate 1.0E-005
Log file "GoBackN-A.log", TCP port 59144, debug mask 0x00
Station A is waiting for station B on TCP port 59144 ... Done.
New epoch: Sat Oct 27 20:56:37 2018
================================================
```

输出信息：站点为 A；程序库版本号 4.0；信道参数：速率 8000bps，传播时延 270ms，误码率 10^{-5}；日志文件名为 GoBackN-A.log，TCP 端口号 59144，控制协议软件调试信息输出类别的控制字 debug_mask 为 0；TCP 连接建立的信息；本次程序运行的时间坐标 0 点设为 2018.10.27 20:56:37。这里的误码率等参数可以调整，详见 8.12“命令行选项”说明。

作为站点 B，启动命令 GoBackN B，函数 protocol_init()的输出与 A 站点类似。

8.6 与网络层模块的接口函数

protocol.h 头文件中的相关定义和函数原型：

```
/* Network Layer functions */
#define PKT_LEN 256
void enable_network_layer(void);
void disable_network_layer(void);
int  get_packet(unsigned char *packet);
void put_packet(unsigned char *packet, int len);
```

两个函数 enable_network_layer()和 disable_network_layer()的功能与教科书中相关示意性程

序中同名函数的功能一致。在相邻的上下层软件实体之间进行有效的数据流量的控制和协调对于一个实用的协议软件的设计非常重要，这种流量控制不仅存在于网络层／数据链路层，也存在于数据链路层／物理层之间。

对于发送方向来说，网络层和数据链路层的约定为：数据链路层在缓冲区满等条件下无法发送分组时通过 disable_network_layer()通知网络层；在能够承接新的发送任务时执行 enable_network_layer()允许网络层发送数据分组。

当网络层有新的分组需要发送并且未被链路层 disable，会产生 NETWORK_LAYER_READY 事件；否则网络层自行缓冲待发送分组。数据链路层在事件处理程序中调用 get_packet(p)函数将分组拷贝到指针 p 指定的缓冲区中，由于分组长度不等，函数返回值为分组长度。

put_packet()函数要求提供两个参数：存放收到分组的缓冲区首地址和分组长度。

程序库在 put_packet()函数的内部实现中增加了统计功能。如果本次调用 put_packet()函数比上次调用该函数的时间间隔超过 2 秒，将给出一个接收方向的报告，格式如下所示：

480.484 .... 1784 packets received, 7611 bps, 95.14%, Err 38 (9.9e-006)

时间坐标为 480.484 秒，收到了 1784 个分组，网络层有效数据传输率 7611bps，实际线路利用率 95.14%，接收方向共检出 38 个帧校验和错误，统计计算出实际误码率 $9.9x10^{-6}$。如果网络层接收方长期接收不到数据，就会长时间看不到这个报告。

发送方向的报告未能给出，可以检查对方站点的接收报告。

8.7 事件驱动函数及程序流程

protocol.h 头文件中的相关定义和函数原型：

```
int wait_for_event(int *arg);
#define NETWORK_LAYER_READY   0
#define PHYSICAL_LAYER_READY  1
#define FRAME_RECEIVED        2
#define DATA_TIMEOUT          3
#define ACK_TIMEOUT           4
```

函数 wait_for_event()导致进程等待，直到一个“事件”发生。可能的事件有上述 5 种，函数返回值为上述 5 种事件之一。参数 arg 用于获得已发生事件的相关信息，仅用于 DATA_TIMEOUT 两种事件，获取产生超时事件的定时器编号。

NETWORK_LAYER_READY：网络层有待发送的分组。此事件发生后才可以调用 get_packet()得到网络层待发送的下一个分组。

PHYSICAL_LAYER_READY：物理层发送队列的长度低于 50 字节。参见“与物理层模块的接口函数”部分。

FRAME_RECEIVED：物理层收到了一整帧。

DATA_TIMEOUT：定时器超时，参数 arg 中返回发生超时的定时器的编号。有关定时器操作，参见“定时器管理”部分。

ACK_TIMEOUT：所设置的搭载 ACK 定时器超时。

数据链路层程序的主控函数的总体流程框架与教科书中的示意性程序类似，但由于所支持的事件种类和流量控制机制不同，在细节处理上不尽相同。程序的示意性流程如下：

```
enable_network_layer();
for (;;) {
    event = wait_for_event(&arg);
    switch (event) {
    case EVENT_NETWORK_LAYER_READY:
        len = get_packet(my_buf);
        … …
        break;
    case EVENT_PHYSICAL_LAYER_READY:
        … …
        break;
    case EVENT_FRAME_RECEIVED:
        rbuf_len = recv_frame(rbuf, sizeof rbuf);
        ... ...
        break;
    case EVENT_ACK_TIMEOUT:
        ... ...
        break;
    case EVENT_DATA_TIMEOUT:
        ... ...
        break;
    }
    if (...)
        enable_network_layer();
    else
        disable_network_layer();
}
```

8.8 与物理层模块的接口函数

protocol.h 头文件中的相关定义和函数原型：

```
void send_frame(unsigned char *frame, int len);
int recv_frame(unsigned char *buf, int size);
int  phl_sq_len(void);
```

函数 send_frame ()用于将内存 frame 处长度为 len 的缓冲区块向物理层发送为一帧，每字节发送需要 1ms，帧与帧之间的边界保留 1ms。

函数 recv_frame()从物理层接收一帧，size 为用于存放接收帧的缓冲区 buf 的空间大小，返回值为收到帧的实际长度。

上下层协议实体之间的数据流量控制问题不仅存在于网络层/数据链路层，也存在于数据链路层/物理层之间。为协调数据链路层和物理层之间的流量，采用的机制是：只要在事件处理周期内至少一次调用过 send_frame()函数，那么，事件等待函数 wait_for_event()会在物理层发送队列低于 50 字节水平时，产生 PHYSICAL_LAYER_READY 事件。例如：物理层当前队列长度 20 字节，链路层调用 send_frame()发送一 7 字节帧，那么，随后的事件等待函数 wait_for_event()会立即产生 PHYSICAL_LAYER_READY 事件；再如：物理层当前队列长度 40 字节，链路层调用 send_frame()发

送一 300 字节帧，受限于信道 8000bps 的带宽，需要发送的数据不可能瞬间发送到线路上，wait_for_event()会在物理层队列由 340 字节降为 50 字节以下（至少需要 290 毫秒）后才产生 PHYSICAL_LAYER_READY 事件。

在 PHYSICAL_LAYER_READY 事件后，如果数据链路层暂时没有需要发送的数据，因系统不会再次送来 PHYSICAL_LAYER_READY 事件，应记录物理层状态，当有数据需要发送时直接发送。物理层事件的这种处理方式类似于硬件中的发送中断。

不顾物理层是否出于准备好状态而调用 send_frame()发送多帧，受限于信道的 8000bps 能力，会导致数据堆积在物理层发送队列的时间较长，等待物理层慢慢把数据发送出去。物理层发送队列最多可以保留 64K 字节。

函数 phl_sq_len()返回当前物理层队列的长度。

8.9 CRC 校验和的产生与验证

protocol.h 头文件中的函数原型：

```
unsigned int crc32(unsigned char *buf, int len);
```

本次实验采用的 CRC 校验方案为 CRC-32，与 IEEE 802.3 以太网校验和生成多项式相同。生成多项式为：

$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x^1+1$

CRC 校验和的生成有硬件产生和软件计算两种方式。硬件产生通过移位寄存器电路实现。教科书中给出了计算 CRC 校验和的方法：根据 CRC 多项式得到一个 33 比特的除数；将待校验的 n 字节帧理解为 8n 比特的二进制数，在后面追加 32 个 0 构成被除数；然后进行“模 2”除法，得到 32 比特余数作为校验和。由于一般通用的 CPU 未提供按位“模 2”除法的指令，需要通过软件模仿这种除法。目前，通过软件计算的方式求校验和，一般用查表并叠加的方式，将逐比特进行的模 2 除法改造为更高效率的逐字节进行模 2 除法，程序库中直接给出了实现校验和的函数调用 crc32()。源程序参考了 TCP/IP 协议簇中的 PPP 协议相关文本 RFC1662。

函数 crc32()返回一个 32 比特整数。设指针 p 的定义为 char *p 并且 p 指向一个缓冲区，缓冲区内有 243 字节数据，为这 243 字节数据生成 CRC-32 校验和，并且把这 32 比特校验和附在 243 字节之后，执行下面的语句：

```
*(unsigned int *)(p + 243) = crc32(p, 243);
```

注意：p 所指缓冲区必须至少有 247 字节有效空间，以防内存访问越界。

验证校验和的方法，对上面的例子，只需要判断 crc32(p, 243 + 4)是否为 0：校验和正确为 0，否则不为 0。

8.10 定时器管理

protocol.h 头文件中的函数原型：

```
/* Timer Management functions */
unsigned int get_ms(void);
void start_timer(unsigned int nr, unsigned int ms);
void stop_timer(unsigned int nr);
void start_ack_timer(unsigned int ms);
void stop_ack_timer(void);
```

get_ms()函数获取当前的时间坐标，单位为毫秒。

start_timer()用于启动一个定时器。两个参数分别为计时器的编号和超时时间值。计时器的编号只

允许在 0~63 之间，超时时间间隔的单位为毫秒。超时发生时，产生 DATA_TIMEOUT 事件，并给出超时的定时器编号。例如：start_timer(3, 1200)启动 3 号定时器，定时器的长度为 1.2 秒。

系统在把截止到调用函数 start_timer()时刻为止已经放入物理层发送队列的数据发送完毕后才开始启动计时（不是从当前时间开始计时）。例如：当前时间坐标为 5.100，物理层发送队列目前有 300 字节，信道速率 8000bps，函数调用 start_timer(3, 1200)会导致 1.5 秒之后时间坐标 6.600 处 3 号定时器产生 DATA_TIMEOUT 事件。

使用 stop_timer()中止一个定时器。在定时器未超时之前直接对同一个编号的定时器执行 start_timer()调用，将按照新的时间设置产生超时事件。

start_ack_timer()和 stop_ack_timer()两个定时器函数为搭载 ACK 机制设置。

start_ack_timer()与 start_timer()有两点不同：首先，定时器启动时刻为当前时刻；其次，在先前启动的定时器未超时之前重新执行 start_ack_timer()调用，定时器将依然按照先前的时间设置产生超时事件 ACK_TIMEOUT。

两种定时器不同的定时处理方式是为了适应数据链路层协议软件开发的需要。

8.11 协议工作过程的跟踪和调试

protocol.h 头文件中的相关定义和函数原型：

```
extern void dbg_event(char *fmt, ...);
extern void dbg_frame(char *fmt, ...);
extern void dbg_warning(char *fmt, ...);
char *station_name(void);
```

协议软件的开发调试需要解决编译错误、内存越界、内存泄漏等语言级别错误。这里所提及的协议软件跟踪调试系统不是指语言级调试，指的是运行期对协议动作进行跟踪和观察分析以确定协议逻辑是否正确。在实用系统中，这种调试机制一般会被加入到发行版本的协议软件中。默认情况下，不产生任何调试信息的输出；通过某种命令可以打开调试开关，边运行边产生协议动作信息的输出。Cisco 等系统中普遍提供这样的协议工作过程的跟踪和分析手段。

调试信息的输出格式和输出内容由协议程序设计者设计，并嵌入到源程序中。程序库内有一个名为 debug_mask 的静态整型变量(默认值 0)用于调试信息的输出控制，可以通过命令行选项对该变量赋值。debug_mask 变量的设置方法参见“命令行选项”。

使用 debug_mask 变量的不同比特位控制不同类别的调试信息输出。这里把调试信息分为两类，一类定义为“事件”，只有在发生某些不正常事件时才产生输出，例如：CRC 校验错，帧超长（可能是因为误码毁掉了帧边界字符），定时器超时重传，等等；另一类定义为“帧收发”，每发送和接收一帧，都打印出相关调试信息便于协议分析。为此，用 debug_mask 的比特 0 控制“事件”信息，比特 1 控制“帧收发”信息。提供了两个函数 dbg_event()和函数 dbg_frame()，当 debug_mask 的 0 号比特为 0 时，dbg_event()的所有输出被忽略，反之，执行与 log_printf 相同的功能；当 debug_mask 的 1 号比特为 0 时，dbg_frame()的所有输出被忽略，反之，执行与 log_printf 相同的功能。

函数 station_name()获取当前进程所对应的站点名，为字符串”A”或者”B”。

8.12 命令行选项

在启动可执行程序文件(.EXE 文件)运行时，可以在命令行中附带一些选项对程序的执行进行控制。这些选项用来指定启动的新进程是站点 A 还是站点 B，控制分组序列发生器的工作模式（分组序列发生器的工作模式必须做到两个站点选项完全一致），控制前述的 debug_mask 取值以控制调试信息的输出，等等。设所生成的可执行文件为 datalink.exe，那么，直接执行不带任何参数的该命令会给出帮助信息。

Usage:
  ./datalink <options> <station-name>

Options :
    -?, --help : print this
    -u, --utopia : utopia channel (an error-free channel)
    -f, --flood : flood traffic
    -i, --ibib : set station B layer 3 sender mode as IDLE-BUSY-IDLE-BUSY-...
    -n, --nolog : do not create log file
    -d, --debug=<0-7>: debug mask (bit0:event, bit1:frame, bit2:warning)
    -p, --port=<port#> : TCP port number (default: 59144)
    -b, --ber=<ber> : Bit Error Rate (received data only)
    -l, --log=<filename> : using assigned file as log file
    -t, --ttl=<seconds> : set time-to-live
i.e.
    ./datalink -fd3 -b 1e-4 A
    ./datalink --flood --debug=3 --ber=1e-4 A

单字母的选项可以连写在一起，例如：下面的命令指定了 f, u, d 三个选项：datalink -fud3 A

表 1 命令行单字母选项功能表

短选项	长选项	功能
-u	--utopia	(Utopia)设置本站接收方向的信道误码率为 0。默认值为 1.0e-5
-f	--flood	(Flood)洪水模式。本站网络层有无穷量分组流及时供应给数据链路层发送。默认情况下，站点 A 以较平缓方式断断续续不停产生分组流，站点 A 以 100 秒为周期发一段停一段
-i	--ibib	仅对站点 B 有意义。站点 B 默认工作方式下，第一个 100 秒有数据发送，第二个 100 秒极少数据发送，第三个 100 秒又有数据发送，.....，如此 BUSY-IDLE-BUSY -IDLE-.....反复。指定 i 选项后，站点 B 的分组序列产生模式与此相反，以 IDLE-BUSY- IDLE-BUSY-.....周期反复。IDLE 期间大约每 4 秒才发送一帧。
-n	--nolog	(No log)取消日志文件。如果需要连续 24 小时执行程序又不打算让日志信息占用磁盘空间，可以使用此选项。默认情况下，日志文件的位置和命名参见 8.2
-d3	--debug=3	设定 debug_mask 变量的值为 0~7 其中之一。用于控制程序中 dbg_event()和 dbg_frame()以及 dbg_warning()的输出。变量 debug_mask 的默认值为 0
-p10002	--port=10002	指定 TCP 通信的端口号为 10002，默认为端口 59144，如果与 Windows 系统中其他程序冲突或者在同一台计算机中启动多对“站点 A/站点 B”通信，那么需要指定其他端口号。
-l file.log	-log=file.log	自己指定一个日志文件名 file.log 替代进程的默认日志文件。
-b 2e-4	--ber==2e-4	指定信道误码率（必须小于 1），使用 C 语言浮点数常数格式，如：误码率 2×10^{-4}，表示为 2.0E-4

例如：datalink -fd3 –p 10002 –l c:\temp\a.log –b 2.0E-4 A

8.13 错误信息

执行程序(.EXE 文件)运行时，库程序如果遇到错误，就打印错误信息，并立刻中止程序的运行。

表 2 库文件中的错误信息列表

类别	错误信息及说明
参数错	Station name must be 'A' or 'B' 启动可执行文件时必须在命令行参数中指定站点名
检测到链路层工作失败	Network Layer: incorrect packet length Network Layer received a bad packet from data link layer 发送端站点使用伪随机数公式生成了分组流，分组中每个字节的内容都是通过公式计算所得，接收端利用与发送端相同的公式计算并与收到的分组流进行比对，发现错误，将给出这两条信息之一
操作系统环境问题	Windows Socket DLL Error Failed to create TCP socket 这类错误很少出现。如果出现，说明 Windows 安装有问题，无法进行 socket 通信
	**** WARNING: System too busy, sleep 15 ms, but be awakened 61 ms later 警告信息，不会导致程序中止运行，但可能影响算法的线路利用率统计指标。 检测到系统忙碌：进程主动请求睡眠 15ms，但是被唤醒后发现时间已逝去 61ms。 关闭系统中其它的运行程序。
TCP 通信故障	Station A failed to bind TCP port 站点 A 作为 TCP 服务器端运行，绑定 TCP 端口号失败，将给出此错误信息。默认端口号 59144，可以通过命令行选项指定端口号。应注意 Windows 网络放火墙或所安装病毒监控软件的配置
	Station B failed to connect station A 站点 B 作为 TCP 客户端运行，连接 127.0.0.1 地址的 TCP 端口 59144 或在命令行选项中指定的端口。TCP 连接失败，将给出此错误信息。应检查站点 A 的程序是否已经启动，端口号是否与站点 B 的端口号一致
内存管理	No enough memory 内存被消耗光，没有充足的内存。出现此错误，说明程序中存在“内存泄漏”问题
	Memory used by 'protocol.lib' is corrupted by your program 库程序监测到它专用的内存被篡改。检查程序中是否有内存访问越界的问题
物理层发送队列溢出	Physical Layer Sending Queue overflow 物理层发送队列溢出（队列最多可以缓冲 64K 字节）
函数调用错	recv_frame(): Receiving Queue is empty 未产生 FRAME_RECEIVED 事件就执行 recv_frame()获取物理层接收到的数据
	get_packet(): Network layer is not ready for a new packet 未产生 NETWORK_LAYER_READY 事件就执行 get_packet()从网络层获取下一个待发送分组
	start_timer(): timer No. must be 0~128 系统最多支持 129 个定时器，定时器编号太大

8.14 样例程序文件 datalink.c

样例程序实现了简单的全双工“停-等”协议，未设 ACK 定时器，收到数据就立刻回复 ACK。

分别在两个命令行窗口运行 datalink A 和 datalink B，那么会启动两个站运行。

如果运行 datalink –d3 A 和 datalink –d3 B，那么，会打印出协议运行信息。协议运行信息的输出，也是在 datalink.c 中设定的。

将样例程序文件 datalink.c 内容全部删掉，就可以在这个文件中编辑自己的协议程序。

9. 正确性测试及性能测试

滑动窗口机制的两个主要目标：实现有误码信道环境下的无差错传输；充分利用传输信道的带宽。完成无差错传输只是达到了基本目标，追求信道实际利用率是关键。测试以下几种情况下你所设计的软件能否正常工作 20 分钟，记录线路利用率。在实验报告中分析这些数据，给出实际线路利用率离极限效率之间尚有多大差距，并解释产生这样结果的原因。

注意：协议软件正常工作的标志是网络层发送端持续发送分组的情况下接收端连续成功地接收到分组，put_packet()函数接纳收到的分组后会给出一个报告，参见 8.6，这个报告应当过一小段时间就出现一次；从某时间点开始再也不出现这个报告，你的软件可能死锁了，协议失败。

表 3 性能测试记录表

序号	命令	说明	运行时间(秒）	接收方线路利用率(%)		存在的问题
				A	B	
1	datalink au datalink bu	无误码信道数据传输				
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，慢发 100 秒”				
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				
4	datalink af datalink bf	站点 A/B 的分组层都洪水式产生分组				
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}				

10. 研究与探索的问题

10.1 CRC 校验能力

假设本次实验中所设计的协议用于建设一个通信系统。这种“在有误码的信道上实现无差错传输”的功能听起来很不错，但是后来该客户听说 CRC 校验理论上不可能 100%检出所有错误。这的确是事实。你怎样说服他相信你的系统能够实现无差错传输？如果传输一个分组途中出错却不能被接收端发现，算作一次分组层误码。该客户使用本次实验描述的信道，客户的通信系统每天的使用率 50%，即：每天只有一半的时间在传输数据，那么，根据你对 CRC32 检错能力的理解，发生一次分组层误码事件，平均需要多少年？从因特网或其他参考书查找相关材料，看看 CRC32 有没有充分考虑线路误码的概率模型，实际校验能力到底怎样。你的推算是过于保守了还是夸大了实际性能？如果你给客户的回答不能让他满意这种分组层误码率，你还有什么措施降低发生分组层误码事件的概率，这些措施需要什么代价？

10.2 CRC 校验和的计算方法

本次实验中 CRC32 校验和的计算直接调用了一个简单的库函数，8.8 节中的库函数 crc32()是从 RFC1662 中复制并修改而来。在 PPP 相关协议文本 RFC1662.TXT（以另外单独一个文件提供）中含有计算 CRC-32 和 CRC-16 的源代码，浏览这些源代码。教材中给出了手工计算 CRC 校验和的方法，通过二进制“模 2”除法求余数。这些源代码中却采用了以字节值查表并叠加的方案，看起来计算速度很快。你

能分析出这些算法与我们课后习题中手工进行二进制“模 2”除法求余数的算法是等效的吗？算法中设置的查表数组 crc_table[256]数组是怎样构造出来的？你能否写一段 C 语言程序，按照模 2 除法的规则，用你的程序生成速查表 crc_table 的 256 个数字？或者，给出一个例子手工验证表中的某项是怎样得来的。在 x86 系列计算机上为某一帧计算 32 位 CRC32 校验和，比较一下：为相同帧计算 16 位 CRC16 校验和，所花费的 CPU 时间会多一倍吗？本次实验课提供的 crc32 函数只有两个参数，分别为缓冲区首地址和缓冲区长度，这似乎足够了，可是，RFC1662 给出的源代码样例中的函数 pppfcs32(fcs, cp, len)需要三个参数，为什么要这样设计？

10.3 程序设计方面的问题

8.10 节提出的协议软件的跟踪功能有什么意义？你的程序实现这样的功能了吗？程序库中获取时间坐标的函数 get_ms()不是 C 语言标准库中的函数，你能自己实现一个这样的函数吗？

在“C 语言程序设计”课程中学习过 printf 函数，printf 风格的函数特点是参数数目不确定，类似的标准库函数还有 fprintf，sprintf，scanf 等。为了便于调试程序，程序库中提供了日志文件和屏幕窗口同步输出的 lprintf 函数(参见 8.3)，这些函数都不是标准的 C 语言库函数，调用风格与 printf 类似。可以参考给出的源代码，看看 printf 风格的函数是怎样实现的。

8.9 节中给出了两个函数 start_timer()和 start_ack_timer()，它们都是定时器函数，两个函数启动定时器的时机不同，而且在定时器到时之前重新调用函数对原残留时间的处理方式也不同，为什么要这样设计？

10.4 软件测试方面的问题

验证所完成的程序能否在各种情况下都能够正确工作，是软件测试环节的主要目的。表 3 中列出了七种测试方案，设计这么多种测试方案的目的是什么？分析每种测试方案，每种方案主要是为了瞄准你的协议软件中可能出问题的哪些环节？或者说，你的协议软件存在什么问题时，测试会失败？你觉得还存在有哪些问题是这些测试尚未覆盖的？这些测试方案和验证协议正确性的手段由指导教师给出，如果是由你自己独立完成整个协议的设计和测试，你会采用哪些手段来验证你的程序能正确工作？针对本次实验的具体问题，你能不能提出一种更高效的软件测试方案？本次实验所提供的程序库还有哪些不足，怎样才能对协议开发提供更方便的支撑，这关系到在整体软件开发过程中的不同模块间的功能划分问题，给出你的建议。

10.5 对等协议实体之间的流量控制

在教科书中多次提及“流量控制”问题，这个问题的确很重要。在本次实验所设计的程序中，多多少少也考虑了上下层软件实体之间的数据流量控制问题。你认为你所设计的滑动窗口协议软件有没有解决两个站点的数据链路层对等实体之间的流量控制问题？如果是已经解决了，那么是怎样解决的？如果尚未解决，那么还需要对协议进行哪方面的改进？

10.6 与标准协议的对比

如果现实中有两个相距 5000 公里的站点要利用你所设计的协议通过卫星信道进行通信，还有哪些问题需要解决？实验协议离实用还有哪些差距？你觉得还需要增加哪方面的功能？从因特网或其他资料查阅 LAPB 相关的协议介绍和对该协议的评论，用成熟的 CCITT 链路层协议标准对比你所实现的实验性协议，实验性协议的设计还遗漏了哪些重要问题？

11. 实验报告要求

下面是应提交实验报告的内容提纲和每项目的具体要求。实验完成后，必须以电子版和纸质两种方式提交源程序和实验报告。

11.1 实验内容和实验环境描述

描述本次实验的任务、内容和实验环境。

11.2 软件设计

给出程序的数据结构，模块之间的调用关系和功能，程序流程。

(1) 数据结构：数据结构是整个程序的要点之一，程序维护者充分了解数据结构就可以对主要算法和处理流程有个基本的理解。描述程序中自定义结构体中各成员的用途，定义的全局变量和主函数中的变量的变量名和变量所起的作用。

(2) 模块结构：给出程序中所设计的子程序所完成的功能，子程序每个参数的意义。给出子程序之间的程序调用关系图。

(3) 算法流程：画出流程图，描述算法的主要流程。

11.3 实验结果分析

(1) 描述你所实现的协议软件是否实现了有误码信道环境中无差错传输功能。

(2) 程序的健壮性如何，能否可靠地长时间运行。

(3) 协议参数的选取：滑动窗口的大小，重传定时器的时限，ACK 搭载定时器的时限，这些参数是怎样确定的？根据信道特性数据，分组层分组的大小，以及你的滑动窗口机制，给出定量分析，详细列举出选择这些参数值的具体原因。

(5) 理论分析：根据所设计的滑动窗口工作机制(Go-Back-N 或者选择重传)，推导出在无差错信道环境下分组层能获得的最大信道利用率；推导出在有误码条件下重传操作及时发生等理想情况下分组层能获得的最大信道利用率。给出理论推导过程。理论推导的目的是得到信道利用率的极限数据。为了简化有误码条件下的最大利用率推导过程，可以对问题模型进行简化，比如：假定超时重传的数据帧的回馈 ACK 帧可以 100%正确传输，但是简化问题分析的这些假设必须不会对整个结论产生较大的误差。

(6) 实验结果分析：你的程序运行实际达到了什么样的效率，比对理论推导给出的结论，有没有差距？给出原因。有没有改进的办法？如果没有时间把这些方法付诸编程实施，介绍你的方案。

(7) 存在的问题：在“表 3 性能测试记录表”中给出了几种测试方案，在测试中你的程序有没有失败，或者，虽未失败，但表现出来的性能仍有差距，你的程序中还存在哪些问题?

11.4 研究和探索的问题

前面列出的“可研究和探索的问题”，有哪些问题你有了答案或者自己的见解？给出你的结论，并详细阐述你的理由或见解。

11.5 实验总结和心得体会

如果一切 100%顺利，编辑的程序一次编译就通过，运行一次就正确，那么完成本次实验的代码编写和调试工作大约需要 4~6 个小时。你花的时间超过了这个预测吗？描述在调试过程中都遇到了哪些问题和解决的过程。

(1) 完成本次实验的实际上机调试时间是多少？

(2) 编程工具方面遇到了哪些问题？包括 Windows 环境和 VC 软件的安装问题。

(3) 编程语言方面遇到了哪些问题？包括 C 语言使用和对 C 语言操控能力上的问题。

(4) 协议方面遇到了哪些问题？包括协议机制的设计错误，发现协议死锁，或者不能正确工作，协议参数的调整等问题。

(5) 开发库方面遇到了哪些问题？包括库程序中的 BUG，库函数文档不够清楚导致误解，库函数在所提供的功能结构上的缺憾导致编程效率低下。这些问题或建议影响不同模块之间功能界限的划分。

(6) 总结本次实验，你在 C 语言方面，协议软件方面，理论学习方面，软件工程方面等哪些方面上有所提高？

11.6 源程序文件

按照附录一“源程序书写格式”要求的源程序书写规范，格式化你的 C 语言源程序。提供 C 语言源程序文件.c 和.h，仅提供你自己编写的 C 源程序文件，不要把实验材料中的 lprintf.c 之类的源文件和工程文件以及生成的中间文件提交上来。

附录一 源程序书写格式

尽管 C 语言在语法规定上对书写格式没有严格的限制，但是源程序的排版格式应当遵从多数程序员的惯例，以便于程序的维护。许多同学可能受中文单词间不需要空格的影响以及 C 语言教科书中未对书写格式提出建议，导致书写的 C 语言源程序缺少必要的空格，密密麻麻的语法元素拥挤在一起，阅读起来需要首先“断词”。按照惯例应当添加适当的空格，断开相关的“语法单词”。下面列举出书写方面应注意的几个问题。后面两页附上的源代码样例节选于 Linux 的源程序文件 ping.c，源代码样例的每行都带有行号，对书写方面应注意的问题提供了实际例子的参考行号。

附表 1　C 语言源程序书写格式建议

项目	说明	参考行号
适当的空行	根据程序的上下文，分成逻辑上的几段，段与段空行	7，12，33
语句的层次缩进	每个层次缩进 4 个空格	13~16
双目运算符	运算符两侧各加一个空格	18~20，51
三目运算符	运算符两侧各加一个空格，如： a > b ? a : b	
单目运算符	与它作用的变量之间不要加空格	9，32，65
关键字	关键字与它后面的括号之间保留一个空格	8，13
函数名	函数名与后面的括号之间不要保留空格，与关键字处理不同	24，32，79
宏名字	宏名字与函数名类似，名字与后面的括号之间不要保留空格	64，69
逗号	逗号后保留一个空格	24，28，80
分号	分号后保留一个空格，如：for (i = 0; i < n; i++)	
括号	左圆括号（或方括号）之后，右括号之前不要保留空格	18，24，64
换行	应当放在两行的内容，不要挤在一行内	59~60，65~66
行的长度	每行长度尽量控制在 80 字符内	76~78
注释	避免无聊的注释；避免程序修改了，不改注释，误导他人	
花括号	if，for，while，switch 等花括号排版风格在整个文件中一致	8~11，64~71
命名	变量、函数及结构体的域，命名风格在整个文件中一致	1，49

花括号的书写风格有两种：

```
if (xxxxx) {
    yyyyyyyyyyyyyyyyyyyy
    yyyyyyyyyyyyyyy
} else {
    zzzzzzzzzzzzzzzz
    zzzzzzzzzzzzzzzz
}
```

```
if (xxxxx)
{
    yyyyyyyyyyyyyyyyyyyy
    yyyyyyyyyyyyyyy
}
else
{
    zzzzzzzzzzzzzzzz
    zzzzzzzzzzzzzzzz
}
```

第一种风格起源于早期 UNIX 的内核源代码，即 C 语言诞生的地方；第二种风格起源于 Pascal 程序员 begin/end 关键字的书写习惯向 C 语言的转变。目前，在 Microsoft 世界和 Linux 世界中，两种书写风格都很常见，第一种风格略占上风。无论选择哪种风格必须做到同一个源程序文件中一致，不要两种风格混用。

变量和函数命名方法也有两种风格。一种是匈牙利命名法（源自 Microsoft 的一位匈牙利籍天才程序员），如：SendAckFrame；另一种方法为传统的下划线分割法，如：send_ack_frame。同一个源程序文件中的命名风格应做到一致。

```c
static int in_cksum(unsigned short *buf, int sz)
{
   int nleft = sz;
   int sum = 0;
   unsigned short *w = buf;
   unsigned short ans = 0;

   while (nleft > 1) {
      sum += *w++;
      nleft -= 2;
    }

    if (nleft == 1) {
       *(unsigned char *)&ans = *(unsigned char *)w;
       sum += ans;
    }

    sum = (sum >> 16) + (sum & 0xFFFF);
    sum += (sum >> 16);
    ans = ~sum;
    return ans;
}

static void unpack(char *buf, int sz, struct sockaddr_in *from)
{
    struct icmp *icmppkt;
    struct iphdr *iphdr;
    struct timeval tv, *tp;
    int hlen, dupflag;
    unsigned long triptime;

    gettimeofday(&tv, NULL);

    /* check IP header */
    iphdr = (struct iphdr *) buf;
    hlen = iphdr->ihl << 2;

    /* discard if too short */
    if (sz < (datalen + ICMP_MINLEN))
       return;

    sz -= hlen;
    icmppkt = (struct icmp *) (buf + hlen);
```

```
44      if (icmppkt->icmp_id != myid)
45          return;           /* not our ping */
46
47      if (icmppkt->icmp_type == ICMP_ECHOREPLY) {
48          nreceived++;
49          tp = (struct timeval *) icmppkt->icmp_data;
50
51          if ((tv.tv_usec -= tp->tv_usec) < 0) {
52              tv.tv_sec--;
53              tv.tv_usec += 1000000;
54          }
55          tv.tv_sec -= tp->tv_sec;
56
57          triptime = tv.tv_sec * 10000 + (tv.tv_usec / 100);
58          tsum += triptime;
59          if (triptime < tmin)
60              tmin = triptime;
61          if (triptime > tmax)
62              tmax = triptime;
63
64          if (TST(icmppkt->icmp_seq % MAX_DUP_CHK)) {
65              nrepeats++;
66              nreceived--;
67              dupflag = 1;
68          } else {
69              SET(icmppkt->icmp_seq % MAX_DUP_CHK);
70              dupflag = 0;
71          }
72
73          if (options & O_QUIET)
74              return;
75
76          printf("%d bytes from %s: icmp_seq=%u", sz,
77                  inet_ntoa(*(struct in_addr *)&from->sin_addr.s_addr),
78                  icmppkt->icmp_seq);
79          printf(" ttl=%d", iphdr->ttl);
80          printf(" time=%lu.%lu ms", triptime / 10, triptime % 10);
81          if (dupflag)
82              printf(" (DUP!)");
83          printf("\n");
84      }
85  }
```