《计算机网络》实验报告

年级、专业、班级	2021 级计算机科学与技术（卓越）2 班、2021 级计算机科学与技术（卓越）1 班	姓名	胡欣凯、李宽宇
实验题目	协议数据单元封装与解封装		

教师评价：

□ 算法/实验过程正确；　□ 源程序/实验内容提交　□ 程序结构/实验步骤合理；

□ 实验结果正确；　□ 语法、语义正确；　□ 报告规范；

其他：

评价教师签名：

一、实验内容

完成数据链路层、网络层、运输层的协议数据单元封装和解封装。我们实现了 UDP 用户数据报、TCP 报文段、IPv4 数据报、PPP 帧、Ethernet II 的 MAC 帧协议的封装和解封装，并且实现了 PPP 协议透明传输的字节填充和零比特填充。我们对每一个协议数据单元的封装和解封装分别进行了测试，并进行了数据链路层、网络层、运输层组合的报文传输测试。完整项目代码上传在 GitHub 网站 https://github.com/XinkaiHu/computer-network-protocols。

二、实验过程或算法

1. 设计思路

在各协议数据单元封装时，除 PPP 帧的零比特填充外，主要以字节为基本操作单元，少量以比特为操作单元；C 语言数据类型的最小存储大小是 1 字节，不同数据类型具有不同的存储大小，且能够方便地完成位运算操作和算术运算操作。因此考虑使用 C 语言中存储大小为 1 的 char 或者 unsigned char 数组保存各种数据，通过位运算完成不足 1 字节的数据赋值，通过强制类型转换完成 2 字节、4 字节的数据赋值。

2. 基本工具

我们使用 CPU 体系结构为 x86_64 体系结构，编译器版本为

报告创建时间：2023 年 12 月 1 日

```
xinkai@LAPTOP-777VF7EM:~/cpp$ gcc --version
gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

由此定义不同存储大小（1 字节、2 字节、4 字节和 8 字节）的数据类型。

```
typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned int uint32_t;
typedef unsigned long uint64_t;
```

定义数据传输类型 data_t，包含字节数组的数据和数据长度。

```
typedef struct data_t {
  uint8_t* value;
  uint32_t size;
} data_t;
```

为方便完成零比特填充和查看封装、解封装结果，我们完成了从字节数组与二进制字符串相互转换、从字节数组到十六进制字符串的转换工具函数。

- 字节数组转换为二进制字符串：

```
void char_to_binary(data_t const input, data_t* output) {
  output->size = input.size << 3;
  output->value = (uint8_t*)malloc(output->size + 1);

  for (int i = 0; i < input.size; ++i) {
    for (int j = 0; j < 8; ++j) {
      output->value[i * 8 + j] = ((input.value[i] >> (7 - j)) & 1) ? '1' : '0';
    }
  }
  output->value[output->size] = '\0';
}
```

- 二进制字符串转换为字节数组：

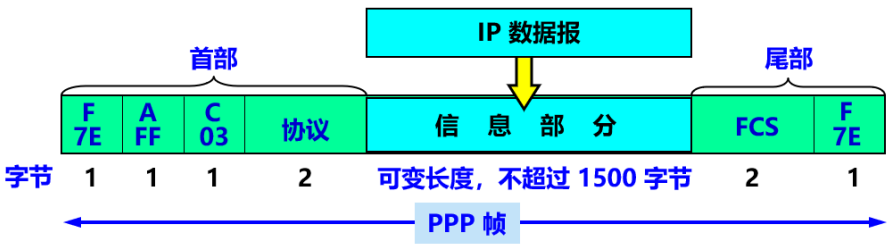
```
void binary_to_char(data_t const input, data_t* output) {
  assert((input.size & 0X7) == 0);
  output->size = input.size >> 3;
  output->value = (uint8_t*)malloc(output->size);

  for (int i = 0; i < output->size; ++i) {
    output->value[i] = 0;
    for (int j = 0; j < 8; ++j) {
      output->value[i] |= (input.value[i * 8 + j] == '1' ? 1 : 0) << (7 - j);
    }
  }
}
```

- 字节数组转换为十六进制字符串：

```
void char_to_hex(data_t const input, data_t* output) {
  uint8_t const hex_table[] = "0123456789ABCDEF";
  output->size = input.size << 1;
  output->value = (uint8_t*)malloc(output->size + 1);

  for (int i = 0; i < input.size; ++i) {
    output->value[i << 1] = hex_table[(input.value[i] >> 4) & 0xF];
    output->value[1 + (i << 1)] = hex_table[input.value[i] & 0XF];
  }
  output->value[output->size] = '\0';
}
```

3. 数据链路层 PPP 协议

PPP 协议的帧格式为



其中协议字段需要根据情况指定。

PPP 协议的特点包括：简单；通过位于首部和尾部的帧定界符封装成帧；通过异步传输中使用字节填充法、同步传输时使用零比特填充法实现透明传输；通过 CRC 冗余码进程差错检测等。

3.1. PPP 帧封装

```
void send_ppp(data_t const input, data_t* output, PPP_PROTOCOL const protocol) {
```

首部包含 4 个字段，尾部包含 2 个字段，总长度为 IP 数据报长度加 8。其中信息部分长度不超过 1500 字节。

```
  assert(input.size <= 1500);
  output->size = input.size + 8;
  output->value = (uint8_t*)malloc(output->size);
```

首部的标志字段为 0X7E；地址字段为 0XFF；控制字段通常置为 0X03。

```
  output->value[0] = 0X7E;
  output->value[1] = 0XFF;
  output->value[2] = 0X03;
```

协议字段取值的情形为

协议字段取值	说明
0X0021	信息字段为 IP 数据报

0X8021	信息字段为网络控制数据
0XC021	信息字段为 PPP 链路控制数据
0XC023	信息字段为鉴别数据

定义协议字段枚举类型，其值分别对应上述四种情形。

```
typedef enum PPP_PROTOCOL {
  PPP_PROTOCOL_IPD,
  PPP_PROTOCOL_NCD,
  PPP_PROTOCOL_LCD,
  PPP_PROTOCOL_AD
} PPP_PROTOCOL;
```

根据信息字段类型设置协议字段取值。此处由于协议字段占 2 字节，因此通过强制类型转换完成赋值。

```
switch (protocol) {
  case PPP_PROTOCOL_IPD:
    *(uint16_t*)(output->value + 3) = 0X0021;
    break;
  case PPP_PROTOCOL_NCD:
    *(uint16_t*)(output->value + 3) = 0X8021;
    break;
  case PPP_PROTOCOL_LCD:
    *(uint16_t*)(output->value + 3) = 0XC021;
    break;
  case PPP_PROTOCOL_AD:
    *(uint16_t*)(output->value + 3) = 0XC023;
    break;
  default:
    assert(0);
}
```

将输入的 IP 数据报复制到信息字段。

```
memcpy(output->value + 5, input.value, input.size);
```

设置帧校验序列。此处由于帧校验序列占 2 字节，因此使用的是 CRC-16 冗余码，通过强制类型转换完成赋值。

```
*(uint16_t*)(output->value + (5 + input.size)) = crc16(*output, input.size + 5);
```

其中 CRC-16 冗余码的计算方法为

```
uint16_t crc16(data_t const input, uint16_t const n) {
  uint8_t i;
  uint16_t crc_value;
  uint16_t size;
  uint8_t* value;

  size = n;
  value = input.value;
  crc_value = 0XFFFF;
  while (size--) {
    crc_value ^= *value++;
    for (i = 0; i < 8; ++i) {
      if (crc_value & 0X0001) {
        crc_value = (crc_value >> 1) ^ 0XA001;
      } else {
        crc_value >>= 1;
      }
    }
  }

  crc_value = ~crc_value;
  return crc_value;
}
```

最后，在尾部置帧结束符。

```
output->value[7 + input.size] = 0X7E;
```

3.2. 透明传输

PPP 协议在异步传输时，使用**字节填充法**。

```
void byte_stuffing(data_t const input, data_t* output) {
```

字节填充法的规则是：除帧开始符和帧结束符为 0X7E 外，当 PPP 帧内部出现帧定界符 0X7E 或填充字符 0X7D 时，在其前方填充 0X7D，并将其减去 0X20；出现 ASCII 码中的控制字符（ASCII 码值小于 0X20）时，在其前方填充 0X7D，并将其加上 0X20。

在代码实现时，为防止造成内存空间浪费，首先检查需要填充的字节数，再分配内存并完成填充。

```
void byte_stuffing(data_t const input, data_t* output) {
  assert(input.value[0] == 0X7E && input.value[input.size - 1] == 0X7E);
  uint32_t input_pos;
  uint32_t output_pos;

  output->size = input.size;
  for (int i = 1; i < input.size - 1; ++i) {
    if (input.value[i] == 0X7E || input.value[i] == 0X7D ||
        input.value[i] < 0X20) {
      ++output->size;
    }
  }

  output->value = (uint8_t*)malloc(output->size);

  output->value[0] = input.value[0];
  output->value[output->size - 1] = input.value[input.size - 1];
  for (output_pos = 1, input_pos = 1; input_pos < input.size - 1; ++input_pos) {
    if (input.value[input_pos] == 0X7E || input.value[input_pos] == 0X7D) {
      output->value[output_pos++] = 0X7D;
      output->value[output_pos++] = input.value[input_pos] - 0X20;
    } else if (input.value[input_pos] < 0X20) {
      output->value[output_pos++] = 0X7D;
      output->value[output_pos++] = input.value[input_pos] + 0X20;
    } else {
      output->value[output_pos++] = input.value[input_pos];
    }
  }
}
```

PPP 协议在同步传输时，使用**零比特填充法**。

零比特填充法的规则是：除帧开始符和帧结束符为 0B01111110 外，当 PPP 帧内部出现连续 5 个 1 时将在其后方填充一个 0 比特，从而防止出现与帧定界符完全相同的比特组合。

在代码实现时，首先将字节数组转换为二进制字符串，再进行填充，以应对零比特填充后 PPP 帧长度不是整数字节的情况。为防止造成内存空间浪费，首先检查需要填充的比特数，再分配内存并完成填充。

```
void bit_stuffing(data_t const input, data_t* output) {
  assert(input.value[0] == 0X7E && input.value[input.size - 1] == 0X7E);

  data_t data_bit;
  uint32_t one_count;
  uint32_t input_pos;
  uint32_t output_pos;

  char_to_binary(input, &data_bit);
  output->size = data_bit.size;
  one_count = 0;

  for (int i = 8; i < data_bit.size - 8; ++i) {
    if (data_bit.value[i] == '1') {
      ++one_count;
    } else {
      one_count = 0;
    }

    if (one_count == 5) {
      ++output->size;
      one_count = 0;
    }
  }

  output->value = (uint8_t*)malloc(output->size);
  memcpy(output->value, data_bit.value, 8);
  memcpy(output->value + (output->size - 8),
         data_bit.value + (data_bit.size - 8), 8);
  one_count = 0;

  for (input_pos = 8, output_pos = 8; input_pos < data_bit.size - 8;
       ++input_pos) {
    output->value[output_pos++] = data_bit.value[input_pos];
    if (data_bit.value[input_pos] == '1') {
      ++one_count;
    } else {
      one_count = 0;
    }
    if (one_count == 5) {
      output->value[output_pos++] = '0';
    }
  }

  free(data_bit.value);
}
```

3.3. 透明传输的数据复原

根据透明传输过程，实现其逆过程。同样地，在代码实现时，首先检查了恢复后的数据长度，再进行内存分配。

字节填充法的数据复原：

```
void parse_byte_stuffing(data_t const input, data_t* output) {
  assert(input.value[0] == 0X7E && input.value[input.size - 1] == 0X7E);
  uint32_t input_pos;
  uint32_t output_pos;

  output->size = input.size;
  for (int i = 1; i < input.size - 1; ++i) {
    if (input.value[i] == 0X7D) {
      --output->size;
    }
  }

  output->value = (uint8_t*)malloc(output->size);

  output->value[0] = input.value[0];
  output->value[output->size - 1] = input.value[input.size - 1];
  for (output_pos = 1, input_pos = 1; input_pos < input.size - 1; ++input_pos) {
    if (input.value[input_pos] == 0X7D) {
      if (input.value[input_pos + 1] == 0X5E ||
          input.value[input_pos + 1] == 0X5D) {
        output->value[output_pos++] = input.value[input_pos + 1] + 0X20;
        ++input_pos;
      } else if (input.value[input_pos + 1] < 0X40) {
        output->value[output_pos++] = input.value[input_pos + 1] - 0X20;
        ++input_pos;
      } else {
        assert(0);
      }
    } else {
      output->value[output_pos++] = input.value[input_pos];
    }
  }
}
```

零比特填充法的数据复原：

```
void parse_bit_stuffing(data_t const input, data_t* output) {
  data_t data_bit;
  uint32_t one_count;
  uint32_t input_pos;
  uint32_t output_pos;

  data_bit.size = input.size;
  one_count = 0;

  for (int i = 8; i < input.size - 8; ++i) {
    if (input.value[i] == '1') {
      ++one_count;
    } else {
      one_count = 0;
    }
    assert(one_count < 6);
    if (one_count == 5) {
      --data_bit.size;
    }
  }

  data_bit.value = (uint8_t*)malloc(data_bit.size);
  memcpy(data_bit.value, input.value, 8);
  memcpy(data_bit.value + (data_bit.size - 8), input.value + (input.size - 8),
         8);
  one_count = 0;

  for (input_pos = 8, output_pos = 8; input_pos < input.size - 8; ++input_pos) {
    data_bit.value[output_pos++] = input.value[input_pos];
    if (input.value[input_pos] == '1') {
      ++one_count;
    } else {
      one_count = 0;
    }
    if (one_count == 5) {
      ++input_pos;
    }
  }

  binary_to_char(data_bit, output);
  assert(output->value[0] == 0X7E && output->value[output->size - 1] == 0X7E);

  free(data_bit.value);
}
```

在复原时，对数据的合理性进行了检查，包括复原后数据首尾应为帧定界符、数据长度为整数字节。

3.4. PPP 帧的解封装

```
void parse_ppp(data_t const input, data_t* output, PPP_PROTOCOL* protocol) {
```

信息字段的长度比 PPP 帧的长度少 8，且长度不大于 1500 字节。

```
  output->size = input.size - 8;
  assert(output->size <= 1500);
  output->value = (uint8_t*)malloc(output->size);
```

根据协议字段判断信息类型。

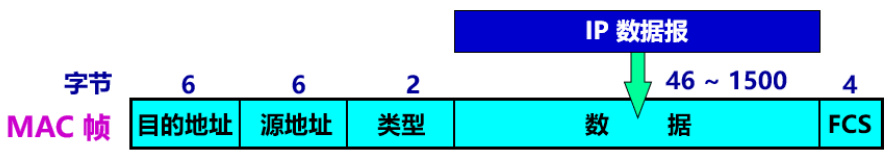
```
switch (*(uint16_t*)(input.value + 3)) {
  case 0X0021:
    *protocol = PPP_PROTOCOL_IPD;
    break;
  case 0X8021:
    *protocol = PPP_PROTOCOL_NCD;
    break;
  case 0XC021:
    *protocol = PPP_PROTOCOL_LCD;
    break;
  case 0XC023:
    *protocol = PPP_PROTOCOL_AD;
    break;
  default:
    assert(0);
}
```

复制信息字段。最后判断帧校验序列是否正确。

```
memcpy(output->value, input.value + 5, output->size);

assert(*(uint16_t*)(input.value + (5 + output->size)) ==
       crc16(input, output->size + 5));
```

4. 数据链路层 Ethernet II 协议

Ethernet II 的 MAC 帧格式为：



其中目的地址、源地址为以太网的 MAC 层中的 MAC 地址；类型标注上一层（IP 层）使用的是什么协议；我们使用 CRC-32 冗余码作为帧校验序列。

4.1. Ethernet II 的 MAC 帧封装

```
void send_ethernet_v2(data_t const input, data_t* output,
                      data_t const dest_addr, data_t const src_addr,
                      ETHERNET_V2_TYPE const type) {
```

MAC 帧中数据长度介于 46 字节到 1500 字节之间，当数据长度不足 46 字节时需要填充至 46 字节；目的地址和源地址长度为 6 字节，

```
  assert(input.size <= 1500);
  assert(src_addr.size == 6 && dest_addr.size == 6);
  output->size = input.size < 46 ? 46 : (input.size + 18);
  output->value = (uint8_t*)malloc(output->size);
```

将目的地址和源地址复制到 MAC 帧中。

```
  memcpy(output->value, dest_addr.value, 6);
  memcpy(output->value + 6, src_addr.value, 6);
```

通过查找资料，类型字段的取值和情形为：

类型字段取值	说明
0X0800	上一层使用 IPv4 协议
0X0806	上一层使用 ARP 协议
0X8864	上一层使用 PPPoE 协议
0X8100	上一层使用 IEEE 802.1q 协议
0X86DD	上一层使用 IPv6 协议
0X8847	上一层使用 MPLS 协议

定义类型字段的枚举类型。

```
typedef enum ETHERNET_V2_TYPE {
  ETHERNET_V2_TYPE_IPv4,
  ETHERNET_V2_TYPE_ARP,
  ETHERNET_V2_TYPE_PPPoE,
  ETHERNET_V2_TYPE_802_1Q,
  ETHERNET_V2_TYPE_IPv6
} ETHERNET_V2_TYPE;
```

根据上一层使用的协议类型，赋值协议字段。

```
  switch (type) {
    case ETHERNET_V2_TYPE_IPv4:
      *(uint16_t*)(output->value + 12) = 0X0800;
      break;
    case ETHERNET_V2_TYPE_ARP:
      *(uint16_t*)(output->value + 12) = 0X0806;
      break;
    case ETHERNET_V2_TYPE_PPPoE:
      *(uint16_t*)(output->value + 12) = 0X8864;
      break;
    case ETHERNET_V2_TYPE_802_1Q:
      *(uint16_t*)(output->value + 12) = 0X8100;
      break;
    case ETHERNET_V2_TYPE_IPv6:
      *(uint16_t*)(output->value + 12) = 0X86DD;
      break;
    default:
      assert(0);
  }
```

将上一层数据赋值到数据字段。

```
  memcpy(output->value + 14, input.value, input.size);
```

当数据字段长度不足 46 时，用 0 填充。

```
  if (input.size < 46) {
    memset(output->value + (14 + input.size), 0, 46 - input.size);
  }
```

最后，填入帧校验序列 CRC-32 冗余码。

```
  *(uint32_t*)(output->value + (14 + input.size)) = crc32(*output, 14 + input.size);
```

其中，CRC-32 冗余码的计算方法与 CRC-16 冗余码的计算方法类似。

```
uint32_t crc32(data_t const input, uint16_t const n) {
  uint8_t i;
  uint32_t crc_value;
  uint16_t size;
  uint8_t* value;

  size = n;
  value = input.value;

  crc_value = 0XFFFFFFFF;
  while (size--) {
    crc_value ^= *value++;
    for (i = 0; i < 8; ++i) {
      if (crc_value & 0X0001) {
        crc_value = (crc_value >> 1) ^ 0XEDB88320;
      } else {
        crc_value >>= 1;
      }
    }
  }

  crc_value = ~crc_value;
  return crc_value;
}
```

4.2. Ethernet II 的 MAC 帧解封装

```
void parse_ethernet_v2(data_t const input, data_t* output, data_t* dest_addr,
                       data_t* src_addr, ETHERNET_V2_TYPE* type) {
```

有效的 MAC 帧长度大于 64 字节。目的地址和源地址分别为 6 字节。

```
  assert(input.size >= 64);

  output->size = input.size - 18;
  output->value = (uint8_t*)malloc(output->size);
  dest_addr->size = 6;
  dest_addr->value = (uint8_t*)malloc(dest_addr->size);
  src_addr->size = 6;
  src_addr->value = (uint8_t*)malloc(src_addr->size);
```

从 MAC 帧中复制目的地址和源地址字段。

```
  memcpy(dest_addr->value, input.value, 6);
  memcpy(src_addr->value, input.value + 6, 6);
```

根据类型字段的值判断上一层使用的协议类型。

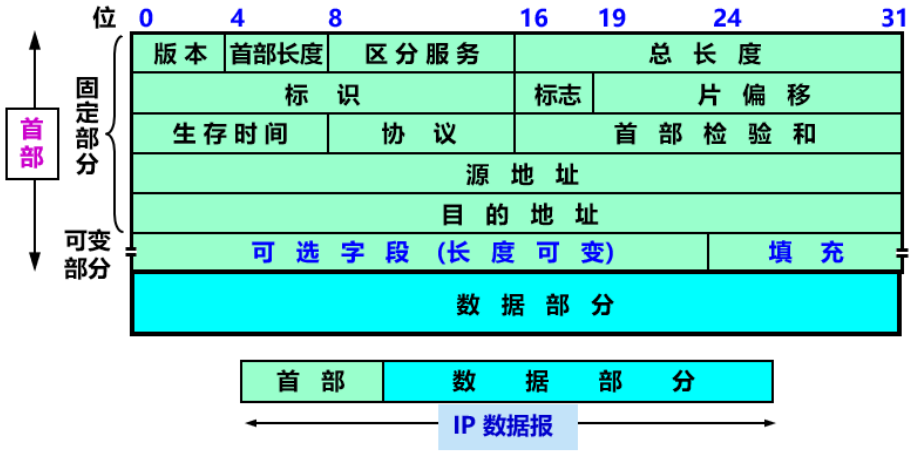
```
switch (*(uint16_t*)(input.value + 12)) {
  case 0X0800:
    *type = ETHERNET_V2_TYPE_IPv4;
    break;
  case 0X0806:
    *type = ETHERNET_V2_TYPE_ARP;
    break;
  case 0X8864:
    *type = ETHERNET_V2_TYPE_PPPoE;
    break;
  case 0X8100:
    *type = ETHERNET_V2_TYPE_802_1Q;
    break;
  case 0X86DD:
    *type = ETHERNET_V2_TYPE_IPv6;
    break;
  default:
    assert(0);
}
```

复制数据字段。检查帧校验序列。

```
memcpy(output->value, input.value + 14, output->size);

assert(*(uint32_t*)(input.value + (14 + output->size)) ==
       crc32(input, 14 + output->size));
```

5. 网络层 IPv4 协议

IPv4 数据报格式为：



5.1. IPv4 数据报的封装

```
void send_ip_v4(data_t const input, data_t* output, uint8_t const diff_serv,
                uint16_t const id, uint8_t const MF, uint8_t const DF,
                uint16_t const offset_byte, uint8_t const TTL,
                IPv4_PROTOCOL_TYPE const protocol, uint32_t const src_addr,
                uint32_t const dest_addr, data_t const optional) {
```

可选字段长度不大于 40 字节；标注字段为 1 比特数据；片偏移以 8 个字节为偏移单位。

```
assert(optional.size <= 40);
assert((MF & 0XFE) == 0 && (DF & 0XFE) == 0);
assert((offset_byte & 0X07) == 0);
```

若可选字段的长度不是 32 位的整数倍，则需要进行填充。

```
uint32_t filling_byte;
filling_byte = (4 - (optional.size & 0X3)) % 4;
uint16_t header_byte = 20 + optional.size + filling_byte;
```

填充后，IP 数据报总长度不大于 65535 字节。

```
output->size = input.size + header_byte;
assert(output->size <= 65535);
output->value = (uint8_t*)malloc(output->size);
```

依次填入各字段。其中版本字段始终为 4，首部长度、标志、片偏移字段长度不为整数字节，需要通过位运算完成赋值。

```
output->value[0] = 4;
output->value[0] |= (header_byte >> 2) << 4;
output->value[1] = diff_serv;
*(uint16_t*)(output->value + 2) = output->size;
*(uint16_t*)(output->value + 4) = id;
output->value[6] = MF;
output->value[6] |= DF << 1;
output->value[7] = 0;
*(uint16_t*)(output->value + 6) |= offset_byte;
output->value[8] = TTL;
```

常用的一些协议和相应的协议字段值为：

协议名	ICMP	IGMP	IP	TCP	EGP	IGP	UDP	IPv6	ESP	AH	ICMP-IPv6	OSPF
协议字段值	1	2	4	6	8	9	17	41	50	51	58	89

定义协议名的枚举类型。

```
typedef enum IPv4_PROTOCOL_TYPE {
  IPv4_PROTOCOL_TYPE_ICMP,
  IPv4_PROTOCOL_TYPE_IGMP,
  IPv4_PROTOCOL_TYPE_IP,
  IPv4_PROTOCOL_TYPE_TCP,
  IPv4_PROTOCOL_TYPE_EGP,
  IPv4_PROTOCOL_TYPE_IGP,
  IPv4_PROTOCOL_TYPE_UDP,
  IPv4_PROTOCOL_TYPE_IPv6,
  IPv4_PROTOCOL_TYPE_ESP,
  IPv4_PROTOCOL_TYPE_AH,
  IPv4_PROTOCOL_TYPE_ICMP_IPv6,
  IPv4_PROTOCOL_TYPE_OSPF
} IPv4_PROTOCOL_TYPE;
```

根据协议类型填入协议字段值。

```
switch (protocol) {
  case IPv4_PROTOCOL_TYPE_ICMP:
    output->value[9] = 1;
    break;
  case IPv4_PROTOCOL_TYPE_IGMP:
    output->value[9] = 2;
    break;
  case IPv4_PROTOCOL_TYPE_IP:
    output->value[9] = 4;
    break;
  case IPv4_PROTOCOL_TYPE_TCP:
    output->value[9] = 6;
    break;
  case IPv4_PROTOCOL_TYPE_EGP:
    output->value[9] = 8;
    break;
  case IPv4_PROTOCOL_TYPE_IGP:
    output->value[9] = 9;
    break;
  case IPv4_PROTOCOL_TYPE_UDP:
    output->value[9] = 17;
    break;
  case IPv4_PROTOCOL_TYPE_IPv6:
    output->value[9] = 41;
    break;
  case IPv4_PROTOCOL_TYPE_ESP:
    output->value[9] = 50;
    break;
  case IPv4_PROTOCOL_TYPE_AH:
    output->value[9] = 51;
    break;
  case IPv4_PROTOCOL_TYPE_ICMP_IPv6:
    output->value[9] = 58;
    break;
  case IPv4_PROTOCOL_TYPE_OSPF:
    output->value[9] = 89;
    break;
  default:
    assert(0);
}
```

填入其他字段。当可选字段长度不是 32 位的整数倍时，用 0 填充。

```
*(uint32_t*)(output->value + 12) = src_addr;
*(uint32_t*)(output->value + 16) = dest_addr;
memcpy(output->value + 20, optional.value, optional.size);
memset(output->value + (20 + optional.size), 0, filling_byte);
```

填入首部校验和字段。

```
*(uint16_t*)(output->value + 10) = 0X0000;
*(uint16_t*)(output->value + 10) = checksum_ip(*output, header_byte);
```

其中首部校验和的计算方法是将首部按 4 字节分段求和，当求和大于 16 位时将超出部分也加入校验和，最后将结果取反码。为保留校验和计算过程中超出 16 位的部分，使用 32 位数据保存校验和的计算中间结果。

```
uint16_t checksum_ip(data_t const input, uint16_t n) {
  uint32_t checksum_help = 0;
  for (uint8_t i = 0; i < n; i += 2) {
    checksum_help += *(uint16_t*)(input.value + i);
  }
  while (checksum_help > 0XFFFF) {
    checksum_help = (checksum_help >> 16) + (uint16_t)checksum_help;
  }
  uint16_t checksum;
  checksum = (uint16_t)checksum_help;
  checksum = ~checksum;
  return checksum;
}
```

最后，将数据部分填入 IP 数据报。

```
  memcpy(output->value + header_byte, input.value, input.size);
```

5.2. IPv4 数据报的解封装

```
void parse_ip_v4(data_t const input, data_t* output, uint8_t* version,
                 uint8_t* header_byte, uint8_t* diff_serv, uint16_t* total_byte,
                 uint16_t* id, uint8_t* MF, uint8_t* DF, uint16_t* offset_byte,
                 uint8_t* TTL, IPv4_PROTOCOL_TYPE* protocol, uint32_t* src_addr,
                 uint32_t* dest_addr, data_t* optional) {
```

解析首部各字段。

```
  assert((input.value[0] & 0X0F) == 4);
  *version = 4;
  *header_byte = (input.value[0] >> 4) << 2;
  *diff_serv = input.value[1];
  *total_byte = *(uint16_t*)(input.value + 2);
  *id = *(uint16_t*)(input.value + 4);
  *MF = input.value[6] & 1;
  *DF = (input.value[6] >> 1) & 1;
  *offset_byte = *(uint16_t*)(input.value + 6) & 0XF8;
  *TTL = input.value[8];
```

根据协议字段取值判断协议类型。

```
switch (input.value[9]) {
  case 1:
    *protocol = IPv4_PROTOCOL_TYPE_ICMP;
    break;
  case 2:
    *protocol = IPv4_PROTOCOL_TYPE_IGMP;
    break;
  case 4:
    *protocol = IPv4_PROTOCOL_TYPE_IP;
    break;
  case 6:
    *protocol = IPv4_PROTOCOL_TYPE_TCP;
    break;
  case 8:
    *protocol = IPv4_PROTOCOL_TYPE_EGP;
    break;
  case 9:
    *protocol = IPv4_PROTOCOL_TYPE_IGP;
    break;
  case 17:
    *protocol = IPv4_PROTOCOL_TYPE_UDP;
    break;
  case 41:
    *protocol = IPv4_PROTOCOL_TYPE_IPv6;
    break;
  case 50:
    *protocol = IPv4_PROTOCOL_TYPE_ESP;
    break;
  case 51:
    *protocol = IPv4_PROTOCOL_TYPE_AH;
    break;
  case 58:
    *protocol = IPv4_PROTOCOL_TYPE_ICMP_IPv6;
    break;
  case 89:
    *protocol = IPv4_PROTOCOL_TYPE_OSPF;
    break;
  default:
    assert(0);
}
```

判断首部校验和是否正确。

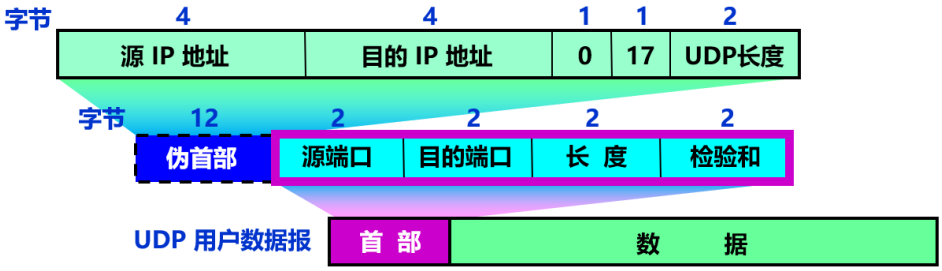
```
assert(0 == checksum_ip(input, *header_byte));
```

解析其他字段和数据。

```
*src_addr = *(uint32_t*)(input.value + 12);
*dest_addr = *(uint32_t*)(input.value + 16);
optional->size = *header_byte - 20;
if (optional->size > 0) {
  optional->value = (uint8_t*)malloc(optional->size);
  memcpy(optional->value, input.value + 20, optional->size);
}
output->size = *total_byte - *header_byte;
output->value = (uint8_t*)malloc(output->size);
memcpy(output->value, input.value + *header_byte, output->size);
```

6. 运输层 UDP 协议

UDP 用户数据报格式为



6.1. UDP 用户数据报的封装

```
void send_udp(data_t const input, data_t* output, uint16_t const src_port,
              uint16_t const dest_port, uint32_t const src_addr,
              uint32_t const dest_addr) {
```

伪首部长度为 12 字节，用于计算校验和。

```
data_t tmp_header;
tmp_header.size = 12;
tmp_header.value = (uint8_t*)malloc(tmp_header.size);
```

填入 UDP 用户数据报首部及数据。

```
output->size = input.size + 8;
output->value = (uint8_t*)malloc(output->size);
*(uint16_t*)output->value = src_port;
*(uint16_t*)(output->value + 2) = dest_port;
*(uint16_t*)(output->value + 4) = 8 + input.size;
memcpy(output->value + 8, input.value, input.size);
```

填入伪首部内容，并计算校验和。

```
*(uint32_t*)tmp_header.value = src_addr;
*(uint32_t*)(tmp_header.value + 4) = dest_addr;
tmp_header.value[8] = 0X00;
tmp_header.value[9] = 0X11;
*(uint16_t*)(tmp_header.value + 10) = output->size;
*(uint16_t*)(output->value + 6) = checksum(*output, tmp_header);
```

其中，校验和的计算方法为：将校验和字段置 0；将首部、数据、伪首部按 16 位分段求和，且数据部分不满 16 位的部分用 0 填充；将结果中超出 16 位的部分加入校验和；校验和取反码。该校验和的计算方法与 IPv4 的首部校验和计算方法类似，但需要额外把伪首部和数据部分加入校验和。

```
uint16_t checksum(data_t const input, data_t const tmp_header) {
  uint32_t checksum_help = 0;
  for (uint8_t i = 0; i < input.size - 1; i += 2) {
    checksum_help += *(uint16_t*)(input.value + i);
  }
  if (input.size & 1) {
    checksum_help += (uint16_t)(input.value[input.size - 1]) << 8;
  } else {
    checksum_help += *(uint16_t*)(input.value + (input.size - 2));
  }
  for (uint8_t i = 0; i < tmp_header.size; i += 2) {
    checksum_help += *(uint16_t*)(tmp_header.value + i);
  }
  while (checksum_help > 0XFFFF) {
    checksum_help = (checksum_help >> 16) + (uint16_t)checksum_help;
  }
  uint16_t checksum;
  checksum = (uint16_t)checksum_help;
  checksum = ~checksum;
  return checksum;
}
```

最后，需要释放为伪首部分配的内存空间。

```
free(tmp_header.value);
```

6.2. UDP 用户数据报的解封装

```
void parse_udp(data_t const input, data_t* output, uint16_t* src_port,
               uint16_t* dest_port, uint32_t const src_addr,
               uint32_t const dest_addr) {
```

通过首部的长度字段验证数据长度是否正确。

```
  assert(input.size == *(uint16_t*)(input.value + 4));
```

设置伪首部内容，检验校验和是否正确。

```
  data_t tmp_header;
  tmp_header.size = 12;
  tmp_header.value = (uint8_t*)malloc(tmp_header.size);

  *(uint32_t*)tmp_header.value = src_addr;
  *(uint32_t*)(tmp_header.value + 4) = dest_addr;
  tmp_header.value[8] = 0X00;
  tmp_header.value[9] = 0X11;
  *(uint16_t*)(tmp_header.value + 10) = input.size;

  assert(0 == checksum(input, tmp_header));
```

解析用户数据报中的数据、源端口和目的端口。

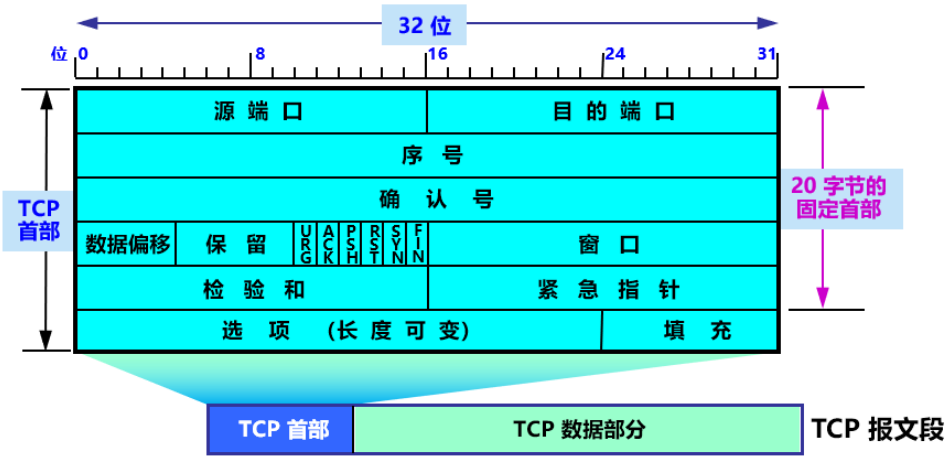
```
  output->size = input.size - 8;
  output->value = (uint8_t*)malloc(output->size);
  memcpy(output->value, input.value + 8, output->size);

  *src_port = *(uint16_t*)input.value;
  *dest_port = *(uint16_t*)(input.value + 2);
```

最后，需要是否为伪首部分配的内存空间。

```
  free(tmp_header.value);
```

7. 运输层 TCP 协议

TCP 报文段的格式为



7.1. TCP 报文段的封装

```
void send_tcp(data_t const input, data_t* output, uint16_t const src_port,
              uint16_t const dest_port, uint32_t const id,
              uint32_t const ack_value, uint8_t const URG, uint8_t const ACK,
              uint8_t const PSH, uint8_t const RST, uint8_t const SYN,
              uint8_t const FIN, uint16_t const window, uint16_t const urg_ptr,
              data_t const optional, uint32_t const src_addr,
              uint32_t const dest_addr) {
```

TCP 报文段中可选字段长度不超过 40 字节；URG、ACK、PSH、RST、SYN、FIN 字段仅有 1 比特。

```
assert(optional.size <= 40);
assert((URG & 0XFE) == 0 && (ACK & 0XFE) == 0 && (PSH & 0XFE) == 0 &&
       (RST & 0XFE) == 0 && (SYN & 0XFE) == 0 && (FIN & 0XFE) == 0);
```

可选字段长度不是 4 字节的整数倍时，需要填充至 4 字节的整数倍。由此可以确定完整的 TCP 报文段长度。

```
uint32_t filling_byte;

filling_byte = (4 - (optional.size & 0X3)) % 4;
uint8_t header_byte = 20 + optional.size + filling_byte;
output->size = header_byte + input.size;
output->value = (uint8_t*)malloc(output->size);
```

依次设置各字段的取值。其中超过 1 字节的通过强制类型转换赋值；不足 1 字节的通过位运算赋值。

```
*(uint16_t*)output->value = src_port;
*(uint16_t*)(output->value + 2) = dest_port;
*(uint32_t*)(output->value + 4) = id;
*(uint32_t*)(output->value + 8) = ack_value;
output->value[12] = header_byte >> 2;
output->value[13] = 0;
output->value[13] |= URG << 2;
output->value[13] |= ACK << 3;
output->value[13] |= PSH << 4;
output->value[13] |= RST << 5;
output->value[13] |= SYN << 6;
output->value[13] |= FIN << 7;
*(uint16_t*)(output->value + 14) = window;
*(uint16_t*)(output->value + 16) = 0;
*(uint16_t*)(output->value + 18) = urg_ptr;
memcpy(output->value + 20, optional.value, optional.size);
memset(output->value + 20 + optional.size, 0, filling_byte);
memcpy(output->value + header_byte, input.value, input.size);
```

创建伪首部，并计算校验和。校验和的计算方法与 UDP 用户数据报中校验和计算方法相同。

```
tmp_header.size = 12;
tmp_header.value = (uint8_t*)malloc(tmp_header.size);

*(uint32_t*)tmp_header.value = src_addr;
*(uint32_t*)(tmp_header.value + 4) = dest_addr;
tmp_header.value[8] = 0;
tmp_header.value[9] = 6;
*(uint16_t*)(tmp_header.value + 10) = output->size;
*(uint16_t*)(output->value + 16) = checksum(*output, tmp_header);

free(tmp_header.value);
```

7.2. TCP 报文段的解封装

```
void parse_tcp(data_t const input, data_t* output, uint16_t* src_port,
               uint16_t* dest_port, uint32_t* id, uint32_t* ack_value,
               uint8_t* offset_byte, uint8_t* URG, uint8_t* ACK, uint8_t* PSH,
               uint8_t* RST, uint8_t* SYN, uint8_t* FIN, uint16_t* window,
               uint16_t* urg_ptr, data_t* optional, uint32_t const src_addr,
               uint32_t const dest_addr) {
```

计算校验和是否正确。

```
data_t tmp_header;
tmp_header.size = 12;
tmp_header.value = (uint8_t*)malloc(tmp_header.size);

*(uint32_t*)tmp_header.value = src_addr;
*(uint32_t*)(tmp_header.value + 4) = dest_addr;
tmp_header.value[8] = 0;
tmp_header.value[9] = 6;
*(uint16_t*)(tmp_header.value + 10) = input.size;
assert(0 == checksum(input, tmp_header));
```

获取各字段和数据的取值。其中首部长度通过数据偏移字段计算得到。数据偏移字段以 4 字节为单位，为方便处理，需要转换为以字节为单位。

```
*offset_byte = input.value[12] << 2;
assert(*offset_byte <= 60);
*src_port = *(uint16_t*)input.value;
*dest_port = *(uint16_t*)(input.value + 2);
*id = *(uint32_t*)(input.value + 4);
*ack_value = *(uint32_t*)(input.value + 8);
*URG = (input.value[13] >> 2) & 1;
*ACK = (input.value[13] >> 3) & 1;
*PSH = (input.value[13] >> 4) & 1;
*RST = (input.value[13] >> 5) & 1;
*SYN = (input.value[13] >> 6) & 1;
*FIN = (input.value[13] >> 7) & 1;
*window = *(uint16_t*)(input.value + 14);
*urg_ptr = *(uint16_t*)(input.value + 18);
optional->size = *offset_byte - 20;
if (optional->size > 0) {
  optional->value = (uint8_t*)malloc(optional->size);
  memcpy(optional->value, input.value + 20, optional->size);
}
output->size = input.size - *offset_byte;
output->value = (uint8_t*)malloc(output->size);
memcpy(output->value, input.value + *offset_byte, output->size);
```

最后，需要是否为伪首部分配的内存空间。

```
free(tmp_header.value);
```

三、实验结果及分析

1. PPP 帧协议封装与解封装测试

测试程序为

```
int main(void) {
  data_t original_data = {.value =
                            "a simple test message for framing and parsing "
                            "of Point-to-Point Protocol",
                          .size = 73};
  PPP_PROTOCOL protocol = PPP_PROTOCOL_NCD;

  data_t ppp_frame;
  data_t ppp_frame_hex;

  data_t parsed_data;
  PPP_PROTOCOL parsed_protocal;

  send_ppp(original_data, &ppp_frame, protocol);
  parse_ppp(ppp_frame, &parsed_data, &parsed_protocal);

  char_to_hex(ppp_frame, &ppp_frame_hex);

  printf("Point-to-Point frame: %s\nSize: %d\n\n", ppp_frame_hex.value,
         ppp_frame.size);
  printf("Parsed data: %s\nSize: %d\nProtocol: %d\n\n", parsed_data.value,
         parsed_data.size, parsed_protocal);

  free(ppp_frame.value);
  free(ppp_frame_hex.value);
  free(parsed_data.value);
}
```

程序运行结果为

```
● xinkai@LAPTOP-777VF7EM:~/cpp$ gcc test/test_ppp.c
● xinkai@LAPTOP-777VF7EM:~/cpp$ ./a.out
  Point-to-Point frame: 7EFF032180612073696D706C652074657374206D6573736167652066F72206672616D696E6720616E6420
  70617273696E67206F6620506F696E742D746F2D506F696E742050726F746F636F6C0071B07E
  Size: 81

  Parsed data: a simple test message for framing and parsing of Point-to-Point Protocol
  Size: 73
  Protocol: 1
```

2. 字节填充与复原

测试程序为

```c
int main(void) {
    data_t original_data = {
        .value = "a simple test message for byte stuffing data", .size = 45};
    PPP_PROTOCOL protocol = PPP_PROTOCOL_NCD;

    data_t ppp_frame;
    data_t ppp_frame_hex;

    data_t ppp_stuffing;
    data_t parsed_stuffing;

    data_t parsed_data;
    PPP_PROTOCOL parsed_protocal;

    send_ppp(original_data, &ppp_frame, protocol);
    byte_stuffing(ppp_frame, &ppp_stuffing);
    parse_byte_stuffing(ppp_stuffing, &parsed_stuffing);
    parse_ppp(parsed_stuffing, &parsed_data, &parsed_protocal);

    char_to_hex(ppp_frame, &ppp_frame_hex);

    printf("Point-to-Point frame: %s\nSize: %d\n\n", ppp_frame_hex.value,
           ppp_frame.size);
    printf("Parsed data: %s\nSize: %d\nProtocol: %d\n\n", parsed_data.value,
           parsed_data.size, parsed_protocal);

    free(ppp_frame.value);
    free(ppp_frame_hex.value);
    free(parsed_data.value);
    free(ppp_stuffing.value);
    free(parsed_stuffing.value);
}
```

程序运行结果为

```
● xinkai@LAPTOP-777VF7EM:~/cpp$ gcc test/test_byte_stuffing.c
● xinkai@LAPTOP-777VF7EM:~/cpp$ ./a.out
  Point-to-Point frame: 7EFF032180612073696D706C652074657374206D6573736167652066F722062797465207374756666696E
  672064617461001A117E
  Size: 53

  Parsed data: a simple test message for byte stuffing data
  Size: 45
  Protocol: 1
```

3. 零比特填充与复原

测试程序为

```c
int main(void) {
  data_t original_data = {
      .value = "a simple test message for bit stuffing data", .size = 44};
  PPP_PROTOCOL protocol = PPP_PROTOCOL_NCD;

  data_t ppp_frame;
  data_t ppp_frame_hex;

  data_t ppp_stuffing;
  data_t parsed_stuffing;

  data_t parsed_data;
  PPP_PROTOCOL parsed_protocal;

  send_ppp(original_data, &ppp_frame, protocol);
  bit_stuffing(ppp_frame, &ppp_stuffing);
  printf("ppp_stuffing: %s\n", ppp_stuffing.value);
  parse_bit_stuffing(ppp_stuffing, &parsed_stuffing);
  parse_ppp(parsed_stuffing, &parsed_data, &parsed_protocal);

  char_to_hex(ppp_frame, &ppp_frame_hex);

  printf("Point-to-Point frame: %s\nSize: %d\n\n", ppp_frame_hex.value,
         ppp_frame.size);
  printf("Parsed data: %s\nSize: %d\nProtocol: %d\n\n", parsed_data.value,
         parsed_data.size, parsed_protocal);

  free(ppp_frame.value);
  free(ppp_frame_hex.value);
  free(parsed_data.value);
  free(ppp_stuffing.value);
  free(parsed_stuffing.value);
}
```

程序运行结果为

```
xinkai@LAPTOP-777VF7EM:~/cpp$ gcc test/test_bit_stuffing.c
xinkai@LAPTOP-777VF7EM:~/cpp$ ./a.out
ppp_stuffing: 0111111011111011100000011001000011000000001100001001000000111001101101001011011010111000001101
1000110010100100000011101000110010101110011011101000010000001101101011001010111001101110011011000010110011101
1100101001000000110011001101111011100100010000001100010011010010111010000100000011100110111010001110101011001
1100110011001101001011011100110011100100000011001000110000101110100011000010000000001011001101000000111111O
Point-to-Point frame: 7EFF032180612073696D706C652074657374206D657373616765206 66F72206269742073747566666696E67
20646174610059A07E
Size: 52

Parsed data: a simple test message for bit stuffing data
Size: 44
Protocol: 1
```

4. Ethernet II 的 MAC 帧封装与解封装

余下的测试程序与前面类似，将数据单元封装后解封装，若解封装后的数据以及各项配置均与输入相同，说明运行结果正确。

输入数据为

```c
data_t original_data = {.value =
                            "a simple test message for framing and parsing "
                            "of Ethernet v2 protocol",
                        .size = 70};
data_t dest_addr = {.value = "12345", .size = 6};
data_t src_addr = {.value = "67890", .size = 6};
ETHERNET_V2_TYPE type = ETHERNET_V2_TYPE_IPv4;
```

程序运行结果为

```
xinkai@LAPTOP-777VF7EM:~/cpp$ gcc test/test_ethernet_v2.c
xinkai@LAPTOP-777VF7EM:~/cpp$ ./a.out
Ethernet v2 Mac frame: 313233343500363738393000000861207369 6D706C652074657374206D6573736167652066 6F722066726
16D696E6720616E642070617273696E67206F662045746865726E65742076322070726F746F636F6C00AD0FA1F5
Size: 88

Parsed data: a simple test message for framing and parsing of Ethernet v2 protocol
Size: 70
Destination address: 12345
Source address: 67890
Type: 0
```

5. IP 数据报的封装与解封装

输入数据为

```
data_t original_data = {
    .value = "a simple test message for framing and parsing of IPv4 protocol",
    .size = 63};
uint8_t diff_serv = 1;
uint16_t id = 1;
uint8_t MF = 1;
uint8_t DF = 0;
uint16_t offset = 24;
uint8_t TTL = 4;
IPv4_PROTOCOL_TYPE protocol = IPv4_PROTOCOL_TYPE_OSPF;
uint32_t src_addr = 0X12345678;
uint32_t dest_addr = 0X87654321;
data_t optional = {.value = "A", .size = 2};
```

程序运行结果为

```
xinkai@LAPTOP-777VF7EM:~/cpp$ gcc test/test_ip.c
xinkai@LAPTOP-777VF7EM:~/cpp$ ./a.out
IPv4 datagram: 64015700010019000459B2717856341221436587410000006120 73696D706C652074657374206D657373616765206
66F72206672616D696E6720616E642070617273696E67206F662049507634207072 6F746F636F6C00
Size: 87

Parsed data: a simple test message for framing and parsing of IPv4 protocol
Header byte: 24
DiffServ: 1
Total byte: 87
ID: 1
MF: 1
DF: 0
Offset byte: 24
TTL: 4
Protocol: 11
Source address: 12345678
Destination address: 87654321
Optional: A
```

6. UDP 用户数据报的封装与解封装

输入数据为

```
data_t original_data = {
    .value = "a simple test message for framing and parsing of UDP datagram",
    .size = 62};
uint32_t src_addr = 0X12345678;
uint32_t dest_addr = 0X87654321;
uint16_t src_port = 0X4919;
uint16_t dest_port = 0X1234;
```

程序运行结果为

```
xinkai@LAPTOP-777VF7EM:~/cpp$ gcc test/test_udp.c
xinkai@LAPTOP-777VF7EM:~/cpp$ ./a.out
UDP datagram: 194934124600B5F8612073696D706C652074657374206D6573736167652066F72206672616D696E6720616E642070
617273696E67206F662065544502064617461616772616D00
Size: 70

Parsed data: a simple test message for framing and parsing of UDP datagram
Size: 62
Source port: 4919
Destination port: 1234
```

7. TCP 报文段的封装与解封装

输入数据为

```
data_t original_data = {
    .value = "a simple test message for framing and parsing of TCP datagram",
    .size = 62};
uint32_t src_addr = 0X12345678;
uint32_t dest_addr = 0X87654321;
uint16_t src_port = 0X4919;
uint16_t dest_port = 0X1234;
uint32_t id = 0X23333333;
uint32_t ack_value = 0X20000000;
uint8_t URG = 0;
uint8_t ACK = 1;
uint8_t PSH = 0;
uint8_t RST = 1;
uint8_t SYN = 0;
uint8_t FIN = 1;
uint16_t window = 2048;
uint16_t urg_ptr = 0;
data_t optional = {.value = "some optional data", .size = 19};
```

程序运行结果为

```
xinkai@LAPTOP-777VF7EM:~/cpp$ gcc test/test_tcp.c
xinkai@LAPTOP-777VF7EM:~/cpp$ ./a.out
TCP datagram: 194934123333332300000020 0AA80008DD710000736F6D65206F7074696F6E616C20646174610000612073696D706C
652074657374206D6573736167652066F72206672616D696E6720616E642070617273696E67206F662065544350206461746167726 16D
00
Size: 102

Parsed data: a simple test message for framing and parsing of TCP datagram
Size: 62
Source port: 4919
Destination port: 1234
ID: 23333333
ACK value: 20000000
Header byte: 40
URG: 0
ACK: 1
PSH: 0
RST: 1
SYN: 0
FIN: 1
Window: 2048
URG pointer: 0
Optional data: some optional data
```

8. 数据链路层、网络层、运输层组合数据传输

数据封装进入 UDP 用户数据报、IP 数据报、Ethernet II 的 MAC 帧并解封装。输入数据为

```
data_t original_data = {
    .value =
        "a simple test message which is \aprocessed into \nUDP datagram, "
        "\nIPv4 datagram, \tEthernet v2.\nAfter all the works, \bit will be "
        "parsed into its original shape.\nHope a good luck.\n\n",
    .size = 177};

uint16_t src_port = 0X1234;
uint16_t dest_port = 0X4919;
uint32_t src_ip_addr = 0X12345678;
uint32_t dest_ip_addr = 0X87654321;

uint8_t diff_serv = 1;
uint16_t ip_id = 1;
uint8_t MF = 1;
uint8_t DF = 0;
uint16_t ip_offset_byte = 24;
uint8_t TTL = 4;
IPv4_PROTOCOL_TYPE ip_protocol = IPv4_PROTOCOL_TYPE_UDP;

data_t src_mac_addr = {.value = "012345", .size = 6};
data_t dest_mac_addr = {.value = "FEDCBA", .size = 6};
ETHERNET_V2_TYPE type = ETHERNET_V2_TYPE_PPPoE;
```

处理过程为

```
send_udp(original_data, &udp_datagram, src_port, dest_port, src_ip_addr,
         dest_ip_addr);
send_ip_v4(udp_datagram, &ip_datagram, diff_serv, ip_id, MF, DF,
           ip_offset_byte, TTL, ip_protocol, src_ip_addr, dest_ip_addr,
           optional);
send_ppp(ip_datagram, &ppp_frame, ppp_protocol);
send_ethernet_v2(ppp_frame, &mac_frame, dest_mac_addr, src_mac_addr, type);
parse_ethernet_v2(mac_frame, &parsed_mac_frame, &parsed_dest_mac_addr,
                  &parsed_src_mac_addr, &parsed_type);
parse_ppp(parsed_mac_frame, &parsed_ppp_frame, &parsed_ppp_protocol);
parse_ip_v4(parsed_ppp_frame, &parsed_ip_datagram, &parsed_version,
            &parsed_header_byte, &parsed_diff_serv, &parsed_total_byte,
            &parsed_ip_id, &parsed_MF, &parsed_DF, &parsed_ip_offset_byte,
            &parsed_TTL, &parsed_protocol, &parsed_src_ip_addr,
            &parsed_dest_ip_addr, &parsed_optional);
parse_udp(parsed_ip_datagram, &parsed_udp_datagram, &parsed_src_port,
          &parsed_dest_port, parsed_src_ip_addr, parsed_dest_ip_addr);
```

程序运行结果为

```
Parsed data: a simple test message which is processed into
UDP datagram,
IPv4 datagram,  Ethernet v2.
After all the works,it will be parsed into its original shape.
Hope a good luck.


Size: 177

Ethernet v2...
Source Mac address: 012345
Destination Mac address: FEDCBA
Type: 2

PPP...
Protocol: 0

IP...
Version: 4
Header byte: 48
DiffServ: 1
Total byte: 233
ID: 0
MF: 1
DF: 0
Offset byte: 0
TTL: 4
Protocol: 6
Source IP address: 12345678
Destination IP address: 87654321
Optional data: Optional configurations.

UDP...
Source port: 1234
Destination port: 4919
```

数据封装进入 TCP 报文段、IP 数据报、PPP 帧，经过零比特填充，复原后解封装。输入数据为

```
original_data.value =
    "another test message which\t is processed into \aTCP datagram,\n IPv4 "
    "datagram, PPP frame\n and stuffed by zero-\bbit. After those works,\n "
    "it will be parsed into its\n\t original shape. Hope a good luck.\n\n";
original_data.size = 199;
```

```
uint8_t diff_serv = 1;
uint16_t ip_id = 1;
uint8_t MF = 1;
uint8_t DF = 0;
uint16_t ip_offset_byte = 24;
uint8_t TTL = 4;
IPv4_PROTOCOL_TYPE ip_protocol = IPv4_PROTOCOL_TYPE_UDP;

uint32_t tcp_id = 0X23333333;
uint32_t ack_value = 0X20000000;
uint8_t URG = 0;
uint8_t ACK = 1;
uint8_t PSH = 0;
uint8_t RST = 0;
uint8_t SYN = 0;
uint8_t FIN = 1;
uint16_t window = 64;
uint16_t urg_ptr = 0;
data_t optional = {.value = "Optional configurations.", .size = 25};

PPP_PROTOCOL ppp_protocol = PPP_PROTOCOL_IPD;
```

处理过程为

```
send_tcp(original_data, &tcp_datagram, src_port, dest_port, tcp_id, ack_value,
         URG, ACK, PSH, RST, SYN, FIN, window, urg_ptr, optional, src_ip_addr,
         dest_ip_addr);
send_ip_v4(tcp_datagram, &ip_datagram, diff_serv, ip_id, MF, DF,
           ip_offset_byte, TTL, ip_protocol, src_ip_addr, dest_ip_addr,
           optional);
send_ppp(ip_datagram, &ppp_frame, ppp_protocol);
bit_stuffing(ppp_frame, &bit_stuffing_stream);
parse_bit_stuffing(bit_stuffing_stream, &parsed_bit_stuffing_stream);
parse_ppp(parsed_bit_stuffing_stream, &parsed_ppp_frame,
          &parsed_ppp_protocol);
parse_ip_v4(parsed_ppp_frame, &parsed_ip_datagram, &parsed_version,
            &parsed_header_byte, &parsed_diff_serv, &parsed_total_byte,
            &parsed_ip_id, &parsed_MF, &parsed_DF, &parsed_ip_offset_byte,
            &parsed_TTL, &parsed_protocol, &parsed_src_ip_addr,
            &parsed_dest_ip_addr, &parsed_optional);
parse_tcp(parsed_ip_datagram, &parsed_tcp_datagram, &parsed_src_port,
          &parsed_dest_port, &parsed_tcp_id, &parsed_ack_value,
          &parsed_tcp_offset_byte, &parsed_URG, &parsed_ACK, &parsed_PSH,
          &parsed_RST, &parsed_SYN, &parsed_FIN, &parsed_window,
          &parsed_urg_ptr, &parsed_optional, src_ip_addr, dest_ip_addr);
```

程序运行结果为

```
Parsed data: another test message which  is processed into TCP datagram,
 IPv4 datagram, PPP frame
 and stuffed by zerobit. After those works,
 it will be parsed into its
        original shape. Hope a good luck.


Size: 199

PPP...
Protocol: 0

IP...
Version: 4
Header byte: 48
DiffServ: 1
Total byte: 295
ID: 590558003
MF: 1
DF: 0
Offset byte: 48
TTL: 4
Protocol: 6
Source IP address: 12345678
Destination IP address: 87654321
Optional data: Optional configurations.

TCP...
Source port: 1234
Destination port: 4919
ID: 23333333
ACK value: 20000000
Header byte: 48
URG: 0
ACK: 1
PSH: 0
RST: 0
SYN: 0
FIN: 1
Window: 64
URG pointer: 0
Optional data: Optional configurations.
```

程序运行结果显示各协议数据单元的封装与解封装、PPP 帧的零比特填充和字节填充均能按预期效果执行。

四、分工说明及自我总结

1. 成员分工

成员	分工
李宽宇	实现各协议数据单元的解封装、实现字节数组与二进制字符串、十六进制字符串转换的工具函数等。
胡欣凯	实现各协议数据单元的封装、各协议封装及解封装测试、代码风格统一整理等。

2. 自我总结