重庆邮电大学

学生实验实习报告册

学年学期：2025-2026 学年（1）学期

课程名称：通信软件开发与应用

学生学院：通信与信息工程学院

专业班级：01042301

学生学号：2023211281

学生姓名：丁同勖

联系电话：18019582857

重庆邮电大学教务处制

课程名称	通信软件开发与应用			**课程编号**	A2012230		
实验地点	YF315			**实验时间**	9 周 9-11 节		
校外指导教师	无			**校内指导教师**	梁燕		
实验名称	Ethernet 帧结构解析程序设计与实现						
评阅人签字		**操作成绩 80%**		**报告成绩 20%**		**总评成绩**	

一、实验目的

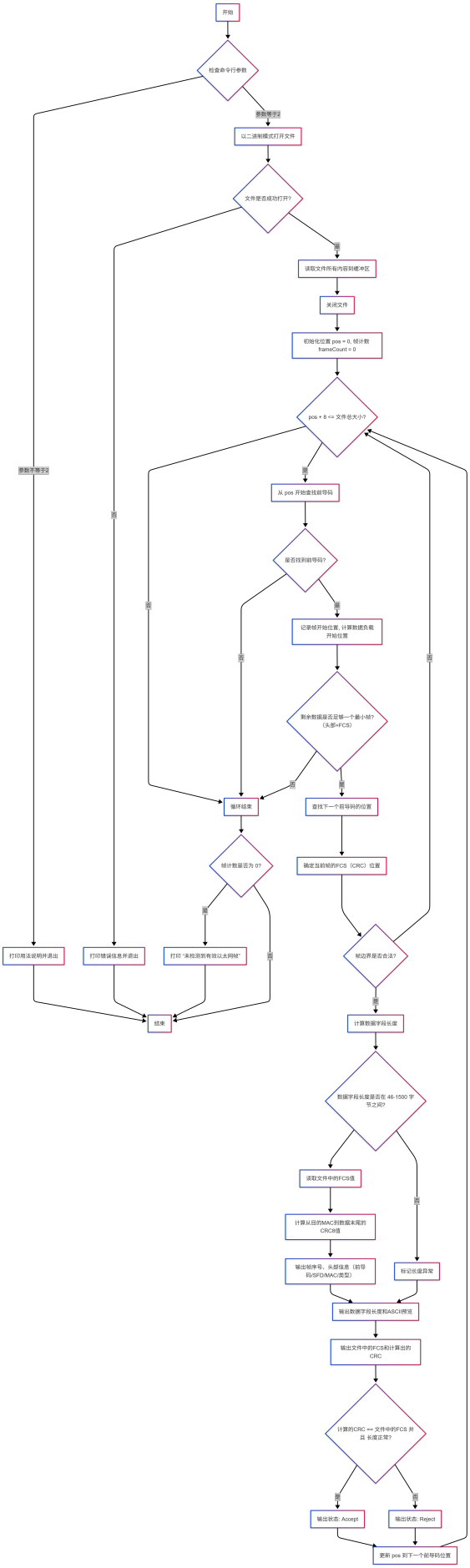
1、掌握以太网帧结构
2、掌握以太网帧同步方法
3、掌握 CRC 校验算法原理及编程实现方法
4、掌握程序编辑、编译、链接、调试等基本概念
5、掌握程序基本排错方法及断点调试技巧

二、实验思路

1. 启动与参数检查:
   - 程序启动后，首先检查命令行参数的数量。
   - 如果参数不为 2（程序名 + 文件路径），则打印程序的正确用法并退出。
2. 文件读取:
   - 如果参数正确，程序尝试以二进制模式打开用户提供的文件。
   - 如果文件打开失败（例如，文件不存在或没有权限），则打印错误信息并退出。
   - 文件成功打开后，程序会一次性将文件的全部二进制内容读取到一个 vector<unsigned char> 缓冲区中，然后关闭文件。
3. 循环解析数据帧:
   - 程序进入一个主循环，从缓冲区(vector)的起始位置开始扫描，直到剩余数据不足以构成一个最小的帧。
   - 查找前导码: 在循环中，它首先寻找以太网帧的起始标记——7 个 0xAA 字节加上 1 个 0xAB（帧前定界符 SFD）组成的 8 字节前导码。
   - 如果找不到前导码，则认为文件中没有更多数据帧，循环结束。
4. 确定帧边界:
   - 定位当前帧: 找到一个前导码后，程序就以此为当前帧的起点。
   - 寻找下一帧: 为了确定当前帧的结束位置，程序会从当前帧的头部之后开始，继续搜索下一个前导码。
   - 确定 FCS 位置:
     - 如果找到了下一个前导码，那么当前帧的最后一个字节（即帧校验序列 FCS 或 CRC）就在下一个前导码的前一个字节。
     - 如果找不到下一个前导码，说明这是文件中的最后一帧，那么文件的最后一个字节就被当作是 FCS。
   - 合法性检查: 程序会进行简单的边界检查，确保根据找到的边界计算出的帧长度是合理的，否则会跳过当前位置，从下一个可能的位置继续搜索。
5. 帧内容解析与校验:
   - 提取头部信息: 程序从前导码之后的位置开始解析，依次提取出：目的 MAC 地址、源 MAC 地址和类型字段。
   - 计算数据长度: 通过 FCS 的位置和头部长度，计算出数据字段（Payload）的实际长度。

- 长度校验: 程序会检查数据字段的长度是否在以太网规定的有效范围内（通常是 46 到 1500 字节）。如果不在，会标记一个“长度异常”。
- CRC 校验:
  - 程序读取帧末尾的 FCS 字节。
  - 然后，它对从“目的 MAC 地址”开始到“数据字段”结束的整个区段，使用 calcCRC8 函数计算出一个 CRC8 校验码。
  - 最后，将计算出的 CRC 与从文件中读取的 FCS 进行比较。

6. 输出结果:
   - 对于每一个找到的潜在数据帧，程序会打印以下信息：
     - 一个自增的序号。
     - 前导码和帧前定界符。
     - 格式化后的目的和源 MAC 地址。
     - 类型字段的十六进制值。
     - 数据字段的长度，并在长度异常时给出提示。
     - 数据字段内容的 ASCII 表示（非打印字符显示为 .）。
     - 文件中的 FCS 值和程序自己计算出的 CRC 值。
     - 最终状态: 如果计算的 CRC 与文件中的 FCS 匹配 并且 数据字段长度在有效范围内，则状态为 "Accept"（接收）；否则为 "Reject"（拒绝）。
7. 循环与结束:
   - 处理完一帧后，程序将搜索位置 pos 更新到下一帧的前导码处，继续循环，直到文件末尾。
   - 如果整个文件扫描完毕，一帧都未成功解析（frameCount 为 0），则打印一条提示信息。
   - 程序正常结束。


开始
检查命令行参数
参数不等于2
参数等于2
以二进制模式打开文件
文件是否成功打开?
否
是
读取文件所有内容到缓冲区
关闭文件
初始化位置 pos = 0, 帧计数 frameCount = 0
pos + 8 <= 文件总大小?
否
是
从 pos 开始查找前导码
是否找到前导码?
否
是
记录帧开始位置, 计算数据负载开始位置
剩余数据是否足够一个最小帧? (头部+FCS)
否
是
循环结束
查找下一个前导码的位置
帧计数是否为 0?
否
是
确定当前帧的FCS（CRC）位置
帧边界是否合法?
否
是
打印用法说明并退出
打印错误信息并退出
打印 "未检测到有效以太网帧"
结束
计算数据字段长度
数据字段长度是否在 46-1500 字节之间?
否
是
读取文件中的FCS值
计算从目的MAC到数据末尾的CRC8值
输出帧序号、头部信息（前导码/SFD/MAC/类型）
标记长度异常
输出数据字段长度和ASCII预览
输出文件中的FCS和计算出的CRC
计算的CRC == 文件中的FCS 并且 长度正常?
否
是
输出状态: Reject
输出状态: Accept
更新 pos 到下一个前导码位置

三、源代码

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <sstream>
#include <vector>
#include <string>
#include <cstdint>
#include <cstring>
#include <array>
#include <cctype>
#include <iterator>

using namespace std;

// ------------------ 工具函数 ------------------

/**
 * @brief 计算CRC8校验码
 * @param data 数据指针
 * @param len 数据长度
 * @return 8位的CRC校验码
 */
uint8_t calcCRC8(const unsigned char* data, size_t len) {
    uint8_t crc = 0x00;
    const uint8_t poly = 0x07;
    for (size_t i = 0; i < len; ++i) {
        crc ^= data[i];
        for (int b = 0; b < 8; ++b) {
            if (crc & 0x80) {
                crc = (uint8_t)((crc << 1) ^ poly);
            }
            else {
                crc = (uint8_t)(crc << 1);
            }
        }
    }
    return crc;
}

/**
 * @brief 将单个字节转换为两位十六进制字符串
 * @param b 字节
 * @return 十六进制表示的字符串
 */
string byteToHex(unsigned char b) {
```

```
    stringstream ss;
    ss << uppercase << hex << setw(2) << setfill('0') << (int)b;
    return ss.str();
}

/**
 * @brief 将6字节的MAC地址格式化为 "XX-XX-XX-XX-XX-XX"
 * @param data 包含MAC地址的字节向量
 * @param start MAC地址在向量中的起始索引
 * @return 格式化后的MAC地址字符串
 */
string macToStr(const vector<unsigned char>& data, int start) {
    stringstream ss;
    for (int i = 0; i < 6; i++) {
        ss << byteToHex(data[start + i]);
        if (i != 5) {
            ss << "-";
        }
    }
    return ss.str();
}

/**
 * @brief 将6字节的MAC地址格式化为 "XX-XX-XX-XX-XX-XX"
 * @param data 指向MAC地址数据的指针
 * @return 格式化后的MAC地址字符串
 */
string macToStr(const unsigned char* data) {
    stringstream ss;
    for (int i = 0; i < 6; i++) {
        ss << uppercase << hex << setw(2) << setfill('0') << (int)data[i];
        if (i != 5) {
            ss << "-";
        }
    }
    ss << dec; // 恢复为十进制流
    return ss.str();
}

/**
 * @brief 将字节数据格式化为ASCII字符串，不可打印字符显示为'.'
 * @param data 数据指针
 * @param len 数据长度
 * @return ASCII表示的字符串
 */
string formatDataAscii(const unsigned char* data, size_t len) {
```

```
    stringstream ss;
    for (size_t i = 0; i < len; ++i) {
        unsigned char c = data[i];
        ss << (isprint(c) ? (char)c : '.');
    }
    return ss.str();
}

// ------------------ 主函数 ------------------
int main(int argc, char* argv[]) {
    if (argc != 2) {
        cout << "用法: " << argv[0] << " [数据帧文件路径]" << endl;
        cout << "示例: Ethernet_Analyzer.exe input" << endl;
        return 0;
    }

    ifstream fin(argv[1], ios::binary);
    if (!fin) {
        cerr << "无法打开文件: " << argv[1] << endl;
        return 1;
    }

    vector<unsigned char> data((istreambuf_iterator<char>(fin)), istreambuf_iterator<char>());
    fin.close();

    size_t pos = 0;
    int frameCount = 0;
    const array<unsigned char, 8> preamble = { {0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAB} };
    const size_t total = data.size();

    while (pos + 8 <= total) {
        // 1. 查找前导码 (7 x 0xAA + 0xAB)
        bool found = false;
        while (pos + 8 <= total) {
            if (memcmp(&data[pos], preamble.data(), preamble.size()) == 0) {
                found = true;
                break;
            }
            ++pos;
        }
        if (!found) {
            break; // 文件末尾未找到完整的前导码
        }

        size_t frameStart = pos; // 指向第一个0xAA
        size_t payloadStart = frameStart + 8; // 跳过7xAA + SFD，指向目的MAC地址
```

```
        // 至少需要 14字节头部 + 1字节FCS
        if (payloadStart + 14 + 1 > total) {
            break;
        }

        // 2. 通过查找下一个前导码来确定当前帧的边界
        size_t nextPreamble = total;
        for (size_t j = payloadStart + 1; j + 8 <= total; ++j) {
            if (memcmp(&data[j], preamble.data(), preamble.size()) == 0) {
                nextPreamble = j;
                break;
            }
        }

        // 3. 确定CRC(FCS)的位置
        size_t crcOffset = 0; // FCS的起始位置 (1 byte)
        if (nextPreamble == total) {
            // 这是文件中的最后一帧
            if (total < 1) break;
            crcOffset = total - 1;
        }
        else {
            // 检查下一帧是否离得太近
            if (nextPreamble < 1 || nextPreamble < payloadStart + 14 + 1) {
                pos = nextPreamble; // 当前帧不合法，跳到下一帧继续
                continue;
            }
            crcOffset = nextPreamble - 1;
        }

        size_t headerOffset = payloadStart;
        size_t dataStart = headerOffset + 14;

        if (crcOffset < dataStart) { // 确保数据段长度至少为0
            pos = (nextPreamble == total) ? total : nextPreamble;
            continue;
        }
        size_t dataLen = crcOffset - dataStart;

        // 新增：对数据字段长度进行检查
        const size_t ETH_MIN_PAYLOAD = 46; //以太网最小有效载荷
        const size_t ETH_MAX_PAYLOAD = 1500; //以太网最大有效载荷(不考虑Jumbo帧)
        bool length_ok = true;
        if (dataLen < ETH_MIN_PAYLOAD) {
            length_ok = false;
```

```
        }
        if (dataLen > ETH_MAX_PAYLOAD) {
            length_ok = false;
        }

        // 4. 读取文件中的FCS并计算CRC
        if (crcOffset + 1 > total) {
            break;
        }
        uint8_t fcs = data[crcOffset];

        // CRC8计算范围：从目的MAC地址到数据字段末尾（不包括FCS）
        size_t crcCalcLen = crcOffset - payloadStart;
        uint8_t calc = calcCRC8(&data[payloadStart], crcCalcLen);

        // 5. 输出解析结果
        ++frameCount;
        cout << "序号: " << setw(2) << setfill('0') << frameCount << endl;
        cout << "----------------------------------------" << endl;
        cout << "前导码:     ";
        for (int i = 0; i < 7; ++i) {
            cout << byteToHex(data[frameStart + i]) << " ";
        }
        cout << endl;
        cout << "帧前定界符: " << byteToHex(data[frameStart + 7]) << endl;

        cout << "目的地址:   " << macToStr(data, headerOffset) << endl;
        cout << "源地址:     " << macToStr(data, headerOffset + 6) << endl;

        // 类型字段（网络字节序）
        unsigned int et = ((unsigned int)data[headerOffset + 12] << 8) | data[headerOffset + 13];
        stringstream ss;
        ss << "0x" << hex << uppercase << setw(4) << setfill('0') << et << dec;
        cout << "类型字段:   " << ss.str() << endl;

        cout << "数据字段长度: " << dec << dataLen << " 字节" << endl;
        if (!length_ok) {
            if (dataLen < ETH_MIN_PAYLOAD) {
                cout << "长度异常: 数据字段长度过短（小于 " << ETH_MIN_PAYLOAD << " 字节），可能为截断帧或错误帧。" << endl;
            } else {
                cout << "长度异常: 数据字段长度过长（大于 " << ETH_MAX_PAYLOAD << " 字节），可能包含额外数据或错误。" << endl;
            }
        }
```

```
        cout << "数据字段(ASCII): " << formatDataAscii(&data[dataStart], dataLen) << endl;

        // 保存当前cout状态，以便在打印十六进制后恢复
        ios oldState(nullptr);
        oldState.copyfmt(cout);
        cout << "CRC校验(文件): 0x" << hex << uppercase << setw(2) << setfill('0') << (int)fcs <<
endl;
        cout << "CRC校验(计算): 0x" << hex << uppercase << setw(2) << setfill('0') << (int)calc <<
endl;
        cout.copyfmt(oldState); // 恢复cout状态

        // 状态：当且仅当CRC匹配且长度正常时接收
        cout << "状态:       " << ((fcs == calc && length_ok) ? "Accept" : "Reject") << endl;
        cout << "----------------------------------------" << endl << endl;

        // 从下一个前导码的位置继续搜索
        pos = (nextPreamble == total) ? total : nextPreamble;
    }

    if (frameCount == 0) {
        cout << "未检测到有效以太网帧。" << endl;
    }

    return 0;
}
```

四、实验结果及分析

五、实验中问题及分析

一开始写的 Ethernet 帧结构解析使用的是 CRC-32 的校验，结果不正确，观察后发现实验中使用的是 CRC-8，修改后输出结果正确。

六、心得体会

通过本次编写和调试以太网帧分析器的实验，我深入理解了以太网帧的理论结构，并将其成功应用于实际的编程实践中。这不仅锻炼了我的 C++编程能力，尤其是在处理二进制数据流方面，更让我对网络底层协议的复杂性和严谨性有了全新的认识。

(一) 对理论知识的深化理解

在动手编程之前，我对以太网帧的认识停留在书本上的结构图：前导码 -> 目的 MAC -> 源 MAC -> 类型 -> 数据 -> FCS。而这次实验让我不得不思考这些字段在真实的、连续的二进制数据流中是如何体现的。

1. 帧边界的确定是核心难点：

   理论上每个帧都是独立的，但在一个文件中，它们是紧密相连的字节流。程序的核心挑战在于如何准确地找到一帧的开始和结束。代码中采用的**“通过查找下一个前导码来确定当前帧的边界”**是一种非常巧妙且实用的策略。这让我认识到，在处理无明确分隔符的数据流时，利用数据本身的特征（如固定的前导码）来同步和定位是关键。

2. FCS 校验的实践意义：

   我亲手实现了 CRC8 校验算法，并通过位运算（异或、移位）来模拟硬件层面的计算过程。当看到自己计算出的 CRC 值与文件中提取的 FCS 值完全一致时，我才真正体会到数据校验在保证网络通信可靠性中的重要作用。这也让我明白了为什么一个看似“错误”的帧（CRC 不匹配或长度异常）也需要被解析和报告出来，因为这本身就是网络诊断的重要信息。

(二) C++编程技能的提升

本次实验也是对 C++编程技能的一次综合运用和提升。

1. 二进制文件 I/O 与数据处理：我学习了如何使用 ifstream 以 ios::binary 模式打开文件，并通过 istreambuf_iterator 将整个文件高效地读入一个 vector<unsigned char> 容器。vector 作为动态缓冲区，在处理大小未知的文件时非常灵活和安全。
2. 内存操作与性能：在查找前导码时，使用 memcmp 函数进行内存块的直接比较，远比自己写循环逐字节判断要高效。这让我体会到，在处理大块数据时，优先选用标准库提供的内存操作函数的重要性。
3. 数据格式化与输出：为了将二进制数据以人类可读的形式（如 XX-XX-XX 格式的 MAC 地址，0xABCD 格式的类型字段）展示出来，我大量使用了 stringstream 和 <iomanip> 中的工具（如 setw, setfill, hex）。这极大地锻炼了我构造复杂输出格式的能力，也让我认识到在恢复流状态（如从 hex 恢复到 dec）时保持代码整洁的重要性。

(三) 设计思路与问题解决

程序的设计并非一帆风顺，其中充满了对边界条件和异常情况的处理。

1. 健壮性设计：程序不仅要能处理“完美”的数据帧，更要能应对各种异常。例如：
    - 文件末尾的数据不足以构成一个完整的帧。
    - 数据帧被截断，导致数据段长度过短。
    - 两个前导码之间距离过近，不满足最小帧长的要求。
    - 文件中根本不包含任何有效的前导码。

    代码中的多处 if 判断（如 pos + 8 <= total）和逻辑（如处理最后一帧的特殊逻辑）正是为了保证程序的健壮性，避免越界访问等运行时错误。
2. 从“假设”到“验证”：最初的思路可能是简单地按固定长度去切分数据，但很快就会发现这是行不通的。最终的方案是一个不断“搜索-验证”的循环：先找到一个疑似帧的起点（前导码），然后根据规则去验证它的结构是否合法，最后才进行解析和输出。这种从“大胆假设”到“小心求证”的逻辑，是处理复杂数据解析任务的通用方法。

(四) 程序的局限性与改进方向

通过本次实验，我也看到了当前程序的局限性和未来可以改进的方向：

1. CRC 算法的简化：为了简化，程序使用了 CRC8，而真实的以太网 V2 帧使用的是 CRC32，算法更复杂但校验能力更强。未来可以尝试实现标准的 CRC32 算法。
2. 协议解析的深度有限：目前程序只解析到以太网头部（数据链路层）。一个自然的扩展是根据“类型”字段（如 0x0800）进一步解析上层协议，如 IP 头部、TCP/UDP 头部等，实现一个更完整的协议分析器。
3. 性能优化：对于非常大的数据文件（如 GB 级别），一次性将文件全部读入内存可能会导致内存耗尽。未来可以优化为基于流的、分块读取和处理的模式。

(五) 总结

总而言之，这次实验是一次理论与实践相结合的宝贵经历。它不仅让我将网络协议的抽象知识具象化，还全方位地锻炼了我的 C++编程能力、逻辑思维和问题解决能力。通过亲手打造一个能“看懂”网络数据帧的工具，我对计算机网络底层的工作原理有了前所未有的深刻理解。