武汉大学计算机学院

本科生课程报告

课　　头　号　：20192021900

课 程 名 称　：多媒体技术

学　　　号　：2017301060030

姓　　　名　：王　虎　林

二〇二〇年六月

郑 重 声 明

本人呈交的课程报告，是在导师的指导下，独立进行研究工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本报告的研究成果不包含他人享有著作权的内容。对本报告所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确的方式标明。

本人签名：王虎林　　日期：2020/6/7

目录

1 选题目的与意义

1.1 选题目的

该实验是对 MPEG 音频第三层解码算法的研究与软件实现。介绍了 MP3 音频压缩使用的几个关键技术:混合滤波器组、心里声学模型、自适应窗口切换技术、哈夫曼编码技术、弹性比特存储技术并剖析了 MP3 的帧结构。重点研究了解码算法，在实现过程中对音频解码(44.1KHz 采样频率，压缩到 64Kbit/s 每通道的编码速率，双声道)采用 MPEG- I 的标准，解码过程包括数据流同步及帧头、边信息的提取、主数据的提取、哈夫曼解码、反量化、立体声处理、重排序、混叠重建、逆向离散余弦变化、频率反转和子带合成几个步骤。

对 MP3 文件进行解码，将数字视音频数据流解码还原成模拟视音频信号的硬件/软件设备。在本次实验中，要求将 MP3 文件格式转换成 PCM 文件格式（脉冲编码调制信号）。并且转换后的 PCM 文件能够正确播放

1.2 选题意义

熟悉音频解码过程，了解音频数据格式，深化课本内容，加强自身能力。

1.3 选题来源

本实验选题来自于百度文库的一篇文档《Mp3 解码流程》(https://wenku.baidu.com/view/9225d40627284b73f242508d.html?fr=search)，参考了其中对于 MP3 文件解码的解码原理与解码方法。其解码流程的后几步(主要是套公式的)主要是在参考项目的开源代码上稍加改进实现的，其余流程均由本人通过阅读参考开源代码后独立实现。

2 实验原理与方案

2.1 MPEG-1 音频标准概述

在 MPEG- I 音频压缩算法标准中提供了以下模式:

1.音频信号采样率可以是 32KHz，44. 1KHz 或 48KHz。

2.压缩后的比特流可以按照单声道、独立的两个声道、立体声、联合立体声等四种模式之一支持单声道或双声道编码。其中，联合立体声模式是利用了立体声通道之间的关联和通道之间相位差的无关性得到的。

3.压缩后的比特率可以从 32Kbit/s 到 224Kbit/s (每声道),也可以使用用户定义的比特率。

4. MPEG 音频编码标准提供了三个独立的压缩层次，使用户可以在复杂性和压缩质量之间权衡选择。层 I 最简单，最适于使用的比特率为 128Kbit/s (每声道)以上。层 II 的复杂度中等，使用的比特率为 128Kbit/s (每声道)左右，其应用包括:数字广播，CD-ROM. 上的声音信号以及 CD-I 和 VCD.层功即我们通常所说的 MP3 (MPEG--I (or II) Audio LayerⅢ)，它最复杂，但音质最佳，比特率为 64Kbit/s (每声道)左右。尤其适用 ISDN.上的声音传输，本文将主要讨论这一层的解码算法。

5.编码后的比特流支持 CRC 校验。

6.MPEG 音频编码标准还支持比特流中附带附加信息。

对于 MPEG- II,增加了低采样频率的编码标准,采样频率可以扩展到 16 KHz、22. 05 KHz、24 KHz。同时压缩后的比特率可以低到 8KHz (每声道)，这有利于实现音质要求不高的现场语音录音。

2.2 MP3 解码流程概述

Mp3 的解码总体上可分为 9 个过程：比特流分解，霍夫曼解码，逆量化处理，立体声处理，频谱重排列，抗锯齿处理，IMDCT 变换，子带合成，pcm 输

出。在进行 MP3 解码时，首先要检测数据流中的同步字来正确确定一帧数据的开始，提取帧头信息，从而得到相应的解码参数，同时分离边信息和主数据。通过对边信息数据解码可得到哈夫曼解码信息和反量化信息，主数据就可以根据哈夫曼解码信息解码出量化之后的数据，量化后数据结合反量化信息就可以得到频域中的数据流。结合帧头中的立体声信息，对反量化结果进行立体声处理后，通过反混叠处理、反离散余弦变换、合成滤波器处理就可以得到原始的 PCM 音频信号。其处理流程图大致如下：

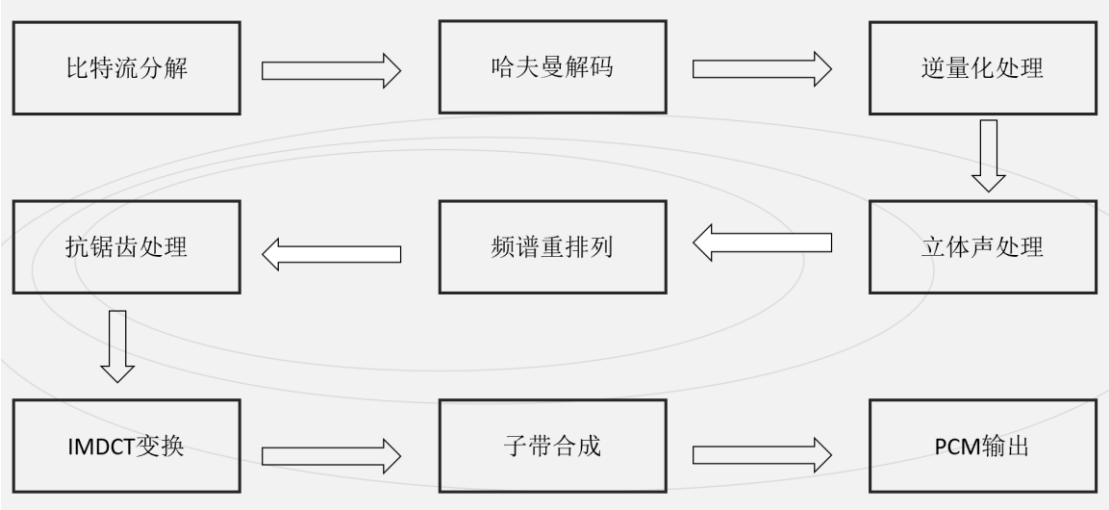

图 2-1　MP3 解码流程图

2.3 比特流分解

比特流分解是指将 MP3 文件以二进制方式打开，然后根据其压缩格式的定义，依次从 MP3 文件中提取出各种信息。其主要流程有查找同步头、头信息解码、CRC 校验码、边信息提取、主信息解码。其流程图如下：

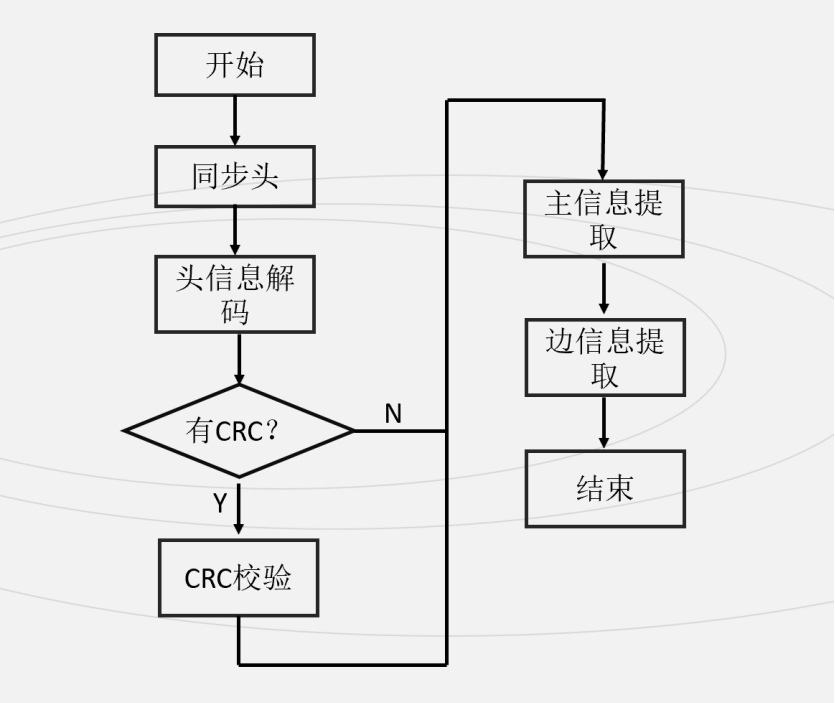

图 2-2　比特流分解流程

其详细过程如下。

2.3.1 帧同步

MP3 是以帧为数据组织形式的。每个帧包含有 1152 个声音采样数据和一些解码需要用的系数数据。但是帧和帧之间并不一定是紧密排列的。之间可能存在空隙，抑或是无用的信息。帧同步的目的在于找出帧头在比特流中的位置，ISO 1172-3 规定，MEPG1 的帧头为 12 比特的“1111　1111　1111”。所以在解码过程中首先要找到这个标志位，以此为起点依次取出需要的对应的系数。

2.3.2 提取头信息

以同步头“1111　1111　1111”为起点，接下来的 20 位属于头信息。头信息包括标识符、拓展标识码、层、保护位、位率检索、取样频率、填充位、加密位、模式、模式扩展、版权位、原件/拷贝位和加重位。具体如下表所示：

起始位	大小	作用	示例
0	11	帧同步标识，11个‘1’。用于定位帧头起始位置	111111111111
11	2	MPEG音频版本 ID： 00：MPEG version2.5 01：reserved 10：MPEG Version 2 11：MPEG Version 1	01
13	2	Layer 序列号: 00 - reserved 01 - Layer III 10 - Layer II 11 - Layer I	01
15	1	Protection bit： 0 - protected by 16 bit CRC following header 1 - no CRC	1
16	4	比特率索引	1001
20	2	采样率索引	11
22	1	Padding bit： 如果设置了，则用一个slot填充数据(slot对框架大小的计算很重要) Layer I的slot大小是4字节，其余情况是1字节。	1
23	1	保护位	1
24	2	channel模式: 00 - 立体声 01 - 联合立体声(立体声) 10 - 双通道(两个单声道) 11 - 单声道	00
26	2	模式扩展	00
28	1	版权位	1
29	1	原始位	1
30	2	加重位	00

表 2-1 数据帧帧头

2.3.3 边信息提取

在每帧 MP3 数据中，对于单声道模式，边信息占用 17 字节的长度；其他模式中边信息占用 32 字节。边信息主要包含 4 个方面的信息：man_data_begin 指针、2 个粒度(granule)组共用的边信息、granule0 的边信息以及 granule1 的边信息。边信息结构表如下所示：

	Main_data_begin	Private_bites	Scfis	Side_info_gr0	Side_info_gr1
单声道	9	5	4	59	59
双声道	9	3	8	118	118

表 2-2　边信息的结构

2.3.4 主数据

MP3 数据帧的主数据部分包括缩放因子和哈夫曼编码数据两部分。

缩放因子：在 MP3 数据中，缩放因子以缩放带位单位，即缩放带中 的每个样点具有相同的缩放因子。这些缩放因子是否为两个粒度公用则是由 scfsi 变量决定。缩放因子所占用的空间由压缩时采用的窗类型决定。

哈夫曼编码数据：在处理 MP3 数据的时候，把每粒度中的 576 个样点分成了 3 个区域（big_value，count1，rzero)。rzero 区域的数据不进行编码。对于 big_value 区域中的样点，又分成了 3 个区域（region）。在该区域中，对两个样点进行统一编码。对于 count1 区域中的样点，4 个样点进行统一的哈夫曼编码。

2.4 哈夫曼解码

在主数据中，紧跟在缩放因子之后的就是哈夫曼编码数据了。每个粒度组的频率线都是用不同的哈夫曼表来进行编码的。编码时，把整个从 0 到奈奎斯特频率的频率范围(共 576 个频率线)分成几个区域，然后再用不同的表编码。划分过程是根据最大的量化值来完成的，它假设较高频率的值有较低的幅度或者根本不需要编码。从高频开始，一对一对的计算量化值等于“0”的数据，此数目记为“rzero”。然后 4 个一组的计算绝对值不超过“1”的量化值(也就是说，其中只可能有一 l，0 和 1 共 3 个可能的量化级别)的数目，记为“count1”，在此区域只应用了两个哈夫曼编码表，郎四元组的哈夫曼表 A 和 B(hA，hB)。最后，剩下的偶数个值的对数记为“big—values”，在此区域值应用了 32 个哈夫曼编码表，即哈夫曼表 O～31(h0～h31)。

此后，为增强哈夫曼编码性能，进一步划分了频谱。也就是说，对 0～big_values*2 的区域(姑且称为大值区)再细化，目的是为了得到更好的错误健壮性和更好的编码效率。在不同的区域内应用了不同的哈夫曼编码表 O～31。具体使用哪一个表由 grle[gr][ch].table—select[region]给出。从帧边信息表中可以看到：

当 window_switching_flag:“l”时，只将大值区再细分为 2 个区，此时 regionl_count 无意义，此时 region0_count 的值是标准默认的；但当 window_switching_flag=“0”时，由 region0_count 和 region1_count 再将大值区分为 3 个区。但是由于 region0_count，region1_count 是根据从 0～576 个频率线划分的，因此有可能超过了 big—values*2 的范围，此时以 big—values*2 为准。Region0_count 和 region1_count 表示的只是一个索引值，具体频带要根据标准中的缩放因子频带表来查得。

在程序实现上，哈夫曼表逻辑存储采用了广义表结构，物理存储上使用数组结构。查表时，先读入 4bit 数据，以这 4bit 数据作为索引，其指向的元素有两种类型，一种是值结构，另一种是链表指针式结构，在链表指针式结构中给出了还需要读 取的 bit 数，及一个偏移值。如果索引指向的是一个值结构，则这个值结构就包含了要查找的数据。如果索引指向的是一个链表指针式结构，则还需再读取其中 指定的比特数，再把读取出的比特数同偏移值相加，递归的找下去，直到找到值结构为止。

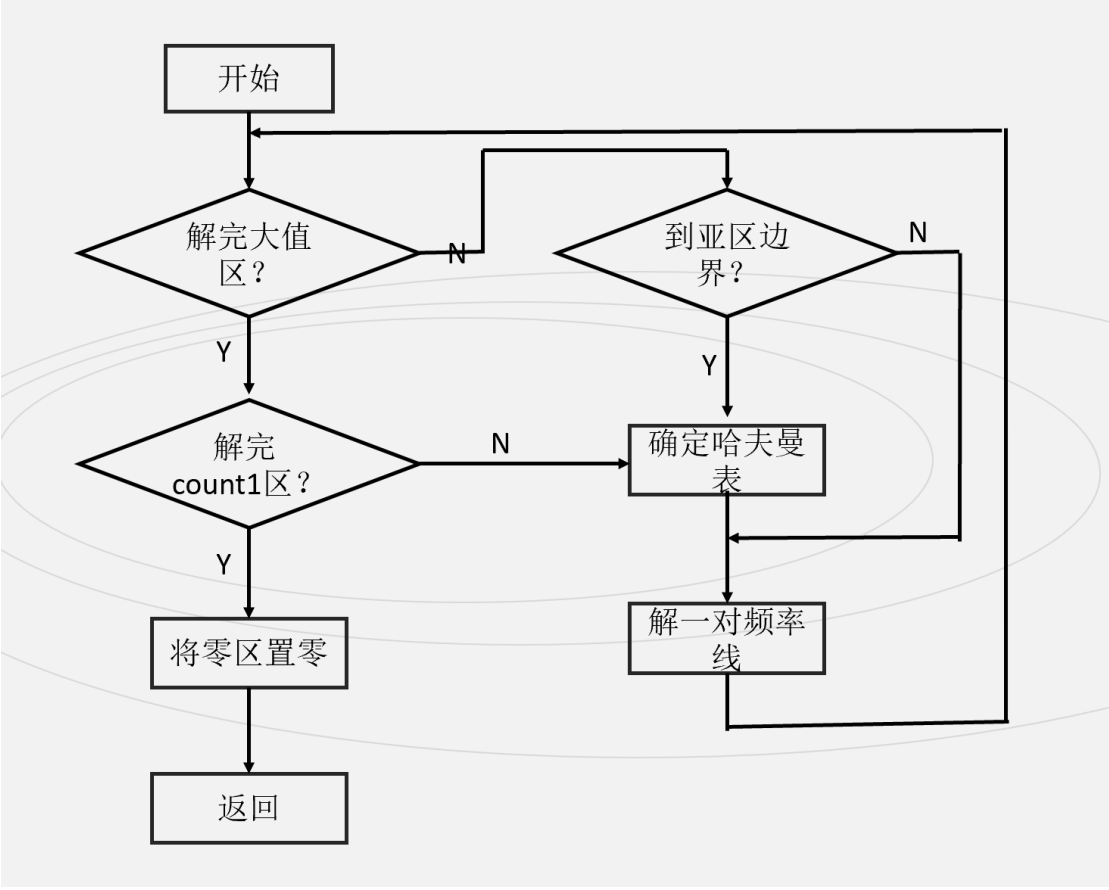

图 2-3　哈夫曼解码流程

2.5 逆量化处理

反量化就是基于前面步骤中所得到的哈夫曼解码数据，根据反量化全缩放公式和帧边信息，对于不同的块类型采用不同的公式以恢复 576 个频率线的真实值。公式如下所示。

$$XR_i = sign(IS_i) \cdot |IS_i|^{\frac{4}{3}} \cdot 2^{\frac{1}{4}A-B}$$

对于长块：A = global_gain – 210

B = (scalefac_scale + 1) / 2 · (scalefac_l[sfb] + preflag · pretab[sfb])

对于短块：A = global_gain – 210 – 8 · subblock_gain[w]

B = (scalefac_scale + 1) / 2 · scalefac_s[sfb][w]

2.6 重排序

反量化过程中得出的频谱值并不是按相同顺序排列的。在编码的 MDCT 过程中，对于长窗产生的频谱值先按子带然后按频率排列；对于短窗，产生的频谱值时按子带、窗、频率的顺序排列的。为了提高哈夫曼编码效率，短窗中的数据被重新排序，按照子带、频率、窗的顺序排列。解码时，重排序过程就是将短窗中的频谱值重新排列。变量 window_switch_flag 和 block_type 可以用来确定是否重排序。

2.7 立体声处理

MPEG-1 Layer III 中除了支持简单的单声道和立体声（相互独立的两个声道）外，还支持更为复杂的联合立体声(joint stereo)模式：M/S 立体声（M/S stereo）和强度立体声（intensity stereo）。

M/S 立体声模式：传送的是规格化的中间/旁边声道 M_i/S_i 值，而不是左右声

道 L_i/R_i。值 M；在左声道中传送，值 S_i 在右声道中传送。当两个声道高度相关时，这种模式比较适合，这就意味着“和”信号中要比“差”信号中包含较多的信息，一般左右声道的值是很接近的，采用 M/S 立体声模式后，只需要传送一个均值和一个小值，这样可以减少传输的比特数。M/S 立体声处理是无损的。在解码时，左右两个声道 L_i/R_i 可以通过下面的公式来重建，其中 i 表示的是频率线的指数：

$$L_i = \frac{M_i + S_i}{\sqrt{2}}$$

$$R_i = \frac{M_i - S_i}{\sqrt{2}}$$

强度立体声模式：在 Layer III 中，强度立体声不是像第一层和第二层那样通过使用一对比例因子来完成，而是左声道仍传送缩放因子，右声道传送立体声位置 is_pos[sfb]。编码器把一些高频子带的输出编码为单个的“和”信号 L+R，而不是分别独立的传送左和右的子带信号，左右声道的平衡可以通过比例因子来传输。解码器通过单个 L+R($=L'_i$)信号来重构左右两个声道的信号，右声道的平衡比例因子用 is_pos_{sfb}，m 来表示。解码时利用下面三个公式解出左右声道信号：

$$is\_ratio_{sfb} = \tan(is\_pos_{sfb} \cdot \frac{\pi}{12})$$

$$L_i = L'_i \cdot \frac{is\_ratio_{sfb}}{1 + is\_ratio_{sfb}}$$

$$R_i = L'_i \cdot \frac{1}{1 + is\_ratio_{sfb}}$$

根据帧头信息中的模式位（mode）和模式扩展位（mode_extension）的值可以确定何时应用 M/S 立体声和强度立体声解码公式。

1．若(modeI=01)，左右声道使用各自对立体声边信息对主要数据解码，获得通道独立的样本数据。

2．若(mode=01)＆(mode_extension=01)，则对右声道 Zero part 部分使用强度立体声解码(直至 is_pos。。=7)，其余部分为独立的左右声道数据。

3．若(mode=01)＆(mode_extension；10)，则对全部频率线用 M／S 立体声方式解码。

4．若(mode=01)＆(mode—extension=11)，则对右声道 Zero_part 部分使用强度立体声解码(直至 is—pos；m=7)，其余使用用 M／s 立体声方式解码。

2.8 抗锯齿处理

抗锯齿处理就是消除伪信号。相邻两个子带间互相干扰造成失真，为了减小这种影响，在信号送入 IMDCT 之前作消混叠处理。消混叠通过对子带作 8 点的蝶形变换实现。其相关算法较为简单这里不再赘述。

2.9 IMDCT 变换

MDCT 的目的在于进行时域到频域的转换，减少信号的相关性，使得信号的压缩 可以更加高效地完成，而它的反变换 IMDCT 的目的在于将信号还原为没有变换之 前的数值，使频域值向时域值过渡。

其公式如下：

$$x_i = \sum_{k=0}^{\frac{n}{2}-1} X_k \cos\left(\frac{\pi}{2n}(2i+1+\frac{n}{2})(2k+1)\right) \qquad \text{for i=0 to n-1}$$

For long block n=36, for short block n=12.

在进行了 IMDCT 变换之后，需对频率信号进行加窗、覆盖、叠加。其公式在程序中以有体现，就不具体说明。

2.10 频率反转

在 IMDCT 之后，进入合成多相滤波之前必须进行频率反转补偿以校正多相滤波器 组的频率反转。方法是将奇数号子带的奇数个采样值乘以-1。

2.11 子带合成

子带合成滤波器将 32 个带宽相等的子带中的频域信号反变换成时域信号。子带合成是逆向离散余弦变换后的一个通道中 32 个子带的样值，经过一系列运算还原出 32 个 PCM 数字音频信号的过程。子带合成过程先将 32 个子带样值进行逆向离散余弦变换，生成 64 个中间值，将这 64 个中间值转入到一个长为 1024 点的类似先进先出 FIFO 的缓存，再在这 1024 个值中抽取一半，构成一个 512

个值的矢量，进行加窗运算，最后将加窗结果进行叠加生成 32 个时域输出。

2.12 PCM 音频输出

将最后的文件写入到 PCM 文件中。

3 主要代码分析

3.1 源代码文件说明

本程序主要功能模块有三个，分别是：common.h，decoder.h，Huffman.h。并通过 main.cpp 调用头文件中的函数。还有两个资源文件 dewindow.txt，huffdec.txt，主要用于子带合成和哈夫曼解码中。项目的工程结构如下图所示：

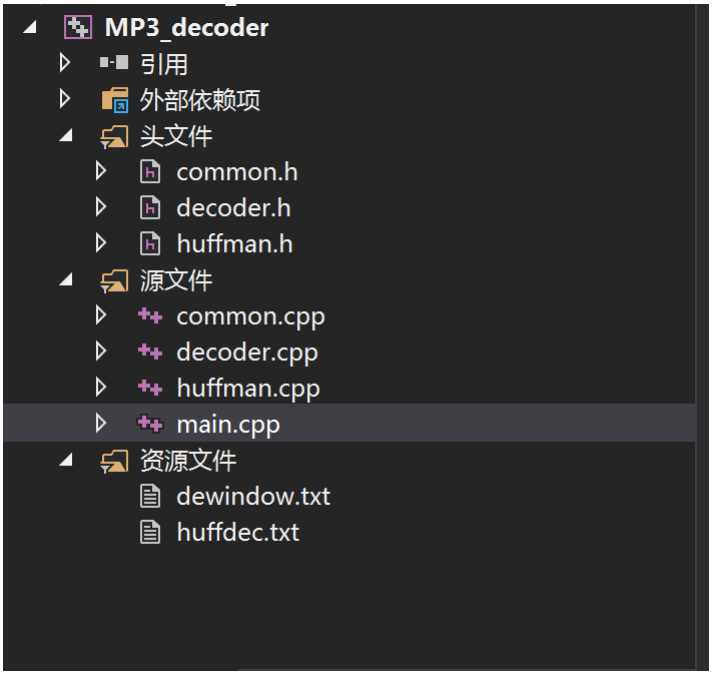

图 3-1　项目工程结构图

头文件/资源文件说明：

1．common.h：解码流程中几乎所有的变量类型说明，一些对文件流的基本操作函数的定义和宏定义都在该头文件中说明。其他的头文件都依赖该文件。

2.　decoder.h：主要是用于 MP3 的解码器，是程序中最核心的部分，负责解码 MP3 二进制流，除了哈夫曼解码以外，其余所有解码操作函数都在该偷吻将中声明。

3.　Huffman.h：主要声明哈夫曼解码过程中所需要的函数和结构体类型。

3.2 main.cpp 主要流程说明

```
int main(int argc, char** argv)
{
    FILE* musicout;
    Bit_stream_struc bs;
    frame_params fr_ps;
    III_side_info_t III_side_info;
    III_scalefac_t III_scalefac;
    unsigned int old_crc;
    layer info;
    int sync, clip;
    int done = FALSE;
    unsigned long frameNum = 0;
    unsigned long bitsPerSlot;
    unsigned long sample_frames;

    typedef short PCM[2][SSLIMIT][SBLIMIT];
    PCM* pcm_sample;

    pcm_sample = (PCM*)mem_alloc((long)sizeof(PCM), (char*)"PCM Samp");

    fr_ps.header = &info;

    if ((fopen_s(&musicout,argv[2], "w+b")) != 0)
    {
        printf("无法创建文件: \"%s\"\n", argv[2]);
        exit(1);
    }

    //创建比特流
    open_bit_stream_r(&bs, argv[1], BUFFER_SIZE);

    sample_frames = 0;
    while (!end_bs(&bs))
    {
        //查找同步头
        sync = seek_sync(&bs, SYNC_WORD, SYNC_WORD_LENGTH);
        if (!sync)
        {
            done = TRUE;
            printf("\n 帧无法被定位\n");
            break;
```

```
        }

        //头信息提取
        decode_info(&bs, &fr_ps);
        hdr_to_frps(&fr_ps);
        frameNum++;
        if (info.error_protection)
            buffer_CRC(&bs, &old_crc);

        //判断是否是 mp3
        switch (info.lay)
        {
        case 3:
        {
            int nSlots, main_data_end, flush_main;
            int bytes_to_discard, gr, ch, ss, sb;
            static int frame_start = 0;

            bitsPerSlot = 8;

            //读边信息
            III_get_side_info(&bs, &III_side_info, &fr_ps);


            //提取主信息
            nSlots = main_data_slots(fr_ps);

            for (; nSlots > 0; --nSlots)
            {
                hputbuf((unsigned int)getbits(&bs, 8), 8);
            }
            main_data_end = hsstell() / 8;

            if (flush_main = (hsstell() % 8))
            {
                hgetbits((int)(bitsPerSlot - flush_main));
                main_data_end++;
            }

            bytes_to_discard = frame_start - main_data_end - III_side_i
nfo.main_data_begin;
            if (main_data_end > 4096)
            {
                frame_start -= 4096;
```

```
                rewindNbytes(4096);
            }

            frame_start += main_data_slots(fr_ps);
            if (bytes_to_discard < 0)
            {
                frameNum--;
                break;
            }
            for (; bytes_to_discard > 0; --bytes_to_discard)
            {
                hgetbits(8);
            }

            clip = 0;
            for (gr = 0; gr < 2; gr++)
            {
                double lr[2][SBLIMIT][SSLIMIT], ro[2][SBLIMIT][SSLIMIT]
;
                for (ch = 0; ch < fr_ps.stereo; ch++)
                {
                    long int is[SBLIMIT][SSLIMIT];
                    int part2_start;
                    part2_start = hsstell();

                    //比例因子提取
                    III_get_scale_factors(&III_scalefac, &III_side_info
, gr, ch, &fr_ps);

                    //hufman 解码
                    III_hufman_decode(is, &III_side_info, ch, gr, part2
_start, &fr_ps);

                    III_dequantize_sample(is, ro[ch], &III_scalefac, &(
III_side_info.ch[ch].gr[gr]), ch, &fr_ps);
                }
                III_stereo(ro, lr, &III_scalefac, &(III_side_info.ch[0]
.gr[gr]), &fr_ps);
                for (ch = 0; ch < fr_ps.stereo; ch++)
                {
                    double re[SBLIMIT][SSLIMIT];
                    double hybridIn[SBLIMIT][SSLIMIT];  /* Hybrid filte
r input */
```

```
                    double hybridOut[SBLIMIT][SSLIMIT]; /* Hybrid filte
r out */
                    double polyPhaseIn[SBLIMIT];        /* PolyPhase In
put. */
                    III_reorder(lr[ch], re, &(III_side_info.ch[ch].gr[g
r]), &fr_ps);
                    //抗锯齿处理
                    III_antialias(re, hybridIn, /* Antialias butterflie
s. */
                        &(III_side_info.ch[ch].gr[gr]), &fr_ps);
                    //IMDCT
                    for (sb = 0; sb < SBLIMIT; sb++)
                    { /* Hybrid synthesis. */
                        III_hybrid(hybridIn[sb], hybridOut[sb], sb, ch,
                            &(III_side_info.ch[ch].gr[gr]), &fr_ps);
                    }
                    for (ss = 0; ss < 18; ss++) //多相频率倒置
                        for (sb = 0; sb < SBLIMIT; sb++)
                            if ((ss % 2) && (sb % 2))
                                hybridOut[sb][ss] = -hybridOut[sb][ss];
                    for (ss = 0; ss < 18; ss++)
                    { //多相合成
                        for (sb = 0; sb < SBLIMIT; sb++)
                            polyPhaseIn[sb] = hybridOut[sb][ss];
                        //子带合成
                        clip += SubBandSynthesis(polyPhaseIn, ch,
                            &((*pcm_sample)[ch][ss][0]));
                    }
                }
                //PCM 输出
                /* Output PCM sample points for one granule(颗粒). */
                out_fifo(*pcm_sample, 18, &fr_ps, done, musicout, &samp
le_frames);
            }
            if (clip > 0)
                printf("\n%d samples clipped.\n", clip);
        }
        break;
        default:
            printf("\nOnly layer III supported!\n");
            exit(1);
            break;
        }
    }
```

```
    close_bit_stream_r(&bs);
    fclose(musicout);
    printf("\nDecoding done.\n");
    return 0;
}
```

4 实验结构与分析

程序运行前需要传入 MP3 文件路径的参数，为了简化测试过程，MP3 文件的路径预定义定义(使用本地某一 MP3 文件的路径)而无需传参，但是在实际过程中需要带有该 MP3 文件的路径。

结果如下图所示：

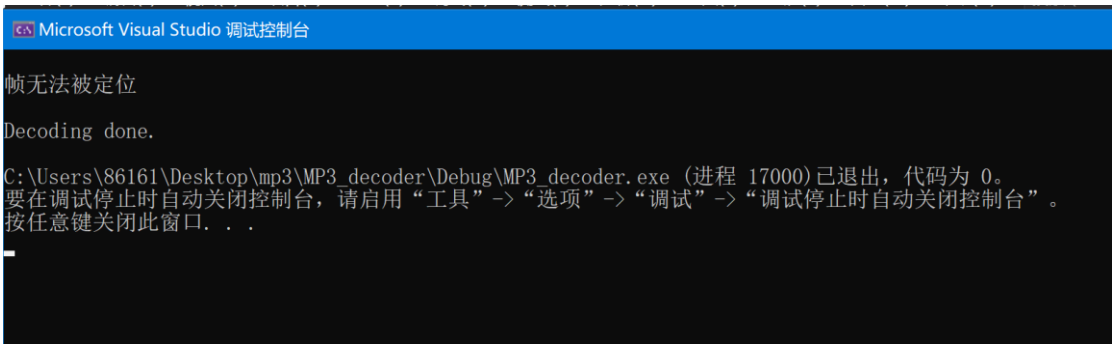

图 4-1 运行结果

由上图可知，该文件正确结束，并生成 musicout.pcm 文件，并且能够通过软件直接播放该文件。

5 心得体会

这次多媒体技术课程，基于在多媒体相关媒体的编码和解码了解下，我选择了这次项目的题目为 MP3 音频格式的解码。虽然在课程上已经讲解了相关压缩编码的知识，但是对与解压缩方面讲解的不是很多，所以选择这次课题也是对自己知识面的补充以及个人能力的增强。

我在做该项目的过程中，也遇到了不少的困难，比如 Huffman 解码实现以及后反编码的公式等。但是通过在网络上如 CSDN 网站、百度查找相关解决方法，通过和同学讨论也得到了不少的帮助，这才是得我的项目能够顺利运行。整个课程设计对我来讲不仅是一次很好的学习过程，更是一次锻炼的机会。

通过这次学习实践，我对 MP3 解码算法有了比较深刻的认识，编程能力也有了进一步的提升。很遗憾的是，当目前为止，我只是将 MP3 解码成 pcm 格式，但是 pcm 格式文件的播放还是需要其他软件来播放。

参考文献

[2]百度文库 Mp3 解码算法流程 .李国辉

https://wenku.baidu.com/view/2178dfe5a5e9856a5712600c.html?fr=search .2015-12-21

[3]百度文库 MP3 软件解码器的研究与实现 .李菁菁

https://wenku.baidu.com/view/a0d1b24dfe4733687e21aad5.html?fr=search&pn=50 .2011-04-26

[4]百度文库 mp3 解码算法原理详解

https://wenku.baidu.com/view/0dc1c929647d27284b735149.html .2018-07-01

[5]百度文库 Mp3 解码流程

https://wenku.baidu.com/view/9225d40627284b73f242508d.html?fr=search .2015-04-08

教师评语评分
【总分】
【评语】 评阅教师签名： 日期：2019.6.12