第 1 章 声明 2
第 2 章 Mp3 解码算法流程 3
2.1. Mp3 文件格式 3
2.1.1. Audio Sequence 3
2.1.2. Audio Frame 3
2.1.3. Header 4
2.1.4. Error Check 4
2.1.5. Audio data , Layer III 4
2.1.6. Main_data 5
2.1.7. Huffmancodebits 6
2.1.8. Ancillary data 7
2.2. 数据项的含义 7
2.2.1. Header 7
2.2.2. Error Check 9
2.2.3. Side information 10
2.3. Mp3 解码算法所用的基本概念 11
2.3.1. 子带和缩放因子频带 11
2.3.2. Huffman 码表的选择 13
2.3.3. huffman 码表的特点 14
2.3.4. 缩放因子(scalefactor) 14
2.3.5. 节的长短块切换 15
2.4. mp3 解码具体流程 16
2.4.1. 预处理（Preprocessing） 16
2.4.2. Huffman decoding 16
2.4.3. 反量化（Requantization） 17
2.4.4. 重排序（Reordering） 17
2.4.5. 立体声解码（Stereo decoding） 17
2.4.6. 混叠消除（Alias reduction） 18
2.4.7. IMDCT 变换 18
2.4.8. 子带合成滤波(Synthesis filter bank) 19
第 3 章 libmad 解码程序源代码分析 21
3.1. 码流读取 21
3.2. 桢的同步 22
3.3. 桢头解码 25
3.4. sideinfo 解码 26
3.5. main_data 的读取 27
3.6. 缩放因子解码 29
3.7. huffman 解码 31
3.8. 反量化（requantization） 34
3.9. 重排序（reordering） 35
3.10. IMDCT 变换 37
3.11. 子带合成滤波（synthesis filter bank） 38
附录 A libmad 的交叉编译过程 44

第1章

声明

作者：李国辉　ghli@aiar.xjtu.edu.cn

第2章 Mp3 解码算法流程

MP3 的全称为 MPEG1 Layer-3 音频文件， MPEG 音频文件是 MPEG1 标准中的声音部分，也叫 MPEG 音频层，它根据压缩质量和编码复杂程度划分为三层，即 Layer1、Layer2、Layer3，且分别对应 MP1、MP2、MP3 这三种声音文件，并根据不同的用途，使用不同层次的编码。MPEG 音频编码的层次越高，编码器越复杂，压缩率也越高，MP1 和 MP2 的压缩率分别为 4：1 和 6：1-8：1，而 MP3 的压缩率则高达 10：1-12：1。一分钟 CD 音质的音乐，未经压缩需要 10MB 的存储空间，而经过 MP3 压缩编码后只有 1MB 左右。不过 MP3 对音频信号采用的是有损压缩方式，为了降低声音失真度，MP3 采取了“心理声学模型”，即编码时先对音频文件进行频谱分析，然后再根据心理声学模型把谱线分成若干个阈值分区，并计算每个阈值分区的阈值，接着通过量化和熵编码对每个谱线进行编码，最后形成具有较高压缩比的 MP3 文件，并使压缩后的文件在回放时能够达到比较接近原音源的声音效果。

2.1. Mp3 文件格式

MP3 文件以一帧为一个编码单元，各帧编码数据是独立的。为了清晰而准确地描述 mp3 文件格式，下面采用位流语法描述，这种语法格式与 c 语言近似，易于理解，且描述清晰。其中粗体表示码流中的数据项，bslbf 代表位串，即“ Bit string, left bit first ”，uimsbf 代表无符号整数，即”unsinged integer, most significant bit first”，数字表示该数据项所占的比特数。

2.1.1. Audio Sequence

```
audio sequence()
 {
  while (true)
  {
    frame()
  }
 }
```

2.1.2. Audio Frame

```
frame()
{
    header()
    error_check()
    audio_data()
    ancillary_data()
```

}

2.1.3. Header

header()		
{		
syncword	12	bslbf
ID	1	bslbf
layer	2	bslbf
protection_bit	1	bslbf
bitrate_index	4	bslbf
sampling_frequency	2	bslbf
padding_bit	1	bslbf
private_bit	1	bslbf
mode	2	bslbf
mode_extension	2	bslbf
copyright	1	bslbf
original/home	1	bslbf
emphasis	2	bslbf
}		

2.1.4. Error Check

error_check()		
{		
if (protection_bit==0)		
crc_check	16	rpchof
}		

2.1.5. Audio data , Layer III

audio_data()		
{		
main_data_begin	9	uimsbf
if(mode==single_channel) **private_bits**	5	bslbf
else **private_bits**	3	bslbf
for(ch=0;ch<nch;ch++)		
for(scfsi_band=0;scfsi_band<4;scfsi_band++)		
scfsi[ch][scfsi_band]	1	bslbf
for(gr=0;gr<2;gr++)		
for(ch=0;ch<nch;ch++){		
part2_3_length[gr][ch]	12	uimsbf

big_values[gr][ch]	9	uimsbf
global_gain[gr][ch]	8	uimsbf
scalefac_compress[gr][ch]	4	bslbf
window_switching_flag[gr][ch]	1	bslbf
if(window_switching_flag[gr][ch]){		
block_type[gr][ch]	2	bslbf
mixed_block_flag[gr][ch]	1	uimsbf
for(region=0;region<2;region++)		
table_select[gr][ch][region]	5	bslbf
for(window=0;window<3;window++)		
subblock_gain[gr][ch][window]	3	uimsbf
}		
else{		
for(region=0;region<3;region++)		
table_select[gr][ch][region]	5	bslbf
region0_count[gr][ch]	4	bslbf
region1_count[gr][ch]	3	bslbf
}		
preflag[gr][ch]	1	bslbf
scalefac_scale[gr][ch]	1	bslbf
count1table_select[gr][ch]	1	bslbf
}		
main_data		
}		

2.1.6. Main_data

main_data()		
{		
for(gr=0;gr<2;gr++)		
for(ch=0;ch<nch;ch++){		
if((window_switching_flag[gr][ch]==1)&&(block_type[gr][ch]==2)){		
if(mixed_block_flag[gr][ch]){		
for(sfb=0;sfb<8;sfb++)		
scalefac_l[gr][ch][sfb]	0..4	uimsbf
for(sfb=3;sfb<12;sfb++)		
for(window=0;window<3;window++)		
scalefac_s[gr][ch][sfb][window]	0..4	uimsbf
}		
else{		
for(sfb=0;sfb<12;sfb++)		
for(window=0;window<3;window++)		
scalefac_s[gr][ch][sfb][window]	0..4	uimsbf
}		

```
}
    else{
        if((scfsi[ch][0]==0||(gr==0))
            for(sfb=0;sfb<6;sfb++)
                scalefac_l[gr][ch][sfb]                    0..4    uimsbf
        if((scfsi[ch][1]==0||(gr==0))
            for(sfb=6;sfb<11;sfb++)
                scalefac_l[gr][ch][sfb]                    0..4    uimsbf
        if((scfsi[ch][2]==0||(gr==0))
            for(sfb=11;sfb<16;sfb++)
                scalefac_l[gr][ch][sfb]                    0..3    uimsbf
        if((scfsi[ch][3]==0||(gr==0))
            for(sfb=16;sfb<21;sfb++)
                scalefac_l[gr][ch][sfb]                    0..3    uimsbf
    }
    Huffmancodebits()
    }
    for(b=0;b<no_of_ancillary_bits;b++)
       ancillary_bit       1      bslbf
}
```

2.1.7. Huffmancodebits

```
Huffmancodebits(){
    for(l=0;l<big_values*2;l+=2){
      hcod[|x|][|y|]                                       0..19   bslbf
      if(|x|==15&&linbits>0)   linbitsx                    1..13   uimsbf
      if(x!=0)signx                                        1       bslbf
      if(|y|==15&&linbits>0)   linbitsy                    1..13   uimsbf
      if(y!=0)   signy                                     1       bslbf
       is[l]=x
       is[l+1]=y
}
    for(;l<big_values*2+count1*4;l+=4){
      hcod[|v|][|w|][|x|][|y|]                             1..6    bslbf
      if(v!=0)   signv                                     1       bslbf
      if(w!=0)   signw                                     1       bslbf
      if(x!=0)   signx                                     1       bslbf
      if(y!=0)   signy                                     1       bslbf
       is[l]=v
       is[l+1]=w
       is[l+2]=x
       is[l+3]=y
    }
```

```
    for(;l<576;l++)
      is[l]=0
}
```

2.1.8. Ancillary data

```
ancillary_data(){
    if((layer==1||layer==2))
      for(b=0;b<no_of_ancillary_bits;b++)
        ancillary_bit                                    1          bslbf
}
```

2.2.数据项的含义

2.2.1. Header

Ø Syncword

同步头，表示一帧数据的开始，共 12 位，全 1 即 0XFFF。

表格 2-1 Layer

Layer	
'11'	Layer I
'10'	Layer II
'01'	Layer III
'00'	reserved

表格 2-2 Bitrate_index

bitrate_index	bitrate specified (kBit/s)		
	Layer I	Layer II	Layer III
'0000'	free	free	free
'0001'	32	32	32
'0010'	64	48	40
'0011'	96	56	48
'0100'	128	64	56
'0101'	160	80	64
'0110'	192	96	80
'0111'	224	112	96
'1000'	256	128	112
'1001'	288	160	128
'1010'	320	192	160

'1011'	352	224	192
'1100'	384	256	224
'1101'	416	320	256
'1110'	448	384	320
'1111'	forbidden	forbidden	forbidden

Ø ID

算法标识位，“ 1 ”表示 MPEG 音频，“ 0 ”保留。

Ø Layer

用来说明是哪一层编码，如表格 2-1 Layer 所示。

Ø Protection_bit

用来表明冗余信息是否被加到音频流中，以进行错误检测和错误隐蔽。“ 1 ”表示未增加，“ 0 ”表示增加。

Ø Bitrate_index

用来指示该帧的 bitrate，如表格 2-2 Bitrate_index 所示。

Ø Sampling_frequency

用来指示采样频率，如表格 2-3 Sampling_frequency 所示。

表格 2-3 Sampling_frequency

sampling_frequency	frequency specified (kHz)
'00'	44.1
'01'	48
'10'	32
'11'	reserved

Ø Padding_bit

如果该位为 1，那么帧中包含一个额外槽，用于把平均位率调节到采样频率，否则该位必须为 0。在采样频率为 44.1kHz 时，填补是必要的，在自由格式中也可能需要填补。

Ø Private_bit

留做私用，没有定义。

Ø Mode

定义通道模式，如表格 2-4 Mode 所示。

表格 2-4 Mode

mode	mode specified
'00'	stereo
'01'	joint_stereo (intensity_stereo and/or ms_stereo)
'10'	dual_channel
'11'	single_channel

Ø Mode_extension

用来标识采用了哪一种 joint_stereo，具体对应的频带范围隐含在算法中，如表格 2-5 Mode_extension 所示。

表格 2-5 Mode_extension

mode_extension	
'00'	subbands 4-31 in intensity_stereo, bound==4
'01'	subbands 8-31 in intensity_stereo, bound==8
'10'	subbands 12-31 in intensity_stereo, bound==12
'11'	subbands 16-31 in intensity_stereo, bound==16

Ø Copyright

表明版权用，“ 1 ”表示有版权，“ 0 ”表示没有版权。
Original/copy ：表明原版还是复制，“ 1 ”表示原版，“ 0 ”表示复制。
Emphasis ：表明加重音类型，如表格 2-6 Emphasis 所示。

表格 2-6 Emphasis

emphasis	emphasis specified
'00'	none
'01'	50/15 microseconds
'10'	reserved
'11'	CCITT J.17

2.2.2. Error Check

CRC 校验的基本思想是利用线性编码理论，在发送端根据要传送的 k 位二进制码序列，以一定的规则产生一个校验用的监督码（既 CRC 码）r 位，并附在信息后边，构成一个新的二进制码序列数共(k+r)位，最后发送出去。在接收端，则根据信息码和 CRC 码之间所遵循的规则进行检验，以确定传送中是否出错。在 MP3 协议中采用了 CRC-16 生成 CRC

码，其生成多项式如下：

$$G(x) = X^{16} + X^{15} + X^{2} + 1$$

2.2.3. Side information

Side information 指的是在 audio_data 中 main_data 之前的一部分信息。这部分提供了解码中一些辅助的信息，用来帮助整个解码过程。为了帮助理解这一部分数据项的含义，会在下面大致阐述 mp3 解码所用的基本概念。

Ø Main_data_begin

表示一帧数据 main data 的开始位置。它表示 main data 相对于该帧同步头的负偏移。这里涉及到一个 bit reservoir 的技术，它改变了每帧的可用比特数为常数的限制，而是围绕一个长时间的平均值（目标比特率）变化。因为 MP3 的编码方式是采用 Huffman 编码，所以编码后每一帧的数据长度是不一样的，可能有的大于目标比特率为每一帧分配的空间，有的可能小于这个空间，所以为了增加空间利用率，当前帧未使用完的空间可以保存起来留给后面需要的帧使用，因此每一帧的 main data 开始位置可能在它的 header 和 side information 之前，而 main_data_begin 就是用来指示这个开始位置的，这种技术就叫做 bit reservoir。

Ø Private_bits

留做私用。

Ø Scfsi、scalefac_compress

参考 2.3.4 缩放因子(scalefactor)

Ø Part_2_3_length

表示 main data 中 scalefactor 和 huffman 数据所占用的比特数。

Ø Global_gain

全局量化步长。

Ø Window_switching_flag、block_type和 mixed_block_flag

参考 2.3.5 节的长短块切换

Ø Table_select、big_values、count1_table_select、region0_count 和 region1_count

参考 2.3.2 Huffman 码表的选择

Ø Subblock_gain

短窗块量化时所用的增益偏移量。

Ø Preflag和 scalefac_scale

在反量化过程中对压缩数据还原时用到的变量。

2.3. Mp3 解码算法所用的基本概念

2.3.1. 子带和缩放因子频带

压缩之后的 mp3 数据是以一桢为单位的，每一桢分为两节（granule），这两节在编解码时相对独立，从每一节中可以解码出 576 个 pcm 数据，两节可解出 1152 个 pcm 数据。从二进制 101……码流中得到我们所需要的信息的第一步就是 huffman 解码，huffuman 编码信息存放在每一节中的 Huffmancodebits( )中，通过 huffman 解码可以得到 576 个值，这 576 个值在不同节类型（参考 2.3.5 节的长短块切换）下有不同的含义，下面分情况描述：

u 该节为长块

这 576 个值代表 576 条频率线上的值，它们是时域中 576 个 pcm 值经过时频变换的结果。这 576 条频率线从低到高分为 32 个子带，每个子带包含 18 条频率线。解出来的这 576 个值是整数，需要进行反量化变成浮点数，反量化的过程并不是一条频率线为单位进行的，而是若干条频率线为单位进行的，这若干条频率线组成了频带（band），叫做缩放因子频带（scalefactor band）,顾名思义，在一个缩放因子频带内的频率线在反量化时共用缩放因子。在 44.1khz 的采样率下，缩放因子频带的划分如所示：

表格 2-7 长块的缩放因子频带划分

scalefactor band	width of band	index of start	index of end
0	4	0	3
1	4	4	7
2	4	8	11
3	4	12	15
4	4	16	19
5	4	20	23
6	6	24	29
7	6	30	35
8	8	36	43
9	8	44	51

10	10	52	61
11	12	62	73
12	16	74	89
13	20	90	109
14	24	110	133
15	28	134	161
16	34	162	195
17	42	196	237
18	50	238	287
19	54	288	341
20	76	342	417

其中，频率线 418 至 575 不需要归属于某一个 scalefactor band, 因为属于这一个频带的频率线在进行反量化时，系统提供默认的反量化因子。

- 该节为短块

这 576 个值代表 192 条频率线上的值，这 192 条频率线从低到高分为 32 个子带，每个子带包含 6 条频率线。每一条频率线上有三个值，分别属于三个窗（window0、window1 和 window2）。这 192 条频率线被划分为若干缩放因子频带，在 44.1khz 情况下，划分方式如下：

表格 2-8　短块的缩放因子频带划分

scalefactor band	width of band	index of start	index of end
0	4	0	3
1	4	4	7
2	4	8	11
3	4	12	15
4	6	16	21
5	8	22	29
6	10	30	39
7	12	40	51
8	14	52	65
9	18	66	83
10	22	84	105
11	30	106	135

其中，频率线 136 至 192 不需要归属于某一个 scalefactor band, 因为属于这一个频带的频率线在进行反量化时，系统提供默认的反量化因子。按先后顺序解出来的这 576 个值先是按缩放因子频带从低到高排列，在每一个缩放因子频带内，按 window0，window1，window2 排列，在每一个 window 中，频率线从低到高排列。

- 该节是混合块

在这种情况下，按先后顺序解出来的 576 个值分为两部分，第一部分（前 36 个值）是长块部分，故它们代表 36 条频率线，这 36 条频率线（参考表格 2-7　长块的缩放因子频带划分）划分为 8 个缩放因子频带；第二部分（后 540 个值）是短块部分，它们代表 180 个频

率线，每个频率线上有三个值，分别属于 window0、window1 和 window2，这 180 个频率线（参考表格 2-8　短块的缩放因子频带划分）划分为 9 个缩放因子频带（scalefactor band3 à scalefactor band11）。

综合上述三种情况，这 576 值排列方式如下所示：

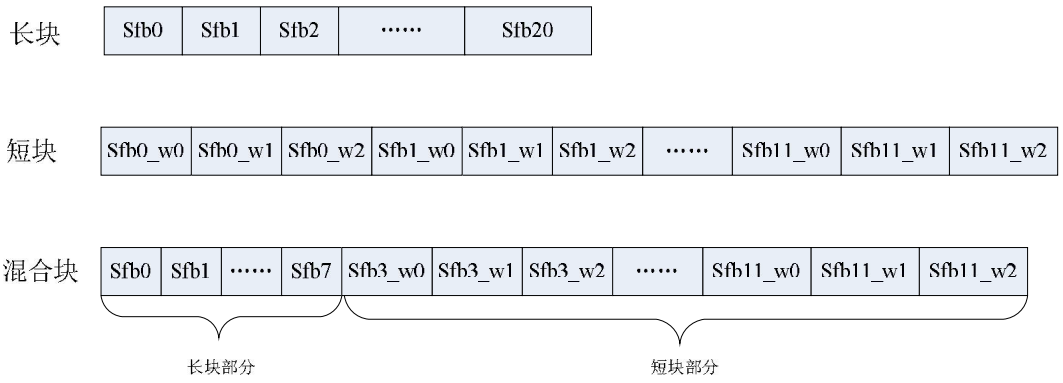

图 2-1 不同情况下 huffman 解码得到的 576 个值的含义

2.3.2. Huffman码表的选择

从 Huffmancodebits( )中解码得到 576 个值的过程不是一个简单的查表过程，这涉及到换表的过程。在解码时，当从一个缩放因子频带过渡到另一个缩放因子频带时，huffman 码表可能需要改变。Huffman 解码得到的 576 个值分为三个部分，如下所示：

```
xxxxxxxxxxxxx-----------------000000000000000000000000000000
|            |                 |                               |
1        bigvalues*2        bigvalues*2+count1*4           iblen

The values 000 are all zero.
The values --- are -1,0 or +1. Their number is a multiple of 4.
The values xxx are not bound.
Iblen is 576.
```

图 2-2　bigvalues 和 count1 的含义

在大值区(xxx),得到的值较大，一共有 bigvalues*2 个值，每两个值一起编码；在小值区(---),值只能为-1，0，+1，一共有 count1*4 个值，每四个值一起编码；在零值区(000)值为零，不需要编码。

在不同区域编码时，用到的 huffman 表是不一样的。

- 在大值区编码时，为了进一步提高编码效率，大值区又分为三个区域：region0、region1 和 region2，在不同区域用不同的 huffman 表编码。region 的划分是以缩放因子频带为单位划分的。在 side information 中，region0_count[gr][ch] 和 region1_count[gr][ch] 提供了划分信息。region0_count+1 表示在 region0 区的缩放因子频带的个数，region1_count+1 表示在 region1 区的缩放因子频带的个数。需要说明的是，如果是短块或者

混合块中的短块部分，一个缩放因子频带被计数三次，例如，对于短块来说，region0_count 为 8 意味着 region1 从 scalefactor band 3 开始。Region2 区的长度在 side information 中并没有给出，但是根据 big_values[gr][ch]、region0_count 和 region1_count 可以计算出来。在得到大值区 region 的划分之后，就可以根据 table_select[gr][ch][region]来选择在每个区域所用的 huffman 码表，一共有 32 个 huffman 码表可供选择，在 mp3 官方协议**错误！未找到引用源。**AnnexB Table 3-B.7 中给出了这 32 个表。

- 在小值区，所用 huffman 表的选择信息由 count1table_select[gr][ch]提供。需要说明的是，小值区的长度是 count1*4，虽然在 side information 中并没有 count1，但解码程序知道在耗尽 part2_3_length[gr][ch]长度的码流之后就可以判断已经达到了小值区的末尾。

2.3.3. huffman码表的特点

大值区的 huffuman 表一个入口项可得到两个值，小值区的 huffman 表一个入口项可得到四个值。大值区的 huffman 表有一个参数为 linbits(见 2.1.7)。当 linbit 为 0 时，该 huffman 表只能用来编码小于等于 15 的数。当 linbit 不为 0 时，该 huffman 表可用来编码值大于 15 的数，当用这样的 huffman 码表编码时，在 hcod[|x|][|y|]之后的码流中有 linbit 位，这长度为 linbit 的位串表示无符号整数，它与 x(或者 y)相加后表示 x(或者 y)真正的编码值。

2.3.4. 缩放因子 (scalefactor)

如前所述，一个缩放因子频带内的频率线在反量化时共用缩放因子，在码流中，缩放因子被编码于 main_data 中(见 2.1.6)。要解码得到缩放因子，首先需要知道该缩放因子所占的比特数，在 side information 中 scale_compress[gr][ch]提供了这样的信息，首先需要查找如下表格：

表格 2-9 scale_compress

scale_compress	slen1	slen2
0	0	0
1	0	1
2	0	2
3	0	3
4	3	0
5	1	1
6	1	2
7	1	3
8	2	1
9	2	2

10	2	3
11	3	1
12	3	2
13	3	3
14	4	2
15	4	3

下面针对不同的节（块）类型说明 slen1 和 slen2 的含义。

u 该节为长块

slen1 表示缩放因子频带 0 到 10 所用缩放因子的长度；slen2 表示缩放因子频带 11 到 20 所用缩放因子的长度。

u 该节为短块

slen1 表示缩放因子频带 0 到 5 所用缩放因子的长度；slen2 表示缩放因子频带 6 到 11 所用缩放因子的长度。

u 该节为混合块

在这种情况下，长块部分（sfb0 到 sfb7）和短块部分（sfb3 到 sfb5）所用的缩放因子的长度相同，为 slen1；短块部分（sfb6 到 sfb11）所用的缩放因子长度相同，为 slen2。

为了进一步地减少 mp3 码流的大小，节 1（granule1）有时会共用节 0(granule 0)的缩放因子信息，是否共用由字段 scfsi[scfsi_band]来决定，如下所示：

表格 2-10 scfsi

scfsi[scfsi_band]	
0	scalefactors are transmitted for each granule
1	scalefactors transmitted for granule 0 are also valid for granule 1

表格 2-11 scfsi_band

scfsi_band	scalefactor bands (see Annex B,Table3-B.8)
0	0,1,2,3,4,5
1	6,7,8,9,10
2	11…15
3	16…20

只有 granule1 的长块才可以共用前一节的缩放因子信息，对于短块来说，scfsi 为 0。知道缩放因子的共用就不难理解在 2.1.6（Main_data）中位流的组织形式。

2.3.5. 节的长短块切换

节的类型由 block_type 来定义,如果 window_switching_flag 未置位，那么 block_type 的值为 0。如果 window_switching_flag 置位，那么 block_type 由字段 block_type[gr][ch]给出，

如下所示：

表格 2-12　block_type

block_type[gr]	
0	reserved
1	start block
2	3 short windows
3	end block

当 block_type 为 0、1、3 时，该节属于长块。当 block_type 为 2 时，如果 mixed_block_flag[gr][ch]为 0，则该节为短块；mixed_block_flag[gr][ch]为 1，则该节为混合块。长块和短块在解码时算法有较大区别。

2.4. mp3 解码具体流程

MP3 解码的流程如下所示，解码的主要过程包括 Preprocessing、Huffman decoding、Requantization、Reordering、Stereo decoding、Alias reduction、IMDCT、Frequency inversion、Synthesis filter bank，最后输出原始的 PCM 数据。

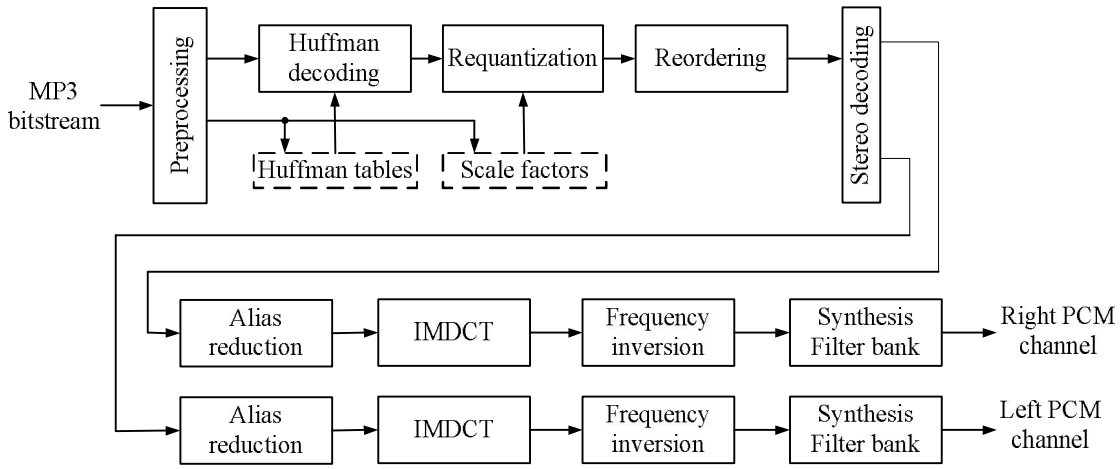

图 2-3　解码流程图

2.4.1.　预处理（Preprocessing）

这个步骤主要是完成 Header 和 Side information 的解码，得到后面解码所需要的一些信息，并保存起来。

2.4.2.　Huffman decoding

在 2.3.1（子带和缩放因子频带）和 2.3.2（Huffman 码表的选择）中已经详细介绍了，在这里就不再重叙。

2.4.3. 反量化（Requantization）

经过 Huffman 解码之后的值必须经过反量化的处理，反量化过程根据使用的 windows 使用不同的反量化运算公式，其反量化的公式如下：

u 短块：

$$xr_i = sign(is_i) * |is_i|^{\frac{4}{3}} * 2^{\frac{1}{4}(global\_gain[gr] - 210 - 8 * subblock\_gain[window][gr])} * 2^{-(scalefac\_multiplier * scalefac\_s[gr][ch][sfb][window])}$$

公式 2-1

u 长块：

$$xr_i = sign(is_i) * |is_i|^{\frac{4}{3}} * 2^{\frac{1}{4}(global\_gain[gr] - 210)} * 2^{-(scalefac\_multiplier * (scalefac\_l[sfb][ch][gr] + preflag[gr] * pretab[sfb]))}$$

公式 2-2

u 混合块：

对于短块部分，按公式 2-1 反量化；对于长块部分，按公式 2-2 反量化。参考（图 2-1 不同情况下 huffman 解码得到的 576 个值的含义）。

isi 表示是第 i 个完成 Huffman decoding 的值，先将该值开 4/3 次方，这步一般通过查表完成。global_gain 及 preflag 的值可以从 side information 中得到，当 side information 中的 scalefac_scale＝0 时，scalefac_multiplier＝0.5，scalefac_scale＝1 时则 scalefac_multiplier＝1，scalefac_l 及 scalefac_s 为从 scale_factor 所解出来的量化因子的值，preflag 则是 MP3 标准中规定中所设定的常数值，而 210 则是系统中需要用来衡量的一个标准值。

2.4.4. 重排序（Reordering）

MP3 编码器为了使 Huffman 编码更加有效率，对短块和混合块中的短块部分进行了 Reordering，因此解码器要按照这个 Reordering 的方法 Reverse Reordering。需注意，该步骤只作用于短块和混合块中的短块部分。

2.4.5. 立体声解码（Stereo decoding）

在这里，假设两个声道独立编解码，不对立体声进行解释。有关立体声处理的详细信息，请参考 mp3 官方协议。

2.4.6. 混叠消除（Alias reduction）

为了避免相邻的两个子频带之间的混迭，在编码和解码中都需要进行 alias reduction 去混迭的处理，这个运算可以看成是对任意两个相邻子频带连续做 8 次的 butterfly 的运算，该运算只对长块和混合块中的长块部分使用。具体算法如下：

```
for(sb=1;sb<32;sb++)
    for(i=0;i<8;i++){
xar[18*sb-1-i]=xr[18*sb-1-i]Cs[i]-xr[18*sb+i]Ca[i]
xar[18*sb+i]=xr[18*sb+i]Cs[i]+xr[18*sb-1-is]Ca[i]
}
```

更形象一点，用图表示如下：

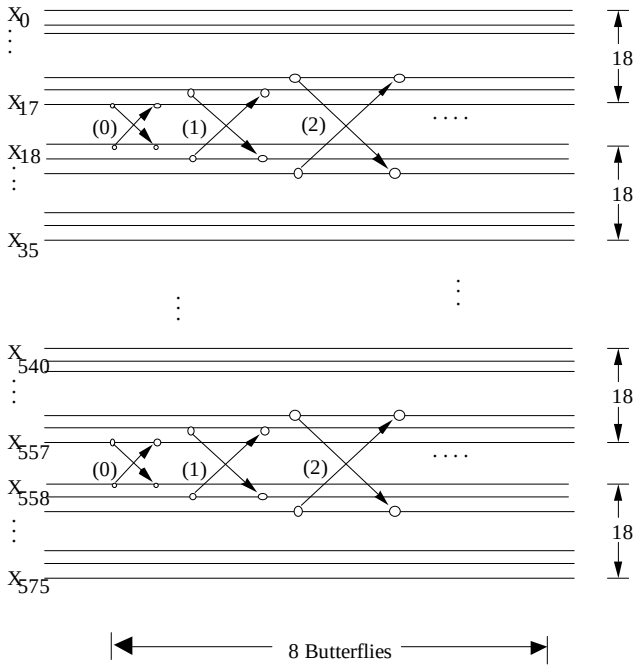

图 2-4 混叠消除示意图

2.4.7. IMDCT变换

对于长块及混合块中的长块部分，IMDCT transform 作用在一个子带的 18 个频率线上，产生 36 个输出；对于短块以及混合块中的短块部分，IMDCT transform 作用在一个子带的 6 个频率线上，产生 12 个输出，连续作用三次，则也产生 36 个输出，IMDCT transform 的公式如下所示，长块时 n 为 36，短块时 n 为 12。

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

做完 IMDCT transform 之后，还要对输出的 36 个值做加窗运算，窗口函数是根据 side information 中的 block_type 决定的，窗口函数与 block_type 之间的关系如下：

- block_type=0 (normal window)

$$z_i = x_i \sin\left(\frac{\pi}{36}\left(i+\frac{1}{2}\right)\right) \quad \text{for i = 0 to 35}$$

- block_type=1 (start block)

$$z_i = \begin{cases} x_i \sin\left(\frac{\pi}{36}\left(i+\frac{1}{2}\right)\right) & \text{for i = 0 to 17} \\ x_i & \text{for i = 18 to 23} \\ x_i \sin\left(\frac{\pi}{12}\left(i-18+\frac{1}{2}\right)\right) & \text{for i = 24 to 29} \\ 0 & \text{for i = 30 to 35} \end{cases}$$

- block_type=3 (stop block)

$$z_i = \begin{cases} 0 & \text{for i = 0 to 5} \\ x_i \sin\left(\frac{\pi}{12}\left(i-6+\frac{1}{2}\right)\right) & \text{for i = 6 to 11} \\ x_i & \text{for i = 12 to 17} \\ x_i \sin\left(\frac{\pi}{36}\left(i+\frac{1}{2}\right)\right) & \text{for i = 18 to 35} \end{cases}$$

- block_type=2 (short block)

$$y_i^{(j)} = x_i^{(j)} \sin\left(\frac{\pi}{12}\left(i+\frac{1}{2}\right)\right) \quad \text{for i = 0 to 11, j = 0 to 2}$$

$$z_i = \begin{cases} 0 & \text{for i = 0 to 5} \\ y_{i-6}^{(1)} & \text{for i = 6 to 11} \\ y_{i-6}^{(1)} + y_{i-12}^{(2)} & \text{for i = 12 to 17} \\ y_{i-12}^{(2)} + y_{i-18}^{(3)} & \text{for i = 18 to 23} \\ y_{i-18}^{(3)} & \text{for i = 24 to 29} \\ 0 & \text{for i = 30 to 35} \end{cases}$$

对于长块，属于前三种情况其中一种；对于短块，属于最后一种情况；对于混合块，长块部分属于第一种情况，短块部分属于最后一种情况。

最后一个步骤是对 32 个子频带中所计算出来的 *Zi* 进行 overlapping，它是将当前子带计算出来的 36 个值的低 18 个值与前一个块相应子带计算出来的 36 个值的高 18 个值进行重叠相加，当前子带的后 18 个值被保存起来，用在下一个块中。公式如下：

$$\begin{aligned} \text{result}_i &= z_i + s_i && \text{for i = 0 to 17} \\ s_i &= z_{i+18} && \text{for i = 0 to 17} \end{aligned}$$

在进行 IMDCT 变换之后，不再有长块、短块的概念，只需知道得到的 576 个值从低到高分为 32 个子带，每个子带 18 个值。

2.4.8. 子带合成滤波 (Synthesis filter bank)

这部分是 MP3 解码的最后一个部分了，它负责从 IMDCT 的输出值中把 PCM 值还原出来，它可以分成五个步骤，首先是 Matrixing 运算，它从 32 个子带的每个子带中取出一个值组成 32 个值送入一个矩阵中进行运算，然后把输出的 64 个结果放入一个 1024 的先入先出(FIFO)的缓存中，接着从 1024 值中取出一半，组成一个 512 矢量，并对这 512 矢量进行加

窗运算，加窗系数 Di 由 mp3 官方协议 AnnexB Table3-B.3 提供。最后将加窗结果进行叠加生成 32 个时域 pcm 输出，具体流程如下图所示，其中

$$N_{ik} = \cos[\frac{(2k+1)(i+16)\pi}{64}]$$

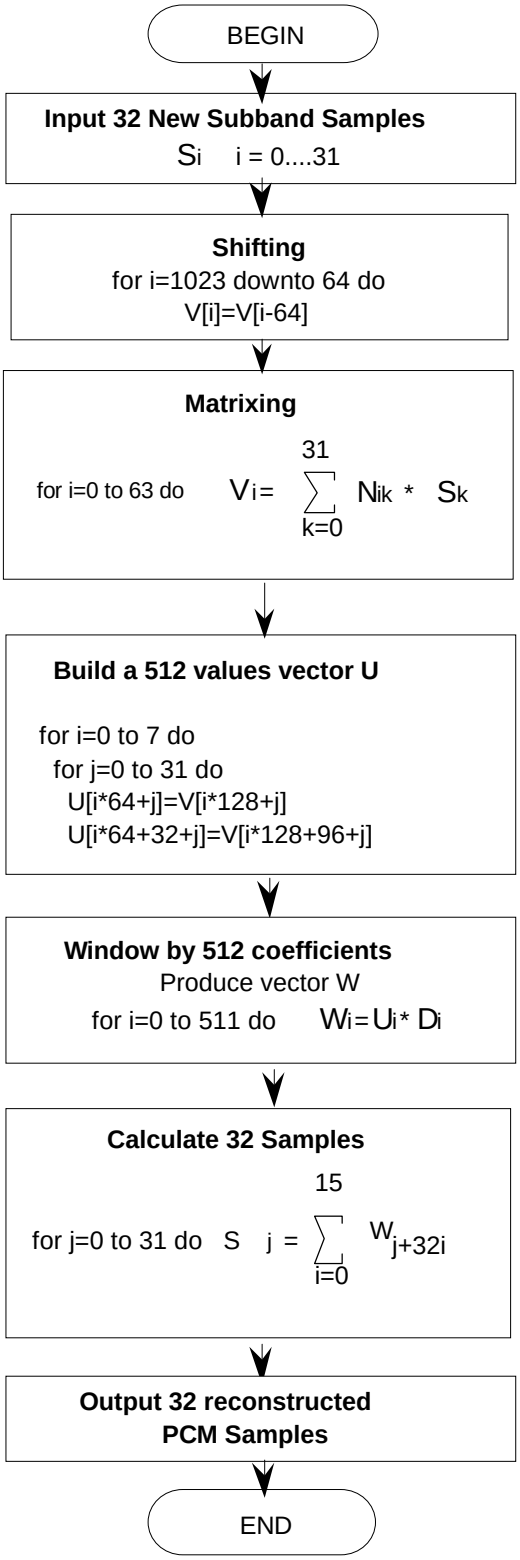

图 2-5 子带合成滤波器流程图

第3章 libmad 解码程序源代码分析

libmad 是专门面向嵌入式应用的 mp3 解码程序，它用定点运算模拟浮点运算，因此不需要处理器有浮点运算功能。libmad 对 mp3 解码中关键部分采用了优化的算法，这些优化算法能够大幅度地减少计算量，而且大多应用于 mp3 解码的 VLSI 实现中。由于以上的特点，libmad 非常适用于嵌入式应用。libmad 的版权归属于 Underbit Technologies, Inc.

libmad 包含的源文件主要有：bit.c、stream.c、decoder.c、frame.c、layer3.c、synth.c、huffman.c。下面按照解码的顺序对源代码分模块说明。

3.1. 码流读取

解码程序的输入就是二进制码流，因此码流读取是很重要、很基础的功能模块。码流的读取是以比特为单位的，而 cpu 读写内存是以字节为单位的，故在两者之间需由相关函数架起桥梁。这个函数就是 mad_bit_read( )，定义于 bit.c：

```
137 unsigned long mad_bit_read(struct mad_bitptr *bitptr, unsigned int len)
138 {
139   register unsigned long value;
140
141   if (bitptr->left == CHAR_BIT)
142     bitptr->cache = *bitptr->byte;
143
144   if (len < bitptr->left) {
145     value = (bitptr->cache & ((1 << bitptr->left) - 1)) >>
146       (bitptr->left - len);
147     bitptr->left -= len;
148
149     return value;
150   }
151
152   /* remaining bits in current byte */
153
154   value = bitptr->cache & ((1 << bitptr->left) - 1);
155   len   -= bitptr->left;
156
157   bitptr->byte++;
158   bitptr->left = CHAR_BIT;
159
160   /* more bytes */
161
162   while (len >= CHAR_BIT) {
```

```
163       value = (value << CHAR_BIT) | *bitptr->byte++;
164       len   -= CHAR_BIT;
165     }
166
167     if (len > 0) {
168       bitptr->cache = *bitptr->byte;
169
170       value = (value << len) | (bitptr->cache >> (CHAR_BIT - len));
171       bitptr->left -= len;
172     }
173
174     return value;
175}
```

以上涉及到一个数据结构 struct mad_bitptr：

```
struct mad_bitptr {
  unsigned char const *byte;
  unsigned short cache;
  unsigned short left;
};
```

结构体 mad_bitptr 指向码流中的要一个要读取的比特，该比特所在的字节由 byte 确定，该比特在字节中的位置由 left 确定，如果 left 为 8，则该比特为（*byte）的最高为（MSB）。另外，成员 cache 为字节数据缓冲，也就是(*byte)。

我们看函数 mad_bit_read（）的第 144 行，((1<<bitptr->left)-1)表示一个蒙板,一个 8 位的数与它相与可表示低 left 位的值,将这个值右移((bitptr->left – len)位，即为要读取的若干比特位的值。函数的第 170 行也是同样的原理，只是要读取的比特位较长，涉及到多个，读者可自行揣摩。

另外，在 bit.c 中，还有若干码流处理函数：mad_bit_bitsleft（）、mad_bit_nextbyte（）、mad_bit_length（）、mad_bit_skip（）。

3.2.桢的同步

每一桢的开头有同步字，即 syncword,为 12 个比特位（0xFFF），用以标明一桢的开始，但通过在码流中查找同步字来定位桢是很费事的，因此 mp3 解码采用了另一种快速的定位手段。在 mp3 码流中，相邻两个同步字之间的距离（也就是一桢的长度）是 slot 的整数倍，其中 slot 就是一个字节。一桢的长度要么是 N 个字节，要么是 N+1 字节。其中 N 这样计算：

$$N=144*\frac{bitrate}{sampling\_frequency}$$

上式计算出来的 N 如果不是整数，应该被截短（truncated）。当桢头中的 Padding_bit 为

1 时，桢长为 N+1；当 Padding_bit 为 0 时，桢长为 N。

桢的同步通过函数 mad_header_decode（）完成，定义于 frame.c：

[mad_frame_decode（） à mad_header_decode（）]

```
300 int mad_header_decode(struct mad_header *header, struct mad_stream *stream)
301 {
302     register unsigned char const *ptr, *end;
303     unsigned int pad_slot, N;
304
305     ptr = stream->next_frame;
306     end = stream->bufend;
307
……

364    /* begin processing */
365     stream->this_frame = ptr;
366     stream->next_frame = ptr + 1;    /* possibly bogus sync word */
367
368     mad_bit_init(&stream->ptr, stream->this_frame);
369
370     if (decode_header(header, stream) == -1)
371         goto fail;

……

377     /* calculate free bit rate */
378     if (header->bitrate == 0) {
379         if ((stream->freerate == 0 || !stream->sync ||
380           (header->layer == MAD_LAYER_III && stream->freerate > 640000)) &&
381         free_bitrate(stream, header) == -1)
382             goto fail;
383
384         header->bitrate = stream->freerate;
385         header->flags    |= MAD_FLAG_FREEFORMAT;
386     }
387
388     /* calculate beginning of next frame */
389     pad_slot = (header->flags & MAD_FLAG_PADDING) ? 1 : 0;
390
391     if (header->layer == MAD_LAYER_I)
392         N = ((12 * header->bitrate / header->samplerate) + pad_slot) * 4;
393     else {
394         unsigned int slots_per_frame;
395
```

```
396         slots_per_frame = (header->layer == MAD_LAYER_III &&
397                            (header->flags & MAD_FLAG_LSF_EXT)) ? 72 : 144;
398
399         N = (slots_per_frame * header->bitrate / header->samplerate) + pad_slot;
400       }

……

410       stream->next_frame = stream->this_frame + N;

……

426       return 0;
……
}
```

第 305 行从 stream 结构体中获取下一个需要解码的桢的首地址，然后在 365 行传递给 stream->this_frame,于是从前一桢过渡到它的下一桢。在得到桢长 N 之后，更新 stream->next_frame（410 行），为下一桢的解码做好准备。

另外，结构体 stream 定义如下：

```
struct mad_stream {
  unsigned char const *buffer;          /* input bitstream buffer */
  unsigned char const *bufend;          /* end of buffer */
  unsigned long skiplen;                /* bytes to skip before next frame */

  int sync;                             /* stream sync found */
  unsigned long freerate;               /* free bitrate (fixed) */

  unsigned char const *this_frame;      /* start of current frame */
  unsigned char const *next_frame;      /* start of next frame */
  struct mad_bitptr ptr;                /* current processing bit pointer */

  struct mad_bitptr anc_ptr;            /* ancillary bits pointer */
  unsigned int anc_bitlen;              /* number of ancillary bits */

  unsigned char (*main_data)[MAD_BUFFER_MDLEN];
                                        /* Layer III main_data() */
  unsigned int md_len;                  /* bytes in main_data */

  int options;                          /* decoding options (see below) */
  enum mad_error error;                 /* error code (see above) */
};
```

关于 stream 结构体中一些成员的含义在 3.5 节(main_data 的读取)中解释。

3.3.桢头解码

桢头解码有函数 decode_header（）完成，定义于 frame.c：

[mad_header_decode（） à decode_header（）]

```
static
int decode_header(struct mad_header *header, struct mad_stream *stream)
{
  unsigned int index;

  header->flags        = 0;
  header->private_bits = 0;

  /* header() */

  /* syncword */
  mad_bit_skip(&stream->ptr, 11);

  /* MPEG 2.5 indicator (really part of syncword) */
  if (mad_bit_read(&stream->ptr, 1) == 0)
    header->flags |= MAD_FLAG_MPEG_2_5_EXT;

  /* ID */
  if (mad_bit_read(&stream->ptr, 1) == 0)
    header->flags |= MAD_FLAG_LSF_EXT;
  else if (header->flags & MAD_FLAG_MPEG_2_5_EXT) {
    stream->error = MAD_ERROR_LOSTSYNC;
    return -1;
  }

……

}
```

桢头解码所得到的信息（ID、Layer、Bitrate、Samping_frequency……）存放于结构体 mad_header 中，定义如下：

```
struct mad_header {
  enum mad_layer layer;          /* audio layer (1, 2, or 3) */
  enum mad_mode mode;            /* channel mode (see above) */
  int mode_extension;            /* additional mode info */
  enum mad_emphasis emphasis;    /* de-emphasis to use (see above) */
```

```
  unsigned long bitrate;        /* stream bitrate (bps) */
  unsigned int samplerate;      /* sampling frequency (Hz) */

  unsigned short crc_check;     /* frame CRC accumulator */
  unsigned short crc_target;    /* final target CRC checksum */

  int flags;                    /* flags (see below) */
  int private_bits;             /* private bits (see below) */

  mad_timer_t duration;         /* audio playing time of frame */
};
```

所要解码的 mp3 文件包含多少桢，decode_header（）就被执行多少次。

3.4. sideinfo 解码

sideinfo 的解码由函数 III_sideinfo（）完成，定义于 layer3.c，知道 sideinfo 的结构，阅读该函数不会有什么困难，我们把它留给读者。sideinfo 解码所得到的信息（main_data_begin、scfsi、part2_3_length、big_values……）存放于结构体 sideinfo 中，定义如下：

```
struct sideinfo {
  unsigned int main_data_begin;
  unsigned int private_bits;

  unsigned char scfsi[2];

  struct granule {
    struct channel {
      /* from side info */
      unsigned short part2_3_length;
      unsigned short big_values;
      unsigned short global_gain;
      unsigned short scalefac_compress;

      unsigned char flags;
      unsigned char block_type;
      unsigned char table_select[3];
      unsigned char subblock_gain[3];
      unsigned char region0_count;
      unsigned char region1_count;

      /* from main_data */
      unsigned char scalefac[39];/* scalefac_l and/or scalefac_s */
    } ch[2];
```

} gr[2];
};

其中，数据成员 scalefac[39]表示缩放因子频带的缩放因子，对于长块，一共有 22 个缩放因子频带（0-20 的缩放因子从码流中获取，21 的缩放因子为 0）。对于短块，在码流的 sideinfo 部分得不到；对于短块，一共有 13 个缩放因子频带（0-11 的缩放因子从码流中获取，12 的缩放因子为 0），其中每个频带有 3 个窗，分别用不同的缩放因子，因此，scalefac[ ]数组的大小为 39（13*3）。缩放因子不存放于 sideinfo 中，而是存放于 main_data 中，后面在对 main_data 解码时，会把缩放因子填入数组 scalefac[39]。

3.5. main_data 的读取

main_data 包含缩放因子和 huffman 编码数据。mp3 编码时，并没有将 main_data 全部存放于当前桢里，而是 main_data 的开头有一部分存放于前一桢里，并且位于当前桢的同步字之前，sideinfo 中 main_data_begin 表示这一部分的字节个数，如下图所示：

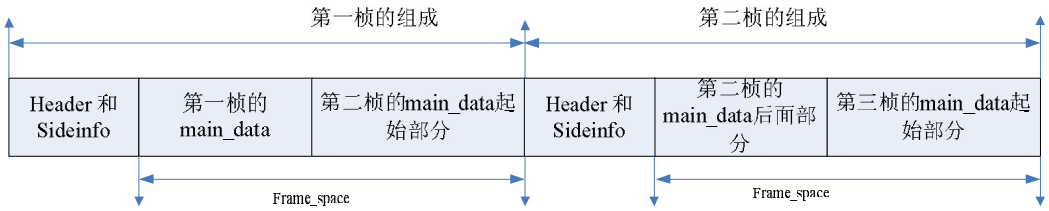

图 3-1 main_data 的组成

main_data 的读取由函数 mad_layer_III（）完成，定义于 layer3.c：

[mad_frame_decode（） à mad_layer_III（）]
2516 int mad_layer_III(struct mad_stream *stream, struct mad_frame *frame)
2517 {

……

/* decode frame side information */

2573 error = III_sideinfo(&stream->ptr, nch, header->flags & MAD_FLAG_LSF_EXT, &si, &data_bitlen, &priv_bitlen);

……

/* find main_data of next frame */

{
struct mad_bitptr peek;
unsigned long header;

```
      mad_bit_init(&peek, stream->next_frame);

      header = mad_bit_read(&peek, 32);
      if ((header & 0xffe60000L) /* syncword | layer */ == 0xffe20000L) {
        if (!(header & 0x00010000L))    /* protection_bit */
      mad_bit_skip(&peek, 16);    /* crc_check */

2596            next_md_begin =
      mad_bit_read(&peek, (header & 0x00080000L) /* ID */ ? 9 : 8);
      }

      mad_bit_finish(&peek);
    }

    /* find main_data of this frame */

2605    frame_space = stream->next_frame - mad_bit_nextbyte(&stream->ptr);

……

2610      md_len = si.main_data_begin + frame_space - next_md_begin;

2612      frame_used = 0;

2614      if (si.main_data_begin == 0) {
2615        ptr = stream->ptr;
2616        stream->md_len = 0;
2617
2618        frame_used = md_len;
2619      }
    else {

2628            mad_bit_init(&ptr,
                *stream->main_data + stream->md_len - si.main_data_begin);

2631            if (md_len > si.main_data_begin) {
2632        assert(stream->md_len + md_len -
2633                    si.main_data_begin <= MAD_BUFFER_MDLEN);

2635        memcpy(*stream->main_data + stream->md_len,
                mad_bit_nextbyte(&stream->ptr),
                frame_used = md_len - si.main_data_begin);
2638        stream->md_len += frame_used;
          }
```

```
  }
2643    frame_free = frame_space - frame_used;

  /* decode main_data */

  if (result == 0) {
2648      error = III_decode(&ptr, frame, &si, nch);
  }

……
  /* preload main_data buffer with up to 511 bytes for next frame(s) */

2670    if (frame_free >= next_md_begin) {
2671      memcpy(*stream->main_data,
2672        stream->next_frame - next_md_begin, next_md_begin);
2673      stream->md_len = next_md_begin;
  }

……

2697    return result;
}
```

在进入 mad_layer_III（）函数之前，桢头信息也被解码出来，从桢头中知道是 layerIII, 因此调用 mad_layer_III（）,而不是 mad_layer_I（）或 mad_layer_II（）。因此，在 mad_layer_III 中，需要 sideinfo 解码（2573 行）。

要读取属于当前桢的 main_data，需要知道 next_md_begin（2596）和 frame_space（2605），两者之差即为属于当前桢的 main_data 部分，也就是 frame_used，而 frame_free 等于 next_md_begin。特别地，当为 mp3 文件的第一桢时，即满足 2614 行的条件，ptr 指向第一桢的 main_data,然后调用 III_decode（）（2648 行），须注意，并没有把第一桢的 main_data 存放于(*stream->main_data)[ ]中；当为 mp3 文件的后续桢时（不是第一桢），则需把 main_data 存放于(*stream->main_data)[ ]，然后使 ptr 指向这个数组的起始位置（2628 行），再调用 III_decode（）。

mad_stream 结构体中的成员 md_len 表示已经读取的当前桢的 main_data 的字节个数，当前一桢被解码时获取了 next_md_begin（2673 行），然后对该成员赋这个初值，然后在当前桢被解码时获取了 frame_used，把它加到成员 md_len 中（2638），此时属于当前桢的 main_data 全部得到。

3.6. 缩放因子解码

读取缩放因子是通过函数 III_scalefactors（）完成的，定义于 layer3.c：

[mad_layer_III（） à III_decode（） à III_scalefactors（）]

```
713 static
714 unsigned int III_scalefactors(struct mad_bitptr *ptr, struct channel *channel,
715                               struct channel const *gr0ch, unsigned int scfsi)
716 {
717     struct mad_bitptr start;
718     unsigned int slen1, slen2, sfbi;
719
720     start = *ptr;
721
722    slen1 = sflen_table[channel->scalefac_compress].slen1;
723    slen2 = sflen_table[channel->scalefac_compress].slen2;
724
725     if (channel->block_type == 2) {
726         unsigned int nsfb;
727
728         sfbi = 0;
729
730         nsfb = (channel->flags & mixed_block_flag) ? 8 + 3 * 3 : 6 * 3;
731         while (nsfb--)
732             channel->scalefac[sfbi++] = mad_bit_read(ptr, slen1);
733
734         nsfb = 6 * 3;
735         while (nsfb--)
736             channel->scalefac[sfbi++] = mad_bit_read(ptr, slen2);
737
738         nsfb = 1 * 3;
739         while (nsfb--)
740             channel->scalefac[sfbi++] = 0;
741     }
742     else {    /* channel->block_type != 2 */
743         if (scfsi & 0x8) {
744             for (sfbi = 0; sfbi < 6; ++sfbi)
745         channel->scalefac[sfbi] = gr0ch->scalefac[sfbi];
746         }
747         else {
748             for (sfbi = 0; sfbi < 6; ++sfbi)
749         channel->scalefac[sfbi] = mad_bit_read(ptr, slen1);
750         }

        ……

779       channel->scalefac[21] = 0;
780     }
781
```

```
782     return mad_bit_length(&start, ptr);
783 }
```

参数 channel 指向当前要读取缩放因子的块，gr0ch 指向一桢中第一节对应的块。如前所述，如果当前块为一桢中第二节对应的块，则它可能共用 gr0ch 中的缩放因子，如第 743 行所示；如果当前块为第一节所对应的块，则 scfsi 必然为 0，因此从码流中读取缩放因子，如第 748 行所示。

3.7. huffman 解码

huffman 解码是比较复杂的一部分，它涉及到 sideinfo 中的 big_values、region0_count、region1_count、table_select、part2_3_length、count1table_select。mp3 在进行 huffman 编码时，对 576 个采样点分为若干区域，对不同的区域采用不同的 huffman 码表，为了行文方便，对采用同一 huffman 码表的区域称为码表区域，需要注意，码表区域的划分与缩放因子的划分并不完全一致。下图给出了某一块的码表区域划分，图中缩放因子频带的划分对应采样率为 44.1khz 的长块。

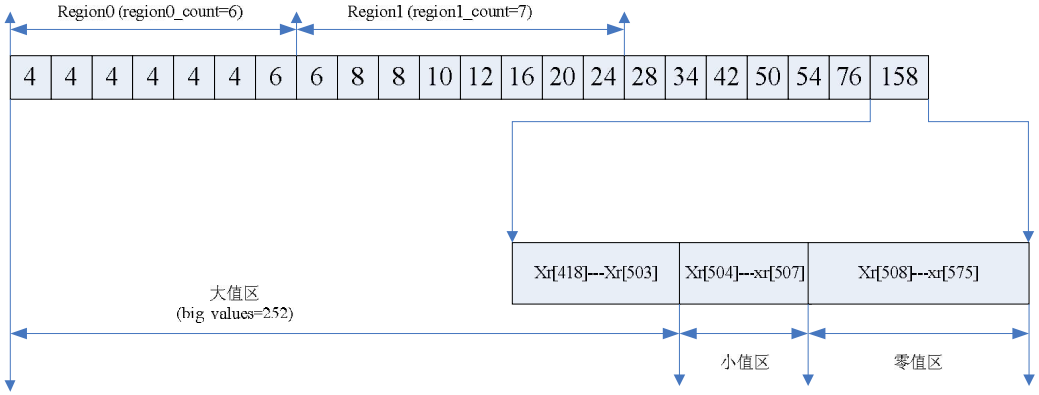

图 3-2 码表区域的划分

对 huffman 数据的解码由函数 III_huffdecode（）完成，定义于 layer3.c：

```
[mad_layer_III（）  à III_decode（） à III_huffdecode（）]
static
enum mad_error III_huffdecode(struct mad_bitptr *ptr, mad_fixed_t xr[576],
                              struct channel *channel,
                              unsigned char const *sfbwidth,
                              unsigned int part2_length)
{

946     bits_left = (signed) channel->part2_3_length - (signed) part2_length;
   if (bits_left < 0)
      return MAD_ERROR_BADPART3LEN;


   /* align bit reads to byte boundaries */
956     cachesz    = mad_bit_bitsleft(&peek);
```

```
957     cachesz += ((32 - 1 - 24) + (24 - cachesz)) & ~7;

959     bitcache      = mad_bit_read(&peek, cachesz);
960     bits_left -= cachesz;

962     xrptr = &xr[0];

  /* big_values */
  {
972       sfbound = xrptr + *sfbwidth++;
973       rcount    = channel->region0_count + 1;

975       entry         = &mad_huff_pair_table[channel->table_select[region = 0]];

987       big_values = channel->big_values;

989       while (big_values-- && cachesz + bits_left > 0) {

994         if (xrptr == sfbound) {
995         sfbound += *sfbwidth++;

      /* change table if region boundary */
999         if (--rcount == 0) {
1000          if (region == 0)
1001              rcount = channel->region1_count + 1;
1002          else
1003              rcount = 0;    /* all remaining */

1005          entry         = &mad_huff_pair_table[channel->table_select[++region]];
    }
  }

1034          pair       = &table[MASK(bitcache, cachesz, clumpsz)];

       /* x (0..1) */
1119 value = pair->value.x;

1131 xrptr[0] = MASK1BIT(bitcache, cachesz--) ?
            -requantized : requantized;

      /* y (0..1) */
1137 value = pair->value.y;

1149 xrptr[1] = MASK1BIT(bitcache, cachesz--) ?
```

```
            -requantized : requantized;

1154            xrptr += 2;
      }
    }

    /* count1 */
    {
1166        table = mad_huff_quad_table[channel->flags & count1table_select];

1170        while (cachesz + bits_left > 0 && xrptr <= &xr[572]) {
          /* v (0..1) */
1206            xrptr[0] = quad->value.v ?
        (MASK1BIT(bitcache, cachesz--) ? -requantized : requantized) : 0;

          /* w (0..1) */
1211            xrptr[1] = quad->value.w ?
        (MASK1BIT(bitcache, cachesz--) ? -requantized : requantized) : 0;

1214            xrptr += 2;

          /* x (0..1) */
1229            xrptr[0] = quad->value.x ?
        (MASK1BIT(bitcache, cachesz--) ? -requantized : requantized) : 0;

          /* y (0..1) */
1234            xrptr[1] = quad->value.y ?
        (MASK1BIT(bitcache, cachesz--) ? -requantized : requantized) : 0;

1237            xrptr += 2;
      }
    }

    /* rzero */
1263    while (xrptr < &xr[576]) {
1264        xrptr[0] = 0;
1265        xrptr[1] = 0;
1267        xrptr += 2;
    }
1270    return MAD_ERROR_NONE;
}
```

在上述代码中，xrptr 指向下一个要从 huffman 码流中得到的值，xrptr 以 2 为单位递增（1154 行、1214 行、1237 行），递增之后，要判断是否到达了缩放因子频带的边界（994

行），从而进一步判断是否需要换表（999 行）。当 xrptr 到达大值区的边界时（989 行），就转向小值区，需要换表（1166）。小值区的长度在 sideinfo 中并没有直接给出，但当耗尽了 main_data 时（比特数为 part2_3_length）,就意味着小值区的结束，程序中第 946 行和 1170 行可以反映这一点。当小值区结束之后，就进入零值区，直接给采样点赋零值即可。

有一点需要说明，在大值区，由 region0_count、region1_count、big_values 共同决定大值区码表区域的划分，但有些情况下，三者表达的信息并不一致。当发生窗类型切换时（window_switching_flag 为 1），region1_count=36，这个值过大，并不表示具体的含义，只是意味着把大值区划分了两个区域（由 region0_count 和 bigvalues 决定划分方式），只用两种码表。当然，这种情况下，仍然会有小值区和零值区。

3.8.反量化（requantization）

反量化由两个函数完成：III_requantize（）、III_exponents（），定义于 layer3.c：

```
[III_huffdecode（） à III_exponents（）]
static
void III_exponents(struct channel const *channel,
                   unsigned char const *sfbwidth, signed int exponents[39])
{
  signed int gain;
  unsigned int scalefac_multiplier, sfbi;

  gain = (signed int) channel->global_gain - 210;
  scalefac_multiplier = (channel->flags & scalefac_scale) ? 2 : 1;

  if (channel->block_type == 2) {

    /* this is probably wrong for 8000 Hz short/mixed blocks */

    gain0 = gain - 8 * (signed int) channel->subblock_gain[0];
    gain1 = gain - 8 * (signed int) channel->subblock_gain[1];
    gain2 = gain - 8 * (signed int) channel->subblock_gain[2];

    while (l < 576) {
      exponents[sfbi + 0] = gain0 -
    (signed int) (channel->scalefac[sfbi + 0] << scalefac_multiplier);
      exponents[sfbi + 1] = gain1 -
    (signed int) (channel->scalefac[sfbi + 1] << scalefac_multiplier);
      exponents[sfbi + 2] = gain2 -
    (signed int) (channel->scalefac[sfbi + 2] << scalefac_multiplier);

      l      += 3 * sfbwidth[sfbi];
      sfbi += 3;
    }
```

```
    }
    else {    /* channel->block_type != 2 */
          for (sfbi = 0; sfbi < 22; ++sfbi) {
        exponents[sfbi] = gain -
          (signed int) ((channel->scalefac[sfbi] + pretab[sfbi]) <<
                    scalefac_multiplier);
          }
        ……
    }
}
```

顾名思义，III_exponents（）返回一个指数，从代码中可以看出，该指数恰好是公式 2-1、公式 2-2 中 2 的指数部分乘以 4，可以看到在 III_requantize（）中，该返回值会除以 4。

[III_huffdecode（） à III_requantize（）]

```
static
mad_fixed_t III_requantize(unsigned int value, signed int exp)
{
    mad_fixed_t requantized;
    signed int frac;
    struct fixedfloat const *power;

    frac = exp % 4;    /* assumes sign(frac) == sign(exp) */
    exp /= 4;

892    power = &rq_table[value];
893    requantized = power->mantissa;
894    exp += power->exponent;

……
916            requantized <<= exp;

    return frac ? mad_f_mul(requantized, root_table[3 + frac]) : requantized;
}
```

第 892 行根据 huffman 解码得到的 value（$|is_i|$）查表得到$|is_i|^{4/3}$。具体公式为：

$$x^{4/3} = rq\_table[x].\text{mantissa} * 2^{rq\_table[x].\text{exponent}}$$

第 894 行计算得到以 2 为底的指数部分，然后在 916 行通过移位指令得到最终的反量化值。

3.9. 重排序（reordering）

重排序应用于短块以及混合块的短块部分。首先将所有采样点按 window0、window1、

window2 分类，然后按如下示意图重新排序：

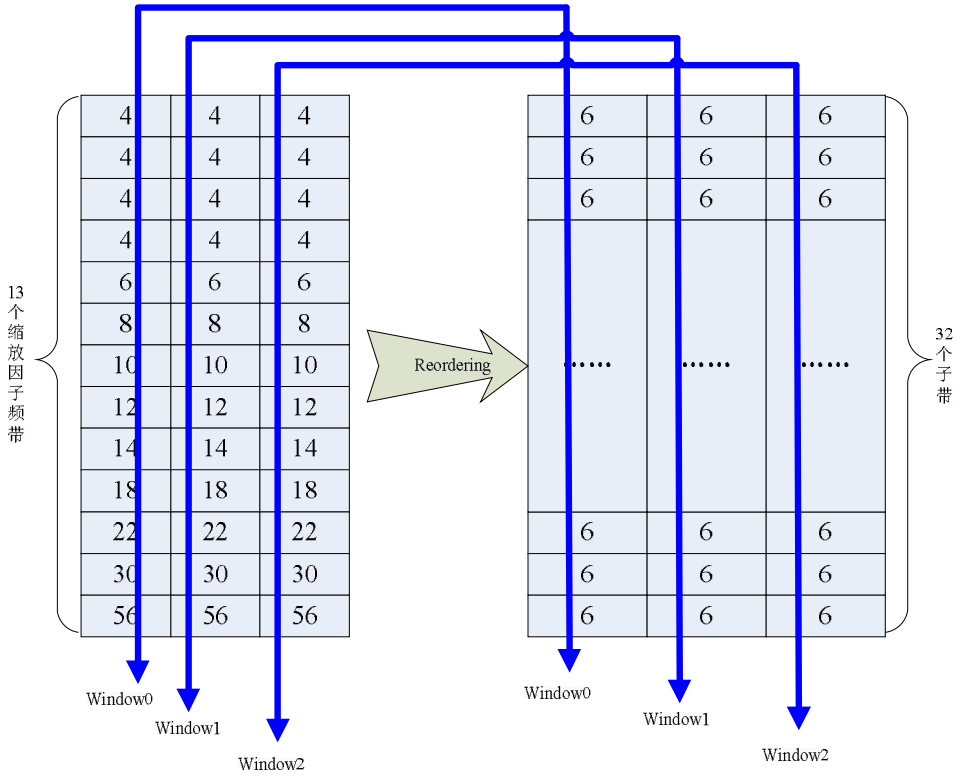

图 3-3

重排序由函数 III_reorder（）完成，定义于 layer3.c：

```
[III_decode（） à III_reorder（）]
static
void III_reorder(mad_fixed_t xr[576], struct channel const *channel,
                 unsigned char const sfbwidth[39])
{
  mad_fixed_t tmp[32][3][6];
  unsigned int sb, l, f, w, sbw[3], sw[3];

  /* this is probably wrong for 8000 Hz mixed blocks */

  sb = 0;
  for (w = 0; w < 3; ++w) {
    sbw[w] = sb;
    sw[w]  = 0;
  }

  f = *sfbwidth++;
  w = 0;

  for (l = 18 * sb; l < 576; ++l) {
```

```
      if (f-- == 0) {
        f = *sfbwidth++ - 1;
1309          w = (w + 1) % 3;
      }

1312      tmp[sbw[w]][w][sw[w]++] = xr[l];

1314      if (sw[w] == 6) {
1315        sw[w] = 0;
1316        ++sbw[w];
1317      }
   }

   memcpy(&xr[18 * sb], &tmp[sb], (576 - 18 * sb) * sizeof(mad_fixed_t));
}
```

第 1312 行完成重排序功能，对每一个窗口（w），有窗口内的计算器 sw[w]（从 0 到 5 变化）和子带计数器 sbw[w]（从 0 到 31），如第 1314 ~第 1316 行所示。w 的变化如第 1309 行所示。

3.10. IMDCT 变换

在 libmad 中，IMDCT 采用 Szu-Wei Lee’s 提出的快速算法。IMDCT 由函数 III_imdct_l（）和 III_imdct_s（）完成，分别对应长块和短块，定义于 layer3.c。读者可对照算法原理自行研究。在完成 IMDCT 之后，需要进行 overlapping。overlapping 将 imdct 输出的 36 个值 z_i 分为两部分，前半部分与上一个块相应子带的后半部分相加，后半部分保存起来用于下一个块的 overlapping。这部分由函数 III_overlap（）完成，定义于 layer3.c：

```
[III_decode（） à III_overlap（）]
static
void III_overlap(mad_fixed_t const output[36], mad_fixed_t overlap[18],
                 mad_fixed_t sample[18][32], unsigned int sb)
{
   unsigned int i;

2264    for (i = 0; i < 18; ++i) {
2265      sample[i][sb] = output[i +    0] + overlap[i];
2266      overlap[i]       = output[i + 18];
   }
}
```

第 2265 行 sample 数组的数据组织形式 sample[i][sb]，i 表示 18 个时域输出，sb 表示 32 个子带，这样是为了方便后续的子带合成滤波。
另外，该函数在 III_decode 被调用时所赋的实参如下：

```
2453 /* long blocks */
2454        for (sb = 0; sb < 2; ++sb, l += 18) {
2455            III_imdct_l(&xr[ch][l], output, block_type);
2456            III_overlap(output, (*frame->overlap)[ch][sb], sample, sb);
2457        }
```

从第 2456 行可以看出，imdct 输出的值是与上一个块的相应子带（(*frame à overlap)[ch][sb]）叠加
其中，各变量的定义如下：

```
2379 mad_fixed_t xr[2][576];
2422 mad_fixed_t output[36];
2420 mad_fixed_t (*sample)[32] = &frame->sbsample[ch][18 * gr];
```

上面涉及到 frame 结构体的定义：

```
struct mad_frame {
  struct mad_header header;         /* MPEG audio header */

  int options;                      /* decoding options (from stream) */

  mad_fixed_t sbsample[2][36][32];    /* synthesis subband filter samples */
  mad_fixed_t (*overlap)[2][32][18];  /* Layer III block overlap data */
};
```

3.11. 子带合成滤波（synthesis filter bank）

子带合成滤波是 mp3 解码中非常耗费时间的关键流程，其中涉及到从 32 个值变换到 64 个值的矩阵运算，对于该矩阵运算，libmad 采用 Konstantinos Konstantinides 提出的方法**错误！未找到引用源。**，该方法将矩阵运算进行一系列变化，最后归于 32 点 dct 变换，而 dct 变换有类似于 FFT 的快速算法（FCT），从该矩阵运算到 32 点的 dct 变换如下图所示：

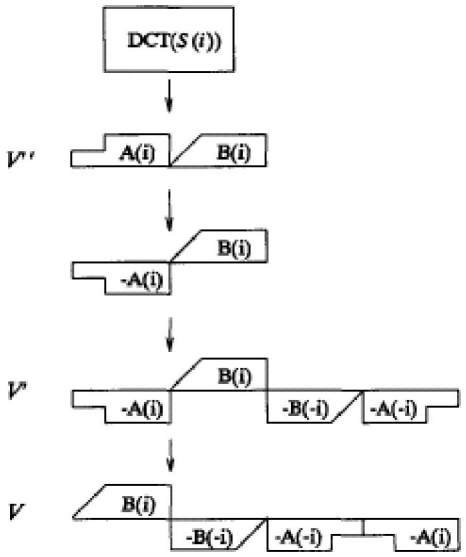

图 3-4 从 dct 到矩阵运算

上图中，V（1*64）表示矩阵的输出，A、B 都是长度为 1*16 的矢量，（A，B）表示 32 点 dct 的输出。

在得到矩阵的输出之后，需将这 64 个值送入到长度为 1024 的 FIFO 中，然后从 FIFO 中取出 512 进行加窗运算，如下图所示：

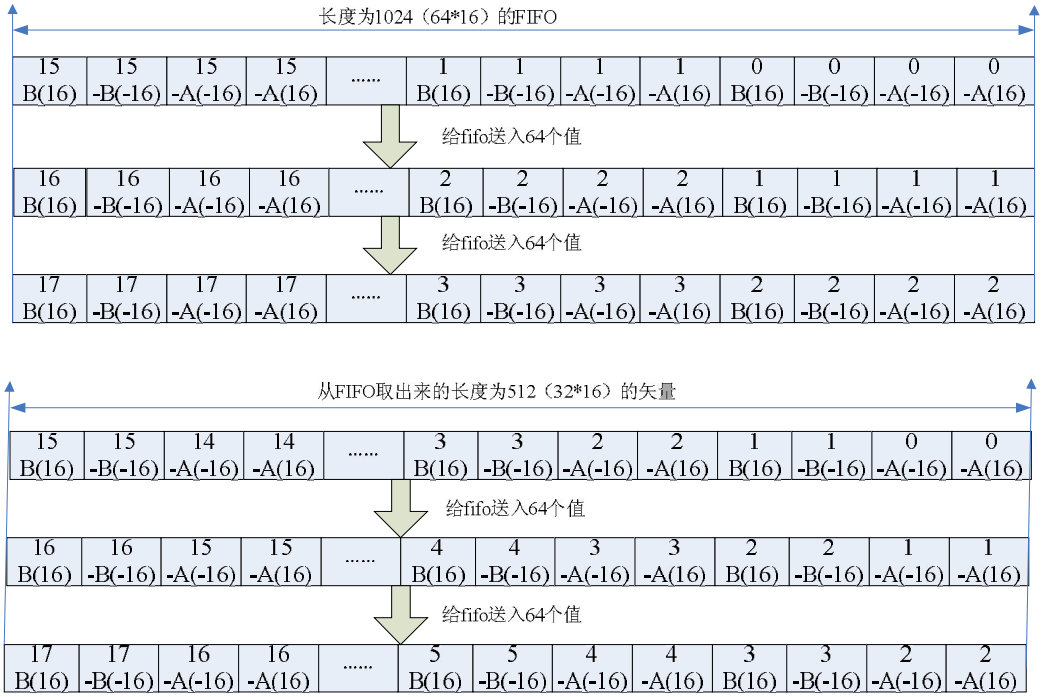

图 3-5 FIFO 和抽取的矢量

为清晰起见，送入的 64 个值按先后顺序被冠以标号（0、1、2、……、16、17、……），仔细观察上图，可知，每当给 fifo 送入 64 个值，取出来的 512 个矢量某个标号的 32 个值要变化一次，观察标号 3，它从 B(16)、-B(16)变化到-A(-16)、-A(16)，再变化到 B(16)、-B(16)。另外，每送入 64 个值，最先进入的标号消失，新的标号产生，且两者标号之差恰好等于 16。上述的这种规律可用于编程之中，从而避免 FIFO 在送入 64 个值时产生的数据搬送。

得到了 512 个矢量之后，如下图所示可得 32 个 pcm 值输出。

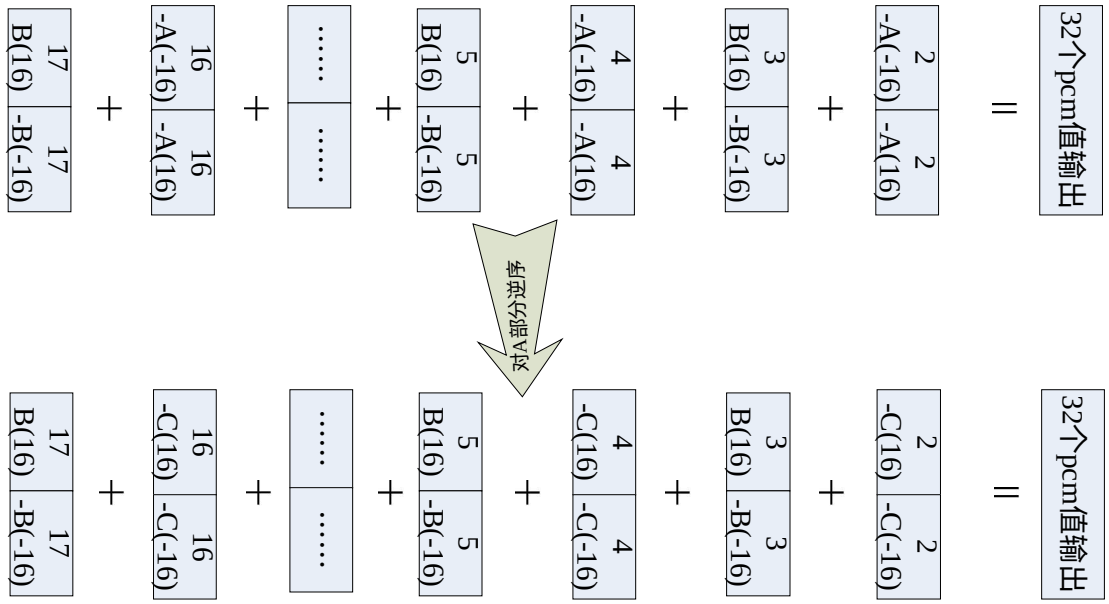

图 3-6 向量相加输出 pcm

其中 C(16)等于 A(-16)，也就是对 A(16)的反转，他们都是长度为 16 的矢量。libmad 中的函数 dct32（）输出的就是 C(16)（hi）和 B（16）(lo)，这一点需要特别注意。libmad 中用于子带滤波的是 synth_full（)，定义于 synth.c：

```
[mad_synth_frame（） à synth_full（）]
static
void synth_full(struct mad_synth *synth, struct mad_frame const *frame,
          unsigned int nch, unsigned int ns)
{
……
569   for (ch = 0; ch < nch; ++ch) {
570     sbsample = &frame->sbsample[ch];
571     filter     = &synth->filter[ch];
572     phase      = synth->phase;
573     pcm1        = synth->pcm.samples[ch];

575     for (s = 0; s < ns; ++s) {
576       dct32((*sbsample)[s], phase >> 1,
577      (*filter)[0][phase & 1], (*filter)[1][phase & 1]);

579       pe = phase & ~1;
580       po = ((phase - 1) & 0xf) | 1;

       /* calculate 32 samples */
584       fe = &(*filter)[0][ phase & 1][0];
585       fx = &(*filter)[0][~phase & 1][0];
```

```
586         fo = &(*filter)[1][~phase & 1][0];

……
613         pcm2 = pcm1 + 30;

615         for (sb = 1; sb < 16; ++sb) {
616       ++fe;
617       ++Dptr;

619  /* D[32 - sb][i] == -D[sb][31 - i] */
      ptr = *Dptr + po;
      ML0(hi, lo, (*fo)[0], ptr[ 0]);
      MLA(hi, lo, (*fo)[1], ptr[14]);
      MLA(hi, lo, (*fo)[2], ptr[12]);
      MLA(hi, lo, (*fo)[3], ptr[10]);
      MLA(hi, lo, (*fo)[4], ptr[ 8]);
      MLA(hi, lo, (*fo)[5], ptr[ 6]);
      MLA(hi, lo, (*fo)[6], ptr[ 4]);
      MLA(hi, lo, (*fo)[7], ptr[ 2]);
      MLN(hi, lo);

      ptr = *Dptr + pe;
      MLA(hi, lo, (*fe)[7], ptr[ 2]);
      MLA(hi, lo, (*fe)[6], ptr[ 4]);
      MLA(hi, lo, (*fe)[5], ptr[ 6]);
      MLA(hi, lo, (*fe)[4], ptr[ 8]);
      MLA(hi, lo, (*fe)[3], ptr[10]);
      MLA(hi, lo, (*fe)[2], ptr[12]);
      MLA(hi, lo, (*fe)[1], ptr[14]);
      MLA(hi, lo, (*fe)[0], ptr[ 0]);

642         *pcm1++ = SHIFT(MLZ(hi, lo));

      ptr = *Dptr - pe;
      ML0(hi, lo, (*fe)[0], ptr[31 - 16]);
      MLA(hi, lo, (*fe)[1], ptr[31 - 14]);
      MLA(hi, lo, (*fe)[2], ptr[31 - 12]);
      MLA(hi, lo, (*fe)[3], ptr[31 - 10]);
      MLA(hi, lo, (*fe)[4], ptr[31 -    8]);
      MLA(hi, lo, (*fe)[5], ptr[31 -    6]);
      MLA(hi, lo, (*fe)[6], ptr[31 -    4]);
      MLA(hi, lo, (*fe)[7], ptr[31 -    2]);

      ptr = *Dptr - po;
```

```
      MLA(hi, lo, (*fo)[7], ptr[31 -  2]);
      MLA(hi, lo, (*fo)[6], ptr[31 -  4]);
      MLA(hi, lo, (*fo)[5], ptr[31 -  6]);
      MLA(hi, lo, (*fo)[4], ptr[31 -  8]);
      MLA(hi, lo, (*fo)[3], ptr[31 - 10]);
      MLA(hi, lo, (*fo)[2], ptr[31 - 12]);
      MLA(hi, lo, (*fo)[1], ptr[31 - 14]);
      MLA(hi, lo, (*fo)[0], ptr[31 - 16]);

664         *pcm2-- = SHIFT(MLZ(hi, lo));

      ++fo;
        }
……
684          phase = (phase + 1) % 16;
      }
   }
}
```

在分析源代码之前，不妨先看一下数组（*filter）的组织形式，参考图 3-7 (*filter)数组的组织形式，指针 fe（584 行）、fo（586 行）分别指向图中不同行、不同列的数组的第一行，这两个数组或者是(*filter)[0][0]和(*filter)[1][1]，或者是(*filter)[0][1]和(*filter)[1][0]。phase 加 1（684 行）意味 1024FIFO 中被送入 64 个值，注意在循环（615 行）中，每一次循环产生两个 pcm 值（613、642、664 行），这两个循环所用的 fe、fo 是一样的，因为在图 3-6（向量相加输出 pcm）计算 32 个 pcm 值有对称性，因此实际上数据存储时，只需存 B(16)、C（16），而不需要存储 B（-16）和 C(-16)。另外根据对称性计算时，有取负操作，这是通过变化加窗系数来做做到的（619 行）。

synth_full（）涉及到两个数据结构：

```
struct mad_synth {
   mad_fixed_t filter[2][2][2][16][8];    /* polyphase filterbank outputs */
                              /* [ch][eo][peo][s][v] */

   unsigned int phase;                    /* current processing phase */

   struct mad_pcm pcm;                    /* PCM output */
};

struct mad_pcm {
   unsigned int samplerate;               /* sampling frequency (Hz) */
   unsigned short channels;               /* number of channels */
   unsigned short length;          /* number of samples per channel */
   mad_fixed_t samples[2][1152];          /* PCM output samples [ch][sample] */
};
```

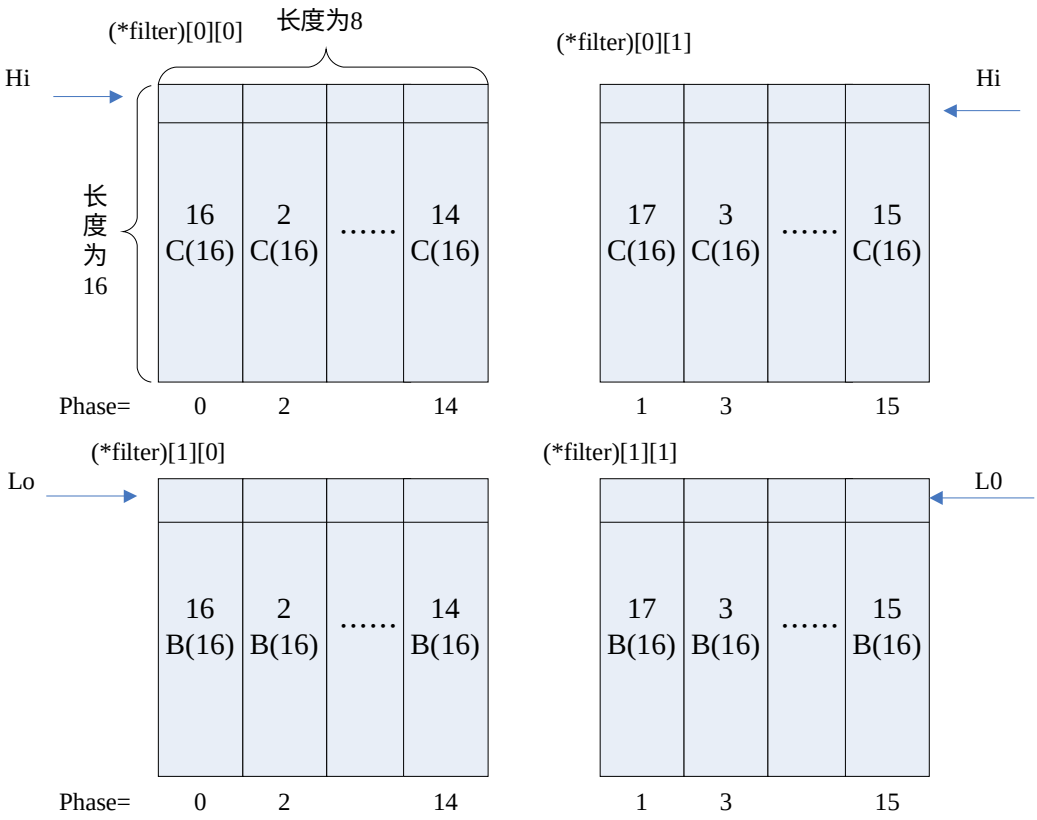

图 3-7 (*filter)数组的组织形式

附录 A　libmad 的交叉编译过程

我们所使用的用于 mp3 解码的处理器是 leon2，它的指令体系为 sparcv8 标准。下面阐述针对该指令体系的交叉编译过程。

1．解压缩从网上下载的压缩包：tar　–xzvf　libmad-0.15.1b.tar.gz。

2．进入目录 libmad-0.15.1b，建立 build 目录：mkdir build

3．进入目录 build，执行：../configure　--enable-fpm=sparc --host=sparc-elf CFLAGS="-Wall -g -O -fforce-mem -fforce-addr -fthread-jumps -fcse-follow-jumps -fcse-skip-blocks -fexpensive-optimizations -fregmove -fschedule-insn2 -mflat"

4．执行：make　make install ，这时生成的库被安装到/usr/local/lib/libmad.a ；头文件 mad.h 被安装到/usr/local/include/

5．进入顶层文件（包含 main 函数的文件）所在的目录，执行：sparc-elf-gcc　-O　-mflat -mv8 -I /usr/local/include mp3_top.c /usr/local/lib/libmad.a -nostartfiles -Ttext=0x40000000　-o mp3.out

6．反汇编：sparc-elf-objdump -D -h mp3.out >mp3.lst