MP3 详解-MP3 代码的总体框架

Mp3 解码过程了解

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。

为了解上述 9 个过程的由来，简要描述 mp3 的压缩流程。声音是一个模拟信号，对声音进行采样，量化，编码将得到 PCM 数据。PCM 又称为脉冲编码调制数据，是电脑可以播放的最原始的数据，也是 MP3 压缩的源。为了达到更大的数据压缩率，MPEG 标准采用子带编码技术将 PCM 数据分成 32 个子带，每个子带都是独立编码的（参考《数字音频原理与应用》221 页）。然后将数据变换到频域下分析，MPEG 采用的是改进的离散余弦变换，也可以使用傅利叶变换（参考《数字音频原理与应用》225）。再下来为了重建立体声进行了频谱按特定规则的排列，随后立体声处理，处理后的数据按照协议定义进行量化。为了达到更大的压缩，再进行霍夫曼编码。最后将一些系数与主信息融合形成 mp3 文件。

解码是编码的反过程大概如下：

- 所谓比特流分解是指将 mp3 文件以二进制方式打开，然后根据其压缩格式的定义，依次从这个 mp3 文件中取出头信息，边信息，比例因子信息等。这些信息都是后面的解码过程中需要的。（这部分是代码理解中的难点）。

- 霍夫曼编码是一种无损压缩编码，属于熵编码。Mp3 的解码可以通过公式实时进行数据的解码，但往往采用的是通过查表法实现解码（节省了 CPU 时间资源）。（这部分是 mp3 解码工作量中最大的一部分，也是代码理解中的难点）。

- 逆量化处理只是几个公式的操作，代码理解中不难

- 立体声处理：这部分的处理也只是对几个公式的操作，代码理解不难，但原理上理解有些难度（**参考：了解下面的部分可以较好地理解代码中的立体声处理函数 Joint Stereo 是一种立体声编码技巧，主要分为 Intensity Stereo(IS)和 Mid/Side (M/S) stereo

两种。 IS 的是在比较低流量时使用，利用了人耳对于低频讯号指向性分辨能力的不足，将音讯资料中的低频分解出来合成单声道资料，剩余的高频资料则合成另一个单声道资料，并另外纪录高频资料的位置资讯，来重建立体声的效果。例如钢琴独奏的录音就可以利用这种方法在有限的资料流量中减少音场资讯却大幅增加音色资讯。Mid/Side (M/S) stereo 在左右声道资料相似度大时常被用到，纪录方式是将左右声道音讯合并 (L+R) 得到新的一轨，再将左右声道音讯相减 (L-R) 得到 另外一轨，然后再将这两轨资料用上面提到听觉心理学模型与滤波器处理。Mid/Side (M/S) stereo 与 IS 一样的是利用部分相位 (phase) 资讯的损失来换得较高的音色纪录资讯。一般的 MP3 是 Mid/Side stereo 和 Intensity Stereo 交替使用的）

- 频谱重排列，抗锯齿处理，IMDCT 变换，子带合成：这 4 个过程都是对若干公式的操作代码易懂，至于为什么要用这些公式，估计需要对 MPEG 编码有个了解才行。

- PCM 的输出是与 c 语言对文件的处理相关的。对文件的处理在比特流分解和霍夫曼解码中最先接触到。看到了上述的两个环节，这一部分难度为 0。

综上，对代码的理解难点集中在前两个环节，一是比特流分解，二是霍夫曼解码。比特流的分解难点在于 c 语言编程知识的应用。通过这部分的学习可以进一步熟悉 c 语言的应用，和一些算法的精短描述（很简约但是有些不好懂）。霍夫曼解码部分的学习可以掌握霍夫曼的查表解码方法，很有学习价值的…… -_-

2：MP3 解码主程序段直观了解

以下罗列一份最简单的 MP3 解码的主程序段代码，目的是对 MP3 解码主程序极其大致的结构有个直观的认识。

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"
#include "decode.h"

void 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), "PCM Samp");
if (argc==1) {
        printf("Useage:decode file.mp3 output.pcm\n");
        return;
}

fr_ps.header = &info;

if ((musicout = fopen(argv[2], "w+b")) == NULL) {
        printf ("Could not create \"%s\".\n", argv[2]);
```

```
        exit(1);
}

open_bit_stream_r(&bs, argv[1], BUFFER_SIZE);

sample_frames = 0;
while(!end_bs(&bs)) {
        //尝试帧同步
        sync = seek_sync(&bs, SYNC_WORD, SYNC_WORD_LENGTH);
        if (!sync) {
                done = TRUE;
                printf("\nFrame cannot be located\n");
                out_fifo(*pcm_sample, 3, &fr_ps, done, musicout, &sample_frames);
                break;
        }
        //解码帧头
        decode_info(&bs, &fr_ps);
        //将 fr_ps.header 中的信息解读到 fr_ps 的相关域中
        hdr_to_frps(&fr_ps);
        //输出相关信息
        if(frameNum == 0)
            WriteHdr(&fr_ps);
        printf("\r%05lu", frameNum++);
        if (info.error_protection)
                buffer_CRC(&bs, &old_crc);
        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;

	//取 Side 信息
	III_get_side_info(&bs, &III_side_info, &fr_ps);
	nSlots = main_data_slots(fr_ps);

	 //读主数据(Audio Data)
	for (; nSlots > 0; nSlots--)    /* read main data. */
		hputbuf((unsigned int) getbits(&bs,8), 8);
	main_data_end = hsstell() / 8; /*of privious frame*/
	if ( flush_main=(hsstell() % bitsPerSlot) ) {
		hgetbits((int)(bitsPerSlot - flush_main));
		main_data_end ++;
	}
	bytes_to_discard = frame_start - main_data_end - III_side_info.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) {
        printf("Not enough main data to decode frame %d.    Frame discarded.\n",
                        frameNum - 1); 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];
        //主解码
        for (ch=0; ch<fr_ps.stereo; ch++) {
                long int is[SBLIMIT][SSLIMIT];      /*保存量化数据*/
                int part2_start;
                part2_start = hsstell();
                //获取比例因子
                III_get_scale_factors(&III_scalefac,&III_side_info, gr, ch, &fr_ps);
                //Huffman 解码
                III_hufman_decode(is, &III_side_info, ch, gr, part2_start, &fr_ps);
                //反量化采样
                III_dequantize_sample(is, ro[ch], &III_scalefac, &(III_side_info.ch[ch].gr[gr]), ch, &fr_ps);
        }
        //立体声处理
        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 filter input */
```

```
            double hybridOut[SBLIMIT][SSLIMIT];/* Hybrid filter out */
            double polyPhaseIn[SBLIMIT];          /* PolyPhase Input. */

            III_reorder(lr[ch], re, &(III_side_info.ch[ch].gr[gr]), &fr_ps);
            //抗锯齿处理
            III_antialias(re, hybridIn, /* Antialias butterflies. */
                          &(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, &sample_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;
}
```

1：比特流分解代码分析

这部分的代码分析是最难的，比重也是最大的。（呵呵，看之前不妨做好心理准备）这一部分包括查找帧同步，头信息提取，边信息提取，主信息提取，比例因子提取等内容。

- 所谓帧同步：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。MPEG 标准规定每个帧以 1111 1111 111B 作为标志开始，所以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。实现该功能的函数名为 seek_sync( )。

- 所谓头信息提取：以同步头 1111 1111 111B 为起点，接下来的 21 位属于头信息。比如，同步头接下来的 2 位比特是 MPEG Audio 版本号信息，根据这个信息我们可以清楚的知道这个音乐文件采用的是什么压缩方式。

- 所谓边信息的提取：就是从文件中取出后面解码需要的大部分参数

- 所谓主信息提取：在 1 帧数据里面，除了头信息，边信息外的信息称为 audio data（主信息），为了方便后面的解码，程序将 audio data 数据整个提取出来存放在全局缓冲区 buf 中

- 所谓比例因子提取：根据之前已经得到的一些系数，在 audio data 中提取出相应的比特就是我们所需要的比例因子。

代码分析：

```
#include <stdio.h>

#include <stdlib.h>

#include "common.h"

#include "decode.h"

void main(int argc, char**argv)

{

////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////

//比特流分解前的准备工作开始

    FILE *musicout;                          变量定义
```

Bit_stream_struc bs; 变量定义

frame_params fr_ps; 变量定义

III_side_info_t III_side_info; 变量定义

III_scalefac_t III_scalefac; 变量定义

unsigned int old_crc; 变量定义

layer info; 变量定义

int sync, clip; 变量定义

int done = FALSE; 变量定义

unsigned long frameNum=0; 变量定义

unsigned long bitsPerSlot; 变量定义

unsigned long sample_frames; 变量定义

typedef short PCM[2][SSLIMIT][SBLIMIT]; 变量定义

说明：声明 PCM 为短型的 3 维数组名，以后就可以用 PCM 来定义变量这 3 个维的含义分别是 PCM[2][SSLIMIT][SBLIMIT]中的 2 表示每个颗粒

SSLIMIT 是个常量 32，表示每个子带。SBLIMIT 是常量 18 表示每个子带里的每个数据。比如 PCM[0][0][0]将存放的是第 1 个颗粒中的第 1 个子带中的第 1 个数据。

PCM *pcm_sample; 变量定义

说明：定义 pcm_sample 指向一个 3 维数组的首地址

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

说明：为这个 3 维数组分配相应的内存空间。该空间命名为 PCM Samp

fr_ps.header = &info;

说明：让管理帧参数的结构体 fr_ps 中的指针 header 与 info 指向相同具体作用后面说

if ((musicout = fopen(argv[2], "w+b")) == NULL) {

说明：这个程序在执行前，用户要给两个参数，一个是要解码的 mp3 文件名对应 argv[1]，一个是保存解码数据的文件名对应 argv[2]。这句话的含义是以“二进制写入”的方式建立文件名为用户提供的 argv[2]的文件，如果建立失败，musicout 将返回 null，一旦返回 null 就要执行下面的提示出错语句。

printf ("Could not create \"%s\".\n", argv[2]);

exit(1);

}

//比特流分解前的准备工作结束

//

//

//比特流分解开始（包括*头信息解码，*边信息解码，*主数据读取，*比例因子解码）

//

//

open_bit_stream_r(&bs, argv[1], BUFFER_SIZE);

说明：这个函数的作用是根据用户提供的参数 argv[1]找到要解码的 MP3 文件，用二进制的方式打开它，并用帧结构体 bs(bitstream 的缩写)来“映像”这个二进制文件（所谓映像主要是将结构体的某些指针指向这个二进制文件以便于后面的操作）。BUFFER_SIZE 是个常量 4096，提供给函数让函数建立 4096 个字节的缓冲区来存放语音数据。

sample_frames = 0;

while(!end_bs(&bs)) {

说明：end_bs(&bs)这个函数比较简单，作用在于返回结构体 bs 中的参数值 bs->eobs 帧结束标志，意思是如果帧还没有操作完毕那么要始终进行下面的操作，直到帧所有的比特信息都被处理过

sync = seek_sync(&bs, SYNC_WORD, SYNC_WORD_LENGTH);

从结构体 bs 指向的文件中查找同步头 SYNC_WORD (即是 1111 1111 111B)

如果找到同步头返回 1 给 sync 否则返回 0

if (!sync) {

done = TRUE;

printf("\nFrame cannot be located\n");

如果没找到同步头提示出错并跳出程序循环

break;

}

如果找到了同步头则开始解码帧头

decode_info(&bs, &fr_ps);

函数 decode_info 的作用是：以同步头为起点，根据头信息的组织形式，依次取出信息，并存放到结构体中统一管理。比如说，根据协议，我们知道紧跟着同步头后面的 1 位比特是版本信息，那么就用函数 getbits(&bs,1)取出一位比特存入 fr_ps->header）->version 中，在下来的 2 位是协议层数信息，

所以用 getbits(&bs,2)取出 2 位比特存入 fr_ps->header）->lay。以下的以此类推直到根据头信息的组织形式取出所有的头信息（头信息有 32 位，其中 11 位是同步头信息）。在头信息中，对解码有用的是比特率信息，采样率信息，声道模式等，至于版本号，版权保护信息等呵呵只是个样子摆设。

hdr_to_frps(&fr_ps);

函数 hdr_to_frps 的作用是：对帧结构体 fr_ps 的其他参数根据取出头信息进行初始化，方便后面的解码，这些参数使用到后再说明，对该函数不具体解释。只将代码列在后面。

frameNum++;（计数解码到第几帧，初始化时为 0）

if (info.error_protection)

buffer_CRC(&bs, &old_crc);

这两句话的意思是，如果帧有纠错机制，那么要从文件中再提出 16 个比特作为纠错校验位。函数 buffer_CRC 很简单也不作介绍了，代码罗列在后。

switch (info.lay) {

case 3:

{

检查从头信息取出的参数 info.lay，当它是 3 的时候，才说明该文件是 mp3 文件，才有必要进行下一步的操作，不是的话直接退出

```
int nSlots, main_data_end, flush_main;

int bytes_to_discard, gr, ch, ss, sb;

static int frame_start = 0;

bitsPerSlot = 8;

III_get_side_info(&bs, &III_side_info, &fr_ps);
```

函数 III_get_side_info 的作用是根据帧结构体 fr_ps 内的相关信息，从文件结构体 bs 指向的文件中提取出边信息，存放到边信息结构体 III_side_info 中统一管理。（这个函数的理解有些困难，主要难在对 mp3 数据组织形式的认识）。边信息里面包含了语音主信息开始位置信息，比例因子解码系数信息等等。说明，边信息里面包含着非常非常多的参数信息，都是直接与解码相联系的。到目前为止只是明白这些参数的大概含义，以及他们在解码中是如何使用的。他们的具体含义分析起来要涉及到编码过程，很难进行，目前这一块的工作我还没有进行下去。

```
nSlots = main_data_slots(fr_ps);
```

说明：在在协议中，帧数据扣掉头信息和边信息后剩下的信息称为 (Audio Data)。在解码过程中需要先将 Audio Data 另存到一个缓冲区中，以方便解码。该函数的作用是通过公式计算出 1 帧数据的字节长度，再减去头信息和边信息占用的字节数。（剩下的字节数就是 Audio Data 所占用的字节数。）

```
for (; nSlots > 0; nSlots--)    /* read main data. */

hputbuf((unsigned int) getbits(&bs,8), 8);
```

在这里函数 hputbuf 的作用是从文件中提取出字节 nSlots 次并将每次提取的字节信息存放在全局缓冲区 buf 中，注意和 bs->buf 区别，两个表示的是不同的缓冲区。这样一来，Audio Data 数据就存放在全局缓冲区 buf 中了。这个函数不做做具体解释，程序罗列在后

```
main_data_end = hsstell() / 8; /*of privious frame*/
```

程序需要做的是：因为之前已经开辟了一个全局缓冲区 buf，并建立了一个新的结构提来管理这个大小是 4096 字节的 buf。程序希望的是将多帧的 Audio Data 通过循环都提取出来存放在这个缓冲区里统一处理。函数 hsstell 是返回全局缓冲区 buf 的比特位置的，每向 buf 放入一个字节，结构体中相应的比特位置计数参量就会加 8。main_data_end = hsstell() / 8 标识的就是 buf 已经放入了多少个字节，实际上就是指明了上一帧 Audio Data 在 buf 中的结束位置。

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

为了确保 buf 中的数据是以字节位组织形式的，有上面的操作（有待更具体的分析，有点想不通，因为它取数据的时候就是以字节为单位存入 buf 的，又何必担心 buf 不是以字节为组织形式，而再加上这 3 句话来确保它是以字节为单位组织的数据）

。。。。。。接下来的 10 句话难了我 1 个多月呢。。。。。。

```
1 bytes_to_discard = frame_start - main_data_end - III_side_info.main_data_begin ;
```

先要知道 frame_start 初始化为 0，还要知道，Audio Data 的有效数据不一定是直接跟在边信息以后，它的开始位置是由 III_side_info.main_data_begin 指定的，在边信息和有效主信息之间会存在垃圾信息，这是我们需要把它扣除的，这 10 话的作用也在于此

```
2   if( main_data_end > 4096 ) {    frame_start -= 4096;
3       rewindNbytes( 4096 );
4   }
5   frame_start += main_data_slots(fr_ps);
```

```
6    if (bytes_to_discard < 0) {
7        frameNum - 1;
8        break;
9    }
10   for (; bytes_to_discard > 0; bytes_to_discard--) hgetbits(8);
```

一开始 frame_start=0，经过第一句程序后 bytes_to_discard 肯定为负值，main_data_end 的取值也一定还没有到 4096，所以 2，3，4 句程序是不执行的，经过 5 句后 frame_start 由 0 变成了标识上一帧 audio data 长度的值，实际上也就是标识了下一帧的开始位置，执行 6，7，8 句直接跳出了 switch，但依然在 while(!end_bs(&bs))死循环体内，还会不断地执行到这 10 句话。

从第二次开始，执行到第一句后 bytes_to_discard 反映的就是确确实实的两帧主数据之间的垃圾信息了。可以参看附图，这里有些问题：按我的理解 maid_data_begin 一定是反映主数据相对帧结束位置的长度，而不是相对帧起点的长度。（这里理解上有些不确定，总体上这 10 句话就是为了去掉垃圾字节所作的工作）

```
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();

变量 part2_start 是对全局 buf 比特位置的一个标识，往后的解码对象已经不是结构体 bs 指向的 bs->buf，而是全局 buf，对全局 buf 的第一个操作是上述的取出垃圾信息丢弃，第二个操作是将要进行的比例因子解码。比例因子的解码是从 part2_start 位置开始的，此前的信息是垃圾信息。

注意函数 sstell 的操作对象是 bs->buf, hsstell 的操作对象是 buf。同样的 getbits 的操作对象是 bs->buf，而 hgetbits 的操作对象是 buf。这些函数的功能相同，只是操作对象不一样，所以不重复介绍相应的一些函数

```
//获取比例因子开始

III_get_scale_factors(&III_scalefac,&III_side_info, gr, ch, &fr_ps);
```

函数 III_get_scale_factors 的作用自然是获取比例因子。比例因子将用于后面解码的相关计算

```
//比特流分解结束（包括*头信息解码，*边信息解码，*主数据读取，*比例因子解码）

/////////////////////////////////////////////////////

/////////////////////////////////////////////////////

//霍夫曼解码开始

/////////////////////////////////////////////////////

/////////////////////////////////////////////////////
```

。。。。。。

2：霍夫曼解码代码分析

霍夫曼编码是基于概率统计的无损压缩，是一种变长变码。这一块的了解还不是很彻底，在接下来的工作里，首先是对这个解码程序的调试，而最先要进行的调试是霍夫曼编解码这模块。这个 mp3 程序是 turbor C 下运行的程序，比特流分解部分已经调试无误通过。紧接着要做的工作是用 C 语言编写一个压缩，解压缩的简单程序。通过这个工作进一步了解霍夫曼编码的运作。最终彻底了解认识 mp3 上霍夫曼解码工作。霍夫曼解码是 mp3 程序工作运算量的 1/3 以上，这一块应该有进一步深入的必要性。

代码分析：

//Huffman 解码开始

//

//

III_hufman_decode(is, &III_side_info, ch, gr, part2_start, &fr_ps);

该函数的作用是通过之前得到的一系列相关的参数，将每个颗粒的 576 个数据，霍夫曼解码出数据，并存放在 32 行 18 列的二维变量 is 中。具体的解码方法看这个函数的具体解释。

//Huffman 解码结束

//

//

3：程序其他解码代码分析

这其中包括//反量化采样//立体声处理//重排列//抗锯齿处理//imdct//多相频率倒置//多相合成//子带合成//PCM 输出等内容。除了 PCM 输出以外，其他部分的处理仅仅是规则和公式的应用，程序解读难度不大，已经没有必要一句一句的解释了。至于为什么用这些公式和规则，要解释起来，呵

呵我可没有那个本事。在这里我值简单的介绍数据的流向，和大概处理。处理方法原理，涉及到编码，和信号处理的很多东西，暂时没有办法说出来。

反量化采样：数据流向是经霍夫曼解码出的数据 is[2][32][18],反量化解码到 ro[2][32][18]数据的大概处理就是在 is 中调用数据，通过若干复杂的公式和规则计算出新的数据存到 ro[2][32][18]。[2]表示 2 个颗粒，[32]表示 32 个子带，[18]表示每个子带的 18 个系数

立体声处理：数据流向是经反量化后的数据 ro[2][32][18]，经过立体声处理后得到的数据存入变量 lr[2][32][18]中。数据大概的处理是：这里的规则很多，也很麻烦。举简单地说，如果帧采用的立体声模式是 ms 模式，那么它的立体声处理方法相对简单，只用两个公式 lr[0][sb][ss] = (xr[0][sb][ss]+xr[1][sb][ss])/1.41421356;

lr[1][sb][ss] = (xr[0][sb][ss]-xr[1][sb][ss])/1.41421356;

就可以求出处理后的数据，并把它存入 lr 数组就可以了。但如果是 i_stereo 立体声模式那就苦拉。。。。。。，这种模式用到很多规则和公式，理不清，目前。

重排列，抗锯齿处理，imdct,多相频率倒置，多相合成，子带合成也都只是公式和规则的应用，大概有这样的了解就可以了。如果要做程序移植，参考着来一定可以。子带合成后的数据就是我们所需要的 pcm 数据存放在 pcm_sample 指向的空间里面

PCM 输出函数完成的功能也很简单，就是以用户提供的第二个参数为文件名建立文件，并将 pcm 数据逐个保存到这个文件里面。从而完成了 mp3 文件的解码。它的代码如下：

代码分析：

//Huffman 解码结束

//

//

```
        //反量化采样
        III_dequantize_sample(is, ro[ch], &III_scalefac, &(III_side_info.ch[ch].gr[gr]), ch, &fr_ps);
}
//立体声处理
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 filter input */
        double hybridOut[SBLIMIT][SSLIMIT];/* Hybrid filter out */
        double polyPhaseIn[SBLIMIT];          /* PolyPhase Input. */

        III_reorder(lr[ch], re, &(III_side_info.ch[ch].gr[gr]), &fr_ps);
        //抗锯齿处理
        III_antialias(re, hybridIn, /* Antialias butterflies. */
                        &(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, &sample_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;
}
```

3：函数 read_decoder_table 分析

```
int read_decoder_table(FILE *fi)
{
```

该函数的作用是将 34 个供解码的数值存入了变量 ht[n].val 中，以方便以后的解码调用

```
//HTN=34
        int n,i,nn,t;
    unsigned int v0,v1;
        char command[100],line[100];
        for (n=0;n<HTN;n++) {
```

要进行 34 次循环因为有 34 个子表

```
        /* .table number treelen xlen ylen linbits */
        do {
        fgets(line,99,fi);
```

从解码树表文件 fi 中读入 98 个字符然后在最后加“/0”组成字符窜存入树组 line 中，但是如果碰到回车则马上停止读入。

```
        } while ((line[0] == '#') || (line[0] < ' '));
```

因为文件 huffdec.txt 中一行就有一个回车，所以往往取不到 98 个字符，就碰到了回车。

文件 huffdec.txt 中开始的部分是一些无用的说明部分是用#作为开头的，上面的这个循环就是为了去除无效行，直到取出一行有效行后进行下面的操作：

```
sscanf(line,"%s %s %u %u %u %u",command,ht[n].tablename,
    &ht[n].treelen, &ht[n].xlen, &ht[n].ylen, &ht[n].linbits);
```

从一个字符串中读进与指定格式相符的数据存入相应的变量，即取出子表头信息，子表头信息包含表名，解码树长，x 长，y 长，和 linbits 这些参量。在这里的含义就是将.table　0　　0　0　0　0 中的.table 存入 command,0 存入 ht[n].tablename,...

```
if (strcmp(command,".end")==0)
```

判断两个字符串

```
return n;
```

如果 command==.end 说明已经循环完了 34 次，已经到了文件尾了，所以返回就可以了

```
else if (strcmp(command,".table")!=0) {
fprintf(stderr,"huffman table %u data corrupted\n",n);
return -1;
}
```

如果 command!=.end&&!=.table 报错

```
ht[n].linmax = (1<<ht[n].linbits)-1;
```

计算出 linmax 参量（什么用处？）

```
        do {
        fgets(line,99,fi);
        } while ((line[0] == '#') || (line[0] < ' '));
```

取出子表中第 2 行有效行

```
        sscanf(line,"%s %u",command,&t);
```

取出该有效行中的有效参数

```
        if (strcmp(command,".reference")==0) {
        ht[n].ref      = t;
        ht[n].val      = ht[t].val;
        ht[n].treelen    = ht[t].treelen;
        if ( (ht[n].xlen != ht[t].xlen) ||
              (ht[n].ylen != ht[t].ylen)    ) {
         fprintf(stderr,"wrong table %u reference\n",n);
         return (-3);
        };
    while ((line[0] == '#') || (line[0] < ' ') ) {
         fgets(line,99,fi);
```

```
    }
 }
```

文件中 ht[n].ref 的存在只是为了减小解码表的容量，但是增加了解码程序的代码长度

```
    else if (strcmp(command,".treedata")==0) {
    ht[n].ref    = -1;
    ht[n].val = (unsigned char (*)[2])
     calloc(2*(ht[n].treelen),sizeof(unsigned char));
```

位变量 val 动态分配内存空间

```
      if ((ht[n].val == NULL) && ( ht[n].treelen != 0 )){
    fprintf(stderr, "heaperror at table %d\n",n);
    exit (-10);
      }
      for (i=0;i<ht[n].treelen; i++) {
     fscanf(fi,"%x %x",&v0, &v1);
     ht[n].val[i][0]=(unsigned char)v0;
     ht[n].val[i][1]=(unsigned char)v1;
    }
```

将 34 个供解码的数值存入了变量 ht[n].val 中

```
        fgets(line,99,fi); /* read the rest of the line */
        }
        else {
        fprintf(stderr,"huffman decodertable error at table %d\n",n);
        }
        }
        return n;
}
```

4：函数 huffman_decoder 分析

```
int huffman_decoder(struct huffcodetab *h, int *x, int *y, int *v, int *w)
{
```

参数说明这个函数需要一个指向相应霍夫曼表的结构体指针，根据霍夫曼解码表来查表解码，解码出来的数据先存到 x,y,v,w 上，然后在函数外，将值存入变量 is 中。本身这里的程序语句不难理解，前提是对霍夫曼查表法解码由足够的了解。往下的部分就不再作程序本身的介绍。只简单的了解数据的流程就可以了，这里数据是从全局 buf 中一个比特一个比特的取出，根据霍夫曼解码方法，通过对比表格，得到相应的解码后的数据。前面已经介绍了，为了方便霍夫曼解码，数据分了 3 个大区，其中大值区还分为 3 个小区。不同区里的比特选用的解码表不一样，解码出来的数据个数也是不一样的。具体实现方法还没有彻底弄通呢。

```
HUFFBITS level;
```

在前面的文件里面已经定义了#define HUFFBITS unsigned long int

还定义了 HUFFBITS dmask = 1 << (sizeof(HUFFBITS)*8-1);

```
int point = 0;

int error = 1;

level          = dmask;

if (h->val == NULL) return 2;

/* table 0 needs no bits */

if ( h->treelen == 0)

{    *x = *y = 0;

     return 0;

}

/* Lookup in Huffman table. */
```

以下开始霍夫曼的查表法解码，即在解码表中查找对应的解码数据

```
do {

   if (h->val[point][0]==0) {      /*end of tree*/

       *x = h->val[point][1] >> 4;
```

```
            *y = h->val[point][1] & 0xf;
```

在数据组织形式上 val 是一对出现的 val[point][0]表示 val 中第[point]对中的第一个数据

如果第 point 对 val 中的第一个数据是 0 的话，那么 x,y 根据上式确定

```
            error = 0;
            break;
        }
        if (hget1bit()) {
            while (h->val[point][1] >= MXOFF) point += h->val[point][1];
            point += h->val[point][1];
        }
        else {
                while (h->val[point][0] >= MXOFF) point += h->val[point][0];
            point += h->val[point][0];
        }
        level >>= 1;
    } while (level    || (point < ht->treelen) );
```

```
/* Check for error. */

if (error) { /* set x and y to a medium value as a simple concealment */
    printf("Illegal Huffman code in data.\n");
    *x = (h->xlen-1 << 1);
    *y = (h->ylen-1 << 1);
}

/* Process sign encodings for quadruples tables. */

if (h->tablename[0] == '3'
        && (h->tablename[1] == '2' || h->tablename[1] == '3')) {
      *v = (*y>>3) & 1;
      *w = (*y>>2) & 1;
      *x = (*y>>1) & 1;
      *y = *y & 1;
```

```
        /* v, w, x and y are reversed in the bitstream.
             switch them around to make test bistream work. */

/*      {int i=*v; *v=*y; *y=i; i=*w; *w=*x; *x=i;}    MI */

        if (*v)
             if (hget1bit() == 1) *v = -*v;
        if (*w)
             if (hget1bit() == 1) *w = -*w;
        if (*x)
             if (hget1bit() == 1) *x = -*x;
        if (*y)
             if (hget1bit() == 1) *y = -*y;
        }

    /* Process sign and escape encodings for dual tables. */
```

```
    else {

          /* x and y are reversed in the test bitstream.
                Reverse x and y here to make test bitstream work. */

/*        removed 11/11/92 -ag
             {int i=*x; *x=*y; *y=i;}
*/
        if (h->linbits)
           if ((h->xlen-1) == *x)
              *x += hgetbits(h->linbits);
        if (*x)
            if (hget1bit() == 1) *x = -*x;
        if (h->linbits)
           if ((h->ylen-1) == *y)
              *y += hgetbits(h->linbits);
        if (*y)
```

```
            if (hget1bit() == 1) *y = -*y;
        }

    return error;
}
```

三 反量化函数分析

```
void III_dequantize_sample(long int is[SBLIMIT][SSLIMIT], double xr[SBLIMIT][SSLIMIT], III_scalefac_t *scalefac, struct gr_info_s *gr_info, int ch, frame_params *fr_ps)
{
```

参数说明：is 是该函数的解码对象，解码后存入 xr,需要比例因子结构体，边信息结构体，以及帧信息

```
    int ss,sb,cb=0,sfreq=fr_ps->header->sampling_frequency;
    int stereo = fr_ps->stereo;
    int next_cb_boundary, cb_begin, cb_width, sign;

    /* choose correct scalefactor band per block type, initalize boundary */
```

反量化也是以颗粒为单位进行的，霍夫曼解码后的数据还是以颗粒为组织形式

```
if (gr_info->window_switching_flag && (gr_info->block_type == 2) )
    if (gr_info->mixed_block_flag)
        如果该颗粒是短块并且是混合型的短块则 next_cb_boundary 有下面的公式确定
        next_cb_boundary=sfBandIndex[sfreq].l[1];    /* LONG blocks: 0,1,3 */
    else {
        如果是非混合型的短块那么需要的一些解码系数确定如下：
        next_cb_boundary=sfBandIndex[sfreq].s[1]*3; /* pure SHORT block */
        cb_width = sfBandIndex[sfreq].s[1];
        cb_begin = 0;
    }
else
    如果是长块，那么系数确定如下
    next_cb_boundary=sfBandIndex[sfreq].l[1];    /* LONG blocks: 0,1,3 */
```

系数确定后则可以进行解码操作：反量化操作就是一个公式计算过程：

以下便是对 SBLIMIT*SSLIMIT 即 32*18=576 个数据的依次反量化计算；

遵循的公式是 xr[sb][ss] = pow(2.0 , (0.25 * (gr_info->global_gain - 210.0)));

```
/* apply formula per block type */
for (sb=0 ; sb < SBLIMIT ; sb++) {
    for (ss=0 ; ss < SSLIMIT ; ss++) {
        if ( (sb*18)+ss == next_cb_boundary)
            当要解码的数据位置正好落在边界的时候要作适当的调整
            ？？？这个所谓的边界是什么的边界？？？
        { /* Adjust critical band boundary */
            if (gr_info->window_switching_flag && (gr_info->block_type == 2)) {
                if (gr_info->mixed_block_flag) {

                    if (((sb*18)+ss) == sfBandIndex[sfreq].l[8])    {
                        如果这个颗粒是混合型的短块并且数据的位置还与 sfBandIndex[sfreq].l[8])相同
                        那么要对 next_cb_boundary 等系数重新调整
                        next_cb_boundary=sfBandIndex[sfreq].s[4]*3;
                        cb = 3;
                        cb_width = sfBandIndex[sfreq].s[cb+1] -
                                    sfBandIndex[sfreq].s[cb];
```

```
            cb_begin = sfBandIndex[sfreq].s[cb]*3;
        }
        else if (((sb*18)+ss) < sfBandIndex[sfreq].l[8])
            如果这个颗粒是混合型的短块并且数据的位置还小于 sfBandIndex[sfreq].l[8])
            那么要对 next_cb_boundary 等系数重新另一种调整如下
            next_cb_boundary = sfBandIndex[sfreq].l[(++cb)+1];
        else {
            如果这个颗粒是混合型的短块并且数据的位置还大于 sfBandIndex[sfreq].l[8])
            那么要对 next_cb_boundary 等系数重新第三种调整如下
            next_cb_boundary = sfBandIndex[sfreq].s[(++cb)+1]*3;
            cb_width = sfBandIndex[sfreq].s[cb+1] -
                                sfBandIndex[sfreq].s[cb];
            cb_begin = sfBandIndex[sfreq].s[cb]*3;
        }
    }
    else {
        如果是非混合型的短块，进行这样的调整：
```

```
                next_cb_boundary = sfBandIndex[sfreq].s[(++cb)+1]*3;
                cb_width = sfBandIndex[sfreq].s[cb+1] -
                                sfBandIndex[sfreq].s[cb];
                cb_begin = sfBandIndex[sfreq].s[cb]*3;
            }
        }
        else /* long blocks */
            长块的话进行如下的调整：
            ？？调整的意义是什么？？？
            next_cb_boundary = sfBandIndex[sfreq].l[(++cb)+1];
}

/* Compute overall (global) scaling. */
xr[sb][ss] = pow( 2.0 , (0.25 * (gr_info->global_gain - 210.0)));

/* Do long/short dependent scaling operations. */
```

```
if (gr_info->window_switching_flag && (
      ((gr_info->block_type == 2) && (gr_info->mixed_block_flag == 0)) ||
      ((gr_info->block_type == 2) && gr_info->mixed_block_flag && (sb >= 2)) )) {
      如果是一个非混合型的短块，或者是混合型的短块且(sb >= 2)
      xr[sb][ss] *= pow(2.0, 0.25 * -8.0 *
                  gr_info->subblock_gain[(((sb*18)+ss) - cb_begin)/cb_width]);
      xr[sb][ss] *= pow(2.0, 0.25 * -2.0 * (1.0+gr_info->scalefac_scale)
                  * (*scalefac)[ch].s[(((sb*18)+ss) - cb_begin)/cb_width][cb]);
}
else {     /* LONG block types 0,1,3 & 1st 2 subbands of switched blocks */
      xr[sb][ss] *= pow(2.0, -0.5 * (1.0+gr_info->scalefac_scale)
                              * ((*scalefac)[ch].l[cb]
                              + gr_info->preflag * pretab[cb]));
}

/* Scale quantized value. */
```

```
                sign = (is[sb][ss]<0) ? 1 : 0;
                xr[sb][ss] *= pow( (double) abs(is[sb][ss]), ((double)4.0/3.0) );
                if (sign) xr[sb][ss] = -xr[sb][ss];
            }
        }
}
```

四 立体声处理函数分析

void III_stereo(double xr[2][SBLIMIT][SSLIMIT], double lr[2][SBLIMIT][SSLIMIT], III_scalefac_t *scalefac, struct gr_info_s *gr_info, frame_params *fr_ps)

{ 参数说明：操作对象是反量化后的存在 xr 的数据，结果存放在 lr，需要比例因子结构体，颗粒信息结构体，帧结构体

```
    int sfreq = fr_ps->header->sampling_frequency;
    int stereo = fr_ps->stereo;
    int ms_stereo = (fr_ps->header->mode == MPG_MD_JOINT_STEREO) &&
                            (fr_ps->header->mode_ext & 0x2);
    int i_stereo = (fr_ps->header->mode == MPG_MD_JOINT_STEREO) &&
                           (fr_ps->header->mode_ext & 0x1);
```

```
int sfb;
int i,j,sb,ss,ch,is_pos[576];
double is_ratio[576];

/* intialization */
```

立体声处理前的初始化工作：让 is_pos[i]=0111B

Joint Stereo 是一种立体声编码技巧，主要分为 Intensity Stereo(IS)
和 Mid/Side (M/S) stereo 两种。 IS 的是在比较低流量时使用，利用
了人耳对于低频讯号指向性分辨能力的不足，将音讯资料中的低频分解出
来合成单声道资料，剩余的高频资料则合成另一个单声道资料，并另外纪录
高频资料的位置资讯，来重建立体声的效果。例如钢琴独奏的录音就可以利用
这种方法在有限的资料流量中减少音场资讯却大幅增加音色资讯。
Mid/Side (M/S) stereo 在左右声道资料相似度大时常被用到，纪录方式是
将左右声道音讯合并 (L+R) 得到新的一轨，再将左右声道音讯相减 (L-R) 得到
另外一轨，然后再将这两轨资料用上面提到听觉心理学模型与滤波器处理。
Mid/Side (M/S) stereo 与 IS 一样的是利用部分相位 (phase) 资讯的损失
来换得较高的音色纪录资讯。一般的 MP3 是 Mid/Side stereo 和 Intensity Stereo 交替使用的

```
    is_pos[i] = 7;
```

```
if ((stereo == 2) && i_stereo )
```

这里主要要操作的是改变 is_pos[i]如果 is_pos[i]始终是初始化时的值 7，那么立体声处理的时候将没有办法进行模式 i_stereo 的解码。因为 i_stereo 的解码要将音讯资料中的低频分解出来合成单声道资料，剩余的高频资料则合成另一个单声道资料，所以得弄清什么时候数据是低频的，什么时候是高频的以下的代码完成的就是这一识别工作 is_ratio[i]从而产生需要的系数

```
{   if (gr_info->window_switching_flag && (gr_info->block_type == 2))
    {   if( gr_info->mixed_block_flag )
```

如果该帧是立体声的，并且是属于 i_stereo 模式的立体声并且该颗粒是混合型的短块那么参数 max_sfb 先得到确定为 0

```
        {   int max_sfb = 0;

            for ( j=0; j<3; j++ )
            {   int sfbcnt;
                sfbcnt = 2;
```

sfbcnt 的作用是什么？

```
for( sfb=12; sfb >=3; sfb-- )
{下面的操作进行 30 次
 int lines;
 可能表征频带宽度
     lines = sfBandIndex[sfreq].s[sfb+1]-sfBandIndex[sfreq].s[sfb];
     注意：这样一减最小的结果也是 4
     i = 3*sfBandIndex[sfreq].s[sfb] + (j+1) * lines - 1;
     这样计算 i 最小的结果也是 3
     while ( lines > 0 )

     {    if ( xr[1][i/SSLIMIT][i%SSLIMIT] != 0.0 )
       如果反量化的结果不是 0，是 0 的话就不用进行立体声处理了
       选择参数 sfb=-10,lines=-10
          {    sfbcnt = sfb;
               sfb = -10;
               lines = -10;
          }
```

```
            lines--;
            i--;
        }
}
sfb = sfbcnt + 1;

if ( sfb > max_sfb )
      max_sfb = sfb;

while( sfb<12 )
{    sb = sfBandIndex[sfreq].s[sfb+1]-sfBandIndex[sfreq].s[sfb];
      i = 3*sfBandIndex[sfreq].s[sfb] + j * sb;
      for ( ; sb > 0; sb--)
      {    is_pos[i] = (*scalefac)[1].s[j][sfb];
            if ( is_pos[i] != 7 )
                  is_ratio[i] = tan( is_pos[i] * (PI / 12));
            i++;
```

```
            }
            sfb++;
        }
        sb = sfBandIndex[sfreq].s[11]-sfBandIndex[sfreq].s[10];
        sfb = 3*sfBandIndex[sfreq].s[10] + j * sb;
        sb = sfBandIndex[sfreq].s[12]-sfBandIndex[sfreq].s[11];
        i = 3*sfBandIndex[sfreq].s[11] + j * sb;
        for ( ; sb > 0; sb-- )
        {   is_pos[i] = is_pos[sfb];
            is_ratio[i] = is_ratio[sfb];
            i++;
        }
    }
    if ( max_sfb <= 3 )
    {   i = 2;
        ss = 17;
        sb = -1;
```

```
while ( i >= 0 )
{    if ( xr[1][i][ss] != 0.0 )
     {    sb = i*18+ss;
          i = -1;
     } else
     {    ss--;
          if ( ss < 0 )
          {    i--;
               ss = 17;
          }
     }
}
i = 0;
while ( sfBandIndex[sfreq].l[i] <= sb )
     i++;
sfb = i;
i = sfBandIndex[sfreq].l[i];
```

```
            for ( ; sfb<8; sfb++ )
            {    sb = sfBandIndex[sfreq].l[sfb+1]-sfBandIndex[sfreq].l[sfb];
                 for ( ; sb > 0; sb--)
                 {    is_pos[i] = (*scalefac)[1].l[sfb];
                      if ( is_pos[i] != 7 )
                           is_ratio[i] = tan( is_pos[i] * (PI / 12));
                      i++;
                 }
            }
       }
  } else
```

如果该帧是立体声的，并且是属于 i_stereo 模式的立体声并且该颗粒是非混合型的短块

```
  {    for ( j=0; j<3; j++ )
       {    int sfbcnt;
            sfbcnt = -1;
            for( sfb=12; sfb >=0; sfb-- )
            {    int lines;
```

```
      lines = sfBandIndex[sfreq].s[sfb+1]-sfBandIndex[sfreq].s[sfb];
      i = 3*sfBandIndex[sfreq].s[sfb] + (j+1) * lines - 1;
      while ( lines > 0 )
      {    if ( xr[1][i/SSLIMIT][i%SSLIMIT] != 0.0 )
           {    sfbcnt = sfb;
                sfb = -10;
                lines = -10;
           }
           lines--;
           i--;
      }
}
sfb = sfbcnt + 1;
while( sfb<12 )
{    sb = sfBandIndex[sfreq].s[sfb+1]-sfBandIndex[sfreq].s[sfb];
      i = 3*sfBandIndex[sfreq].s[sfb] + j * sb;
      for ( ; sb > 0; sb--)
```

```
    {    is_pos[i] = (*scalefac)[1].s[j][sfb];
         if ( is_pos[i] != 7 )
              is_ratio[i] = tan( is_pos[i] * (PI / 12));
         i++;
    }
    sfb++;
}

sb = sfBandIndex[sfreq].s[11]-sfBandIndex[sfreq].s[10];
sfb = 3*sfBandIndex[sfreq].s[10] + j * sb;
sb = sfBandIndex[sfreq].s[12]-sfBandIndex[sfreq].s[11];
i = 3*sfBandIndex[sfreq].s[11] + j * sb;
for ( ; sb > 0; sb-- )
{    is_pos[i] = is_pos[sfb];
     is_ratio[i] = is_ratio[sfb];
     i++;
}
```

```
            }
        }
    } else
    {   i = 31;
        ss = 17;
        sb = 0;
        while ( i >= 0 )
        {   if ( xr[1][i][ss] != 0.0 )
            {   sb = i*18+ss;
                i = -1;
            } else
            {   ss--;
                if ( ss < 0 )
                {   i--;
                    ss = 17;
                }
            }
```

```
}
i = 0;
while ( sfBandIndex[sfreq].l[i] <= sb )
      i++;
sfb = i;
i = sfBandIndex[sfreq].l[i];
for ( ; sfb<21; sfb++ )
{    sb = sfBandIndex[sfreq].l[sfb+1] - sfBandIndex[sfreq].l[sfb];
      for ( ; sb > 0; sb--)
      {    is_pos[i] = (*scalefac)[1].l[sfb];
            if ( is_pos[i] != 7 )
                  is_ratio[i] = tan( is_pos[i] * (PI / 12));
            i++;
      }
}
sfb = sfBandIndex[sfreq].l[20];
for ( sb = 576 - sfBandIndex[sfreq].l[21]; sb > 0; sb-- )
```

```
            {    is_pos[i] = is_pos[sfb];
                 is_ratio[i] = is_ratio[sfb];
                 i++;
            }
        }
}

for(ch=0;ch<2;ch++)
    for(sb=0;sb<SBLIMIT;sb++)
        for(ss=0;ss<SSLIMIT;ss++)
            lr[ch][sb][ss] = 0;

if (stereo==2)
    for(sb=0;sb<SBLIMIT;sb++)
        for(ss=0;ss<SSLIMIT;ss++) {
            i = (sb*18)+ss;
            if ( is_pos[i] == 7 ) {
```

```
            if ( ms_stereo ) {
                lr[0][sb][ss] = (xr[0][sb][ss]+xr[1][sb][ss])/1.41421356;
                lr[1][sb][ss] = (xr[0][sb][ss]-xr[1][sb][ss])/1.41421356;
            }
            else {
                lr[0][sb][ss] = xr[0][sb][ss];
                lr[1][sb][ss] = xr[1][sb][ss];
            }
        }
        else if (i_stereo ) {
            lr[0][sb][ss] = xr[0][sb][ss] * (is_ratio[i]/(1+is_ratio[i]));
            lr[1][sb][ss] = xr[0][sb][ss] * (1/(1+is_ratio[i]));
        }
        else {
            printf("Error in streo processing\n");
        }
    }
```

```
    else    /* mono , bypass xr[0][][] to lr[0][][]*/
        for(sb=0;sb<SBLIMIT;sb++)
            for(ss=0;ss<SSLIMIT;ss++)
                lr[0][sb][ss] = xr[0][sb][ss];

}
```

五 数据重排列函数分析

```
void III_reorder(double xr[SBLIMIT][SSLIMIT], double ro[SBLIMIT][SSLIMIT], struct gr_info_s *gr_info, frame_params *fr_ps)
{     排列前的数据位置 xr,排列后要存放的位置 ro,需要颗粒结构体，和帧结构体
    int sfreq=fr_ps->header->sampling_frequency;
    int sfb, sfb_start, sfb_lines;
    int sb, ss, window, freq, src_line, des_line;

    for(sb=0;sb<SBLIMIT;sb++)
        for(ss=0;ss<SSLIMIT;ss++)
            ro[sb][ss] = 0;
```

```
if (gr_info->window_switching_flag && (gr_info->block_type == 2)) {
    if (gr_info->mixed_block_flag) {
        /* NO REORDER FOR LOW 2 SUBBANDS */
        for (sb=0 ; sb < 2 ; sb++)
            for (ss=0 ; ss < SSLIMIT ; ss++) {
                ro[sb][ss] = xr[sb][ss];
            }
            在最下面的两个子带中数据的排列不需要变化
            其它的子带要根据协议内容来变化
        /* REORDERING FOR REST SWITCHED SHORT */
        for(sfb=3,sfb_start=sfBandIndex[sfreq].s[3],
            sfb_lines=sfBandIndex[sfreq].s[4] - sfb_start;
            sfb < 13; sfb++,sfb_start=sfBandIndex[sfreq].s[sfb],
            (sfb_lines=sfBandIndex[sfreq].s[sfb+1] - sfb_start))
                for(window=0; window<3; window++)
                    for(freq=0;freq<sfb_lines;freq++) {
```

```
                    src_line = sfb_start*3 + window*sfb_lines + freq;
                    des_line = (sfb_start*3) + window + (freq*3);
                    ro[des_line/SSLIMIT][des_line%SSLIMIT] =
                                        xr[src_line/SSLIMIT][src_line%SSLIMIT];
                }
}
else {    /* pure short */
    for(sfb=0,sfb_start=0,sfb_lines=sfBandIndex[sfreq].s[1];
        sfb < 13; sfb++,sfb_start=sfBandIndex[sfreq].s[sfb],
        (sfb_lines=sfBandIndex[sfreq].s[sfb+1] - sfb_start))
            for(window=0; window<3; window++)
                for(freq=0;freq<sfb_lines;freq++) {
                    src_line = sfb_start*3 + window*sfb_lines + freq;
                    des_line = (sfb_start*3) + window + (freq*3);
                    ro[des_line/SSLIMIT][des_line%SSLIMIT] =
                                        xr[src_line/SSLIMIT][src_line%SSLIMIT];
                }
```

```
        }
    }
    else {      /*long blocks */
        for (sb=0 ; sb < SBLIMIT ; sb++)
            for (ss=0 ; ss < SSLIMIT ; ss++)
                ro[sb][ss] = xr[sb][ss];
    }}
```

一 比特流分解函数分析

1：函数 open_bit_stream_r 分析

```
void open_bit_stream_r(Bit_stream_struc *bs, char *bs_filenam, int size)
{
```

函数的操作对象是（1）用户提供的 mp3 文件名 filenam（2）用来映像 MP3 文件的帧结构体 bs（3）缓冲区的大小 size。

```
    register unsigned char flag = 1;
```

定义变量前加个 register 是因为这个参数 flag 将频繁使用，为了引用的速度够块，前加 register 从而建议处理器分配寄存器来存放这个参数。

```
    if ((bs->pt = fopen(bs_filenam, "rb")) == NULL) {
```

以二进制读出的方式打开用户指定的 mp3 文件，并用 pt 指向这个文件，如果打开失败则执行下面的出错信息。

```
        printf("Could not find \"%s\".\n", bs_filenam);
        exit(1);
    }
    bs->format = BINARY;
    alloc_buffer(bs, size);
    bs->buf_byte_idx=0;
    bs->buf_bit_idx=0;
    bs->totbit=0;
    bs->mode = READ_MODE;
    bs->eob = FALSE;
    bs->eobs = FALSE;
```

为了更好的宏观上用结构体 bs 映像该 mp3 文件，要根据文件的一些特点来对结构体中的一些参数进行赋值。比如用 bs->pt 指向文件，bs->format 设为二进制，bs->buf_byte_idx=0;bs->buf_bit_idx=0;bs->totbit=0;表示这个 mp3 文件现在还没有被操作过，如果从这个文件中取出了 9 位比特，那么相应的 bs->buf_byte_idx； bs->buf_bit_idx bs->totbit=0 会被从新设定为 bs->buf_byte_idx=1 表示当前是第 2 个字节，bs->buf_bit_idx=1 表示当前等待操作的是第 2 个字节的第 2 位，bs->totbit 表示当前等待操作的是第 10 位比特。

```
}
```

2：函数 end_bs 分析

int end_bs(Bit_stream_struc *bs)

{

return(bs->eobs);

返回结束标志，一开始初始化位 false 表示当前帧还没有操作结束，一旦对当前帧的所有数据都操作过后，bs->eobs 为 true

}

3：函数 seek_sync 分析

int seek_sync(Bit_stream_struc *bs, unsigned long sync, int N)

{

该函数的操作对象是结构体 bs 指向的文件，查找目标是变量 sync 定义的 0xfff，查找目标的长度有变量 N 定义这里 N 是 12。

unsigned long aligning;

unsigned long val;

long maxi = (int)pow(2.0, (double)N) - 1;

定义 maxi=[（2 的 N 次方）-1]，在主程序中 N=12,所以 maxi 实际上就是 1111 1111 1111B

aligning = sstell(bs)%ALIGNING;

变量 aligning 的作用在于确保同步头的识别是从字节边界开始的。比如说在二进制文件里面存在这么一段数据：0111 1111 1111 0101 0011 1100 1111 0010，在这段数据里面，前面连续的 11 个 1 不能算是同步头，因为第一个 1 开始的地方不是字节的边界。如果存在这么一段数据 1111 1111 1111 0101 0011 1100 1111 0010，那么前面连续的 11 个 1 就可以确定是同步头了。（understand?:）

函数 sstell 的作用是返回当前要操作的比特位置，比如说，我们已经提取并处理了文件的前 50 个比特，但还是没有找到同步头（mp3 文件的开始往往有一段无用的乱码数据），虽然没有找到，但是结构体中的 bs->totbit 始终在累加计数，此时的它应该是 50，通过调用函数 sstell(&bs)就可以返回出 bs->totbit 的取值。如果让 bs->totbit 对 8 求余，余数为 0 说明，当前要被操作的这个比特是一个新字节的起点。%是求余运算，ALIGNING 是常量 8。

if (aligning)

如果余数不为 0，说明刚要被操作的比特不是以字节为边界的，比如 bs->totbit 是 14，而我们知道第 0，8，16 个比特是以字节为边界的，为了查找同步头，我们希望一开始就把比特位置定位在字节边界。就像刚才比如的 bs->totbit 是 14，从这个位置开始查找 11 个连续的 1，即使找到也没有意义。所以要先进行下面的操作：

getbits(bs, (int)(ALIGNING-aligning));

我们要再提出两个比特让 bs->totbit 变成 16（每提出一个比特，相应的 bs->totbit 会加 1，这是提取比特函数完成的工作）。这里用到了一个新的函数 getbits 是比特提取函数，在下面介绍。

val = getbits(bs, N);

定位到字节边界后，以后每次从文件中提取 11 个比特，只要这 11 个比特是同步头，就说明解码起点找到了。但是请注意 ☺ 程序中 val = getbits(bs, N)的 N 实际上是 12，程序每次提出了 12 个比特来进行比对，这主要是因为如果不这样，万一提出的 11 位数据不是同步头的时候，还得重复进行比特字节边界定位的操作，会变得麻烦。而如果一次提出 12 位，即使不是同步头，重新进行比特字节边界定位的操作也比较方便用如下的 3 句话完成（自己分析这 3 句话哈，不难的呵呵，提示 maxi 是上面定义好的 0xfff）

while (((val&maxi) != sync) && (!end_bs(bs))) {

val <<= ALIGNING;

```
        val |= getbits(bs, ALIGNING);
    }

    if (end_bs(bs))
        return(0);
    else
        return(1);
}
```

4：函数 getbits 分析

```
unsigned long getbits(Bit_stream_struc *bs, int N)
{
```

☺ 该函数是比特分解的核心函数，静下心看还是看得懂的，加油

该函数的操作对象是结构体指向的文件，以及需要取出的比特个数 N

```
    unsigned long val=0;
    register int i;
    register int j = N;
```

register int k, tmp;

if (N > MAX_LENGTH)

程序设定最多一次只能取出 32 个比特，MAX_LENGTH 是常量 32

printf("Cannot read or write more than %d bits at a time.\n", MAX_LENGTH);

bs->totbit += N;

取出多少个比特，结构体中相应的比特位置参数就要相应的计数累加，事实上字节位置也要计数累加，就是每取出 8 个比特，字节位置变化 1，但由于程序中往往取出的比特个数是不一定的，所以字节位置的计数比较困难，相对麻烦一些

while (j > 0) {

如果要取出的比特个数不为 0，那么进行如下的操作：比特位置的计数很方便上面已经实现，字节位置的计数要麻烦一些：要借助到变量 bs->buf_bit_idx,具体实现看下面：（至于为什么要用到字节位置的计数，那是因为缓冲器是一字节位单位组织数据的）

if (!bs->buf_bit_idx) {

注意 bs->buf_bit_idx 的取值是[0-7]，比特位置一直累加，bs->buf_bit_idx 就一直在[0-7]区间里面循环。如果 bs->buf_bit_idx 为 0，说明循环一次了，已经有一个字节长度被处理过了。

以下的代码循环相套可读性很差，得有点耐心分析。。。。。。

为方便分析，在这里加了编号：

1 bs->buf_bit_idx = 8;

2 bs->buf_byte_idx--;

3 if ((bs->buf_byte_idx < MINIMUM) || (bs->buf_byte_idx < bs->eob)) {

```
4     if (bs->eob)
5           bs->eobs = TRUE;
6     else {
7           for (i=bs->buf_byte_idx; i>=0;i--)
8                 bs->buf[bs->buf_size-1-bs->buf_byte_idx+i] = bs->buf[i];
9                 refill_buffer(bs);
10          bs->buf_byte_idx = bs->buf_size-1;
      }
      }
}
```

刚开始的时候 buf_byte_idx 初始化为 0，那么经过语句 2，则它一定是个小于 0 的数，肯定通过 3 句的判断，继而执行第 4 句，以开始的时候，buffer 里还没调入数据，所以 eof(end of buffer)肯定是 false 所以不执行 5 句，而直接执行 7 句，有因为这次的 buf_byte_idx 也不符合 7 句的判断，所以不会执行 8 句，而开始 9 句的执行。（9 句的目的是把数据调入 buffer 中，填满 buffer,让 eof 为真），然后执行 10 句 bs->buf_byte_idx = bs->buf_size-1;（显然 bs->buf_byte_idx 的计数方式与 bs->totbit 的计数方向正好相反。

等第二次循环到这里的时候

2 句的执行说明，已经处理完了一个字节，bs->buf_byte_idx 计数减 1，　　如果 bs->buf_byte_idx 值已经小于 MINIMUM（常量 4）抑或已经是负值的时候，表明这一帧的数据已经处理完毕，令 bs->eobs 为真

```
11 k = MIN(j, bs->buf_bit_idx);
```

```
12 tmp = bs->buf[bs->buf_byte_idx]&putmask[bs->buf_bit_idx];

13 tmp = tmp >> (bs->buf_bit_idx-k);

14 val |= tmp << (j-k);

15 bs->buf_bit_idx -= k;

16 j -= k;
```

[11-16]的含义是：如果要取出的比特数小于 8，比如是 5，那么从缓冲区 buffer 中取出第 buf_byte_idx 个字节数据（注意缓冲区里的数据应该是从高位相低位开始填充的，否则没有办法解释，有待确定），通过逻辑与运算只保留前 5 为比特（在高位）。(这里 putmask[bs->buf_bit_idx]是个数组，int putmask[9]={0x0, 0x1, 0x3, 0x7, 0xf, 0x1f, 0x3f, 0x7f, 0xff};)。保留后左移 3 位，存入 32 为的变量 val 中，因为值取了 5 位，没有超过 8 位，所以 buf_byte_idx 不变

如果取出的比特数大于 8，比如 25，那么。。。。。下面的分析你自己搞定哈，表达出来太麻烦了，呵呵，现在体会到什么是只可意会了呵呵。额外话：春节快乐！！！

```
    }

    return val;

}
```

5：函数 refill_buffer 分析

```
void refill_buffer(Bit_stream_struc *bs)
```

{

register int i=bs->buf_size-2-bs->buf_byte_idx;

register unsigned long n=1;

while ((i>=0) && (!bs->eob)) {

在这里如果(i>=0) && (!bs->eob),说明 buffer 有空余的空间

n=fread(&bs->buf[i--], sizeof(unsigned char), 1, bs->pt);

那么从文件中取出数据一次来填这个缓冲区 buffer，通过反复执行达到填满 buffer 的目的

if (!n)

bs->eob= i+1;

直到填满后，将没办法再填入，返回值 n 开始变为 0，于是 bs->eob= 真，并跳出程序

}

}

6：函数 decode_info 分析

void decode_info(Bit_stream_struc *bs, frame_params *fr_ps)

{ 该函数的操作对象是结构体 bs 所映像的文件，

将解码出来的信息存放在帧结构体 fr_ps 中

layer *hdr = fr_ps->header;

这句话是辅助用的，他的使用让 hdr->version = get1bit(bs);

等效于(fr_ps->header）->version= get1bit(bs);）

hdr->version = get1bit(bs);（取出同步头后，接着取出的 1 个比特是版本信息

hdr->lay = 4-getbits(bs,2); （再接着取出的 2 个比特是协议层数的信息）

hdr->error_protection = !get1bit(bs); /* error protect. TRUE/FALSE */

hdr->bitrate_index = getbits(bs,4);

hdr->sampling_frequency = getbits(bs,2);

hdr->padding = get1bit(bs);

hdr->extension = get1bit(bs);

hdr->mode = getbits(bs,2);

hdr->mode_ext = getbits(bs,2);

hdr->copyright = get1bit(bs);

hdr->original = get1bit(bs);

hdr->emphasis = getbits(bs,2);

注意这里有个新的函数 get1bit(bs)，其实功能等于 getbits(bs,1)，所有就不再额外分析

```
}
```

7：函数 hdr_to_frps 分析

```
void hdr_to_frps(frame_params *fr_ps)
{
    layer *hdr = fr_ps->header;         /* (or pass in as arg?) */

    fr_ps->actual_mode = hdr->mode;
    fr_ps->stereo = (hdr->mode == MPG_MD_MONO) ? 1 : 2;
    fr_ps->sblimit = SBLIMIT;
    if(hdr->mode == MPG_MD_JOINT_STEREO)
        fr_ps->jsbound = js_bound(hdr->lay, hdr->mode_ext);
    else
        fr_ps->jsbound = fr_ps->sblimit;
}
```

8：函数 buffer_CRC 分析

```
void    buffer_CRC(Bit_stream_struc *bs, unsigned int *old_crc)
{
        *old_crc = getbits(bs, 16);
}
```

9：函数 III_get_side_info 分析

```
void III_get_side_info(Bit_stream_struc *bs, III_side_info_t *si, frame_params *fr_ps)
{
```

该函数的操作对象是文件结构体 bs 指向的文件，根据帧结构体 fr_ps 的相关信息，从文件中提取边信息存放到边信息结构体 III_side_info_t *si 中管理。

```
    int ch, gr, i;
    int stereo = fr_ps->stereo;
    si->main_data_begin = getbits(bs, 9);
```

边信息紧接在头信息之后，接在头信息后面的 9 位是第一个边信息即语音主信息开始位置，取出后存放在 si->main_data_begin 中

if (stereo == 1)

说明：stereo 在帧解码出来的头信息中，标志该帧是单声道，还是双声道

如果是单声道那么文件的数据组织形式是，标识语音主信息开始位置的 9 位比特之后，接着的 5 位是 private_bits 信息（解码中没用，呵呵摆设用的）。如果是双声道的，那么只有接着的 3 位是 private_bits 信息

```
    si->private_bits = getbits(bs,5);
else
    si->private_bits = getbits(bs,3);
for (ch=0; ch<stereo; ch++)
    for (i=0; i<4; i++)
        si->ch[ch].scfsi[i] = get1bit(bs);
```

紧接着取出比例因子解码需要的系数 scfsi，每个声道有 4 个比例因子系数，每个比例因子是 1 位比特信息。注意此后的信息解码是以颗粒位单位来进行的。一帧中有一个或者两个声道，每个声道有 1152 个采样数据，协议把这 1152 个数据分为两个颗粒来封装编码。颗粒编码得到的相应的一些系数被放到边信息中的比例因子解码信息之后。对这一部分的系数的解码结果自然要对应相应的颗粒来存放。以下的程序是个循环体，这里取 gr=0,ch=0,来简单说明这个循环体的作用：

```
for (gr=0; gr<2; gr++) {
    for (ch=0; ch<stereo; ch++) {
        si->ch[ch].gr[gr].part2_3_length = getbits(bs, 12);
```

从文件中在取出 12 个比特作为第 1 个声道的第一个颗粒的 part2_3_length 系数，这个系数的作用是：表识 1 区的大小，上面已经介绍过了，1 个

颗粒里面有 576 个数据。实际上采样数据被分为 32 个子带，每个子带经过 mdct 变换后得到 18 个数据组成的 32*18=576 个系数。这 576 个系数经过霍夫曼编码后得到这 576 个数据。在编码的时候，为了方便解码这 576 个数据分为 3 个区，大值区（big_value）,1 区，0 区。大值区又被细分位 3 个小区：0 号区，1 号区，2 号区。分区的目的是为了霍夫曼解码的速度。具体机制，huhu 俺还没彻底清楚，可能的话，呵呵有你来讲。

```
si->ch[ch].gr[gr].big_values = getbits(bs, 9);
```

参数 big_values 用来标识从主数据开始的多少字节是属于大值区

```
si->ch[ch].gr[gr].global_gain = getbits(bs, 8);
```

参数 global_gain 是全局增益因子，解码计算用到

```
si->ch[ch].gr[gr].scalefac_compress = getbits(bs, 4);
```

参数 scalefac_compress 比例因子压缩系数，也是后面的计算用到

```
si->ch[ch].gr[gr].window_switching_flag = get1bit(bs);
```

参数 window_switching_flag 带窗标志，后面的解码常常根据这个标志作一些判断，是否带窗决定了不同的解码方法。窗的具体含义还没有彻底明白，应该跟子带分解，滤波，子带合成，信号处理等有关系。

```
if (si->ch[ch].gr[gr].window_switching_flag) {
    si->ch[ch].gr[gr].block_type = getbits(bs, 2);
    si->ch[ch].gr[gr].mixed_block_flag = get1bit(bs);
    for (i=0; i<2; i++)
        si->ch[ch].gr[gr].table_select[i] = getbits(bs, 5);
    for (i=0; i<3; i++)
```

```
                si->ch[ch].gr[gr].subblock_gain[i] = getbits(bs, 3);
```

如果该颗粒是带窗的那么进行上述的解码。table_select[i]是霍夫曼解码表的选择：协议提供了 32 个表来提供霍夫曼解码使用，具体用那个表，与该参数 table_select[i]相关

```
            }
            if (si->ch[ch].gr[gr].block_type == 0) {
                printf("Side info bad: block_type == 0 in split block.\n");
                exit(0);
            else if (si->ch[ch].gr[gr].block_type == 2
                    && si->ch[ch].gr[gr].mixed_block_flag == 0)
                si->ch[ch].gr[gr].region0_count = 8; /* MI 9; */
            else si->ch[ch].gr[gr].region0_count = 7; /* MI 8; */
si->ch[ch].gr[gr].region1_count = 20 - si->ch[ch].gr[gr].region0_count;
```

上面这些语句的含义是根据窗的类型，来确定大值区中 0 号，1 号，2 号区的起始与结束区间。

```
        }
        else {
```

如果颗粒是不带窗的，那么大值区中 0 号，1 号，2 号区的起始与结束区间又是一种新的分布如下：

```
            for (i=0; i<3; i++)
```

```
                    si->ch[ch].gr[gr].table_select[i] = getbits(bs, 5);
                si->ch[ch].gr[gr].region0_count = getbits(bs, 4);
                si->ch[ch].gr[gr].region1_count = getbits(bs, 3);
                si->ch[ch].gr[gr].block_type = 0;
            }
            si->ch[ch].gr[gr].preflag = get1bit(bs);
            si->ch[ch].gr[gr].scalefac_scale = get1bit(bs);
            si->ch[ch].gr[gr].count1table_select = get1bit(bs);
        }
    }
}
```

10：函数 main_data_slots 分析

```
int main_data_slots(frame_params fr_ps)
{
    int nSlots;
    nSlots = (144 * bitrate[2][fr_ps.header->bitrate_index])
```

```
                / s_freq[fr_ps.header->sampling_frequency];
```

在协议中，帧的长度是固定的。用上述的计算公式可以计算出来

在协议中，帧数据扣掉头信息和边信息后剩下的信息称为 (Audio Data)

在解码过程中需要先将 Audio Data 另存到一个缓冲区中，以方便解码

下面的操作是为了计算 Audio Data 的长度，通过帧长度减去头信息，边信息长度

注意 nSlots 表征的是多少字节，而不是多少比特

```
    if (fr_ps.header->padding) nSlots++;
    nSlots -= 4;
    if (fr_ps.header->error_protection)
        nSlots -= 2;
    if (fr_ps.stereo == 1)
        nSlots -= 17;
    else
        nSlots -=32;
    return(nSlots);
}
```

11：函数 hputbuf 分析

```
void hputbuf(unsigned int val, int N)
{
    if (N != 8) {
        printf("Not Supported yet!!\n");
        exit(-3);
    }
    buf[offset % BUFSIZE] = val;
    offset++;
}
```

12：函数 III_get_scale_factors 分析

```
void III_get_scale_factors(III_scalefac_t *scalefac, III_side_info_t *si, int gr, int ch, frame_params *fr_ps)
{
```

以下的代码理解难度不大，让人感觉纷乱的是 mp3 里面参数之间错综复杂的关系，这个函数的作用无非就是根据已经得到的一些信息（比如是窗类型，比例因子解码系数等），从全局 buf 中取出规定位数的比特作为相应的比例因子。更细节的描述，即使写出来呵呵，你也未必有耐心看。

```
int sfb, i, window;
struct gr_info_s *gr_info = &(si->ch[ch].gr[gr]);


if (gr_info->window_switching_flag && (gr_info->block_type == 2)) {
      if (gr_info->mixed_block_flag) { /* MIXED */ /* NEW - ag 11/25 */
            for (sfb = 0; sfb < 8; sfb++)
                  (*scalefac)[ch].l[sfb] = hgetbits(
                        slen[0][gr_info->scalefac_compress]);
            for (sfb = 3; sfb < 6; sfb++)
                  for (window=0; window<3; window++)
                        (*scalefac)[ch].s[window][sfb] = hgetbits(
                              slen[0][gr_info->scalefac_compress]);
            for (sfb = 6; sfb < 12; sfb++)
                  for (window=0; window<3; window++)
                        (*scalefac)[ch].s[window][sfb] = hgetbits(
                              slen[1][gr_info->scalefac_compress]);
            for (sfb=12,window=0; window<3; window++)
```

```
                (*scalefac)[ch].s[window][sfb] = 0;
        }
        else {    /* SHORT*/
            for (i=0; i<2; i++)
                for (sfb = sfbtable.s[i]; sfb < sfbtable.s[i+1]; sfb++)
                    for (window=0; window<3; window++)
                        (*scalefac)[ch].s[window][sfb] = hgetbits(
                            slen[i][gr_info->scalefac_compress]);
                for (sfb=12,window=0; window<3; window++)
                    (*scalefac)[ch].s[window][sfb] = 0;
        }
    }
    else {      /* LONG types 0,1,3 */
        for (i=0; i<4; i++) {
            if ((si->ch[ch].scfsi[i] == 0) || (gr == 0))
                for (sfb = sfbtable.l[i]; sfb < sfbtable.l[i+1]; sfb++)
                    (*scalefac)[ch].l[sfb] = hgetbits(
```

```
                    slen[(i<2)?0:1][gr_info->scalefac_compress]);
            }
            (*scalefac)[ch].l[22] = 0;
        }
    }
```

二 霍夫曼解码函数分析

1：函数 III_hufman_decode 分析

void III_hufman_decode(long int is[SBLIMIT][SSLIMIT], III_side_info_t *si, int ch, int gr, int part2_start, frame_params *fr_ps)

参量说明:（a）is 是一个 32*18 的二维数组用来存放解码后的 576 个数据，既是 PCM 数据经过频率变换得到的系数。

(b)*si 是边信息结构体的入口指针：解码过程需要一些边信息中的系数

(c)ch 是声道标示，gr 是颗粒标示，解码过程是以没个声道的颗粒为单位进行的

(d)part2_start，解码的位置依据

(e)*fr_ps 是帧参量的入口指针：解码过程需要诸如声道数，协议层次的信息来源

```
{
        int i, x, y;
```

int v, w;

struct huffcodetab *h;

这是一个结构体的定义：它将统一对若干个霍夫曼解码表进行管理。

int region1Start;

int region2Start;

int bt = (*si).ch[ch].gr[gr].window_switching_flag && ((*si).ch[ch].gr[gr].block_type == 2);

initialize_huffman();

函数 initialize_huffman 的作用：

(1) 霍夫曼解码初始化：打开协议定义好的霍夫曼解码树表(该表中存在 34 各子表)，在理解之后的内容时，请一定一边读程序，一边参考这个霍夫曼解码树表文件，该文件放在附录里

(2) 取出霍夫曼解码树表文件中 34 个子表的内容以便于解码使用

/* Find region boundary for short block case. */

if (((*si).ch[ch].gr[gr].window_switching_flag) &&

((*si).ch[ch].gr[gr].block_type == 2)) {

如果相应颗粒的窗口标志为 1 并且块类型是 2 的话那么等于说明这个颗粒是短块，它的 576 个系数中前 36 个数属于 0 区，从 36 到 576 的系数属于 1 区。并且不存在属于 2 区的系数。所在区不同，需要的解码表不同，要根据区从 34 个表中选出合适的表来解码。

region1Start = 36; /* sfb[9/3]*3=36 */

```
            region2Start = 576; /* No Region2 for short block case. */
                }
    else {              /* Find region boundary for long block case. */
```

相反如果不符合上述的条件就是长块，长块的数据区分布情况是：

```
        region1Start = sfBandIndex[fr_ps->header->sampling_frequency]
                                    .l[(*si).ch[ch].gr[gr].region0_count + 1]; /* MI */
        region2Start = sfBandIndex[fr_ps->header->sampling_frequency]
                                        .l[(*si).ch[ch].gr[gr].region0_count +
                                        (*si).ch[ch].gr[gr].region1_count + 2]; /* MI */
```

注意：

（1） 首先要了解 sfBandIndex 这一特殊的数据结构。可以理解它是一个特殊的数组，在这个数组里面还包含了两个子数组 l[n]和 s[n]

（2） 根据头信息的采样率和边信息中的相关参数可以确定 0 区，1 区，2 区的范围

（3） 576 个系数大体上分为大值区，1 号区和零区。大值区又分为 0，1，2 三个小区

（4） 大值区解码一次得到两个数据，1 号区解码一次得到 4 个数据。

```
        }
    /* Read bigvalues area. */
```

for (i=0; i<(*si).ch[ch].gr[gr].big_values*2; i+=2) {

（1） 循环执行 big_values 次以下的代码，big_values 是边信息中的一个参数。

（2） 大数值区是成对出现的

（3） 上面已经介绍过 h 是结构体统一对若干个霍夫曼解码表进行管理。

（4） 结果霍夫曼解码表的解读后，解码表数据已经存入了变量 ht[n].val 中，ht 是个全局的管理霍夫曼解码表的结构体

if (i<region1Start) h = &ht[(*si).ch[ch].gr[gr].table_select[0]];

else if (i<region2Start) h = &ht[(*si).ch[ch].gr[gr].table_select[1]];

else h = &ht[(*si).ch[ch].gr[gr].table_select[2]];

先根据数据的位置选择相应的用来解码的霍夫曼表

huffman_decoder(h, &x, &y, &v, &w);

函数 huffman_decoder 的作用就是霍夫曼解码核心程序，解码的数据通过下面的 2 句，保存到变量 is 中

is[i/SSLIMIT][i%SSLIMIT] = x;

is[(i+1)/SSLIMIT][(i+1)%SSLIMIT] = y;

}

大值区的霍夫曼解码到此为止，存放在 is 数组中

/* Read count1 area. */

1 号区的霍夫曼解码：

```
    h = &ht[(*si).ch[ch].gr[gr].count1table_select+32];
```

先选择适当的霍夫曼解码表

```
    while ((hsstell() < part2_start + (*si).ch[ch].gr[gr].part2_3_length ) &&
       ( i < SSLIMIT*SBLIMIT )) {
```

当位比特的位置在 1 号区范围的时候并且 i < SSLIMIT*SBLIMIT 的时候，始终执行如下的动作：

```
        huffman_decoder(h, &x, &y, &v, &w);
        is[i/SSLIMIT][i%SSLIMIT] = v;
        is[(i+1)/SSLIMIT][(i+1)%SSLIMIT] = w;
        is[(i+2)/SSLIMIT][(i+2)%SSLIMIT] = x;
        is[(i+3)/SSLIMIT][(i+3)%SSLIMIT] = y;
        i += 4;
        }

    if (hsstell() > part2_start + (*si).ch[ch].gr[gr].part2_3_length)
```

如果比特位置已经到了 0 值区，进行 0 值区的解码前重新组织一下缓冲区，以便于下一帧的解码。

```
    {   i -=4;
        rewindNbits(hsstell()-part2_start - (*si).ch[ch].gr[gr].part2_3_length);
```

```
    }

    /* Dismiss stuffing Bits */
    if ( hsstell() < part2_start + (*si).ch[ch].gr[gr].part2_3_length )
        hgetbits( part2_start + (*si).ch[ch].gr[gr].part2_3_length - hsstell());

    /* Zero out rest. */
    for (; i<SSLIMIT*SBLIMIT; i++)
        is[i/SSLIMIT][i%SSLIMIT] = 0;
}
```

2：函数 initialize_huffman 分析

```
 void initialize_huffman()
{
```

该函数的作用就是打开协议定义好的霍夫曼解码树表(该表中存在 34 各子表)。MPEG 标准通过大量的统计已经为霍夫曼编码作了规定，也为其解码制作了相应的解码表以供查表。在理解之后的内容时，请一定先看一下这个霍夫曼解码树表文件结构，该文件放在附录里

另一个作用读取霍夫曼解码树表中的数据

```
    FILE *fi;
    if (huffman_initialized) return;
    if (!(fi = OpenTableFile("huffdec.txt") )) {
    如果打开名为 huffdec.txt 霍夫曼解码树表文件出错则提示并推出
        printf("Please check huffman table 'huffdec.txt'\n");
        exit(1);
    }
    if (read_decoder_table(fi) != HTN) {
函数 read_decoder_table 的作用是读取文件指针 fi 指向的霍夫曼解码树表文件，从中读出相应的数据，如果读取失败则提示出错
        fprintf(stderr,"decoder table read error\n");
        exit(4);
    }
    huffman_initialized = TRUE;
}
```