

# **多媒体音视频格式标准基础**

## **Revision History**

<b>Version</b>	<b>Change Description</b>	<b>Date</b>	<b>Reviser</b>
0.1	Initial version	2005-11-07	Gu Jianjun
	Add a-law, Mu-law description	2005-12-11	Gu Jianjun
	Add Real Media file format	2005-12-20	Gu Jianjun

1.	图像/视频基础.....	9
1.1	颜色的基本概念.....	9
1.1.1	颜色的物理性质.....	9
1.1.2	色调、饱和度和亮度.....	9
1.1.3	颜色的混合与互补.....	12
1.2	颜色的空间表示.....	13
1.2.1	颜色空间及分类.....	13
1.2.2	几种典型的颜色空间.....	14
1.2.2.1	RGB 颜色空间.....	14
1.2.2.2	HSI 颜色空间.....	14
1.2.2.3	YUV (Lab) 颜色空间.....	15
1.2.2.4	CMY 颜色空间.....	15
1.2.2.5	YIQ 模型.....	17
1.2.2.6	YCrCb 模型.....	17
1.3	数字图像与视频.....	18
1.3.1	灰度图和彩色图.....	18
1.3.2	矢量图和位图.....	19
1.3.3	分辨率.....	20
1.3.4	图像深度与颜色类型.....	21
1.3.5	色彩深度.....	23
1.3.6	图像数据的容量.....	23
1.3.7	视频概念.....	24
2.	图像/视频压缩技术介绍.....	24
2.1	视频压缩的基本概念.....	24
2.2	颜色空间与压缩图像数据.....	24
2.3	去冗余技术.....	25
2.3.1	空间域压缩技术.....	26
2.3.1.1	预测编码 (Predictive coding) .....	26
2.3.1.2	变换编码 (Transform coding) .....	26
2.3.1.3	量化 (Quantisation) .....	27
2.3.2	熵编码.....	27
2.3.2.1	游程长度编码 (Run Length coding) .....	28
2.3.2.2	霍夫曼编码 (Huffman coding) .....	28
2.3.2.3	算术编码 (Arithmetic coding) .....	29
2.3.2.4	帧内图像压缩编码处理举例.....	30
2.3.3	时域压缩技术.....	32
2.3.3.1	运动估计的运动补偿编码.....	32
2.3.3.4	视频压缩编码中去冗余技术的应用.....	33
2.3.4.1	帧内图像 I 的压缩编码算法.....	33
2.3.4.2	预测图像 P 的压缩编码算法.....	34
2.3.4.3	双向预测图像 B 的压缩编码算法.....	34
3.	视频标准简介.....	35
3.1	H. 261.....	35

3.2 H.263.....	35
3.3 JPEG.....	36
3.4 MJPEG.....	38
3.5 MPEG-1 Video (ISO11172-2) .....	38
3.6 MPEG-2 Video (ISO13818-2) .....	38
3.7 MPEG-4 Video (ISO14496-2) .....	40
3.8 H.264/MPEG-4 AVC.....	41
3.9 Windows Media Video 9 (WMV 9) .....	43
3.10 AVS.....	44
4 各种视频标准功能对比.....	44
4.1 视频标准的演进历程.....	44
4.2 常见压缩格式技术指标对比.....	45
5. 音频基础.....	45
5.1 声音的频率和幅度.....	45
5.2 模拟信号与数字信号.....	46
5.3 采样和量化.....	46
5.3.1 采样频率.....	46
5.3.2 量化精度.....	47
5.4 声音质量与数据率.....	47
6 音频压缩技术简介.....	47
6.1 听觉系统的感知特性.....	48
6.1.1. 人耳听觉曲线.....	48
6.1.2 掩蔽效应.....	49
6.1.2.1 频域掩蔽.....	49
6.1.2.2 时域掩蔽.....	50
6.2 音频编解码技术.....	51
6.2.1 基于波形编码技术.....	51
6.2.1.1 脉冲编码调制编码.....	51
6.2.1.2 预测编码.....	54
6.2.1.3 子带滤波编码.....	55
6.2.1.4 谱带复制技术 (Spectral Band Replication) .....	56
6.2.1.5 变量立体声技术 (Parametric Stereo) .....	57
6.2.2 音源编码技术.....	57
6.2.3 混合编解码.....	58
6.3 音频编码技术与应用.....	58
7. 音频压缩标准简介.....	59
7.1 MPEG-1 Audio.....	59
7.1.1 MPEG1 Layer 1 (ISO11172-3 Layer 1 / MP1) .....	61
7.1.2 MPEG1 Layer 2 (ISO11172-3 Layer 2 / MP2) .....	62
7.1.3 MPEG1 Layer 3 (ISO11172-3 Layer 3 / MP3) .....	62
7.2 MPEG-2 Audio.....	63
7.2.1 MPEG-2 Audio (ISO/IEC 13818-3) .....	64
7.2.2 MPEG 2.5.....	65
7.2.3 MPEG-2 AAC (MP4) .....	66

7.3 MPEG-4 Audio.....	68
7.3.1 MPEG-4 AAC (ISO14496-3) .....	68
7.3.2 AACplus.....	69
7.4 自适应多码率语音传输编解码器 (AMR) .....	70
7.4.1 AMR 编码器原理.....	70
7.4.2 非连续传输编码.....	71
7.5 WMA (Windows Media Audio) .....	71
7.6 乐器数字接口 (MIDI) .....	71
7.6.1 MIDI 的 3 个标准.....	72
7.6.2 波表合成器.....	72
7.6.3 MIDI 播放器结构.....	72
7.6.4 MIDI 播放器中 3 种效果.....	73
8. 各种音频标准对比.....	73
9. 文件格式.....	75
9.1 WAVE File Formats.....	75
9.2 IMA ADPCM.....	78
9.2.1 Fact Chunk.....	78
9.2.2 Wave Format Header.....	78
9.2.3 Block.....	79
9.2.4 Data.....	79
9.2.5 Decoding Algorithm.....	79
9.3 MS ADPCM.....	81
9.3.1 Fact Chunk.....	81
9.3.2 Wave Format Header.....	81
9.3.3 Block.....	82
9.3.4 Data.....	83
9.3.5 Padding.....	83
9.3.6 Decoding Algorithm.....	83
9.4 2Bit IMA ADPCM Algorithm.....	84
9.5 ADPCM DATA Format.....	85
9.6 MPEG Audio.....	87
9.6.1 MPEG Audio Format.....	87
9.6.2 MPEG Audio Frame Header .....	87
9.6.3 VBR Headers.....	89
9.6.4 XING Header.....	89
9.6.5 VBRI Header.....	90
9.6.6 Additional Tags.....	90
9.6.6.1 ID3V1.....	91
9.6.6.2 ID3V2.....	94
9.7 MIDI file format.....	99
9.8 MPEG file format.....	102
9.8.1 MPEG-1 system bitstream.....	102
9.8.2 MPEG-1 video bitstream.....	103
9.8.3 MPEG-2 transport stream.....	105

9.8.4 MPEG-2 program stream.....	105
9.8.5 MPEG-2 video bitstream.....	107
9.8.7 How to distinguish MPEG-1/2 bitstream without additional information .....	109
9.9 AVI file format.....	109
9.10 3GP file format.....	113
9.10.1 Evolution of 3GPP standard.....	113
9.10.2 3GP file format standard (3GPP TS 26.244) .....	114
9.10.2.1 Track Atoms.....	115
9.10.2.2 Media Atoms.....	115
9.10.2.3 Track Header.....	116
9.10.2.4 Media Header Atoms.....	117
9.10.2.5 Video Media Information Atoms .....	117
9.10.2.6 Sound Media Information Atoms .....	117
9.10.2.7 Sample Table Atoms.....	118
9.11 Real Media file format.....	119
9.11.1 Tagged File Formats.....	120
9.11.2 Sections of a RealMedia File.....	120
9.11.3 Header Section.....	120
9.11.4 RealMedia File Header.....	121
9.11.5 Properties Header.....	122
9.11.6 Media Properties Header.....	124
9.11.7 Logical Stream Organization.....	126
9.11.8 Logical Stream Organization.....	127
9.11.9 LogicalStream Structure.....	127
9.11.10 NameValueProperty Structure.....	129
9.11.11 Content Description Header.....	130
9.11.12 Data Section.....	131
9.11.13 Data Chunk Header.....	131
9.11.14 Data Packet Header.....	132
9.11.15 Index Section.....	134
9.11.16 Index Section Header.....	134
9.11.17 Index Record.....	135
9.11.18 Metadata Section.....	136
9.11.19 Metadata Section Header .....	136
9.11.20 Metadata Tag.....	136
9.11.21 Metadata Property Structure.....	137
9.11.22 PropListEntry Structure.....	139
9.11.23 Metadata Section Footer.....	140
9.11.24 ID3v1 Tag.....	140
9.12 Real Media file format.....	140
9.12.1 History.....	141
9.12.1.1 DivX and Spyware.....	143
9.12.2 Current Version.....	143

9.12.3 DivX Profiles.....	144
9.12.4 Quality, alternative MPEG-4 ASP implementations.....	145
10 文本编码.....	145
10.1 ASCII.....	145
概述.....	145
基本原理.....	146
优缺点.....	146
10.2 GSM7bit.....	146
概述.....	146
基本原理.....	146
优缺点.....	147
10.3 UCS-4 和 ISO 10646.....	147
概述.....	147
原理.....	147
优缺点.....	147
10.4 Unicode 和 UCS-2.....	147
概述.....	147
基本原理.....	148
优缺点.....	148
10.5 UTF-8.....	149
概述.....	149
基本原理.....	149
优缺点.....	150
10.6 UTF-16.....	150
概述.....	150
基本原理.....	150
优缺点.....	150
10.7 GB2312.....	151
概述.....	151
原理.....	151
优缺点.....	151
10.8 GB 12345.....	151
概述.....	151
基本原理.....	151
优缺点.....	151
10.9 GBK.....	152
概述.....	152
基本原理.....	152
优缺点.....	152
10.10 GB 18030.....	152
概述.....	152
优缺点.....	152
10.11 BIG5.....	153
概述.....	153

基本原理.....	153
优缺点.....	153
10.12 总结和比较表格.....	153

## 1. 图像/视频基础

图形、图像和视频是人类最容易接受的信息，它具有文字不可比拟的优点。图形、图像和视频数据量一般都很大，这对于存储、处理以及在通信线路上传输都带来巨大的负担。但图像和视频信息存在着大量的冗余，可以采用多种方法进行压缩。

### 1.1 颜色的基本概念

颜色是创建完美图像的基础。从许多方面说，在计算机上使用的颜色并没有什么不同，只不过它有一套特定的记录和处理颜色的技术。因此，要理解图像处理软件中所出现的各种有关颜色的术语，首先要具备基本的颜色理论知识。

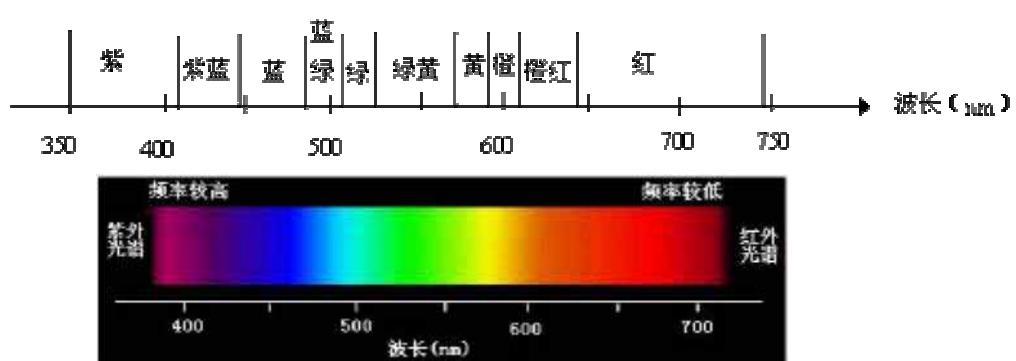
#### 1.1.1 颜色的物理性质

1666年，牛顿以三棱镜分解太阳光，发现看似无色的光线，经过三棱镜时，会依其波长和折射关系，依序分为红、橙、黄、绿、蓝、靛、紫七色光。不同波长的光线的色彩是不同的。其中红光波长较长，蓝光的波长较短。

眼睛对颜色的采样仅用相应于红、绿和蓝色三种锥体细胞，这些锥体细胞采样得到的信号通过大脑产生不同颜色的感觉，这些感觉由国际照明委员会(CIE)作了定义，用颜色的三个特性来区分颜色。这些特性是色调，饱和度和明度，它们是颜色所固有的并且是截然不同的特性。



(a) 电磁波谱



(b) 可见光谱

图 06-01-1 电磁波谱及可见光谱

#### 1.1.2 色调、饱和度和亮度

从人的视觉系统看，颜色可用色调(hue)、饱和度(saturation)和亮度来描述，其中色调与光波的波长有直接关系，亮度和饱和度与光波的幅度有关。人眼看到的任一彩色光都是这三个特性的

综合效果，这三个特性可以说是颜色的三要素。

### (一) 色调

色调(hue)又称为色相，指颜色的外观，取决于可见光谱中的光波的频率，它是最容易把颜色区分开的一种属性。

色调用红、橙、黄、绿、青、蓝、靛、紫(red, orange, yellow, green, cyan, blue, indigo, violet)等术语来刻画。苹果是红色的，这“红色”便是一种色调，它与颜色明暗无关。绘画中要求有固定的颜色感觉，有统一的色调，否则难以表现画面的情调和主题。例如我们说一幅画具红色调，是指它在颜色上总体偏红。

色调的种类很多，如果要仔细分析，可有一千万种以上，但普通颜色专业人士可辨认出的颜色大约可达三百至四百种。黑、灰、白则为无色彩。色调有一个自然次序：红、橙、黄、绿、青、蓝、靛、紫。在这个次序中，当人们混合相邻颜色时，可以获得在这两种颜色之间连续变化的色调。色调在颜色圆上用圆周表示，圆周上的颜色具有相同的饱和度和明度，但它们的色调不同，如图 06-01-2 所示，展开的情况如图 06-01-3 所示色环。

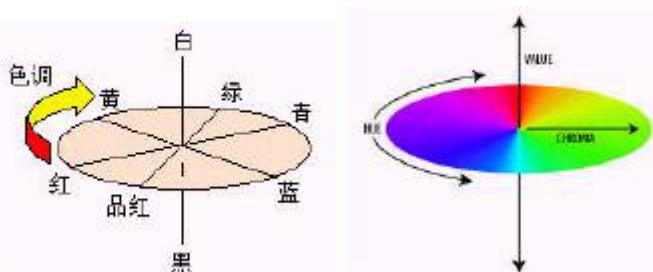


图 06-01-2 圆周表示色调

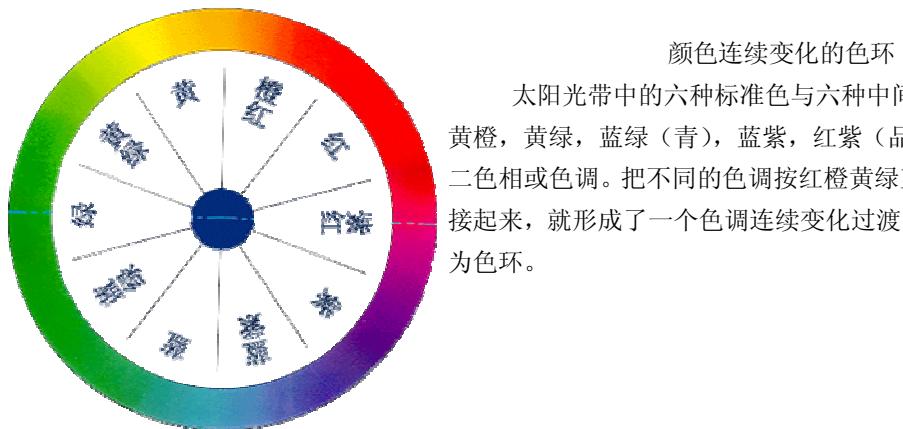


图 06-01-3 色环

用于描述感知色调的一个术语是色彩(colorfulness)。色彩是视觉系统对一个区域呈现的色调多少的感觉，例如，是浅蓝还是深蓝的感觉。

### (二) 饱和度

饱和度(saturation)是颜色的纯洁性，可用来区别颜色明暗的程度。当一种颜色渗入其他光成分愈多时，就说颜色愈不饱和。完全饱和的颜色是指没有渗入白光所呈现的颜色，例如仅由单一光波组成的光谱色就是完全饱和的颜色。饱和度在颜色圆上用半径表示，如图 06-01-4(a)所示。沿径向方向上的不同颜色具有相同的色调和明度，但它们的饱和度不同。例如在图 06-01-4(b)所示的七种颜色，它们具有相同的色调和明度，但具有不同的饱和度，左边的饱和度最浅，右边的饱和度最深。

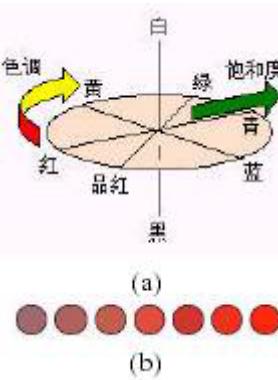


图 06-01-4 半径大小表示饱和度的深浅

### (三) 明度、亮度与光亮度

#### 1. 明度

根据国际照明委员会的定义，明度(brightness)是视觉系统对可见物体辐射或者发光多少的感知属性。

有色表面的明度取决于亮度和表面的反射率。由于感知的明度与反射率不是成正比，而认为是一种对数关系，因此在颜色度量系统中使用一个比例因子(例如，0~10)来表示明度。人们也已经发现，对于用不同光谱特性但发射流明数量相同的两个表面，它们被认为有相同的明度。明度的一个极端是黑色(没有光)，另一个极端是白色，在这两个极端之间是灰色。

在许多颜色系统中，明度常用垂直轴表示，如图 06-01-5(a)所示。例如在图 06-01-5(b)所示的七种颜色，它们具有相同的色调和饱和度，但它们的明度不同，底部的明度最小，顶部的明度最大。

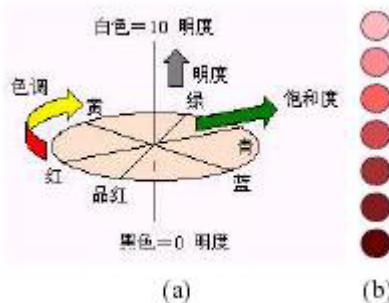


图 06-01-5 垂直轴表示明度

用纯正的颜色相互比较所产生的明暗差别是明度的典型例子。在纯正光谱中，黄色的明度最高，显得最亮；其次是橙、绿；再其次是红、蓝；紫色明度最低，显得最暗(图 06-01-6)。



图 06-01-6 纯正的颜色的明暗差别

明度和人的感知有关，目前还无法用物理设备来测量。

#### 2. 亮度

根据国际照明委员会的定义，亮度(luminance)是用反映视觉特性的光谱敏感函数加权之后得到的辐射功率(radiant power)，用单位面积上反射或者发射的光的强度表示。它的幅度与物理功率成正比，并在 555 nm 处达到峰值。在 CIE XYZ 系统中，亮度用 Y 表示。严格地说亮度应该使用像烛光/平方米(cd/m<sup>2</sup>)这样的单位来度量，但实际上是由指定的亮度即白光作参考，并把它标称化为 1 或者 100 个单位。例如，监视器用亮度为 80 cd/m<sup>2</sup>的白光作参考，并指定 Y = 1。

#### 3. 光亮度

$$L^* = 116 \times \sqrt[3]{Y/Y_n} - 16$$

其中  $(Y/Y_n) = 0.008856$ , Y 是 CIE XYZ 系统中定义的亮度,  $Y_n$  是参考白色光的亮度。

光亮度用作颜色空间的一个维, 而明度(brightness)则仅限用于发光体, 该术语用来描述反射表面或者透射表面。对计算机显示器显示的颜色, 除使用明度(brightness)之外, 也可使用光亮度(lightness)。因为虽然监视器是发射光的物体, 但显示的颜色是相对于监视器的白光而言的。

由于明度很难度量, 通常可以用亮度(luminance)即辐射的能量来度量。

#### (四) 色调、饱和度与亮度的关系

淡色的饱和度比浓色要低一些(图 06-01-7)。



图 06-01-7 淡色和浓色的饱和度对比

饱和度还和亮度有关, 同一色调越亮或越暗越不纯。

对于同一色调的彩色光, 饱和度越深, 颜色越鲜明或者说越纯, 相反则越淡(图 06-01-8)。



图 06-01-8 饱和度与纯度

在饱和的彩色光中增加白光的成分, 相当于增加了光能, 因而变得更亮了, 但是它的饱和度却降低了。若增加黑色光的成分, 相当于降低了光能, 因而变得更暗, 其饱和度也降低了。

饱和度越高, 颜色越艳丽、越鲜明突出, 越能发挥其颜色的固有特性。但饱和度高的颜色容易让人感到单调刺眼。饱和度低, 色感比较柔和协调, 可混色太杂则容易让人感觉浑浊, 色调显得灰暗。

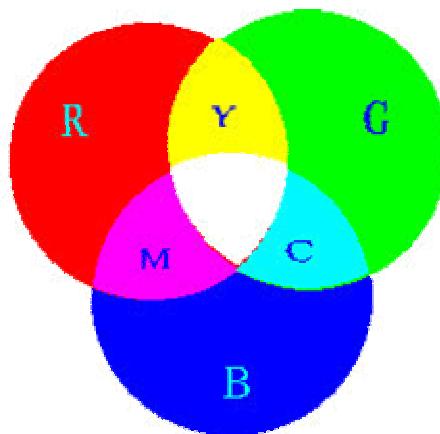
### 1.1.3 颜色的混合与互补

#### (一) 光的三基色模型

颜色的混合与颜料的混合不同。色光的基色或原色为红(R, Red)、绿(G, Green)、蓝(B, Blue)三色。

#### (二) 色光混合

三基色以不同的比例相混合, 可成为各种色光, 但基色却不能由其他色光混合而成。色光的混合是光量的增加, 所以三基色相混合(相加)而成白光(图 06-01-9)。



R—Red, 红

B—Blue, 蓝

G—Green, 绿

C—Cyan, 青

M—Magenta, 品红

Y—Yellow, 黄

图 06-01-9 相加混色

#### (三) 互补色

如果两种色光相混合能形成白光，则这两种色光互为补色（Complementary Colors）。如图 06-01-9 所示 R、C；G、M；B、Y 互为补色。互补色是彼此之间最不一样的颜色，这就是人眼能看到除了基色之外 其他色的原因。

## 1.2 颜色的空间表示

### 1.2.1 颜色空间及分类

#### (一) 颜色空间

颜色常用颜色空间来表示。颜色空间是用一种数学方法形象化表示颜色，人们用它来指定和产生颜色。例如，对于人来说，我们可以通过色调、饱和度和明度来定义颜色；对于显示设备来说，人们使用红、绿和蓝磷光体的发光量来描述颜色；对于打印或者印刷设备来说，人们使用青色、品红色、黄色和黑色的反射和吸收来产生指定的颜色。

颜色空间通常用 3 维模型表示，空间中的颜色能够看到或者使用颜色模型产生。颜色空间中的颜色通常用代表 3 个参数的 3 维坐标来描述，其颜色要取决于所使用的坐标。

图 06-02-1 表示使用色调、饱和度和明度构造的一种颜色空间，称为 HSB (hue, saturation and brightness) 颜色空间。色调用角度来标定，通常红色标为 0°，青色标为 180°；在径向方向上饱和度的深浅用离开中心线的距离表示；明度用垂直轴表示。

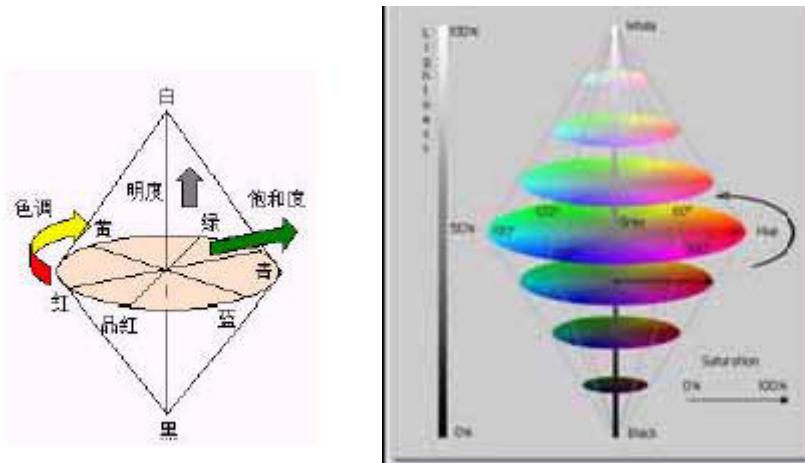


图 06-02-1 色调-饱和度-明度颜色空间

#### (二) 颜色空间分类

从技术上角度区分，颜色空间可考虑分成如下三类：

**RGB 型颜色空间/计算机图形颜色空间：**这类模型主要用于电视机和计算机的颜色显示系统。例如，RGB，HSI，HSL 和 HSV 等颜色空间。在显示技术和印刷技术中，颜色空间经常被称为颜色模型 (color mode)。“颜色空间”侧重于颜色的表示，而“颜色模型”侧重于颜色的生成。

**YUV 型颜色空间/电视系统颜色空间：**由广播电视需求的推动而开发的颜色空间，主要目的是通过压缩色度信息以有效地播送彩色电视图像。例如，YUV，YIQ，ITU-R BT. 601 Y' C<sub>b</sub>C<sub>r</sub>，ITU-R BT. 709 Y' C<sub>b</sub>C<sub>r</sub> 和 SMPTE-240M Y' P<sub>b</sub>P<sub>r</sub> 等颜色空间。

**XYZ 型颜色空间/CIE 颜色空间：**这类颜色空间是由国际照明委员会定义的颜色空间，通常作为国际性的颜色空间标准，用作颜色的基本度量方法。国际照明委员会定义的颜色空间是与设备无关的颜色表示法，在科学计算中得到广泛应用。对不能直接相互转换的两个颜色空间，可利用这类颜色空间作为过渡性的颜色空间，例如，CIE 1931 XYZ，L<sup>\*</sup>a<sup>\*</sup>b，L<sup>\*</sup>u<sup>\*</sup>v 和 LCH 等颜色空间就可作为过渡性的转换空间。

## 1.2.2 几种典型颜色空间

### 1.2.2.1 RGB 颜色空间

计算机颜色显示器显示颜色的原理与彩色电视机一样，都是采用 R、G、B 相加混色的原理，通过发射出三种不同强度的电子束，使屏幕内侧覆盖的红、绿、蓝磷光材料发光而产生颜色。这种颜色的表示方法称为 RGB 颜色空间表示。在多媒体计算机技术中，用得最多的是 RGB 颜色空间表示（图 06-01-9）。

根据三基色原理，用基色光单位来表示光的量，则在 RGB 颜色空间，任意色光 F 都可以用 R、G、B 三色不同分量的相加混合而成：

$$F = r [R] + g [G] + b [B]$$

RGB 颜色空间还可以用一个三维的立方体来描述（图 06-02-5）。

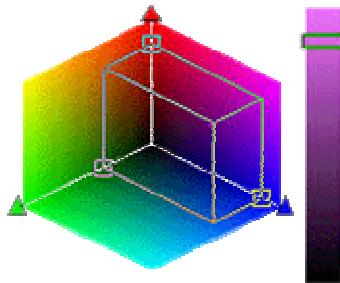


图 06-02-5 RGB 颜色空间

我们可知自然界中任何一种色光都可由 R、G、B 三基色按不同的比例相加混合而成，当三基色分量都为 0（最弱）时混合为黑色光；当三基色分量都为 k（最强）时混合为白色光。任一颜色 F 是这个立方体坐标中的一点，调整三色系数 r、g、b 中的任一系数都会改变 F 的坐标值，也即改变了 F 的色值。RGB 颜色空间采用物理三基色表示，因而物理意义很清楚，适合彩色显像管工作。然而这一体制并不适应人的视觉特点。因而，产生了其他不同的颜色空间表示法。

### 1.2.2.2 HSI 颜色空间

HSI (Hue, Saturation and Intensity) 颜色空间是从人的视觉系统出发，用色调 (Hue)、色饱和度 (Saturation 或 Chroma) 和亮度 (Intensity 或 Brightness) 来描述颜色。HSI 颜色空间可以用一个圆锥空间模型来描述（图 06-02-6）。

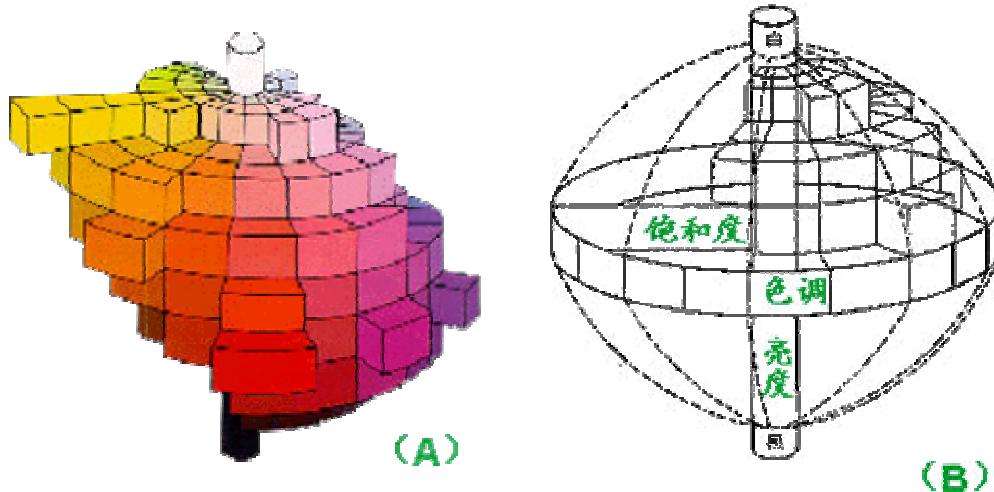


图 06-02-6 HSI 颜色圆锥空间模型

通常把色调和饱和度通称为色度，用来表示颜色的类别与深浅程度。由于人的视觉对亮度的敏感程度远强于对颜色浓淡的敏感程度，为了便于颜色处理和识别，人的视觉系统经常采用 HSI 颜色空间，它比 RGB 颜色空间更符合人的视觉特性。在图像处理和计算机视觉中大量算法都可在 HSI 颜色空间中方便地使用，它们可以分开处理而且是相互独立的。因此，在 HSI 颜色空间可以大大简化图像分析和处理的工作量。

HSI 颜色空间和 RGB 颜色空间只是同一物理量的不同表示法，因而它们之间存在着转换关系，如公式所示：

$$\begin{cases} I = \frac{R+G+B}{3} \\ H = \frac{1}{360} [90 - Arc \tan(\frac{F}{\sqrt{3}}) + \{0, G > B \geq 180, G < B\}] \\ S = 1 - [\frac{\min(R, G, B)}{I}] \end{cases}$$

其中  $F = \frac{2R-G-B}{G-B}$

### 1.2.2.3 YUV (Lab) 颜色空间

在现代彩色电视系统中，通常采用三管彩色摄像机或彩色 CCD（电耦合器件）摄像机，它把得到的彩色图像信号，经分色、分别放大校正得到 RGB，再经过矩阵变换电路得到亮度信号 Y 和两个色差信号 R-Y、B-Y，最后发送端将亮度和色差三个信号分别进行编码，用同一信道发送出去。这就是我们常用的 YUV 颜色空间。

采用 YUV 颜色空间的重要性是它的亮度信号 Y 和色度信号 U、V 是分离的。如果只有 Y 信号分量而没有 U、V 分量，那么这样表示的图就是黑白灰度图。彩色电视采用 YUV 空间正是为了用亮度信号 Y 解决彩色电视机与黑白电视机的兼容问题，使黑白电视机也能接收彩色信号。

根据美国国家电视制式委员会 NTSC 制式的标准，当白光的亮度用 Y 来表示时，它和红、绿、蓝三色光的关系可用如下式的方程描述：

$$Y = 0.3 R + 0.59 G + 0.11 B$$

这就是常用的亮度公式。色差 U、V 是由 B-Y、R-Y 按不同比例压缩而成的。YUV 颜色空间与 RGB 颜色空间的转换关系如下：

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.3 & 0.59 & 0.11 \\ -0.15 & -0.29 & 0.44 \\ 0.61 & -0.52 & -0.096 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

如果要由 YUV 空间转化成 RGB 空间，只要进行相应的逆运算即可。

与 YUV 颜色空间类似的还有 Lab 颜色空间，它也是用亮度和色差来描述颜色分量，其中 L 为亮度、a 和 b 分别为各色差分量。

### 1.2.2.4 CMY 颜色空间

彩色印刷或彩色打印的纸张是不能发射光线的，因而印刷机或彩色打印机就只能使用一些能够吸收特定的光波而反射其他光波的油墨或颜料。油墨或颜料的 3 基色是青 (Cyan)、品红 (Magenta)

和黄(Yellow)，简称为 CMY。青色对应蓝绿色，品红对应紫红色。理论上说，任何一种由颜料表现的颜色都可以用这三种基色按不同的比例混合而成，这种颜色表示方法称 CMY 颜色空间表示法。彩色打印机和彩色印刷系统都采用 CMY 颜色空间。

用 CMY 模型产生的颜色被称为相减色，是因为它减少了为视觉系统识别颜色所需要的反射光。在 CMY 相减混色中，三基色等量相减时得到黑色；等量黄色(Y)和品红(M)相减而青色(C)为 0 时，得到红色(R)；等量青色(C)和品红(M)相减而黄色(Y)为 0 时，得到蓝色(B)；等量黄色(Y)和青色(C)相减而品红(M)为 0 时，得到绿色(G)。这些三基色相减结果如图 06-02-7 所示。

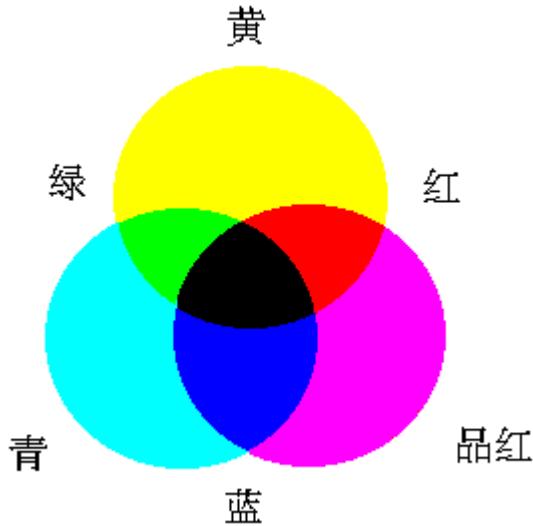


图 06-02-7 三基色相减

CMY 空间正好与 RGB 空间互补，也即用白色减去 RGB 空间中的某一颜色值就等于同样颜色在 CMY 空间中的值。RGB 空间与 CMY 空间的互补关系如表 06-02-1 所示。

表 06-02-1 RGB 空间与 CMY 空间的互补关系

RGB 相加混色	CMY 相减混色	对应颜色
0 0 0	1 1 1	黑色
0 0 1	1 1 0	蓝色
0 1 0	1 0 1	绿色
0 1 1	1 0 0	青色
1 0 0	0 1 1	红色
1 0 1	0 1 0	品红色
1 1 0	0 0 1	黄色
1 1 1	0 0 0	白色

根据这个原理，很容易把 RGB 空间转换成 CMY 空间。由于彩色墨水和颜料的化学特性，用等量的 CMY 三基色得到的黑色不是真正的黑色，因此在印刷术中常加一种真正的黑色 (black ink)，所以 CMY 又写成 CMYK。

实际应用中，一幅图像在计算机中用 RGB 空间显示；用 RGB 或 SHI 空间编辑处理；打印输出时要转换成 CMY 空间；如果要印刷，则要转换成 CMYK 四幅印刷分色图，用于套印彩色印刷品。

### 1.2.2.5 YIQ 模型

YIQ 模型与 YUV 模型非常类似，是在彩色电视制式中使用的另一种重要的颜色模型，NTSC 彩色电视制式中使用。这里的 Y 表示亮度，I、Q 是两个彩色分量。YIQ 和 RGB 的对应关系用下面的方程式表示：

$$Y = 0.299R + 0.587G + 0.114B$$

$$I = 0.596R - 0.275G - 0.321B$$

$$Q = 0.212R - 0.523G + 0.311B$$

或者写成矩阵的形式，

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.523 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

### 1.2.2.6 YCrCb 模型

YC<sub>r</sub>C<sub>b</sub> 模型适用于计算机用的显示器。它也是使用 Y、C<sub>r</sub> 和 C<sub>b</sub> 来分别表示一种亮度分量信号和两种色度分量信号。YC<sub>r</sub>C<sub>b</sub> 模型与 RGB 空间的转换关系如下：

$$Y = 0.299R + 0.578G + 0.114B$$

$$C_r = (0.500R - 0.4187G - 0.0813B) + 128$$

$$C_b = (-0.1687R - 0.3313G + 0.500B) + 128$$

或者写成矩阵的形式，

$$\begin{bmatrix} Y \\ C_r \\ C_b \end{bmatrix} = \begin{bmatrix} 0.299 & 0.578 & 0.114 \\ 0.500 & -0.4187 & -0.0813 \\ -0.1687 & -0.3313 & 0.500 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix}$$

RGB 与 YC<sub>r</sub>C<sub>b</sub> 之间的变换关系可写成如下的形式，

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 1.4020 & 0 \\ 1 & -0.7141 & -0.3441 \\ 1 & 0 & 1.7720 \end{bmatrix} \begin{bmatrix} 0 \\ C_r - 128 \\ C_b - 128 \end{bmatrix}$$

### YCbCr 采样格式

4:4:4 采样就是说三种元素 Y, C<sub>b</sub>, C<sub>r</sub> 有同样的分辨率，这样的话，在每一个像素点上都对这三种元素进行采样。数字 4 是指在水平方向上对于各种元素的采样率，比如说，每四个亮度采样点就有四个 C<sub>b</sub> 的 C<sub>r</sub> 采样值。4:4:4 采样完整地保留了所有的信息值。4:2:2 采样中（有时记为 YUY2），色度元素在纵向与亮度值有同样的分辨率，而在横向则是亮度分辨率的一半（4:2:2 表示每四个亮度值就有两个 C<sub>b</sub> 和 C<sub>r</sub> 采样。）4:2:2 视频用来构造高品质的视频彩色信号。

在流行的 4:2:0 采样格式中（常记为 YV12）C<sub>b</sub> 和 C<sub>r</sub> 在水平和垂直方向上有 Y 分辨率的一半。4:2:0 有些不同，因为它并不是指在实际采样中使用 4:2:0，而是在编码史中定义这种编码方法是用来区别于 4:4:4 和 4:2:2 方法的。4:2:0 采样被广泛地应用于消费应用中，比如视频会议，数字电视和 DVD 存储中。因为每个颜色差别元素中包含了四分之一的 Y 采样元素量，那么 4:2:0 YCbCr 视频需要刚好 4:4:4 或 RGB 视频中采样量的一半。

4:2:0 采样有时被描述是一个“每像素 12 位”的方法。这么说的原因可以从对四个像素的采样中

看出。使用 4:4:4 采样，一共要进行 12 次采样，对每一个 Y, Cb 和 Cr，就需要  $12 \times 8 = 96$  位，平均下来要  $96/4 = 24$  位。使用 4:2:0 就需要  $6 \times 8 = 48$  位，平均每个像素  $48/4 = 12$  位。

4:1:1 采样格式中 Cb 和 Cr 在水平方向上是 Y 分辨率的 1/4，在垂直方向方向和 Y 分辨率相同。4:1:1 采样通常应用在 DV 存储中。

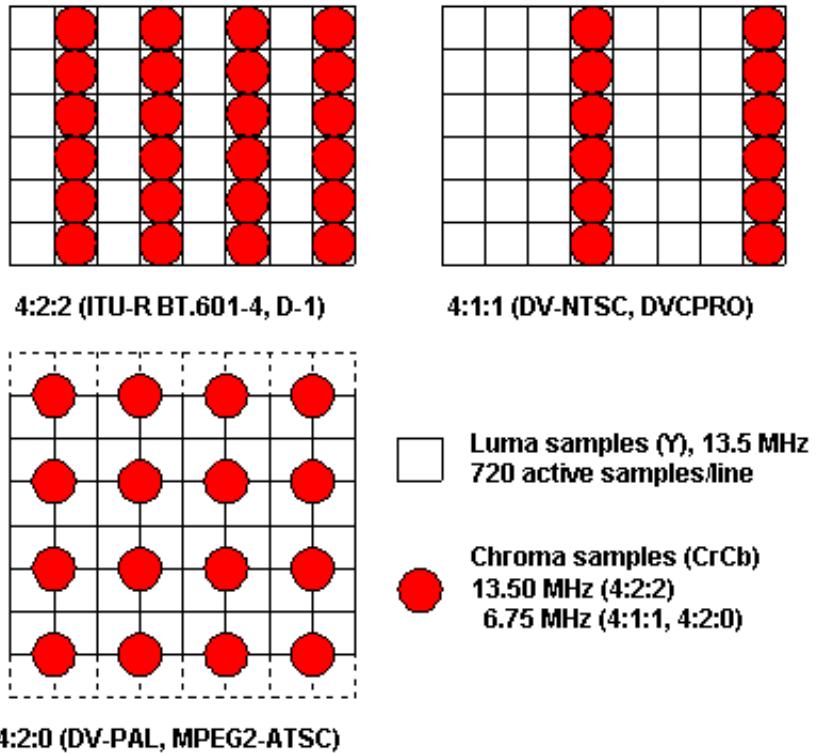


Figure 1. A comparison of chroma sampling in 4:2:2, 4:1:1, and 4:2:0 formats.

### 1.3 数字图像与视频

人眼能识别的自然景象或图像原来也是一种模拟信号，为了使计算机能够记录和处理图像、图形，必须首先使其数字化。数字化后的图像、图形称为数字图像、数字图形，一般也简称为图像、图形。根据描述方法不同，计算机中的图形图像可分为矢量图和位图两种基本形式。根据图像包含色彩信息的多少，数字图像可分为黑白图、灰度图和彩色图。

#### 1.3.1 灰度图和彩色图

灰度图按照灰度等级的数目来划分。只有黑白两种颜色的图像称为单色图像。每个像素的像素值用 1 位二进制数存储，它的值只有“0”或者“1”。彩色图像可按照颜色的数目来划分，分为 8 位、16 位、真彩色等等。图中所示就是黑白图、灰度图和彩色图。



### 1.3.2 矢量图和位图

矢量图 (vector-based image) 是用一系列计算机指令来描述图形，如画点、画曲线、画圆、画矩形等。实际上是用数学的方法来描述一幅图像。例如用矢量图描述一个圆，图像中纪录的是圆心位置，圆的半径以及圆弧的粗细、颜色等信息。其优点是：(1) 缩放、旋转、移动时图像不会失真。

(2) 存储和传输时数据量较小。缺点是：(1) 图像显示时花费时间比较长。(2) 真实世界的彩色图像难以转化为矢量图。

位图，是用像素点来描述或映射的图，也即位映射图 (bit-mapped image)。位图在内存中也就是一组计算机内存位 (bit) 组成，这些位定义图像中每个像素点的颜色和亮度。位图一般也称为图像。

位图可以采用将自然图像进行模数转换 (AD) 的方式来获取，这个过程称为图像的扫描。一幅位图是由许多描述每个像素的数据组成的，这些数据通常称为图像数据，而这些数据作为一个文件来存储，这种文件称为图像文件。

其优点是：(1) 显示速度快。(2) 真实世界的图像可以通过扫描仪、数码相机、摄像机等设备方便的转化为点位图。缺点是：(1) 存储和传输时数据量比较大。(2) 缩放、旋转时算法复杂且容易失真。

通过位图与矢量图的比较，可以进一步了解这两者的关系和含义（表 06-03-1）。

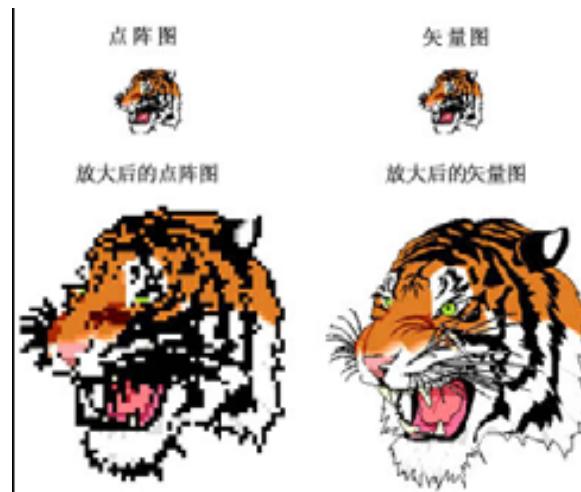


表 06-03-1 位图与矢量图比较

	文件内容	容量	显示速度	应用特点
--	------	----	------	------

矢量图	图形指令	与图的复杂程度有关	图越复杂,需执行的指令越多, 显示越慢	易于编辑, 适于“绘制”和“创建”。但表现力受限
位图	图像点阵数据	与图的尺寸、颜色有关	与图的容量有关	适于“获取”和“复制”, 表现力丰富, 但编辑较复杂

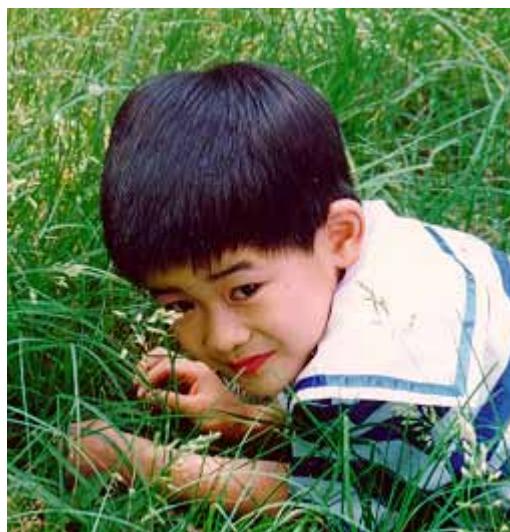
### 1.3.3 分辨率

分辨率包括显示分辨率和图像分辨率。

显示分辨率是确定屏幕上显示图像的区域的大小。显示分辨率有最大显示分辨率和当前显示分辨率之别。最大显示分辨率是由物理参数, 即显示器和显示卡(显示缓存)决定的。当前显示分辨率是由当前设置的参数决定的。

图像分辨率是确立组成一幅图像的像素数目, 图像分辨率用每英寸多少点(dpi, dot per inch)表示, 指组成一幅图像的像素密度的度量方法。对同样大小的一幅原图, 如果数字化时图像分辨率高, 则组成该图的像素点数目越多, 看起来就越逼真。图像分辨率在图像输入/输出时起作用; 它决定图像的点阵数。而且, 不同的分辨率会造成不同的图像清晰度。

按照不同的图像分辨率来扫描图像, 可以看出图像分辨率与显示分辨率的关系和不同。如果图像的点数大于显示分辨率的点数, 则该图像在显示器上只能显示出图像的一部分。只有当图像大小与显示分辨率相同时, 一幅图像才能充满整屏。对于14英寸的显示器, 当显示分辨率设置为800×600时, 屏幕上的一英寸约有72个像素点, 这时若用72dpi的图像分辨率来扫描一幅图像, 则屏幕上显示的图像大小基本就是原图的大小(图06-03-3)。



图像A 200dpi



图像B 50dpi



图像C 图像B放大  
四倍

图 06-03-3 按照不同的图像分辨率来扫描图像

如果用 200 dpi 来扫描一幅彩色照片的局部，得到一幅  $253 \times 265$  个像素的图 06-03-3 A。如果用 50 dpi 来扫相同的画面，则得到  $63 \times 66$  个像素的图 06-03-3 B。图 06-03-3 B 的边长只有图像 A 的四分之一大小。如果要使图像 B 在屏幕上达到图像 A 的大小，则需把图像 B 的点阵放大，如一点变成颜色相同的四点，与图像 A 相比，所得到的图 06-03-3 C 显然很粗糙。

#### 1.3.4 图像深度与颜色类型

图像深度是指位图中记录每个像素点所占的位数，它决定了彩色图像中可出现的最多颜色数，或者灰度图像中的最大灰度等级数。图像的颜色需用三维空间来表示，如 RGB 颜色空间，而颜色的空间表示法又不是惟一的，所以每个像素点的图像深度的分配还与图像所用的颜色空间有关。以最常用的 RGB 颜色空间为例，图像深度与颜色的映射关系主要有真彩色、伪彩色和直接色。

(一) 真彩色 (true-color): 真彩色是指图像中的每个像素值都分成 R、G、B 三个基色分量，每个基色分量直接决定其基色的强度，这样产生的颜色称为真彩色。例如图像深度为 24，用 R: G: B = 8: 8: 8 来表示颜色，则 R、G、B 各用 8 位来表示各自基色分量的强度，每个基色分量的强度等级为  $2^8 = 256$  种。图像可容纳  $2^{24} = 16M$  种颜色。这样得到的颜色可以反映原图的真实颜色，故称真彩色。

(二) 伪彩色 (pseudo-color): 伪彩色图像的每个像素值实际上是一个索引值或代码，该代码值作为颜色查找表 (CLUT, Color Look-Up Table) 中某一项的入口地址，根据该地址可查找出包含实际 R、G、B 的强度值。这种用查找映射的方法产生的颜色称为伪彩色。用这种方式产生的颜色本身是真的，不过它不一定反映原图的颜色。在 VGA 显示系统中，调色板就相当于颜色查找表。从 16 色标准 VGA 调色板的定义可以看出这种伪彩色的工作方式 (表 06-03-2)。调色板的代码对应 RGB 颜色的入口地址，颜色即调色板中 RGB 混合后对应的颜色。

表 06-03-2 16 色标准 VGA 调色板

代码	R	G	B	颜色名称	效果
0	0	0	0	黑 (Black)	
1	0	0	128	深蓝 (Navy)	
2	0	128	0	深绿 (Dark Green)	

3	0	128	128	深青(Dark Cyan)	
4	128	0	0	深红(Maroon)	
5	128	0	128	紫(Purple)	
6	128	128	0	橄榄绿(Olive)	
7	192	192	192	灰白(Light gray)	
8	128	128	128	深灰(Dark gray)	
9	0	0	255	蓝(blue)	
10	0	255	0	绿(green)	
11	0	255	255	青(cyan)	
12	255	0	0	红(red)	
13	255	0	255	品红(magenta)	
14	255	255	0	黄(Yellow)	
15	255	255	255	白(white)	

伪彩色一般用于 65K 色以下的显示方式中。标准的调色板是在 256K 色谱中按色调均匀地选取 16 种或 256 种颜色。一般应用中，有的图像往往偏向于某一种或几种色调，此时如果采用标准调色板，则颜色失真较多。因此，同一幅图像，采用不同的调色板显示可能会出现不同的颜色效果（图 06-03-4）。

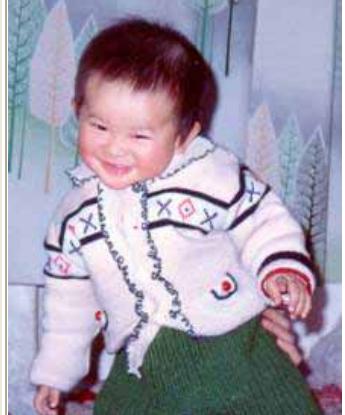
图像	调色板	说明
	无	真彩色显示，无需调色板



图 06-03-4 调色板的影响

(三) 直接色 (direct-color): 直接色的获取是通过每个像素点的 R、G、B 分量分别作为单独的索引值进行变换，经相应的颜色变换表找出各自的基色强度，用变换后的 R、G、B 强度值产生的颜色。

直接色与伪彩色相比，相同之处是都采用查找表，不同之处是前者对 R、G、B 分量分别进行查找变换，后者是把整个像素当作查找的索引进行查找变换。因此，直接色的效果一般比伪彩色好。

直接色与真彩色比，相同之处是都采用 R、G、B 分量来决定基色强度，不同之处是前者的基色强度是由 R、G、B 经变换后得到的，而后者是直接用 R、G、B 决定。在 VGA 显示系统中，用直接色可以得到相当逼真的彩色图像，虽然其颜色数受调色板的限制而只有 256 色。

### 1.3.5 色彩深度

存储每个像素信息所用的二进制位数。像素深度决定彩色图像每个像素可能有的颜色数，或者确定灰度图像每个像素可能有的灰度级数。黑白图像的每个像素只有 1bit 信息，在 16 色图像中，颜色的编号可以是 0~15 间的任一个值，由于存储 16 种不同的颜色需要 4 个信息位，所以 16 色模式叫“4 比特”模式。同样，在 256 色模式中，每个象素颜色编号的取值可高达 255，要存储象素的颜色需要 8 个信息位。而我们常用的 RGB565 格式就是分别用 5bit、6bit、5bit 表示红色、绿色、蓝色分量，可以表示  $2^{16} = 65536$  种颜色。

### 1.3.6 图像数据的容量

在扫描生成一幅图像时，实际上就是按一定的图像分辨率和一定的图像深度对模拟图片或照片进行采样，从而生成一幅数字化的图像。图像的分辨率越高、图像深度越深，则数字化后的图像效果

越逼真、图像数据量也越大。按照像素点及其深度映射的图像数据大小可用下面的公式来估算：

$$\text{图像数据量} = \text{图像的总像素} \times \text{图像深度} / 8 \text{ (Byte)}$$

一幅  $640 \times 480$ 、真彩色的图像，其文件大小约为：

$$640 \times 480 \times 24 / 8 = 1 \text{ M (Bytes)}$$

通过以上的分析，我们可知如果要确定一幅图像的参数，要考虑的因素一是图像的容量，二是图像输出的效果。在多媒体应用中，更应考虑好图像容量与效果的关系。由于图像数据量很大，因此，数据的压缩就成为图像处理的重要内容之一。

### 1.3.7 视频概念

静止的画面叫图像 (picture)。连续的图像变化每秒超过 24 帧 (frame) 画面以上时，根据视觉暂留原理，人眼无法辨别每帧单独的静态画面，看上去是平滑连续的视觉效果。这样的连续画面叫视频 (video)。视频是在时间上连续的一系列图像帧的集合。当连续图像变化每秒低于 24 帧画面时，人眼有不连续的感觉，叫动画 (cartoon)。

## 2. 图像/视频压缩技术介绍

### 2.1 视频压缩的基本概念

数据压缩的基本方法分两类：一种是将相同的或相似的数据或数据特征归类，使用较少的数据量描述原始数据，达到减少数据量的目的。这种压缩一般为无损压缩。第二类方法是利用人眼的视觉特性有针对性地简化不重要的数据，以减少总的数据量。这种压缩一般为有损压缩，只要损失的数据不太影响人眼主观接收的效果，就可采用。

图像数据的压缩主要基于两点原理：

1. 原始图像信息存在着很大的冗余度，数据之间存在着相关性，如相邻像素之间颜色的相关性及相邻图像帧之间内容的相关性等。

2. 其次是因为在多媒体系统的应用领域中，人眼是图像信息的接收端。因此，可利用人的视觉对于边缘急剧变化不敏感（视觉掩盖效应），以及人眼对图像的亮度信息敏感、对颜色分辨率弱的特点实现高压缩比，而解压缩后的图像信号仍有着满意的主观质量。

图像压缩的主要参数之一是图像压缩比。图像压缩比的定义与音频数据压缩比的定义类似，即为压缩后的图像数据量与压缩前的图像数据量之比：

$$\text{图像数据压缩比} = \frac{\text{压缩后的图像数据量}}{\text{压缩前的图像数据量}}$$

### 2.2 颜色空间与压缩图像数据

由于人眼对颜色细节的分辨能力远比对亮度细节的分辨能力低，若把人眼刚能分辨的黑白相间的条纹换成不同颜色的彩色条纹，那么眼睛就不再能分辨出条纹来。如图 06-04-1 所示，等宽的蓝红相间的彩条，蓝绿相间的彩条和黑白相间的条纹比较。使眼睛逐渐远离屏幕，当你分辨不出彩条时，黑白条还能分辨出来。

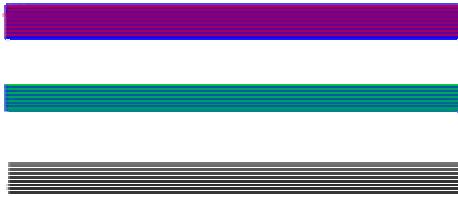


图 06-04-1 亮度和颜色分辨率

根据这个原理，利用不同的颜色空间也能压缩图像数据。保持亮度分量的分辨率而把彩色分量的分辨率降低，这样并不会明显降低图像的质量。实际中可以把几个相邻像素的颜色值当作相同的颜色值来处理，也即用“大面积着色原理”，从而减少所需的存储容量。

实际应用中的示例如采用 RGB 和 YUV 颜色空间来记录真彩色图像。RGB 空间的图像深度为 R: G: B=8: 8: 8，而 YUV 空间的图像深度可为 Y: U: V=8: 4: 4 或者是 Y: U: V=8: 2: 2。后者具体的做法是对亮度信号 Y，每个像素都数字化为 8bit（256 级亮度），而 U, V 色差信号每四个像素用一个 8 bit 数据表示，即粒度变大，相当于每个像素只用了 2 bit 数据。这样，将一个像素用 24bit 表示压缩为用 12bit 表示，存储空间压缩一倍，压缩比为 1: 2，而人的眼睛却基本感觉不出来。电视信号的传送就是根据这一原理。

## 2.3 去冗余技术

视频图像数据有极强的相关性，也就是说有大量的冗余信息。其中冗余信息可分为空域冗余信息和时域冗余信息。压缩技术就是将数据中的冗余信息去掉（去除数据之间的相关性），压缩技术包含帧内图像数据压缩技术、帧间图像数据压缩技术和熵编码压缩技术。

### ● 时域压缩技术

使用帧间编码技术(Interframe coding)可去除时域冗余信息，它包括以下两部分：

#### — 运动补偿

运动补偿是通过先前的局部图像来预测、补偿当前的局部图像，它是减少帧序列冗余信息的有效方法。

#### — 运动估计

运动估计是从视频序列中抽取运动信息的一整套技术。

注：通用的压缩标准都使用基于块的运动估计和运动补偿。

### ● 空间域压缩技术

主要使用帧内编码技术(Intraframe coding)：

#### — 预测编码

预测编码的基本思想是：根据数据的统计特性得到预测值，然后传输图像像素与其预测值的差值信号，使传输的码率降低，达到压缩的目的。预测编码方法简单经济，编码效率较高。

#### — 变换编码

帧内图像和预测差分信号都有很高的空域冗余信息。变换编码将空域信号变换到另一正交矢量空间，使其相关性下降，数据冗余度减小。

#### — 量化编码

经过变换编码后，产生一批变换系数，对这些系数进行量化，使编码器的输出达到一定的位率。这一过程导致精度的降低。

### ● 熵编码

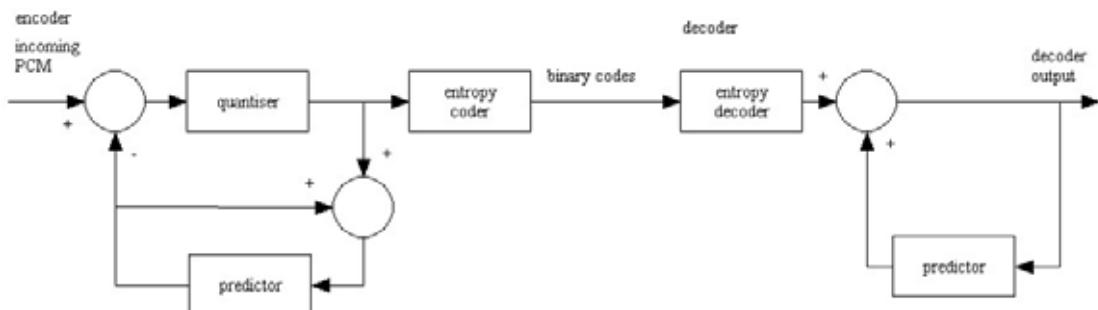
熵编码是无损编码。它对变换、量化后得到的系数和运动信息，进行进一步的压缩。

### 2.3.1 空间域压缩技术

#### 2.3.1.1 预测编码 (Predictive coding)

电视图像基本上是由面积较大的像块(如蓝天、大地、服装等)组成。虽然每个像块的幅值各不相同，但像块内各样值的幅度是相近的或相同的，幅值跃变部分相当于像块的轮廓，只占整幅图像的很小一部分。帧间相同的概率就更大了，静止图像相邻帧间的相应位置的像素完全一样，这意味着前后像素之差或前后帧间相应位置像素之差为零或差值小的概率大，差值大的概率小。这就是差值编码(DPCM: differential pulse code modulation)的基本想法。

如果差值编码中小幅度出现的机会增加，由于其对应的码长较短，总数码率会进一步减小。如果能猜出下一个样值，那么差值就会是零，当然这种情况是没有意义的，因为若预先知道下一样值，就不需要进行通信了。但可以肯定，如果我们不仅利用前后样值的相关性，同时也利用其它行、其它帧的像素的相关性，用更接近当前样值的预测值与当前样值相减，小幅度差值就会增加，总数码率就会减小，这就是预测编码的方法。其原理框图见图。



#### 2.3.1.2 变换编码 (Transform coding)

变换编码将空域信号变换到另一正交矢量空间，使其相关性下降，数据冗余度减小。目前常用的变换编码方法是离散余弦变换。

离散余弦变换(Discrete cosine Transform)简称 DCT。DCT 是一个无损的，可逆的数学过程，它把空间幅度数据转化为空间频率数据。在用于视频压缩时，这一运算过程是以亮度采样和相应的色差采样构成的 8\*8 点的块为单位进行的。左上角的 DCT 系数反映块的 DC(直流)分量，位于 DC 分量下方的系数代表着逐渐增高的垂直空间频率，位于 DC 分量右侧的系数代表着逐渐增高的水平空间频率，其他系数则代表垂直水平空间频串的不同组合。

由于视频图象的自然属性，大多数图像的高频分量较小，相当于图像高频成分的系数经常为零，加上人眼对高频成分的失真不太敏感，所以可用更粗的量化，因此传送变换系数所用的数码率要大大小于传送图像像素所用的数码率。到达接收端后再通过反离散余弦变换回到样值，虽然会有一定的失真，但人眼是可以接受的。

### 2.3.1.3 量化 (Quantisation)

严格说 DCT 本身并不能进行码率压缩，因为 64 个样值仍然得到 64 个系数，如图 4-2 所示。这里给出了一个  $8 \times 8$  像块的具体例子，经 DCT 变换后，比特数增加了。在这个例子中样值是 8 比特，从  $0 \sim 225$  得到的 XX，即直流分量的最大值是原来 256 的  $64/8$  倍，即  $0 \sim 2047$ ，交流分量的范围是  $-1024 \sim 1023$ 。只是在经过量化后，特别是按人眼的生理特征对低频分量和高频分量设置不同的量化，会使大多数高频分量的系数变为零。一般说来，人眼对低频分量比较敏感，而对高频分量不太敏感。因此对低频分量采用较细的量化，而对高频分量采用较粗的量化。

所谓量化，即根据不同的要求，设置不同的量化等级，从而降低码率。

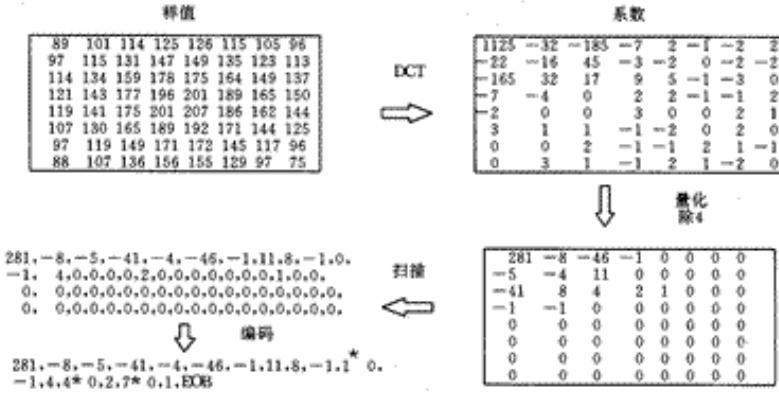


图 4-2 DCT 变换例

### 2.3.2 熵编码

#### Entropy(熵)的概念

1. 熵(Entropy)是信息量的度量方法，它表示某一事件出现的消息越多，事件发生的可能性就越小，数学上就是概率越小。
2. 某个事件的信息量用  $I_i = -\log_2 p_i$  表示，其中  $p_i$  为第  $i$  个事件的概率， $0 < p_i \leq 1$

#### 信源 $S$ 的熵的定义

按照仙农(Shannon)的理论，信源  $S$  的熵定义为

$$H(S) = \eta = \sum_i p_i \log_2 (1/p_i)$$

其中  $p_i$  是符号  $s_i$  在  $S$  中出现的概率； $\log_2 (1/p_i)$  表示包含在  $s_i$  中的信息量，也就是编码  $s_i$  所需要的位数。

例如，有一幅 40 个象素组成的灰度图像，灰度共有 5 级，分别用符号 A、B、C、D 和 E 表示，40 个象素中出现灰度 A 的象素数有 15 个，出现灰度 B 的象素数有 7 个，出现灰度 C 的象素数有 7 个等等，如表 4-01 所示。如果用 3 个位表示 5 个等级的灰度值，也就是每个象素用 3 位表示，编码这幅图像总共需要 120 位。

表 4-01 符号在图像中出现的数目

符 号	A	B	C	D	E

出现的次数	15	7	7	6	5
-------	----	---	---	---	---

按照仙农理论，这幅图像的熵为

$$H(S) = (15/40) \log_2 (40/15) + (7/40) \log_2 (40/7) + (7/40) \log_2 (40/7) + (6/40) \log_2 (40/6) + (5/40) \log_2 (40/5) \\ = 2.196$$

这就是说每个符号用 2.196 位表示，40 个象素需用 87.84 位。

熵编码是无损编码。它对变换、量化后得到的系数和运动信息，进行进一步的压缩。主要包括游程长度编码，霍夫曼编码和算术编码。

### 2.3.2.1 游程长度编码 (Run Length coding)

读出数据和表示数据的方式也是减少码率的一个重要因素。读出的方式可以有多种选择，如水平逐行读出、垂直逐列读出、之字型读出和交替读出等，其中之字型读出 (Zig-Zag) 是最常用的一种。由于经 DCT 变换以后，系数大多数集中在左上角，即低频分量区，因此之字型读出实际上是按二维频率的高低顺序读出系数的，这样一来就为游程长度编码 (Runlength Encoding) 创造了条件。所谓游程长度编码是指一个码可同时表示码的值和前面几个零，这样就可以把之字型读出的优点显示出来了。因为之字型读出在大多数情况下出现连零的机会比较多，尤其在最后，如果都是零，在读到最后一个数后只要给出“块结束” (EOB) 码，就可以结束输出，因此节省了很多码率。

游程长度指的是由字构成的数据流中各个字符连续重复出现而形成字符串的长度。基本的游程编码就是在数据流中直接用三个字符来给出上述三种信息，其数据结构如图 4-3 所示。

### 2.3.2.2 霍夫曼编码 (Huffman coding)

霍夫曼编码是可变字长编码 (VLC) 的一种。Huffman 于 1952 年提出一种编码方法，该方法完全依据字符出现概率来构造异字头的平均长度最短的码字，有时称之为最佳编码，一般就叫作 Huffman 编码。

霍夫曼编码的具体方法：先按出现的概率大小排队，把两个最小的概率相加，作为新的概率 和剩余的概率重新排队，再把最小的两个概率相加，再重新排队，直到最后变成 1。每次相加时都将“0”和“1”赋与相加的两个概率，读出时由该符号开始一直走到最后的“1”，将路线上所遇到的“0”和“1”按最低位到最高位的顺序排好，就是该符号的霍夫曼编码。

例：设将信源符号按出现的概率大小顺序排列为

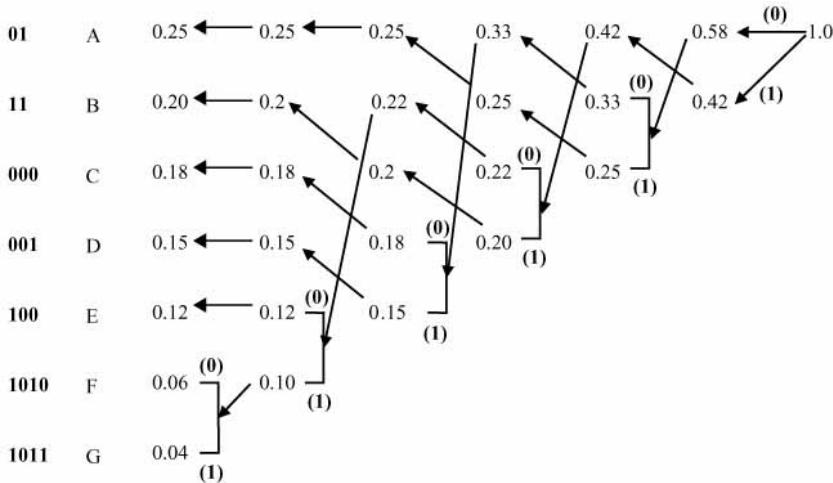
U: ( A B C D E F G )

0.25 0.20 0.18 0.15 0.12 0.06 0.04

给概率最小的两个符号 G 与 F 分别指定为“1”与“0”，然后将它们的概率相加再与原来的 A~E 组合并重新排序成新的原为：

$U' : (A \ B \ C \ D \ E \ FG)$   
 0.25 0.20 0.18 0.15 0.12 0.10

Code Symbol Prob



对 FG 与 E 分别指定“1”与“0”后，再作概率相加并重新按概率排序得  
 $U'' : (0.25 \ 0.22 \ 0.2 \ 0.18 \ 0.15) \dots$   
 直到最后得 : (0.58 0.42)  
 分别给以“0”，“1”为止，如图 4-4 所示。}

例如 G 从至右左，其码字为 1011；

F 按践线将所遇到的“0”和“1”按最低位到最高位的顺序排好，其码字为 1010…

用霍夫曼编码所得的平均比特率为： $\Sigma$  码长 × 出现概率

上例为： $0.25 \times 2 + 0.20 \times 2 + 0.18 \times 3 + 0.12 \times 3 + 0.15 \times 3 + 0.06 \times 4 + 0.04 \times 4 = 2.65$  bit  
 可以算出本例的信源熵为 2.62bit，二者已经是很接近了。

### 2.3.2.3 算术编码 (Arithmetic coding)

算术编码在图像数据压缩标准(如 JPEG, JBIG)中扮演了重要的角色。在算术编码中，消息用 0 到 1 之间的实数进行编码，算术编码用到两个基本的参数：符号的概率和它的编码间隔。信源符号的概率决定压缩编码的效率，也决定编码过程中信源符号的间隔，而这些间隔包含在 0 到 1 之间。编码过程中的间隔决定了符号压缩后的输出。

和其它熵编码方法不同的地方在于其他的熵编码方法通常是把输入的消息分割为符号，然后对每个符号进行编码。算术编码是把一个信源表示为实轴上 0 和 1 之间的一个区间，信源集合中的每一个元素都用来缩短这个区间。

算术编码器的编码过程可用下面的例子加以解释。

例 假设信源符号为 {00, 01, 10, 11}，这些符号的概率分别为 {0.1, 0.4, 0.2, 0.3}，根据这些概率可把间隔 [0, 1] 分成 4 个子间隔：[0, 0.1), [0.1, 0.5), [0.5, 0.7), [0.7, 1)，其中  $[x, y)$

表示半开放间隔，即包含  $x$  不包含  $y$ 。上面的信息可综合在表 4-04 中。

表 4-04 信源符号、概率和初始编码间隔

符号	00	01	10	11
概率	0.1	0.4	0.2	0.3
初始编码 间隔	[0, 0.1)	[0.1, 0.5)	[0.5, 0.7)	[0.7, 1)

如果二进制消息序列的输入为：10 00 11 00 10 11 01。编码时首先输入的符号是 10，找到它的编码范围是 [0.5, 0.7]。由于消息中第二个符号 00 的编码范围是 [0, 0.1]，因此它的间隔就取 [0.5, 0.7] 的第一个十分之一作为新间隔 [0.5, 0.52]。依此类推，编码第 3 个符号 11 时取新间隔为 [0.514, 0.52]，编码第 4 个符号 00 时，取新间隔为 [0.514, 0.5146]，…。消息的编码输出可以是最后一个间隔中的任意数。整个编码过程如图 4-03 所示。

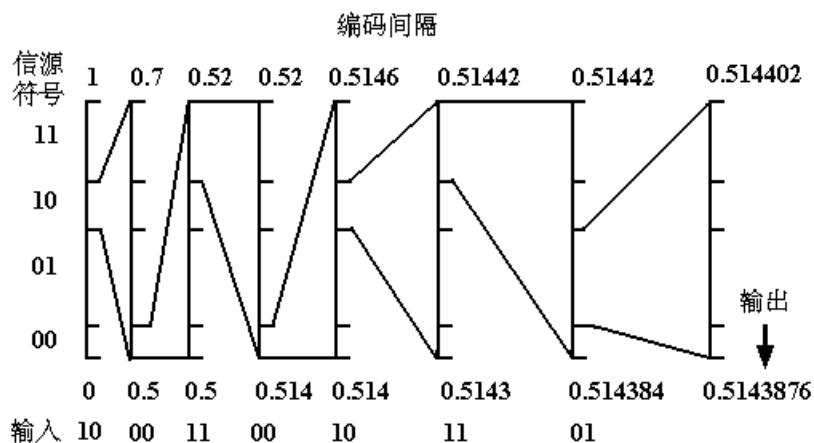


图 4-03 算术编码过程举例

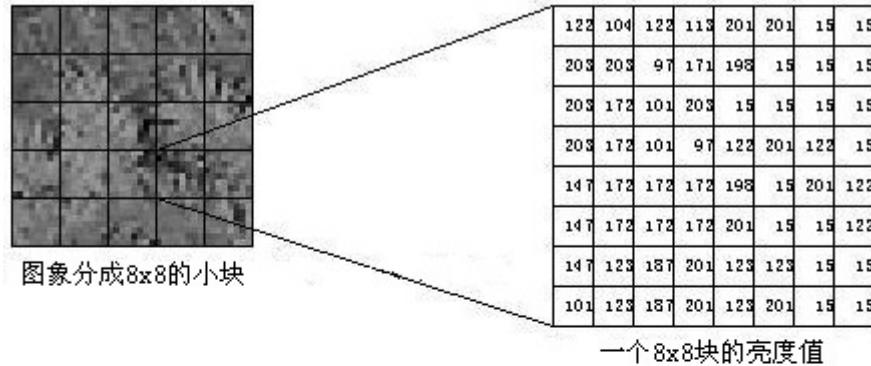
#### 2.3.2.4 帧内图像压缩编码处理举例

JPEG 压缩是典型的帧内图像编码处理过程，具体方法是首先把它转换成 YCrCb 空间表示的图像。每个图像平面分成  $8 \times 8$  的图块，对每个图块进行离散余弦变换 DCT (discrete Cosine Transform)。DCT 变换后经过量化的支流交流分量系数按照 Zig-zag 的形状排序，然后再使用无损压缩技术进行编码。

下面是一个 JPEG 处理流程。原始的图像见图 08-03-11。图 08-03-12 中列出了图像分块、计算 DCT 系数以及系数量化结果。图 08-03-13 为按 Z 形路径将 DCT 系数先经行程编码，最后得到 Huffman 编码。



图 08-03-11 原始图



4.5	4.3	4.1	3.2	1.7	1.1	0.2	0.2
4.2	4.3	3.8	1.6	1.8	0.3	0.1	0.1
3.5	3.3	2.3	1.9	0.5	0.4	0.2	0.3
3.2	2.5	1.1	0.2	0.2	0.5	0	0.2
2.9	1.2	0	0.1	0.4	0.1	0	0
1.1	0	0.4	0.2	0	0	0	0
0.1	0.1	0.2	0	0	0	0	0
0	0.2	0.3	0.3	0	0	0	0

该块的DCT系数

4	4	4	3	1	1	0	0
4	4	3	1	1	0	0	0
3	3	2	1	0	0	0	0
3	2	1	0	0	0	0	0
2	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

量化的DCT系数

图 08-03-12 图像分块、计算 DCT 系数以及系数量化

4	4	4	3	1	0	0
4	4	3	1	1	0	0
3	3	2	1	0	0	0
3	2	1	0	0	0	0
2	1	0	0	0	0	0
1	0	0	0	0	0	0
0	0	0	0	0	0	0

已量化的DCT系数

4	4	4	3	4	4	3	3	3	2	2	2	1	1	1	1	1	1	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Z形排列的系数

4 ! 3	3 ! 1	4 ! 2	3 ! 4	2 ! 3	1 ! 8	0 ! 4
-------	-------	-------	-------	-------	-------	-------

行程编码

00101	00110	0100	0101	00111	110	00010
-------	-------	------	------	-------	-----	-------

Huffman编码

图 08-03-13 系数编码

### 2.3.3 时域压缩技术

#### 2.3.3.1 运动估计的运动补偿编码

这是一种帧间编码的方法，其原理是利用帧间的空间相关性，减小空间冗余度。这是因为两帧之间有很大的相似性，如果将前后两帧相减（移动物体作相应位移）得到的误差作编码所需比特要比帧内编码所需的比特少，帧间差集中在零附近，可以用短的码字传送。

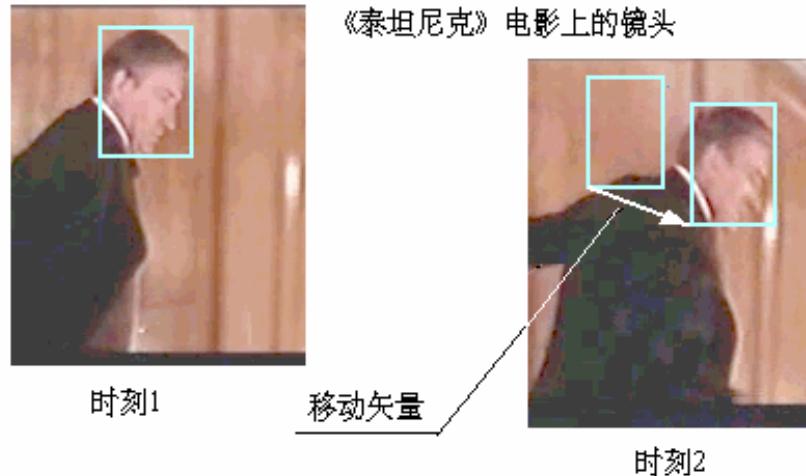


图 10-03 移动矢量的概念

实现帧间编码的方法是运动估计和运动补偿。

当前帧在过去帧的窗口中寻找匹配部分，从中找到运动矢量；

根据运动矢量，将过去帧位移，求得对当前帧的估计；

将这个估计和当前帧相减，求得估计的误差值；

将运动矢量和估计的误差值送到接收端去。

接收端根据收到的运动矢量将过去帧作位移（也就是对当前帧的估计），再加上接收到的误差值，就是当前帧了。

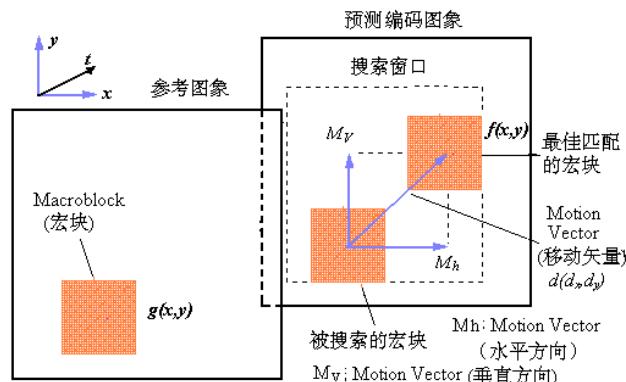


图 4-7 运动估计的全局搜索块匹配 实际上，在做运动估计和运动补偿时，是以  $16 \times 16$  的块（称宏块）逐个进行的，如图 4-6 所示，这是将当前帧划分为  $N \times N$  ( $16 \times 16$ ) 的块。对每一块在过去帧中的一定范围内进行搜索，以求得最优匹配，从而得到运动矢量的估值 ( $M_v, M_h$ )。衡量匹配好坏的准则可以是均方误差最小准则。搜索方法可以是全局搜索法，即对搜索范围内的每一点都计算均方误差，

选最小值即对应最优匹配，如图 4-7 所示。

#### 2.3.4 视频压缩编码中去冗余技术的应用

图像压缩技术基本方法和方法可以归纳成两个要点：① 在空间方向上，图像数据压缩采用 JPEG (Joint Photographic Experts Group) 压缩算法来去掉冗余信息。② 在时间方向上，图像数据压缩采用移动补偿(motion compensation)算法来去掉冗余信息。

根据应用的压缩算法的不同，MPEG 专家组定义了三种图像：帧内图像 I (Intra-coded picture)，预测图像 P (Predictive-coded picture) 和双向预测图像 B (bidirectionally predictive-coded picture)，典型的排列如图 10-01 所示。

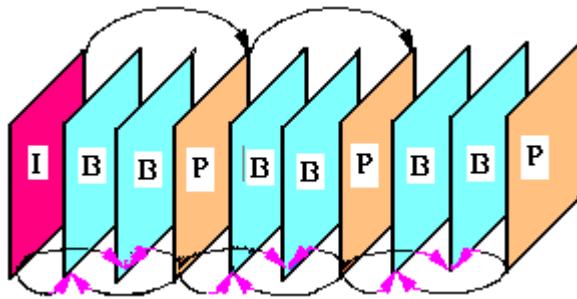


图 10-01 MPEG 专家组定义的三种图像

##### 2.3.4.1 帧内图像 I 的压缩编码算法

帧内图像 I 不参照任何过去的或者将来的其他图像帧，压缩编码采用类似 JPEG 压缩算法，它的框图如图 10-02 所示。如果电视图像是用 RGB 空间表示的，则首先把它转换成 YCrCb 空间表示的图像。每个图像平面分成  $8 \times 8$  的图块，对每个图块进行离散余弦变换 DCT (discrete Cosine Transform)。DCT 变换后经过量化的交流分量系数按照 Zig-zag 的形状排序，然后再使用无损压缩技术进行编码。DCT 变换后经过量化的直流分量系数用差分脉冲编码 DPCM (Differential Pulse Code Modulation)，交流分量系数用行程长度编码 RLE (run-length encoding)，然后再用霍夫曼 (Huffman) 编码或者用算术编码。它的编码框图如图 10-2 所示。

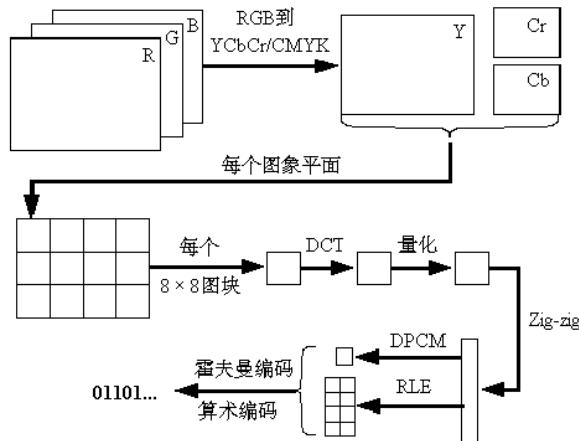


图 10-02 帧内图像 I 的压缩编码算法框图

#### 2.3.4.2 预测图像 P 的压缩编码算法

预测图像的编码也是以图像宏块(macroblock)为基本编码单元，一个宏块定义为  $I \times J$  像素的图像块，一般取  $16 \times 16$ 。预测图像 P 使用两种类型的参数来表示：一种参数是当前要编码的图像宏块与参考图像的宏块之间的差值，另一种参数是宏块的移动矢量。移动矢量的概念可用图 10-03 表示。参考宏块来自于当前编码图像之前的图像。

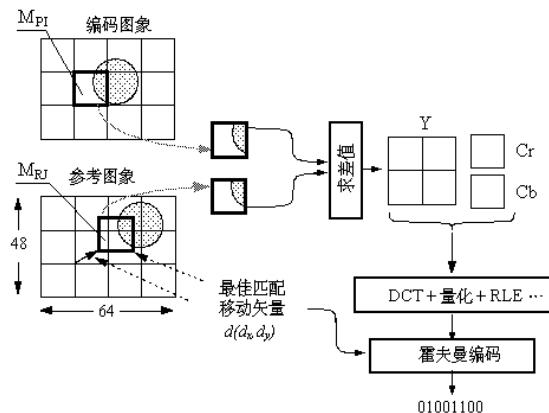


图 10-04 预测图像 P 的压缩编码算法框图

#### 2.3.4.3 双向预测图像 B 的压缩编码算法

双向预测图像 B 的压缩编码框图如图 10-09 所示。具体计算方法与预测图像 P 的算法类似，与 P 图像不同的是参考宏块可以从当前编码图像之前或者之后的图像中选择。

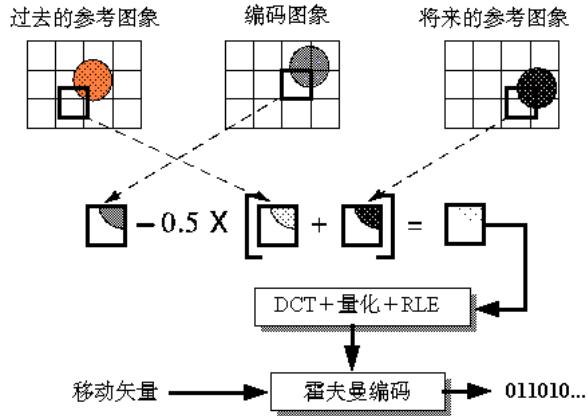


图 10-09 双向预测图像 B 的压缩编码算法框图[5]

### 3. 视频标准简介

#### 3.1 H. 261

H. 261 是由 ITU-T 开发的，第一个使用的数字视频压缩标准。实质上说，之后的所有标准视频编解码器都是基于它涉及的。它使用了常见的 YCbCr 颜色空间，4:2:0 的色度抽样格式，8 位的抽样精度，16x16 的宏块，分块的运动补偿，按 8x8 分块进行的离散余弦变换，量化，对量化系数的 Zig-zag 扫描，run-level 符号影射以及霍夫曼编码。H. 261 只支持逐行扫描的视频输入。只有 I 帧和 P 帧，没有 B 帧，运动估计精度只精确到像素级。支持两种图像扫描格式：QCIF 和 CIF。

H. 261 标准是为 ISDN 设计，主要针对视频会议和视频电话产品中的实时编码和解码设计，压缩和解压缩的信号延时不超过 150ms，码率 px64kbps ( $p=1 \sim 30$ )。

#### 3.2 H. 263

H. 263 是 ITU-T 为低于 64kb/s 的窄带通信信道制定的视频编码标准。它是在 H. 261 基础上发展起来的，其标准输入图像格式可以是 S-QCIF、QCIF、CIF、4CIF 或者 16CIF 的彩色 4:2:0 亚取样图像。H. 263 与 H. 261 相比采用了半象素的运动补偿，并增加了 4 种有效的压缩编码模式。

1. 无限制的运动矢量模式允许运动矢量指向图像以外的区域。当某一运动矢量所指的参考宏块位于编码图像之外时，就用其边缘的图像象素值来代替。当存在跨边界的运动时，这种模式能取得很大的编码增益，特别是对小图像而言。另外，这种模式包括了运动矢量范围的扩展，允许使用更大的运动矢量，这对摄像机运动特别有利。

2. 基于句法的算术编码模式使用算术编码代替霍夫曼编码，可在信噪比和重建图像质量相同的情况下降低码率。

3. 先进的预测模式允许一个宏块中 4 个  $8 \times 8$  亮度块各对应一个运动矢量，从而提高了预测精度；两个色度块的运动矢量则取这 4 个亮度块运动矢量的平均值。补偿时，使用重叠的块运动补偿， $8 \times 8$  亮度块的每个象素的补偿值由 3 个预测值加权平均得到。使用该模式可以产生显著的编码增益，特别是采用重叠的块运动补偿，会减少块效应，提高主观质量。

4. PB-帧模式规定一个 PB-帧包含作为一个单元进行编码的两帧图像。PB-帧模式可在码率增加不多的情况下，使帧率加倍。

#### H. 263 视频压缩标准版本 2

ITU-T 在 H. 263 发布后又修订发布了 H. 263 标准的版本 2，非正式地命名为 H. 263+ 标准。它在

保证原 H.263 标准核心句法和语义不变的基础上，增加了若干选项以提高压缩效率或改善某方面的功能。原 H.263 标准限制了其应用的图像输入格式，仅允许 5 种视频源格式。H.263+标准允许更大范围的图像输入格式，自定义图像的尺寸，从而拓宽了标准使用的范围，使之可以处理基于视窗的计算机图像、更高帧频的图像序列及宽屏图像。

为提高压缩效率，H.263+采用先进的帧内编码模式；增强的 PB-帧模式改进了 H.263 的不足，增强了帧间预测的效果；去块效应滤波器不仅提高了压缩效率，而且提供重建图像的主观质量。

为适应网络传输，H.263+增加了时间分级、信噪比和空间分级，对在噪声信道和存在大量包丢失的网络中传送视频信号很有意义；另外，片结构模式、参考帧选择模式增强了视频传输的抗误码能力。

#### H.263++视频压缩标准

H.263++在 H.263+基础上增加了 3 个选项，主要是为了增强码流在恶劣信道上的抗误码性能，同时为了提高增强编码效率。这 3 个选项为：

选项 U——称为增强型参考帧选择，它能够提供增强的编码效率和信道错误再生能力(特别是在包丢失的情况下)，需要设计多缓冲区用于存储多参考帧图像。

选项 V——称为数据分片，它能够提供增强型的抗误码能力(特别是在传输过程中本地数据被破坏的情况下)，通过分离视频码流中 DCT 的系数头和运动矢量数据，采用可逆编码方式保护运动矢量。

选项 W——在 H.263+的码流中增加补充信息，保证增强型的反向兼容性，附加信息包括：指示采用的定点 IDCT、图像信息和信息类型、任意的二进制数据、文本、重复的图像头、交替的场指示、稀疏的参考帧识别。

### 3.3 JPEG

国际标准化组织于 1986 年成立了 JPEG (Joint Photographic Expert Group) 联合图片专家小组，主要致力于制定连续色调、多级灰度、静态图像的数字图像压缩编码标准。常用的基于离散余弦变换(DCT)的编码方法，是 JPEG 算法的核心内容。

JPEG 标准定义了三个层次：

1. 基本系统 (Baseline Sequential processes)
2. 扩展系统 (Extended DCT Based Process)
3. 特殊无损功能 (Lossless Process)

每个编码解码器必须实现一个称为基本顺序编码器的必备基本系统。实现有损图像压缩，重建图像质量达到人眼难以观察出来的要求。采用的是 8x8 像素自适应 DCT 算法、量化及 Huffman 型的熵编码器。

扩展系统包括了各种的编码方式，如长度可变编码，累进编码，以及分层模式的编码。所有这些编码方法都是基本顺序编码方法的扩展，这些特殊用途的扩展可适用于各种应用。

特殊无损功能（也称作预测无损编码法）采用预测编码及 Huffman 编码（或算术编码），确保在图像被压缩的分辨率下，解压缩不造成初始源数字图像中任何细节的损失。

JPEG 标准制定了四种工作模式：

- (1) 基本顺序的基于 DCT (Baseline Sequential DCT-based) 模式

由 DCT (离散余弦变换) 系数的形成、量化和熵编码三步组成。从左到右，从上到下扫描信号，为每个图像编码。

- (2) 累进的基于 DCT (Progressive DCT-based) 模式

生成 DCT 系数和量化中的关键步骤与基本顺序编码解码器相同。主要的区别在于每个图像部件由多次扫描进行编码而不是仅一次扫描。每次继续的扫描都对图像作了改善，直到达到由量化表建立的图像质量为止。

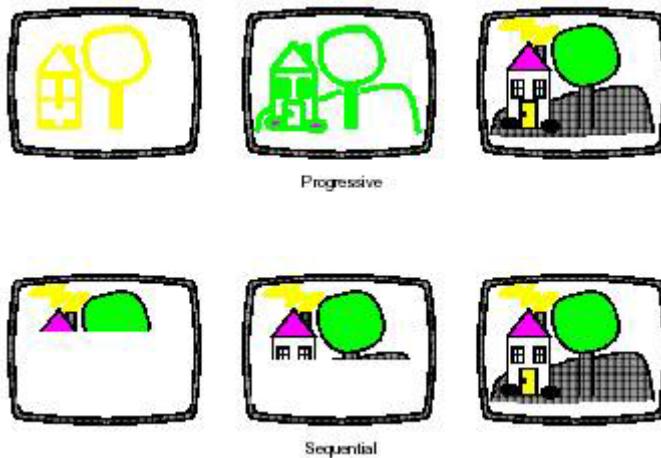


图 08-03-7 演进(上)和顺序(下)显示比较



图 08-03-8 演进编码显示

(a) 第 1 遍, 轮廓极不分明

(b) 第 2 遍, 轮廓不分明

(c) 第 3 遍, 轮廓分明

### (3) 无损 (Lossless) 模式

独立于 DCT 处理, 用来定义一种达到无损连续色调压缩的手段。预测器将采样区域组合起来并基于采样区域预测出邻系统区域。预测出的区域对照着每一区域的完全无损采样进行预测, 同时通过 Huffman 编码法或算术熵编码法对这一差别进行无损编码, 对较好质量的复制通常可达到 2: 1 的压缩率。

### (4) 分层 (Hierarchical) 模式

分层模式提供了一种可实现多种分辨率的手段。每个接续层次上的图像编码在水平或垂直方向上的分辨率都被降低二倍。它所传送的数据包括所支持的最低分辨率图像, 以及用于解码恢复到原有的全分辨率图像所需的、分辨率以 2 的倍数递降的相邻图像的差分信息。

JPEG 算法的平均压缩比为 15: 1 。当压缩比大于 50 倍时将可能出现方块效应。

JPEG 编码的基本处理过程包括: 图像准备, 图像处理, 量化, 和熵编码 (图 08-03-1)。

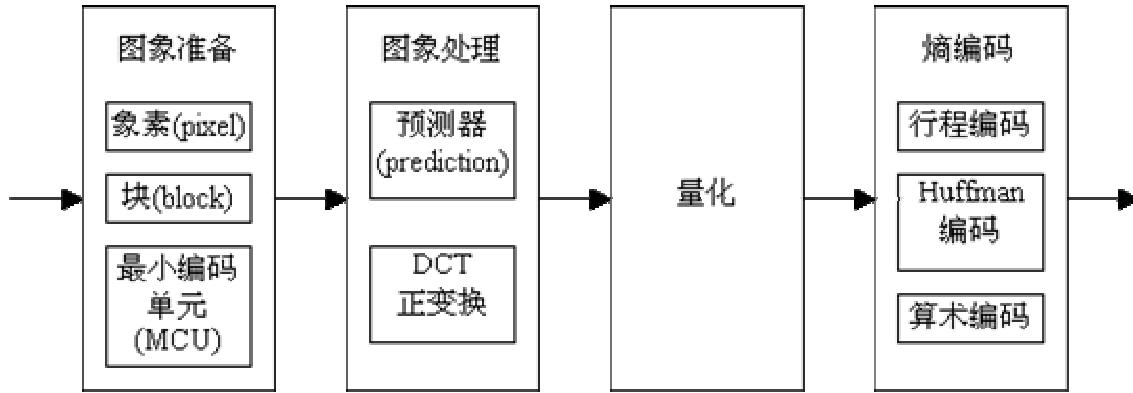


图 08-03-1 JPEG 编码的基本处理过程

### 3.4 MJPEG

MJPEG (Motion JPEG) 压缩技术，主要是基于静态视频压缩发展起来的技术，它的主要特点是基本不考虑视频流中不同帧之间的变化，只单独对某一帧进行压缩。

MJPEG 压缩技术可以获取清晰度很高的视频图像，可以动态调整帧率、分辨率。但由于没有考虑到帧间变化，造成大量冗余信息被重复存储，因此单帧视频的占用空间较大。

### 3.5 MPEG-1 Video (ISO11172-2)

MPEG-1 (ISO/IEC 11172) 是 MPEG 组织着眼于解决多媒体的存储问题于 1992 年提出的一个具有广泛影响的多媒体国际标准。MPEG-1 标准的正式名称为“基于数字存储媒体运动图像和声音的压缩标准”，用于数字存储体上活动图像及其伴音的编码，其典型码率为 1.5Mb/s (在 1.5Mb/s 时基于 MPEG-1 编码的视频质量和 VHS 提供的视频质量相当)。

MPEG-1 视频压缩策略：为了提高压缩比，帧内/帧间图像数据压缩技术必须同时使用。帧内压缩算法与 JPEG 压缩算法大致相同，采用基于 DCT 的变换编码技术，用以减少空域冗余信息。帧间压缩算法，采用预测法和插补法。预测误差可在通过 DCT 变换编码处理，进一步压缩。帧间编码技术可减少时间轴方向的冗余信息。

MPEG-1 视频压缩技术的特点：1. 随机存取；2. 快速正向/逆向搜索；3. 逆向重播；4. 视听同步；5. 容错性；6. 编/解码延迟。

### 3.6 MPEG-2 Video (ISO13818-2)

继成功制定 MPEG-1 之后，MPEG 组织于 1996 年推出解决多媒体传输问题的 MPEG-2 标准。MPEG-2 的正式名称为“通用的图像和声音压缩标准”。MPEG-2 标准最为引人注目的产品是数字电视机顶盒与 DVD。

MPEG-2 主要针对高清晰度电视 (HDTV) 的需要，典型的传输速率为 10~15Mbps，有每秒 30 帧 704\*480 的分辨率，是 MPEG-1 分辨率的四倍。它适用于高要求的广播和娱乐应用程序，如：DSS 卫星广播和 DVD。

MPEG-2 基本算法和 MPEG-1 相同，但是在 MPEG-1 的基础上作了许多重要的扩展和改进。主要的扩展有以下两点：

1. 针对电视图像都是隔行扫描的特性，加入了对隔行场信号的编码支持，提高了对电视信号编码压缩的效率。
2. 针对不同的应用定义了不同的轮廓 (Profile) 和等级 (Level)，不同的轮廓和等级提供

不同的编解码功能和运算复杂度。

现有 MPEG-2 视频标准的技术规范集包括 5 类 (Profile) 4 级 (Level) 组成，并采用分级编码。类和集的若干组合构成 MPEG-2 标准在某种特定应用下的子集。

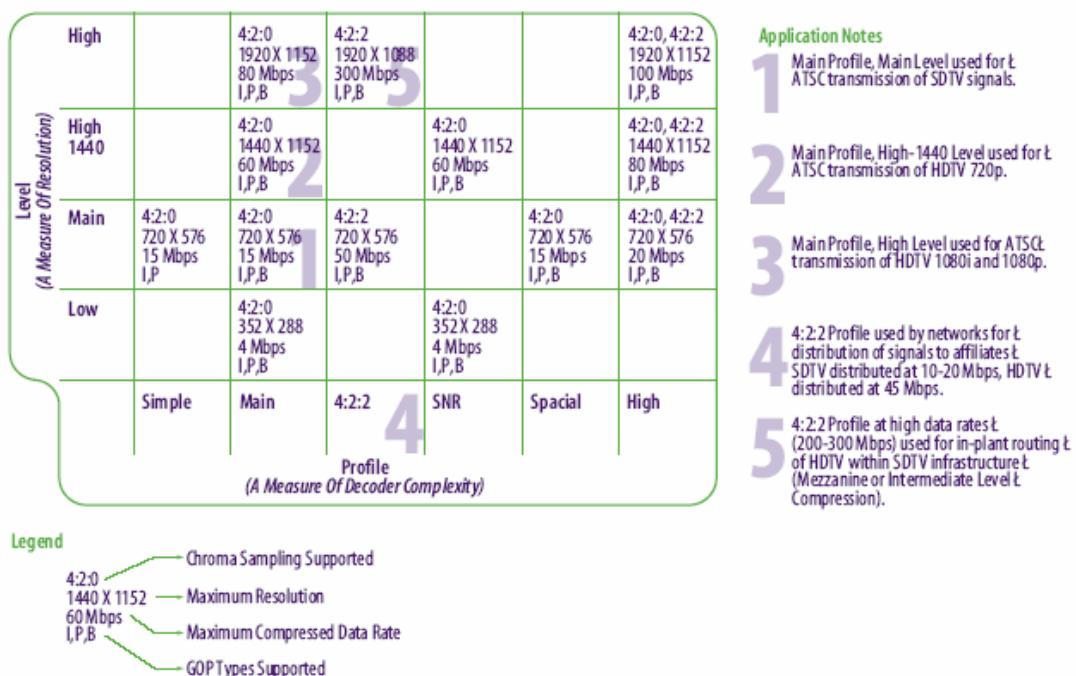
级 (Level) 是指 MPEG-2 的输入格式，标识从有限清晰度的 VHS 质量图像到 HDTV 图像，每一种输入格式编码后都有一个相应的范围。共分 4 级：

- 1) 低级 LL (Low Level)，图像输入格式的像素是 ITU-R Rec. BT 601 格式的 1/4，即  $352 \times 240 \times 30$  或  $352 \times 288 \times 25$ 。
- 2) 主级 ML (Main Level)，图像输入格式符合 ITU-R Rec. BT 601 格式，即  $720 \times 480 \times 30$  或  $720 \times 576 \times 25$ 。
- 3) 1440 高级 H14L (High 1440 Level)，图像宽高比为 4 : 3，采用  $1440 \times 1080 \times 30$  的高清晰度格式。
- 4) 高级 HL (High Level)，图像宽高比为 16 : 9，采用  $1920 \times 1080 \times 30$  的高清晰度格式。

类 (Profile) 是指 MPEG-2 的不同处理方法，每一类都包括压缩和处理方法的一个集合，较高的类意味着采用较多的编码工具集，进行更精细的处理，达到更好的图像质量，同时实现的代价也更大。共分 5 类：

- 1) 简单类 SP (Simple Profile)，使用最少的编码工具集。
- 2) 主类 MP (Main Profile)，它比简单类增加了一种双向预测方法，在相同比特率的情况下，将给出比简单类更好的图像。
- 3) 信噪比可分级类 SNRP (SNR Scaleable Profile) 和空域可分级类 SPP (Spatially Scaleable Profile)，这两种类允许将编码的视频数据分为基本层和附加层，提供了一种多种广播的方式。
- 4) 高级类 HP (High Profile)，应用于图像质量、比特率要求更高的场合。

## MPEG-2 Compression Levels and Profile Chart



实际应用中，可以根据编解码器支持的轮廓和等级不同来判断编解码器的性能。比如：DVD 解码芯片通常支持 MP@ML (Main Profile and Main Level) 解码；支持 HDTV 的 STB 芯片会支持 MP@HL (Main Profile and High Profile) 的 MPEG2 解码，其性能是 DVD 解码芯片的 5 倍。

表 8-01 MPEG-1 和-2 典型的编码参数

	MPEG-1	MPEG-2 (基本型)
标准化时间	1992 年	1994 年 (DIS)
主要应用	CD-ROM 上的数字电视，VCD	数字 TV，DVD
空间分辨率	CIF 格式 (1/4 TV)， $288 \times 360$ 像素	TV， $576 \times 720$ 像素
时间分辨率	25 – 30 帧/秒	50–60 场/秒
位速率	1.5 Mbit/s	15 Mbit/s
质量	相当于 VHS	相当于 NTSC/PAL 电视
压缩率	$20 \sim 30$	$30 \sim 40$

### 3.7 MPEG-4 Video (ISO14496-2)

MPEG-1、MPEG-2、H.261、H.263 都是采用第一代压缩编码技术，着眼于图像信号的统计特性来设计编码器。MPEG-4 是以视听媒体对象为基本单元，采用基于内容的压缩编码，以实现数字视音频、图形合成应用及交互式多媒体的集成。

MPEG-4 实现基于内容交互的首要任务就是把视频/图像分割成不同对象或者把运动对象从背景中分离出来，然后针对不同对象采用相应编码方法，以实现高效压缩。因此视频对象提取与编码是 MPEG-4 视频编码的关键。

和 MPEG-2 一样，MPEG-4 也定义了一系列的框架和等级。目前业内所说的 MPEG-4 一般是指 SP(Simple Profile 简级)或 ASP(Advanced Simple Profile 进阶的简级)，主要针对低码率应用，如因特网上的流媒体、无线网的视频传输及视频存储等，其核心类似于 H.263，并不涉及视频对象提取与编码部分。

Tool	简单级	先进简单级
I 帧	x	x
P 帧	x	x
B 帧		x
Short Header (兼容 H.263)	x	x
AC/DC 预测	x	x
4 运动矢量/无限制运动矢量	x	x
Video Packet Resynchronisation	x	x
Data Partitioning	x	x
Reversible VLC	x	x
H.263 / MPEG-2 量化表		x

全局运动矢量		x
1/4 象素精度的运动估计		x
Interlace 交错场编码		x

Profile, Level	简单级, Level 0	简单级, Level 0b	简单级, Level 1	简单级, Level 2	简单级, Level 3
最大码率 (kbit/s)	64	128	64	128	384
每帧最大宏块数	99	99	99	396	396
最大分辨率 @ 30 帧/秒	-	-	128x96	256x192	CIF
最大分辨率 @ 15 帧/秒	QCIF	QCIF	QCIF	CIF	CIF

Profile, Level	先进 简单级, L0	先进 简单级, L1	先进 简单级, L2	先进 简单级, L3	先进 简单级, L3b	先进 简单级, L4	先进 简单级, L5
最大码率 (kbit/s)	128	128	384	768	1500	3000	8000
每帧最大宏块数	99	99	396	396	396	792	1620
最大分辨率 @ 30 帧/秒	QCIF	QCIF	256x192	CIF	CIF	352x576 704x288	720x576
最大分辨率 @ 15 帧/秒	QCIF	QCIF	CIF	CIF	CIF	352x576 704x288	720x576

### 3.8 H.264/MPEG-4 AVC

JVT 是由 ISO/IEC MPEG 和 ITU-T VCEG 成立的联合视频工作组 (Joint Video Team)，致力于新一代数字视频压缩标准的制定。JVT 标准在 ISO/IEC 中的正式名称为：MPEG-4 AVC(part10) 标准；在 ITU-T 中的名称：H.264（早期被称为 H.26L）。

H264 集中了以往标准的优点，并吸收了以往标准制定中积累的经验，采用简洁设计，使它比 MPEG4 更容易推广。H.264 创造性了多参考帧、多块类型、整数变换、帧内预测等新的压缩技术，使用了更精细的分象素运动矢量 (1/4、1/8) 和新一代的环路滤波器，使得在同等速率下，H.264 能够比 H.263 减小 50% 的码率。

相对于前期的视频压缩标准，H.264 引入了很多先进的技术，包括  $4 \times 4$  整数变换、空域内的帧内预测、1/4 象素精度的运动估计、多参考帧与多种大小块的帧间预测技术等。新技术带来了较高的压缩比，同时大大提高了算法的复杂度。

#### 1. $4 \times 4$ 整数变换

以前的标准，如 H.263 或 MPEG-4，都是采用  $8 \times 8$  的 DCT 变换。H.26L 中建议的整数变换实际上接近于  $4 \times 4$  的 DCT 变换，整数的引入降低了算法的复杂度，也避免了反变换的失配问题， $4 \times 4$  的块可以减小块效应。而 H.264 的  $4 \times 4$  整数变换进一步降低了算法的复杂度，相比 H.26L 中建议的整数

变换，对于 9b 输入残差数据，由以前的 32b 降为现在的 16b 运算，而且整个变换无乘法，只需加法和一些移位运算。新的变换对编码的性能几乎没有影响，而且实际编码略好一些。

## 2. 基于空域的帧内预测技术

视频编码是通过去除图像的空间与时间相关性来达到压缩的目的。空间相关性通过有效的变换来去除，如 DCT 变换、H. 264 的整数变换；时间相关性则通过帧间预测来去除。这里所说的变换去除空间相关性，仅仅局限在所变换的块内，如  $8 \times 8$  或者  $4 \times 4$ ，并没有块与块之间的处理。H. 263+与 MPEG-4 引入了帧内预测技术，在变换域中根据相临块对当前块的某些系数做预测。H. 264 则是在空域中，利用当前块的相临像素直接对每个系数做预测，更有效地去除相临块之间的相关性，极大地提高了帧内编码的效率。H. 264 基本部分的帧内预测包括 9 种  $4 \times 4$  亮度块的预测、4 种  $16 \times 16$  亮度块的预测和 4 种色度块的预测。

## 3. 运动估计

H. 264 的运动估计具有 3 个新的特点：1/4 象素精度的运动估计；7 种大小不同的块进行匹配；前向与后向多参考帧。H. 264 在帧间编码中，一个宏块( $16 \times 16$ )可以被分为  $16 \times 8$ 、 $8 \times 16$ 、 $8 \times 8$  的块，而  $8 \times 8$  的块被称为子宏块，又可以分为  $8 \times 4$ 、 $4 \times 8$ 、 $4 \times 4$  的块。总体而言，共有 7 种大小不同的块做运动估计，以找出最匹配的类型。与以往标准的 P 帧、B 帧不同，H. 264 采用了前向与后向多个参考帧的预测。半象素精度的运动估计比整象素运动估计有效地提高了压缩比，而 1/4 象素精度的运动估计可带来更好的压缩效果。

编码器中运用多种大小不同的块进行运动估计，可节省 15%以上的比特率(相对于  $16 \times 16$  的块)。运用 1/4 象素精度的运动估计，可以节省 20%的码率 (相对于整象素预测)。多参考帧预测方面，假设为 5 个参考帧预测，相对于一个参考帧，可降低 5%~10%的码率。以上百分比都是统计数据，不同视频因其细节特征与运动情况而有所差异。

## 4. 熵编码

H. 264 标准采用的熵编码有两种：一种是基于内容的自适应变长编码(CAVLC)与统一的变长编码(UVLC)结合；另一种是基于内容的自适应二进制算术编码(CABAC)。CAVLC 与 CABAC 根据相临块的情况进行当前块的编码，以达到更好的编码效率。CABAC 比 CAVLC 压缩效率高，但要复杂一些。

## 5. 去块效应滤波器

和 H. 263 和 MPEG-4 一样，H. 264 标准引入了去块效应滤波器，对块的边界进行滤波，滤波强度与块的编码模式、运动矢量及块的系数有关。去块效应滤波器在提高压缩效率的同时，改善了图像的主观效果。在 H. 263 中，去块效应滤波器为可选项。

H. 264 标准分成三个框架(Profile)：Baseline、Main Profile 及 Extended Profile，代表针对不同应用的算法集及技术限定。Baseline 主要包含低复杂度、低延时的技术特征，主要针对交互式的应用，考虑到恶劣环境下的容错性，内容基本都被其它更高级别的 Profile 所包含；Main Profile 是针对更高编码效率的应用，如视频广播；X Profile 的设计主要针对流媒体的应用，在这一框架中所有容错技术、对比特流的灵活访问及切换技术都将包括其中。

Tool	Profile		
	Baseline	Main	Extended
I 帧	x	x	x

P 帧	x	x	x
B 帧		x	x
SI 帧			x
SP Frame			x
自适应变长编码 CAVLC	x	x	x
自适应二进制算术编码 CABAC		x	
数据划分 Data Partitioning			x
交错场编码 Interlaced MBs		x	x
加权预测 Weighted Prediction		x	x
#Slice Groups > 1	x		x

Slice Group: 在 H.264 标准中的图像宏块能够以灵活的宏块组织顺序(FMO)划分为多个 Slice Group; Slice 之间相互独立, 可以任意的顺序传输到解码端(ASO)。在比特流中 Slice 可以使用重复的方式(RS)传输, 在 Slice 数据出错的情况下可用来进行恢复, 增强了图像传输的鲁棒性。同时 Slice 间的相互独立性抑制了错误的空间传播, 提高了比特流的容错性。

level	每帧最大宏块数	最大分辨率 @ 帧率	
1	99	176x144@15fps	128x96@30fps
1. 1	396	—	172x144@30fps
1. 2	396	352x288@15fps	—
1. 3	396	—	252x288@30fps
2	396	—	252x288@30fps
2. 1	792	—	352x480@30fps
2. 2	1620	720x480@15fps	352x480@30fps
3	1620	—	720x480@30fps
3. 1	3600	—	1280x720@30fps
3. 2	5120	—	1280x720@60fps
4	8192	—	2048x1024@30fps
4. 1	8192	—	2048x1024@30fps
4. 2	8192	—	2048x1024@60fps
5	21696	—	2560x1920@30fps
5. 1	36864	—	4096x2048@30fps

可以根据编解码器支持的轮廓和等级不同来判断编解码器的性能。比如: SONY PSP (Play Station Portable) 的 H.264 解码器支持到 H.264/MPEG-4 AVC Main Profile Level 3。由此可知它的最大可以解码 720x480 分辨率 30 帧每秒的 H.264 视频。

### 3.9 Windows Media Video 9 (WMV 9)

Windows Media Video 9 (WMV 9) 是 Microsoft 视频媒体技术的首要 Codec, 它派生于 MPEG-4, 几个专有扩展功能使其可在给定码率下提供更好的图像质量, 在这种意义下, Windows Media Video 9

是流式视频中质量最高的 Codec 之一。一些测试表明，WM9 的视频压缩效率比 MPEG-2、MPEG-4 SP 及 H.263 高很多，而与 H.264 的压缩效率相当。

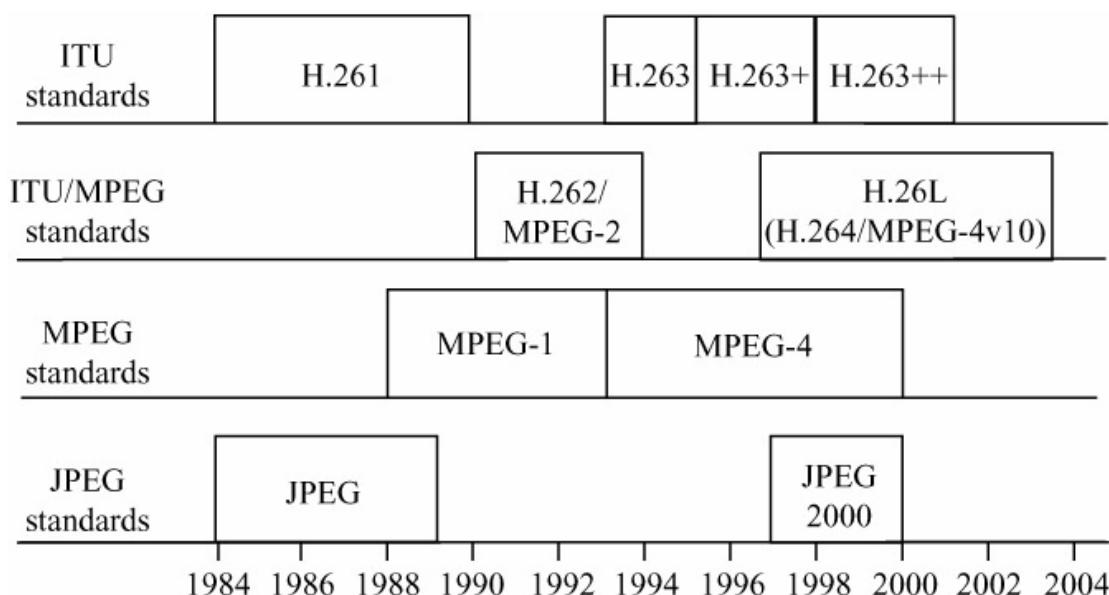
WMV 9 的主要优点包括：本地或网络回放、图像质量优秀、可扩充的媒体类型、部件下载、可伸缩的媒体类型、流的优先级化、多码率合成、多语言支持、环境独立性、丰富的流间关系以及扩展性等。但是 WMV 9 也具有一些缺陷，如系统平台只能采用 Windows 操作系统，跨平台支持能力差；转码生成 WMV 9 的效率较低；流式媒体内容的浏览没有采用 QuickTime 浏览 ISO/ISMA MPEG-4 视频内容的方便等。

### 3.10 AVS

AVS 是由我国自主制定的音/视频编码技术标准，主要面向高清晰度电视、高密度光存储媒体等应用。AVS 标准以当前国际上最先进的 MPEG-4 AVC/H.264 框架为基础，强调自主知识产权，同时充分考虑了实现的复杂度。相对于 H.264，AVS 的主要特点有：(1)  $8 \times 8$  的整数变换与 64 级量化；(2) 亮度和色度帧内预测都是以  $8 \times 8$  块为单位，亮度块采用 5 种预测模式，色度块采用 4 种预测模式；(3) 采用  $16 \times 16$ 、 $16 \times 8$ 、 $8 \times 16$  和  $8 \times 8$  4 种块模式进行运动补偿；(4) 在  $1/4$  象素运动估计方面，采用不同的四抽头滤波器进行半象素插值和  $1/4$  象素插值；(5) P 帧可以利用最多 2 帧的前向参考帧，而 B 帧采用前后各一个参考帧。

## 4 各种视频标准功能对比

### 4.1 视频标准的演进历程



(图一) ITU-T 规范和 ISO/IEC MPEG 标准的演进历程

## 4.2 常见压缩格式技术指标对比

Feature/Standard	MPEG-2	MPEG-4 part2	MPEG-4 part 10/H.264	WMV-9	AVS
Macroblock size	16x16 (frame mode) 16x8 (field mode)	16x16	16x16	16x16	16x16
Prediction block size	8x8	16x16, 16x8, 8x8	16x16, 8x16, 16x8, 8x8, 4x8, 8x4, 4x4	16x16, 8x8	16x16, 8x8
Intra prediction	No	Transform Domain	Spatial Domain	No	8x8 : 5modes
Transform	8x8 DCT	8x8 DCT/Wavelet transform	8x8, 4x4 integer DCT 4x4, 2x2 Hadamard	8x8, 8x4, 4x8, 4x4 integer DCT	Asymmetric 8x8 integer DCT
Quantization	Scalar quantization with step size of constant increment	Vector quantization	Scalar quantization with step size of increase at the rate of 12.5%	Dead zone, uniform scalar quantization	Scalar quantization
Entropy coding	VLC	VLC	VLC, CAVLC, CABAC	Multiple VLC tables	VLC
Pel accuracy	1/2 - pel	1/4-pel	1/4-pel	1/4-pel	1/4-pel
Sub-pel filter	1/2-pel: 2-tap	1/2-pel: 4-tap 1/4-pel: 4-tap	1/2-pel: 6-tap 1/4-pel: 2-tap	1/2-pel: 4-tap 1/4-pel: 4-tap	1/2-pel: 4-tap 1/4-pel: 4-tap
Reference picture	One picture	One Picture	Multiple picture	one picture	Two pictures
Bidirectional prediction mode	forward/backward	forward/backward	forward/backward forward/forward backward/backward 2 vectors	forward/backward 2 motion vectors	forward/backward symmetric 1 motion vectors
Weighted prediction	No	No	Yes	yes	yes
Deblocking filter	No	No	Yes	yes	yes
Picture types	I, P, B	I, P, B	I, P, B, SI, SP	I, P, B	I, P, B
Profiles	5 profiles	8 profiles	7 profiles	3 profiles	1 profile
Playback&Random access	Yes	Yes	Yes	Yes	Yes
Error robustness	Data partitioning, FEC for important packet transmission	Synchronization, Data partitioning, Header extension, Reversible VLCs	Data partitioning, Parameter setting, Flexible macroblock ordering, Redundant slice, SP and SI slices		
Transmission rate	2~15Mbps	64kbps~2Mbps	64kbps ~ 150Mbps	64kbps ~ 45Mbps	64kbps ~ 30Mbps max.
Encoder complexity	Medium	Medium	High	High	High
Compatibility with previous standards	yes	yes	No	No	No

## 5. 音频基础

声音是通过空气等介质传播的一种连续的波，称为声波。声音的强弱体现在声波压力的大小上，音调的高低体现在声音的频率上。声波压力在时间和幅度上都是连续的，即是模拟信号，如图 2-01-7 的波形和谱图所示。声波具有普通波所具有的特性，例如反射（reflection）、折射（refraction）和衍射（diffraction）等。声波既然是波，也用频率（波长）、幅度（响度）和相位等来量度。

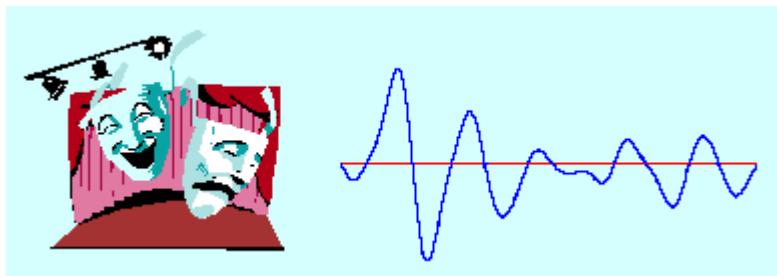


图 2-01 声音是一种连续的波

### 5.1 声音的频率和幅度

声音信号的两个基本参数是频率和幅度。信号的频率是指信号每秒钟变化的次数，用 Hz 表示。人们把频率小于 20 Hz 的信号称为亚音信号，或称为次声波信号 (subsonic)；频率范围为 20 Hz~20 kHz 的信号称为音频 (Audio) 信号；虽然人的发音器官发出的声音频率大约是 80~3400 Hz，但人说话的信号频率通常为 300~3000 Hz，人们把在这种频率范围的信号称为话音 (speech) 信号；高于 20 kHz 的信号称为超音频信号，或称超声波 (ultrasonic) 信号。

声音的幅度，即声音的大小强弱程度，也称为音量。但是相同幅度的声音对不同人感知的音量的大小不同，这取决于各个人的年龄和耳朵的特性。

## 5.2 模拟信号与数字信号

话音信号是典型的连续信号，不仅在时间上是连续的，而且在幅度上也是连续的。在时间上“连续”是指在一个指定的时间范围里声音信号的幅值有无穷多个，在幅度上“连续”是指幅度的数值有无穷多个。我们把在时间和幅度上都是连续的信号称为模拟信号。

在某些特定的时刻对这种模拟信号进行测量叫做采样(sampling)，由这些特定时刻采样得到的信号称为离散时间信号。采样得到的幅值是无穷多个实数值中的一个，因此幅度还是连续的。如果把信号幅度取值的数目加以限定，这种由有限个数值组成的信号就称为离散幅度信号。例如，假设输入电压的范围是  $0.0V \sim 0.7V$ ，并假设它的取值只限定在  $0, 0.1, 0.2, \dots, 0.7$  共 8 个值。如果采样得到的幅度值是  $0.123V$ ，它的取值就应算作  $0.1V$ ，如果采样得到的幅度值是  $0.26V$ ，它的取值就算作  $0.3$ ，这种数值就称为离散数值。我们把时间和幅度都用离散的数字表示的信号就称为数字信号。

## 5.3 采样和量化

声音进入计算机的第一步就是数字化，数字化实际上就是采样和量化。如前所述，连续时间的离散化通过采样来实现，就是每隔相等的一小段时间采样一次，这种采样称为均匀采样(uniform sampling)；连续幅度的离散化通过量化(quantization)来实现，就是把信号的强度划分成一小段一小段，如果幅度的划分是等间隔的，就称为线性量化，否则就称为非线性量化。图 2-02 表示了声音数字化的概念。

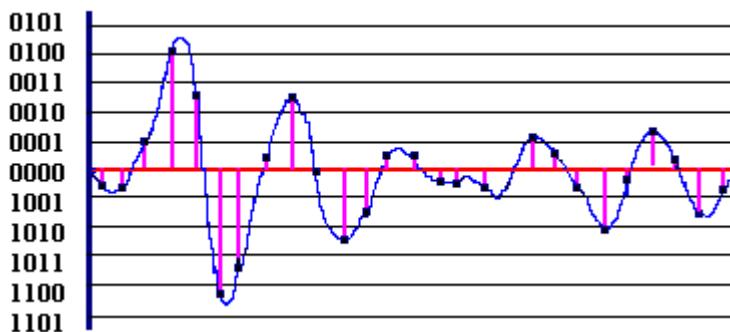


图 2-02 声音的采样和量化

### 5.3.1 采样频率

采样频率的高低是根据奈奎斯特理论(Nyquist theory)和声音信号本身的最高频率决定的。奈奎斯特理论指出，采样频率不应低于声音信号最高频率的两倍，这样就能把以数字表达的声音还原成原来的声音，这叫做无损数字化(lossless digitization)。采样定律用公式表示为

$$f_s \geq 2f \text{ 或者 } T_s \leq T/2$$

其中  $f$  为被采样信号的最高频率。

你可以这样来理解奈奎斯特理论：声音信号可以看成由许许多多正弦波组成的，一个振幅为  $A$ 、频率为  $f$  的正弦波至少需要两个采样样本表示，因此，如果一个信号中的最高频率为  $f_{\max}$ ，采样频

率最低要选择  $2^{f_{\max}}$ 。例如，电话话音的信号频率约为 3.4 kHz，采样频率就选为 8 kHz。

### 5.3.2 量化精度

样本大小是用每个声音样本的位数 bit/s(即 bps)表示的，它反映度量声音波形幅度的精度。例如，每个声音样本用 16 位(2 字节)表示，测得的声音样本值是在 0~65536 的范围里，它的精度就是输入信号的 1/65536。样本位数的大小影响到声音的质量，位数越多，声音的质量越高，而需要的存储空间也越多；位数越少，声音的质量越低，需要的存储空间越少。

量化精度的另一种表示方法是信号噪声比，简称为信噪比(signal-to-noise ratio, SNR)，并用下式计算：

$$SNR = 10 \log [(V_{signal})^2 / (V_{noise})^2] = 20 \log (V_{signal} / V_{noise})$$

其中， $V_{signal}$  表示信号电压， $V_{noise}$  表示噪声电压；SNR 的单位为分贝(db)

例 1：假设  $V_{noise}=1$ ，采样精度为 1 位表示  $V_{signal}=2^1$ ，它的信噪比  $SNR=6$  分贝。

例 2：假设  $V_{noise}=1$ ，采样精度为 16 位表示  $V_{signal}=2^{16}$ ，它的信噪比  $SNR=96$  分贝。

CD audio 的指标是达到 20—20kHz 的频响以及不低于 90DB 的信噪比。根据以上公式可以算出要达到以上指标，音源数字化时必须采样频率要在 40KHz 以上，采样精度要大于等于 16 比特。这就是为什么现在的音乐 CD 上的数据是 44.1KHz 采样 16bit 采样的原因。

### 5.4 声音质量与数据率

根据声音的频带，通常把声音的质量分成 5 个等级，由低到高分别是电话(telephone)、调幅(amplitude modulation, AM)广播、调频(frequency modulation, FM)广播、激光唱盘(CD-Audio)和数字录音带(digital audio tape, DAT)的声音。在这 5 个等级中，使用的采样频率、样本精度、通道数和数据率列于表 2 - 01。

表 2 - 01 声音质量和数据率

质量	采样频率 (kHz)	样本精度 (bit/s)	单道声/ 立体声	数 据 率 (kB/s)	频率范围
电话	8	8	单道声	8	200~3 400 Hz
AM	11.025	8	单道声	11.0	20~15 000Hz
FM	22.050	16	立体声	88.2	50~7 000Hz
CD	44.1	16	立体声	176.4	20~20 000 Hz
DAT	48	16	立体声	192.0	20~20 000 Hz

## 6 音频压缩技术简介

## 6.1 听觉系统的感知特性

与前面章节介绍的波形声音压缩编码(如 ADPCM)和参数编码(如 LPC)不同, MPEG-1 和 MPEG-2 的声音数据压缩编码不是依据波形本身的相关性和模拟人的发音器官的特性, 而是利用人的听觉系统的特性来达到压缩声音数据的目的, 这种压缩编码称为感知声音编码(perceptual audio coding)。进入 20 世纪 80 年代之后, 尤其最近几年, 人类在利用自身的听觉系统的特性来压缩声音数据方面取得了很大的进展, 先后制定了 MPEG-1 Audio, MPEG-2 Audio 和 MPEG-2 AAC 等标准。

### 6.1.1. 人耳听觉曲线

声音的响度就是声音的强弱。在物理上, 声音的响度使用客观测量单位来度量, 即  $\text{dyn}/\text{cm}^2$ (达因/平方厘米)(声压)或  $\text{W}/\text{cm}^2$ (瓦特/平方厘米)(声强)。在心理上, 主观感觉的声音强弱使用响度级“方(phon)”或者“宋(sone)”来度量。这两种感知声音强弱的计量单位是完全不同的两种概念, 但是它们之间又有一定的联系。

当声音弱到人的耳朵刚刚可以听见时, 我们称此时的声音强度为“听阈”。例如, 1 kHz 纯音的声强达到  $10^{-16}\text{W}/\text{cm}^2$ (定义成零 dB 声强级)时, 人耳刚能听到, 此时的主观响度级定为零方。实验表明, 听阈是随频率变化的。测出的“听阈—频率”曲线如图 9-01 所示。图中最靠下面的一根曲线叫做“零方等响度级”曲线, 也称“绝对听阈”曲线, 即在安静环境中, 能被人耳听到的纯音的最小值。

另一种极端的情况是声音强到使人耳感到疼痛。实验表明, 如果频率为 1 kHz 的纯音的声强级达到 120 dB 左右时, 人的耳朵就感到疼痛, 这个阈值称为“痛阈”。对不同的频率进行测量, 可以得到“痛阈—频率”曲线, 如图 9-01 中最靠上面所示的一根曲线。这条曲线也就是 120 方等响度级曲线。

在“听阈—频率”曲线和“痛阈—频率”曲线之间的区域就是人耳的听觉范围。这个范围内的等响度级曲线也是用同样的方法测量出来的。由图 9-01 可以看出, 1 kHz 的 10 dB 的声音和 200 Hz 的 30 dB 的声音, 在人耳听起来具有相同的响度。

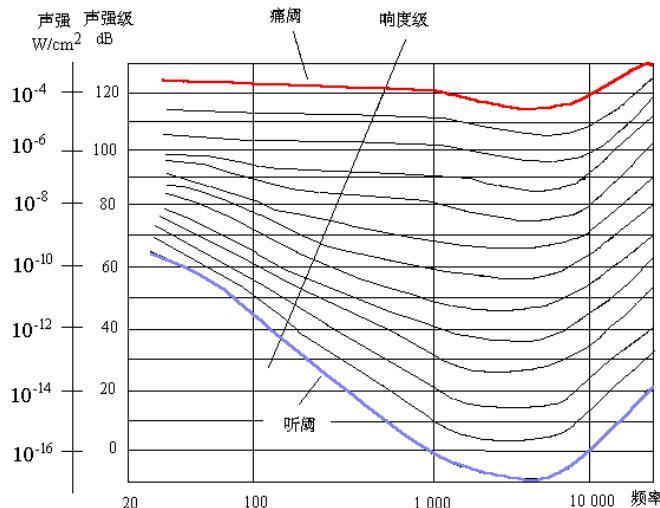


图 9-01 “听阈—频率” 曲线

图 9-01 说明人耳对不同频率的敏感程度差别很大，其中对  $2 \text{ kHz} \sim 4 \text{ kHz}$  范围的信号最为敏感，幅度很低的信号都能被人耳听到。而在低频区和高频区，能被人耳听到的信号幅度要高得多。

### 6.1.2 掩蔽效应

一种频率的声音阻碍听觉系统感受另一种频率的声音的现象称为掩蔽效应。前者称为掩蔽声音 (masking tone)，后者称为被掩蔽声音 (masked tone)。掩蔽可分成频域掩蔽和时域掩蔽。

#### 6.1.2.1 频域掩蔽

一个强纯音会掩蔽在其附近同时发声的弱纯音，这种特性称为频域掩蔽，也称同时掩蔽 (simultaneous masking)。如图 9-03 所示，一个声强为  $60 \text{ dB}$ 、频率为  $1000 \text{ Hz}$  的纯音，另外还有一个  $1100 \text{ Hz}$  的纯音，前者比后者高  $18 \text{ dB}$ ，在这种情况下我们的耳朵就只能听到那个  $1000 \text{ Hz}$  的强音。如果有一个  $1000 \text{ Hz}$  的纯音和一个声强比它低  $18 \text{ dB}$  的  $2000 \text{ Hz}$  的纯音，那么我们的耳朵将会同时听到这两个声音。要想让  $2000 \text{ Hz}$  的纯音也听不到，则需要把它降到比  $1000 \text{ Hz}$  的纯音低  $45 \text{ dB}$ 。一般来说，弱纯音离强纯音越近就越容易被掩蔽。

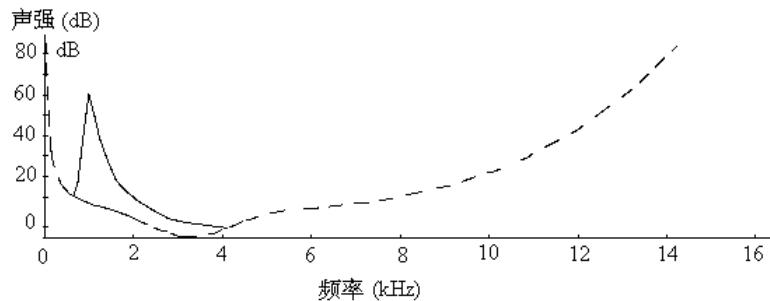


图 9-03 声强为  $60 \text{ dB}$ 、频率为  $1000 \text{ Hz}$  纯音的掩蔽效应

在图 9-04 中的一组曲线分别表示频率为  $250 \text{ Hz}$ 、 $1 \text{ kHz}$ 、 $4 \text{ kHz}$  和  $8 \text{ kHz}$  纯音的掩蔽效应，它们的声强均为  $60 \text{ dB}$ 。从图中可以看到：①在  $250 \text{ Hz}$ 、 $1 \text{ kHz}$ 、 $4 \text{ kHz}$  和  $8 \text{ kHz}$  纯音附近，对其他纯音的掩蔽效果最明显，②低频纯音可以有效地掩蔽高频纯音，但高频纯音对低频纯音的掩蔽作用则不明显。

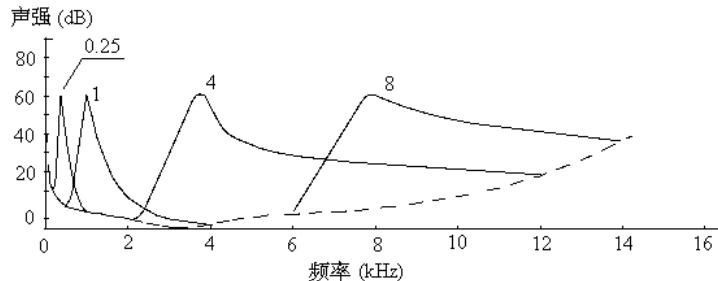


图 9-04 不同纯音的掩蔽效应曲线

由于声音频率与掩蔽曲线不是线性关系，为从感知上来统一度量声音频率，引入了“临界频带 (critical band)”的概念。通常认为，在  $20 \text{ Hz}$  到  $16 \text{ kHz}$  范围内有 24 个临界频带，如表 9-01 所示。临界频带的单位叫 Bark (巴克)，

- 1 Bark = 一个临界频带的宽度
- $f$  (频率) < 500 Hz 的情况下,  $1 \text{ Bark} \approx f / 100$
- $f$  (频率) > 500 Hz 的情况下,  $1 \text{ Bark} \approx 9 + 4 \log(f)$

/1000)

以上我们讨论了响度和掩蔽效应，尤其是人的主观感觉。其中掩蔽效应尤为重要，它是心理声学模型的基础。

表 9-01 临界频带[16]

临界 频带	频率 (Hz)			临界 频带	频率 (Hz)		
	低端	高端	宽度		低端	高端	宽度
0	0	100	100	13	2000	2320	320
1	100	200	100	14	2320	2700	380
2	200	300	100	15	2700	3150	450
3	300	400	100	16	3150	3700	550
4	400	510	110	17	3700	4400	700
5	510	630	120	18	4400	5300	900
6	630	770	140	19	5300	6400	1100
7	770	920	150	20	6400	7700	1300
8	920	1080	160	21	7700	9500	1800
9	1080	1270	190	22	9500	12000	2500
10	1270	1480	210	23	12000	15500	3500
11	1480	1720	240	24	15500	22050	6550
12	1720	2000	280				

### 6.1.2.2 时域掩蔽

除了同时发出的声音之间有掩蔽现象之外，在时间上相邻的声音之间也有掩蔽现象，并且称为时域掩蔽。时域掩蔽又分为超前掩蔽(pre-masking)和滞后掩蔽(post-masking)，如图 9-05 所示。产生时域掩蔽的主要原因是人的大脑处理信息需要花费一定的时间。一般来说，超前掩蔽很短，只有大约 5~20 ms，而滞后掩蔽可以持续 50~200 ms。这个区别也是很容易理解的。

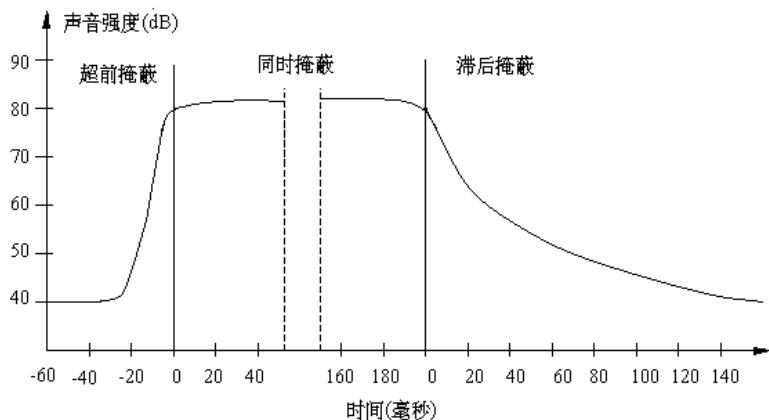


图 9-05 时域掩蔽

## 6.2 音频编解码技术

已有的音频编解码技术分成以下三种类型：基于波形编解码(waveform codecs)，基于音源编解码(source codecs)和混合编解码(hybrid codecs)。一般来说，基于波形编码压缩的声音质量高，但数据率也很高；基于音源编码的声音数据率很低，产生的合成声音的音质有待提高；混合编译码器使用音源编译码技术和波形编译码技术，数据率和音质介于它们之间。

### 6.2.1 基于波形编码技术

波形编解码的想法是，不利用生成声音信号的任何知识而企图产生一种重构信号，它的波形与原始声音波形尽可能地一致。一般来说，这种编译码器的复杂程度比较低，数据速率在 16 kb/s 以上，质量相当高。低于这个数据速率时，音质急剧下降。

#### 6.2.1.1 脉冲编码调制编码

最简单的波形编码是脉冲编码调制(pulse code modulation, PCM)，它仅仅是对输入信号进行采样和量化。典型的窄带话音带宽限制在 4 kHz，采样频率是 8 kHz。如果要获得高一点的音质，样本精度要用 12 位，它的数据率就等于 96 kb/s，这个数据率可以使用非线性量化来降低。例如，可以使用近似于对数的对数量化器(logarithmic quantizer)，使用它产生的样本精度为 8 位，它的数据率为 64 kb/s 时，重构的话音信号几乎与原始的话音信号没有什么差别。这种量化器在 20 世纪 80 年代就已经标准化，而且直到今天还在广泛使用。在北美的压扩(companding)标准是  $\mu$  律( $\mu$ -law)，在欧洲的压扩标准是 A 律(A-law)。它们的优点是编译码器简单，延迟时间短，音质高。但不足之处是数据速率比较高，对传输通道的错误比较敏感。

#### 均匀量化

如果采用相等的量化间隔对采样得到的信号作量化，那么这种量化称为均匀量化。均匀量化就是采用相同的“等分尺”来度量采样得到的幅度，也称为线性量化，如图 3-08 所示。量化后的样本值  $Y$  和原始值  $X$  的差  $E=Y-X$  称为量化误差或量化噪声。

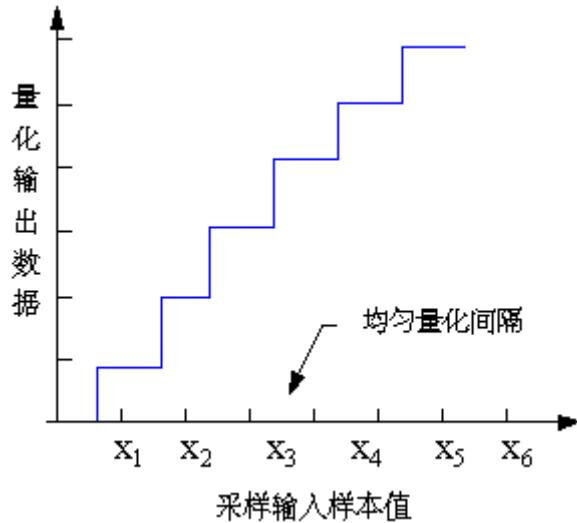


图 3-08 均匀量化

用这种方法量化输入信号时，无论对大的输入信号还是小的输入信号一律都采用相同的量化间隔。为了适应幅度大的输入信号，同时又要满足精度要求，就需要增加样本的位数。但是，对话音信号来说，大信号出现的机会并不多，增加的样本位数就没有充分利用。为了克服这个不足，就出现了非均匀量化的方法，这种方法也叫做非线性量化。

### 非均匀量化

非线性量化的基本想法是，对输入信号进行量化时，大的输入信号采用大的量化间隔，小的输入信号采用小的量化间隔，如图 3-09 所示。这样就可以在满足精度要求的情况下用较少的位数来表示。声音数据还原时，采用相同的规则。

在非线性量化中，采样输入信号幅度和量化输出数据之间定义了两种对应关系，一种称为  $m$  律压扩 (companding) 算法，另一种称为 A 律压扩算法。

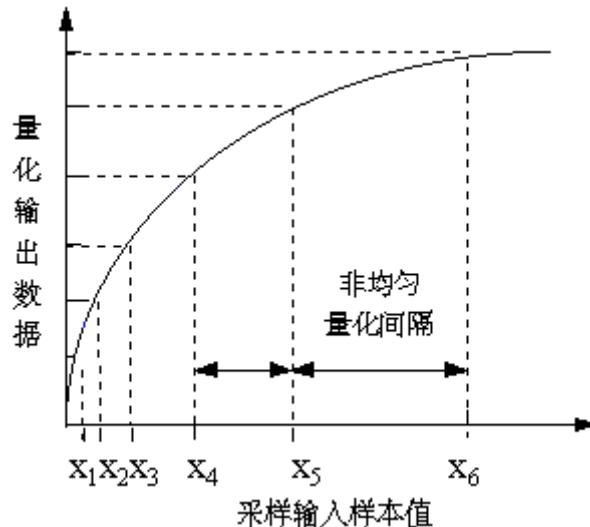


图 3-09 非均匀量化

### μ 律压扩

$\mu$  律 ( $\mu$ -Law) 压扩 (G. 711) 主要用在北美和日本等地区的数字电话通信中，按下面的式子确定量化输入和输出的关系：

$$F_{\mu}(x) = \text{sgn}(x) \frac{\ln(1 + \mu |x|)}{\ln(1 + \mu)}$$

式中： $x$  为输入信号幅度，规格化成  $-1 \leq x \leq 1$ ；

$\text{sgn}(x)$  为  $x$  的极性；

$\mu$  为确定压缩量的参数，它反映最大量化间隔和最小量化间隔之比，取  $100 \leq \mu \leq 500$ 。

由于  $\mu$  律压扩的输入和输出关系是对数关系，所以这种编码又称为对数 PCM。具体计算时，用  $\mu = 255$ ，把对数曲线变成 8 条折线以简化计算过程。

### A 律压扩

A 律 (A-Law) 压扩 (G. 711) 主要用在欧洲和中国大陆等地区的数字电话通信中，按下面的式子确定量化输入和输出的关系：

$$F_A(x) = \text{sgn}(x) \frac{A|x|}{1 + \ln A} \quad 0 \leq |x| \leq 1/A$$

$$F_A(x) = \text{sgn}(x) \frac{1 + \ln(A|x|)}{1 + \ln A} \quad 1/A < |x| \leq 1$$

式中:  $x$  为输入信号幅度, 规格化成  $-1 \leq x \leq 1$ ;

$\text{sgn}(x)$  为  $x$  的极性;

$A$  为确定压缩量的参数, 它反映最大量化间隔和最小量化间隔之比。

$A$  律压扩的前一部分是线性的, 其余部分与  $m$  律压扩相同。具体计算时,  $A=87.56$ , 为简化计算, 同样把对数曲线部分变成折线。

对于采样频率为 8 kHz, 样本精度为 13 位、14 位或者 16 位的输入信号, 使用  $m$  律压扩编码或者使用  $A$  律压扩编码, 经过 PCM 编码器之后每个样本的精度为 8 位, 输出的数据率为 64 kb/s。这个数据就是 CCITT 推荐的 G.711 标准: 话音频率脉冲编码调制(Pulse Code Modulation (PCM) of Voice Frequencies)。

u-law 将 14-bit linear PCM samples frame 压缩到 8-bit logarithmic PCM code words frame  
A-law 将 13-bit linear PCM samples 压缩到 8-bit logarithmic PCM code words

### 6.2.1.2 预测编码

在声音编码中, 一种普遍使用的压缩技术叫做预测技术, 这种技术是企图从过去的样本来预测下一个样本的值。这样做的根据是认为在声音样本之间存在相关性。如果样本的预测值与样本的实际值比较接近, 它们之间的差值幅度的变化就比原始声音样本幅度值的变化小, 因此量化这种差值信号时就可以用比较少的位数来表示差值。这就是差分脉冲编码调制(differential pulse code modulation, DPCM)的基础—对预测的样本值与原始的样本值之差进行编码。

这种编译码器对幅度急剧变化的输入信号会产生比较大的噪声, 改进的方法之一就是使用自适应的预测器和量化器, 这就产生了一种叫做自适应差分脉冲编码调制(adaptive differential PCM, ADPCM)。它的核心想法是: ①利用自适应的思想改变量化阶的大小, 即使用小的量化阶(step-size)去编码小的差值, 使用大的量化阶去编码大的差值, ②使用过去的样本值估算下一个输入样本的预测值, 使实际样本值和预测值之间的差值总是最小。它的编码简化框图如图 04-01-5 所示。

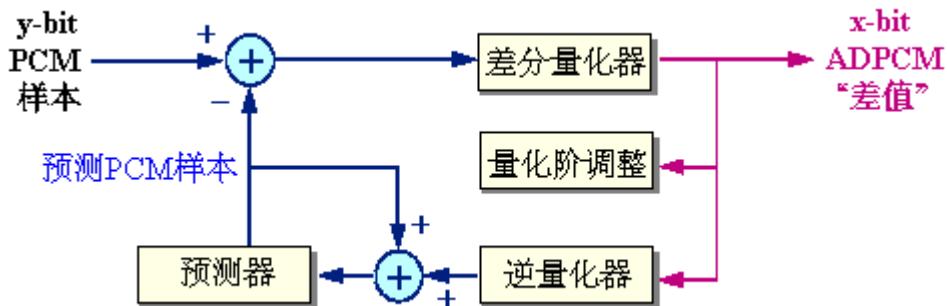


图 04-01-5 ADPCM 方框图

### 6.2.1.3 子带滤波编码

上述的波形编译码技术完全是在时间域里开发的，在时域里的编译码方法称为时域法(time domain approach)。在波形编译码器中，还有一种基于频率域压缩的方法，叫做频域法(frequency domain approach)。例如，在子带编码(sub-band coding, SBC)中，输入的话音信号被分成好几个频带(即子带)，变换到每个子带中的话音信号都进行独立编码，例如使用ADPCM编码器编码，在接收端，每个子带中的信号单独解码之后重新组合，然后产生重构话音信号。它的优点是每个子带中的噪声信号仅仅与该子带使用的编码方法有关系。对听觉感知比较重要的子带信号，编码器可分配比较多的位数来表示它们，于是在这些频率范围里噪声就比较低。对于其他的子带，由于对听觉感知的重要性比较低，允许比较高的噪声，于是编码器就可以分配比较少的位数来表示这些信号。自适应位分配的方案也可以考虑用来进一步提高音质。子带编码需要用滤波器把信号分成若干个子带，这比使用简单的ADPCM编译码器复杂，而且还增加了更多的编码时延。即使如此，与大多数混合编译码器相比，子带编译码的复杂性和时延相对来说还是比较低的。

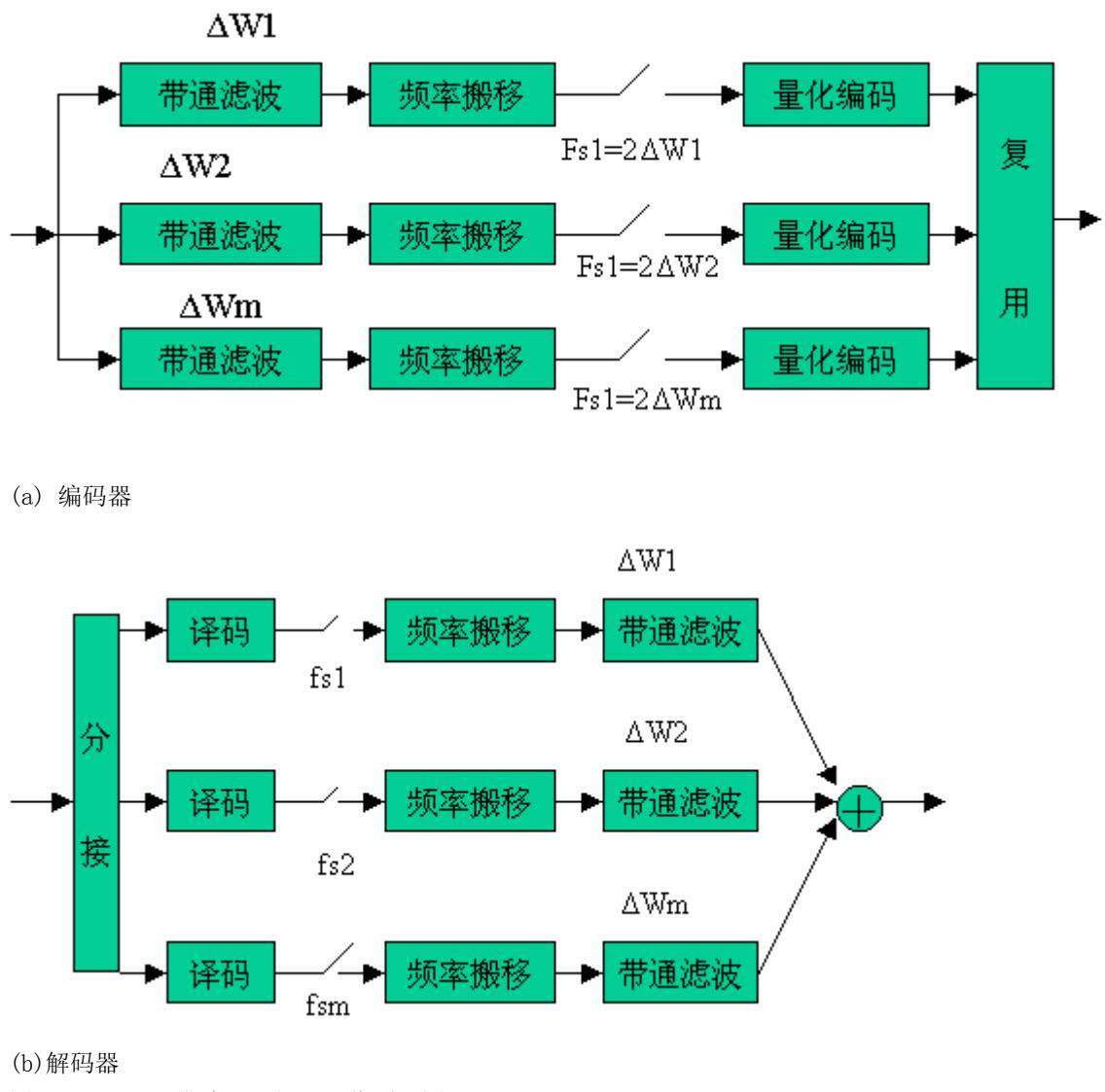


图 04-05-1 子带编码、解码工作原理图

在子带编码中，若各个子带的带宽  $\Delta W_k$  是相同的，则称为等带宽子带编码，否则，称为变带宽

子带编码。

对每个子带分别编码的好处是：

(1) 可以利用人耳(或人眼)对不同频率信号的感知灵敏度不同的特性，在人的听觉(或视觉)不敏感的频段采用较粗糙的量化，从而达到数据压缩的目的。例如，在声音低频子带中，为了保护音调和共振峰的结构，就要求用较小的量化阶、较多的量化级数，即分配较多的比特数来表示样本值。而话音中的摩擦音和类似噪声的声音，通常出现在高频子带中，对它分配较少的比特数。

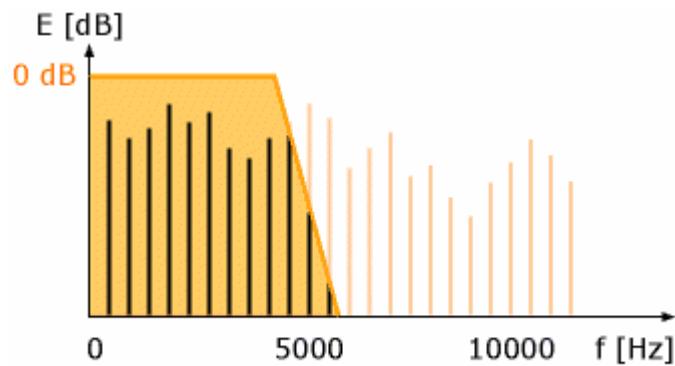
(2) 各个子带的量化噪声都束缚在本子带内，这就可以避免能量较小的频带内的信号被其他频带中量化噪声所掩盖。

(3) 通过频带分裂，各个子带的取样频率可以成倍下降。例如，若分成频谱面积相同的N个子带，则每个子带的取样频率可以降为原始信号取样频率的 $1/N$ ，因而可以减少硬件实现的难度，并便于并行处理。

#### 6.2.1.4 谱带复制技术 (Spectral Band Replication)

这种技术不是独立的。它的主要用途是附加到知觉音频编码 perceptual audio codecs 技术上使能显著地改善其表现。目前，已经实现对mp3 和 MPEG-2 AAC 的 SBR 增强。

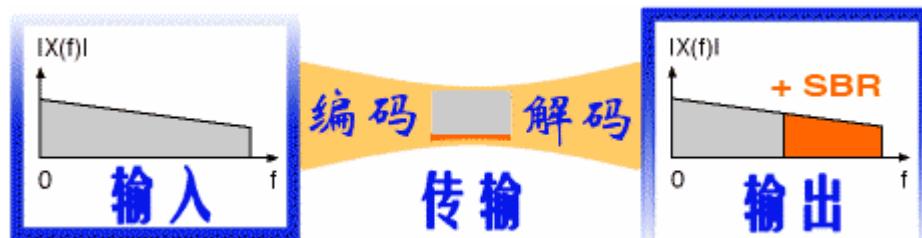
目前，流行的知觉音频编码技术通常都在 128kbps 左右可以达到 CD 音质。低于 128kbps 的时候这些编码方法都显著地出现质量下降的情况。编码器要么就减少音频带宽或者修改立体声影像，要么就产生一些讨厌的噪音信号。原因都是一样的：没有足够的位去表达完整的音频带宽。这两种对编码器的修改方法在某种程度上都是不可接受的。



典型信号的带宽限制图

解决办法之一是利用 SBR 编码。SBR (Spectral Band Replication 频段复制) 是一种新的音频编码增强工具。它提供了改善低位率情况下音频和语音编码的性能的可能。这种方法要么可在指定的位率下增加音频的带宽，要么可以在指定的编码质量要求上改善编码效率。

SBR 可以扩大传统的知觉音频编码算法在地位率时所能提供的带宽。因此它可以相当甚至超过模拟的 FM 音频带宽 (15kHz)。SBR 也可改进窄带音频编码的素质，为广播者提供达到 12kHz 的语音带宽。通常语音编码的带宽都是非常有限的，SBR 的作用不仅仅是改善其语音质量，更重要的是改善了语音的可理解性。SBR 主要是一种后处理技术，虽然在编码器中为了能正确地使播放器播放而要做一些前期的处理工作。



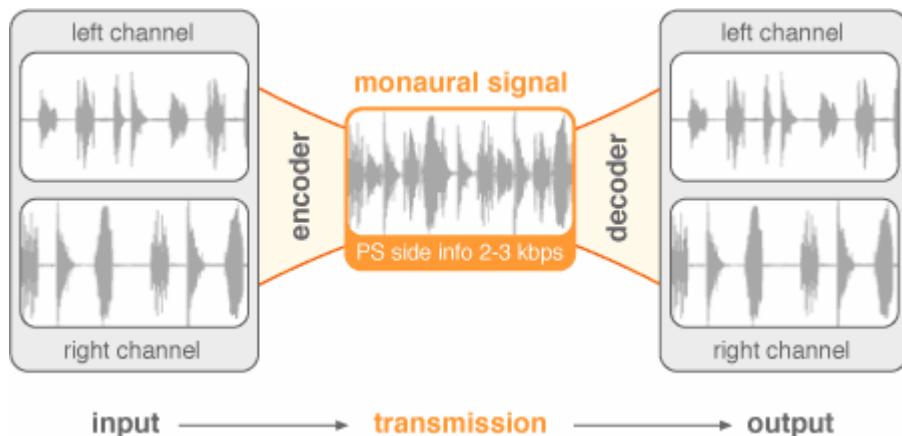
从技术角度来看，SBR 是一种对音频信号高频部分有很高效率的编码方法。当与 SBR 一同使用时，本来的编码器只需要负责传输频带中的低频部分。高频部分是通过 SBR 解码器生成的。在这里，SBR 解码器起到的是跟随在原来的解码器的一个后处理作用。与直接传输频谱信号不同的是，SBR 是根据对原来的编码器所传输的低频信号进行分析从而重新构建出高频信号的。为了保证能准确地重现信号，在编码后的比特流中需要以非常低的数据率来传输一些指导性的信号。重建的方法无论是对和声还是对近似噪声的组成都同样有效。并允许如同在频域上那样在时域上作适当的 SHAPING。结果是 SBR 能提供在非常低的数据率下的全带宽的音频编码，因而能提供与核心编码相比非常明显的压缩效率的提高。

SBR 能在中到低的位率范围上增强知觉音频编码的效率大约 30%（在某些情况下可能更高）。准确的改善程度与所依附的编码器有关。比如 mp3PRO，与 mp3 技术一同使用，可以在 64kbps 得到能与一般的大于 100kbps 的 mp3 音频流相媲美的立体声音频。SBR 还可以与单声道或者双声道甚至 5.1 声道的音频相结合。SBR 最大的优势就是在低数据速率下实现非常高效的编码，如果在高数据速率的情况下，SBR 效果并不明显。

#### 6.2.1.5 变量立体声技术 (Parametric Stereo)

传统的编码器对于立体声的声音都是对每一个声道分别独立编码的。变量立体声技术通过分析音频信号的 2 个声道提取出立体声信息，然后只有一个单声道的声音信号进行传统的编码，提取出的立体声信息和单声道的声音编码一同被编进码流。解码器会根据立体声信息重建立体声音频信号。

因为立体声信息的数据量大大小于一个单声道音频编码的数据量，因此 24kbps 采用 PS 技术编码的码流重建出的声音质量要大大好于相同码率不采用 PS 技术的码流。



#### 6.2.2 音源编码技术

音源编译码的想法是企图从声音波形信号中提取生成声音的参数，使用这些参数通过声音生成模型重构出话音。针对声音的音源编译码器叫做声码器 (vocoder)。在声音生成模型中，声道被等效成一个随时间变化的滤波器，叫做时变滤波器 (time-varying filter)，它由白噪声—无声话音段激励，或者由脉冲串——有声话音段激励。因此需要传送给解码器的信息就是滤波器的规格、发声或者不发声的标志和有声话音的音节周期，并且每隔 10~20 ms 更新一次。声码器的模型参数既可使用时域的方法也可以使用频域的方法确定，这项任务由编码器完成。音源编码的数据率在 2.4 kb/s 左右，

产生的语音虽然可以听懂，但其质量远远低于自然语音。增加数据率对提高合成语音的质量无济于事，这是因为受到语音生成模型的限制。尽管它的音质比较低，但它的保密性能好。

目前音源编码技术的典型应用是 MIDI。MIDI 文件本身只是一堆数字信号而已，不包含任何声音信息。我们知道任何声音都有其波形，如果我们把某种声音的波形记录下来，就可以正确地反映这个声音的实际效果，WAVE 文件就是这种形式，它在任何一台电脑上回放都是一样的。但是 MIDI 实际上是一堆数字信号，它记录的是在音乐的什么时间用什么音色发多长的音等等，而真正用来发出声音的是音源，但是不同声卡，不同软波表，不同硬件音源的音色是完全不同的，所以相同的 MIDI 文件在不同的设备上播放结果会完全不一样。这是 MIDI 的基本特点。但由于 MIDI 文件体积相当小，所以很适合在网络上传播。

### 6.2.3 混合编译码

混合编译码的想法是企图填补波形编译码和音源编译码之间的间隔。波形编译码器虽然可提供高话音的质量，但数据率低于 16 kb/s 的情况下，在技术上还没有解决音质的问题；声码器的数据率虽然可降到 2.4 kb/s 甚至更低，但它的音质根本不能与自然话音相提并论。为了得到音质高而数据率又低的编译码器，历史上出现过很多形式的混合编译码器，但最成功并且普遍使用的编译码器是时域合成-分析(analysis-by-synthesis, AbS)编译码器。这种编译码器使用的声道线性预测滤波器模型与线性预测编码(linear predictive coding, LPC)使用的模型相同，不使用两个状态(有声/无声)的模型来寻找滤波器的输入激励信号，而是企图寻找这样一种激励信号，使用这种信号激励产生的波形尽可能接近于原始话音的波形。AbS 编译码器由 Atal 和 Remde 在 1982 年首次提出，并命名为多脉冲激励(multi-pulse excited, MPE)编译码器，在此基础上随后出现的是等间隔脉冲激励(regular-pulse excited, RPE)编译码器、码激励线性预测 CELP(code excited linear predictive)编译码器和混合激励线性预测(mixed excitation linear prediction, MELP)等编译码器。

## 6.3 音频编码技术与应用

编码技术	应用
脉冲编码调制	Windows WAV RAW data
ADPCM	IMA ADPCM Microsoft ADPCM 2bit ADPCM
子带滤波编码	MPEG1/2 audio, MP3Pro AAC, AACPlus WMA AMR OGG
谱带复制技术	MP3Pro, AACPlus
音源编码技术	MIDI

## 7. 音频压缩标准简介

### 7.1 MPEG-1 Audio

声音的数据量由两方面决定：采样频率和样本精度。对单声道信号而言，每秒钟的数据量(位数)=采样频率×样本精度。要减小数据量，就需要降低采样频率或者降低样本精度。但是人耳可听到的频率范围大约是 20 Hz~20 kHz。根据奈奎斯特理论，要想不失真地重构信号，采样频率不能低于 40 kHz。再考虑到实际中使用的滤波器都不可能是理想滤波器，以及考虑各国所用的交流电源的频率，为保证声音频带的宽度，所以采样频率一般不能低于 44.1 kHz。这样，压缩就必须从降低样本精度这个角度出发，即减少每位样本所需要的位数。

MPEG 声音数据压缩的基础是量化。虽然量化会带来失真，但 MPEG 标准要求量化失真对于人耳来说是感觉不到的。在 MPEG 标准的制定过程中，MPEG-Audio 委员会作了大量的主观测试实验。实验表明，采样频率为 48 kHz、样本精度为 16 比特的声音数据压缩到 256 kb/s 时，即在 6: 1 的压缩率下，即使是专业测试员也很难分辨出是原始声音还是编码压缩后的声音。

MPEG Audio 是一个子带编码系统，声音数据压缩算法的根据是心理声学模型，心理声学模型中一个最基本的概念是听觉系统中存在一个听觉阈值电平，低于这个电平的声音信号就听不到。听觉阈值的大小随声音频率的改变而改变，各个人的听觉阈值也不同。大多数人的听觉系统对 2 kHz~5 kHz 之间的声音最敏感。一个人是否能听到声音取决于声音的频率，以及声音的幅度是否高于这种频率下的听觉阈值。

心理声学模型中的另一个概念是听觉掩饰特性，意思是听觉阈值电平是自适应的，即听觉阈值电平会随听到的频率不同的声音而发生变化。例如，在一般环境下房间里的普通谈话可以听得很清楚，但在摇滚乐环境下同样的普通谈话就听不清楚了。声音压缩算法也同样可以确立这种特性的模型，根据这个模型可取消冗余的声音数据。MPEG Audio 的压缩算法框图如图 9-06 所示。

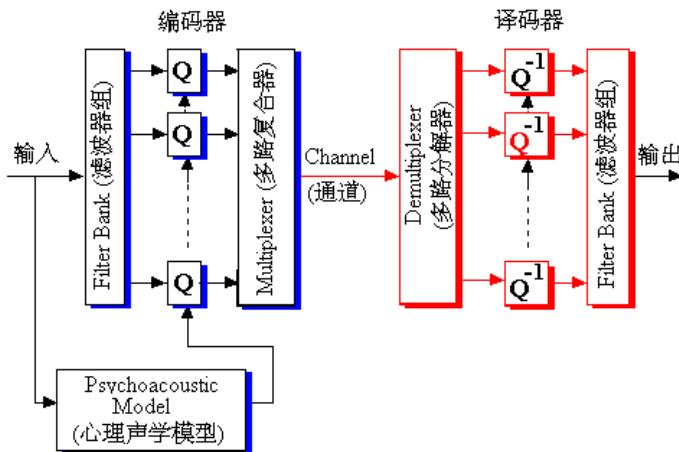


图 9-06 MPEG Audio 压缩算法框图

Dolby AC-3 同样利用人的听觉系统特性来压缩声音数据，它的压缩编码算法框图如图 9-07 所示。有兴趣的读者请浏览网址：<http://atsc.org/stan&rps.html>（浏览日期：1999 年 2 月 3 日）

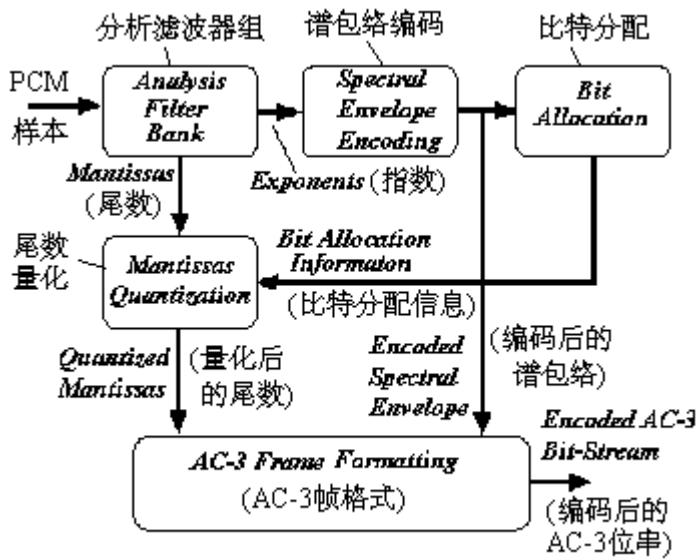


图 9-07 Dolby AC-3 压缩编码算法框图

MPEG 声音压缩定义了 3 个分明的层次，它们的基本模型是相同的。层 1 是最基础的，层 2 和层 3 都在层 1 的基础上有所提高。每个后继的层次都有更高的压缩比，但需要更复杂的编码解码器。MPEG 声音的每一个层都自含 SBC 编码器，其中包含“时间-频率多相滤波器组”、“心理声学模型(计算掩蔽特性)”、“量化和编码”和“数据流帧包装”，而高层 SBC 可使用低层 SBC 编码的声音数据。

MPEG 的声音数据分成帧(frame)，层 1 每帧包含 384 个样本的数据，每帧由 32 个子带分别输出的 12 个样本组成。层 2 和层 3 每帧为 1152 个样本，如图 9-13 所示。

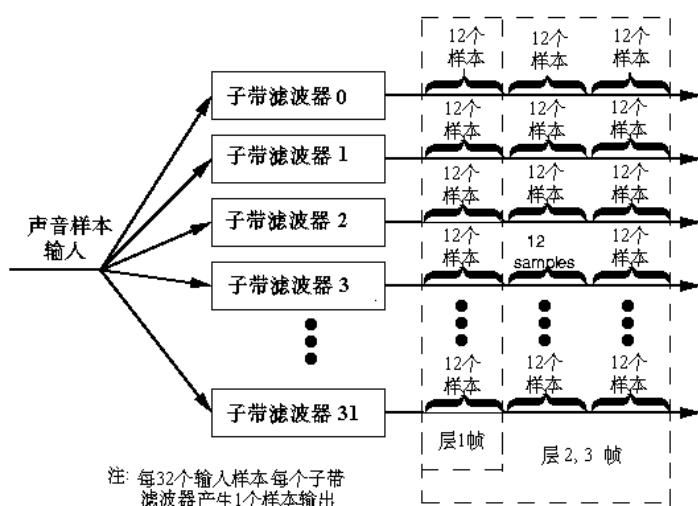


图 9-13 层 1、2 和层 3 的子带样本

MPEG 编码器的输入以 12 个样本为一组，每组样本经过时间-频率变换之后进行一次比特分配并记录一个比例因子(scale factor)。比特分配信息告诉解码器每个样本由几位表示，比例因子用 6 比特表示，解码器使用这个 6 比特的比例因子乘逆量化器的每个输出样本值，以恢复被量化的子带值。比例因子的作用是充分利用量化器的量化范围，通过比特分配和比例因子相配合，可以表示动态范围超过 120 DB 的样本。

### 7.1.1 MPEG1 Layer 1 (ISO11172-3 Layer 1 / MP1)

层 1 和层 2 的比较详细的框图如图 9-14 所示。层 1 的子带是频带相等的子带，它的心理声学模型仅使用频域掩蔽特性。层 1 的“时间-频率多相滤波器组”使用类似于离散余弦变换 DCT(discrete cosine transform)的分析滤波器组进行变换，以获得详细的信号频谱信息。根据信号的频率、强度和音调，滤波器组的输出可用来找出掩蔽阈值，然后组合每个子带的单个掩蔽阈值以形成全局的掩蔽阈值。使用这个阈值与子带中的最大信号进行比较，产生信掩比 SMR 之后再输入到“量化和编码器”。

“量化和编码器”首先检查每个子带的样本，找出这些样本中的最大的绝对值，然后量化成 6 比特，这个比特数称为比例因子(scale factor)。“量化和编码器”然后根据 SMR 确定每个子带的比特分配(bit allocation)，子带样本按照比特分配进行量化和编码。对被高度掩蔽的子带自然就不需要对它进行编码。

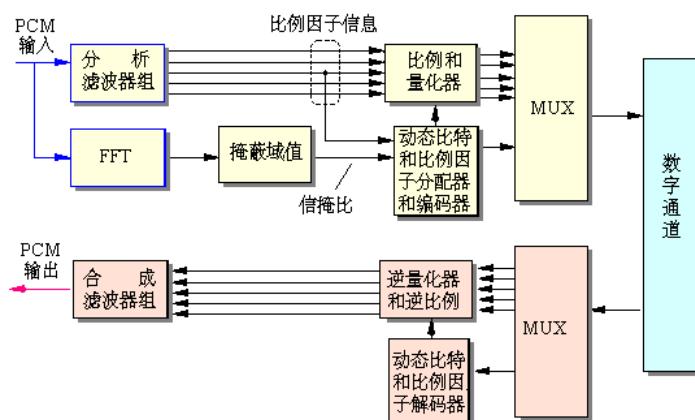


图 9-14 ISO/MPEG audio 层 1 和层 2 编码器和解码器的结构

“数据流帧包装”按规定的帧格式进行包装，实际上就是一个多路复合器 MUX。层 1 的帧结构如图 9-15 所示。每帧都包含：①用于同步和记录该帧信息的同步头，长度为 32 比特，它的结构如图 9-16 所示，②用于检查是否有错误的循环冗余码 CRC(cyclic redundancy code)，长度为 16 比特，③用于描述比特分配的比特分配域，长度为 4 比特，④比例因子域，长度为 6 比特，⑤子带样本域，⑥有可能添加的附加数据域，长度未规定。



图 9-15 层 1 的帧结构

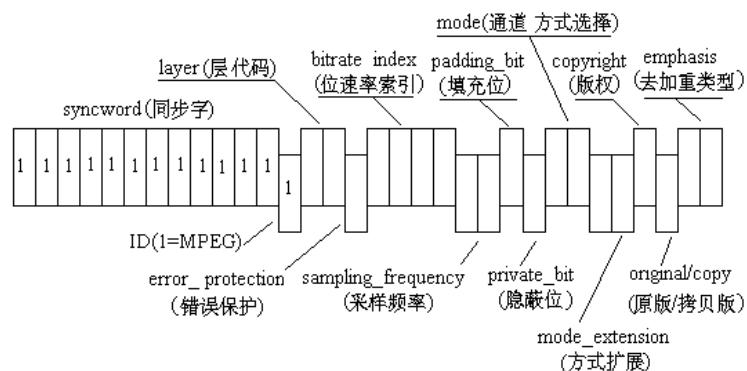


图 9-16 MPEG 声音比特流同步头的格式

### 7.1.2 MPEG1 Layer 2 (ISO11172-3 Layer 2 / MP2)

层 2 对层 1 作了一些直观的改进，相当于 3 个层 1 的帧，每帧有 1152 个样本。它使用的心理声学模型除了使用频域掩蔽特性之外还利用了时间掩蔽特性，并且在低、中和高频段对比特分配作了一些限制，对比特分配、比例因子和量化样本值的编码也更紧凑。由于层 2 采用了上述措施，因此所需的比特数减少了，这样就可以有更多的比特用来表示声音数据，音质也比层 1 更高。

层 1 是对一个子带中的一个样本组(由 12 个样本组成)进行编码，而层 2 和层 3 是对一个子带中的三个样本组进行编码。图 9-13 也表示了层 2 和层 3 的分组方法。

如图 9-17 所示，层 2 使用与层 1 相同的同步头和 CRC 结构，但描述比特分配的位数(即比特数)随子带不同而变化：低频段的子带用 4 比特，中频段的子带用 3 比特，高频段的子带用 2 比特。层 2 比特流中有一个比例因子选择信息(scale factor selection information, SCFSI)域，解码器根据这个域的信息可知道是否需要以及如何共享比例因子。

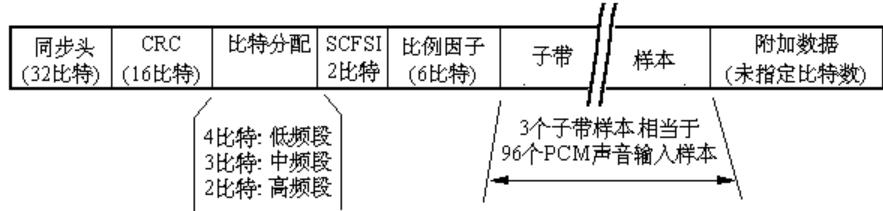


图 9-17 层 2 比特流数据格式

### 7.1.3 MPEG1 Layer 3 (ISO11172-3 Layer 3 / MP3)

层 3 使用比较好的临界频带滤波器，把声音频带分成非等带宽的子带，心理声学模型除了使用频域掩蔽特性和时间掩蔽特性之外，还考虑了立体声数据的冗余，并且使用了霍夫曼(Huffman)编码器。层 3 编码器的详细框图如图 9-18 所示。

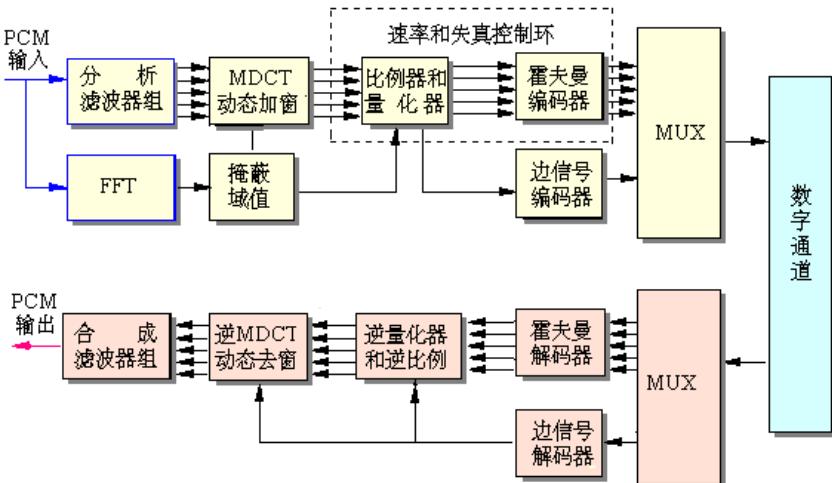


图 9-18 ISO/MPEG audio 层 3 编码器和解码器的结构

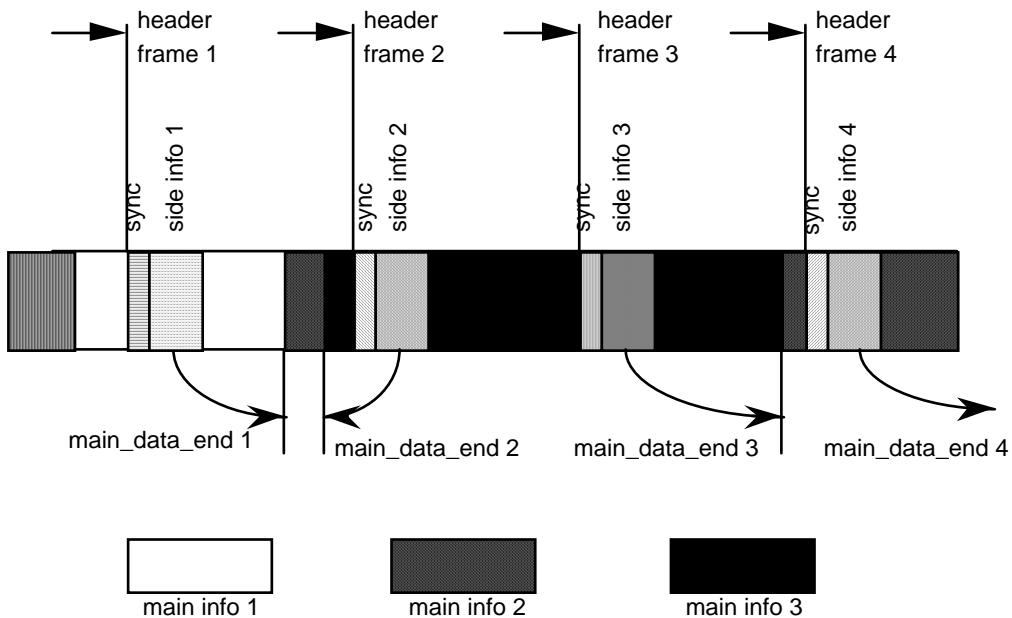
层 3 使用了从 ASPEC(Audio Spectral Perceptual Entropy Encoding) 和 OCF(Optimal Coding In The Frequency domain) 导出的算法，比层 1 和层 2 都要复杂。虽然层 3 所用的滤波器组与层 1 和层 2 所用的滤波器组的结构相同，但是层 3 还使用了改进离散余弦变换(modified discrete cosine transform, MDCT)，对层 1 和层 2 的滤波器组的不足作了一些补偿。MDCT 把子带的输出在频域里进一步细分以达到更高的频域分辨率。而且通过对子带的进一步细分，层 3 编码器已经部分消除了多相滤波器组引入的混迭效应。

层 3 指定了两种 MDCT 的块长：长块的块长为 18 个样本，短块的块长为 6 个样本，相邻变换窗

口之间有 50% 的重叠。长块对于平稳的声音信号可以得到更高的频域分辨率，而短块对跳变的声音信号可以得到更高的时域分辨率。在短块模式下，3 个短块代替 1 个长块，而短块的大小恰好是一个长块的 1/3，所以 MDCT 的样本数不受块长的影响。对于给定的一帧声音信号，MDCT 可以全部使用长块或全部使用短块，也可以长短块混合使用。因为低频区的频域分辨率对音质有重大影响，所以在混合块长模式下，MDCT 对最低频的 2 个子带使用长块，而对其余的 30 个子带使用短块。这样，既能保证低频区的频域分辨率，又不会牺牲高频区的时域分辨率。长块和短块之间的切换有一个过程，一般用一个带特殊长转短或短转长数据窗口的长块来完成这个长短块之间的切换。

除了使用 MDCT 外，层 3 还采用了其他许多改进措施来提高压缩比而不降低音质。虽然层 3 引入了许多复杂的概念，但是它的计算量并没有比层 2 增加很多。增加的主要原因是编码器的复杂度和解码器所需要的存储容量。

层 3 的帧结构和层 1, 2 类似，但是扩展了附加数据段的功能。在层 1, 2 里，附加数据段发的是一些用户自定义的数据，对解码没有实际意义。在层 3 里，允许把一帧的数据分割成几块，放到其他帧的附加数据段中。这样可以极大提高帧的编码效率，在相同的文件长度下，层 3 可以提供比层 1, 2 更多的有效数据。



**Note:** 'info' means information

**Figure 3-A.8. Layer III illustration of granules for frame with no block split in first granule and block split in second granule.**

## 7.2 MPEG-2 Audio

MPEG-2 标准委员会定义了两种声音数据压缩格式，一种称为 MPEG-2 Audio，或者称为 MPEG-2 多通道(Multichannel)声音，因为它与 MPEG-1 Audio 是兼容的，所以又称为 MPEG-2 BC (Backward Compatible)。另一种称为 MPEG-2 AAC (Advanced Audio Coding)，因为它与 MPEG-1 声音格式不兼容，因此通常称为非后向兼容 MPEG-2 NBC(Non-Backward-Compatible)标准。

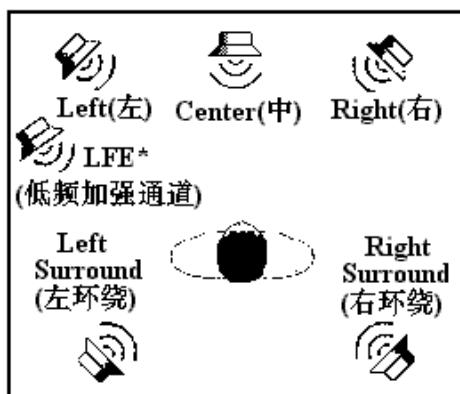
### 7.2.1 MPEG-2 Audio (ISO/IEC 13818-3)

MPEG-2 Audio (ISO/IEC 13818-3) 和 MPEG-1 Audio (ISO/IEC 1117-3) 标准都使用相同种类的编译码器，层-1，-2 和-3 的结构也相同。MPEG-2 声音标准与 MPEG-1 标准相比，MPEG-2 做了如下扩充：①增加了 16 kHz, 22.05 kHz 和 24 kHz 采样频率，②扩展了编码器的输出速率范围，由 32~384 kb/s 扩展到 8~640 kb/s, ③增加了声道数，支持 5.1 声道和 7.1 声道的环绕声。此外 MPEG-2 还支持 Linear PCM(线性 PCM)和 Dolby AC-3(Audio Code Number 3)编码。它们的差别如表 9-05 所示。

表 9-05 MPEG-1 和-2 的声音数据规格

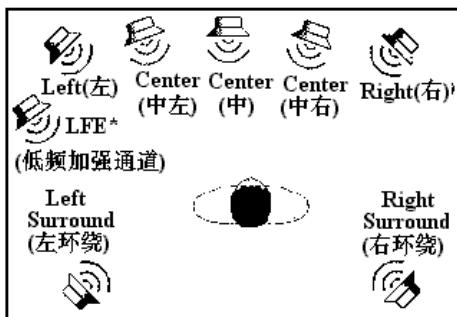
参数名称	Linear PCM	Dolby AC-3	MPEG-2 Audio	MPEG-1 Audio
采用频率	48/96 kHz	32/44.1/48 kHz	16/22.05/24/ 32/44.1/48 kHz	32/44.1/48 kHz
样本精度 (每个样本的比特 数)	16/20/24	压缩 (16 bits)	压缩 (16 bits)	16
最大数据传 输率	6.144 Mb/s	448 kb/s	8~640 kb/s	32 ~ 448 kb/s
最大声道数	8	5.1	5.1/7.1	2

MPEG-2 Audio 的“5.1 环绕声”也称为“3/2-立体声加 LFE”，其中的“.1”就是指 LFE 声道。它的含义是播音现场的前面可有 3 个喇叭声道(左、中、右)，后面可有 2 个环绕声喇叭声道，LFE (low frequency effects) 是低频音效的加强声道，如图 9-19(a) 所示。7.1 声道环绕立体声与 5.1 类似，如图 9-19(b) 所示。



\* LFE: Low Frequency Enhancement  
(3 Hz to 120 Hz)

图 9-19(a) 5.1 声道立体环绕声



\* LFE: Low Frequency Enhancement (3 Hz to 120 Hz)

图 9-19(b) 7.1 声道立体环绕声

Dolby AC-3 支持 5 个声道(左、中、右、左环绕、右环绕和 0.1 kHz 以下的低音音效声道)，声音样本的精度为 20 比特，每个声道的采样率可以是 32 kHz, 44.1 kHz 或者 48 kHz。

MPEG-2 声音标准的第 3 部分(Part 3)是 MPEG-1 声音标准的扩展，扩展部分就是多声道扩展(multichannel extension)，如图 9-20 所示。这个标准称为 MPEG-2 后向兼容多声道声音编码(MPEG-2 backwards compatible multichannel audio coding)标准，简称为 MPEG-2 BC。

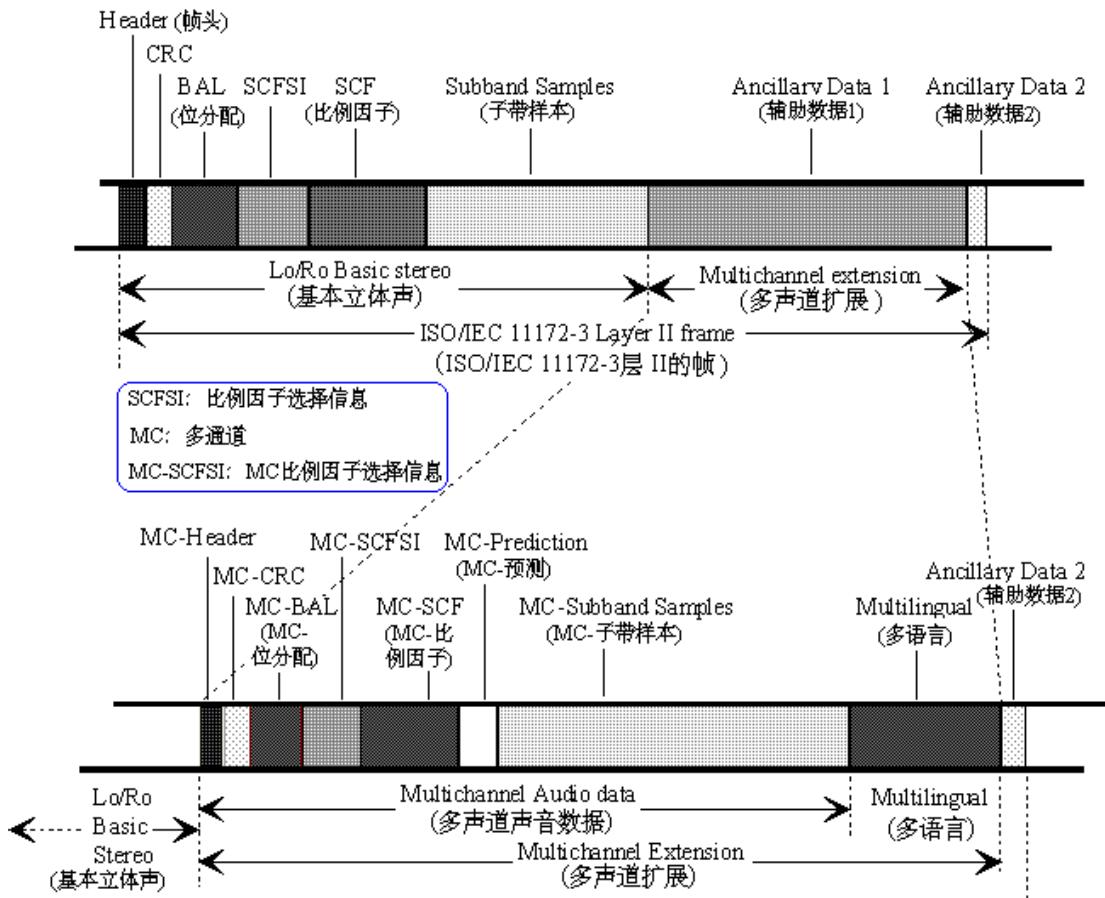


图 9-20 MPEG-2 Audio 的数据块

## 7.2.2 MPEG 2.5

MPEG2.5 是非标准的 MPEG2 编码的改进版本，它和 MPEG2 唯一的不同是增加了支持的信号采样率 8, 11.025 和 12kHz。

MPEG-1 和 MPEG-2 在音频方面都使用相同的音频编解码家族：Layer-1, -2 和 -3 共三层。数字越小，相对应的技术就越简单，越容易实现。MPEG-2 的新音频特性主要表现在 MPEG-2 具有低采样率扩展以满足只具备非常有限的应用范围。新的采样频率为：16, 22.05 或 24 kHz, 比特率则扩展到 8 kbps。

MPEG-2 标准允许比特率低到 8kbps，在该种情况下，实际有效的音频带宽需要做出限制，比如限制到 3khz。因此，实际的采样率会被减少到 8khz。采样率越低，频率分辨率就越好，时间分辨率就越差，流格式中控制信息和音频数据的消耗比就越好。由于 MPEG-2 定义的最低的采样率为 16kHz，于是 Fraunhofer 便对此进行扩展，将原来 MPEG-2 所支持的低采样率再除以 2，得到：8, 11.025, 和 12 kHz，称为“MPEG 2.5”。在第三层的音质表现上，8 kbps @ 8 kHz 或 16 kbps @ 11 kHz 明显比 8 或 16 kbps @ 16 kHz 要好。

### 7.2.3 MPEG-2 AAC (MP4)

MPEG-2 AAC (ISO/IEC 13818-7) 是 MPEG-2 标准中的一种非常灵活的声音感知编码标准，主要使用听觉系统的掩蔽特性来减少声音的数据量，并且通过把量化噪声分散到各个子带中，用全局信号把噪声掩蔽掉。

AAC 支持的采样频率可从 8kHz 到 96kHz，AAC 编码器的音源可以是单声道的、立体声的和多声道的声音。AAC 标准可支持 48 个主声道、16 个低频音效 (LFE, low frequency effects) 加强通道、16 个配音声道 (overdub channel) 或者叫做多语言声道 (multilingual channel) 和 16 个数据流。MPEG-2 AAC 在压缩比为 11:1 (即每个声道的数据率为  $(44.1 \times 16) / 11 = 64$  kbps)，5 声道的总数据率为 320kbps 的情况下，很难区分还原后的声音与原始声音之间的差别。与 MPEG-1 的层 2 相比，MPEG-2 AAC 的压缩率可提高 1 倍，而且质量更高，与 MPEG-1 的层 3 相比，在质量相同的条件下数据率是它的 70%。

AAC 定义的编码和解码的基本结构如图 05-04-4 和图 05-04-5 所示。

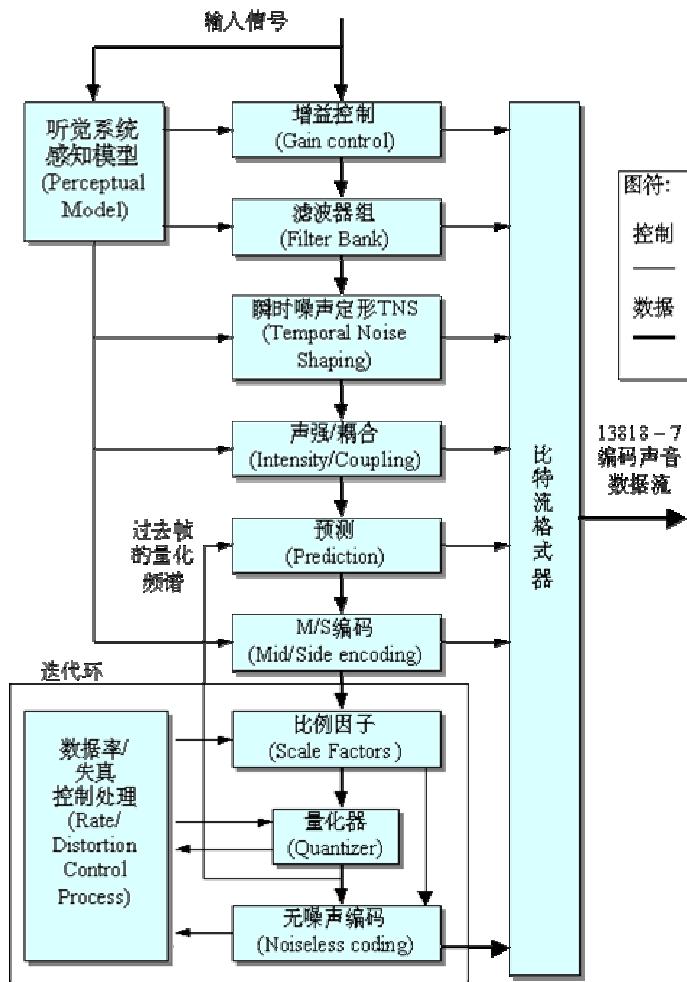


图 05-04-4 MPEG-2 AAC 编码器框图

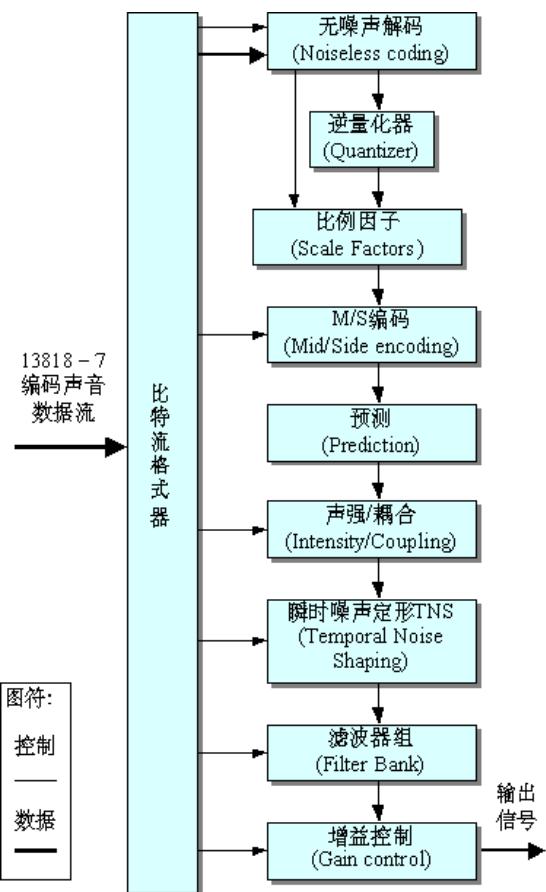


图 05-04-5 MPEG-2 AAC 解码器框图

MPEG-2 AAC 是基于 MPEG 层 3 的编解码器架构建立的，并且保留了层 3 的大部分功能。相对于层 3 编解码器，AAC 通过加入了一下几点新技术提高了性能：

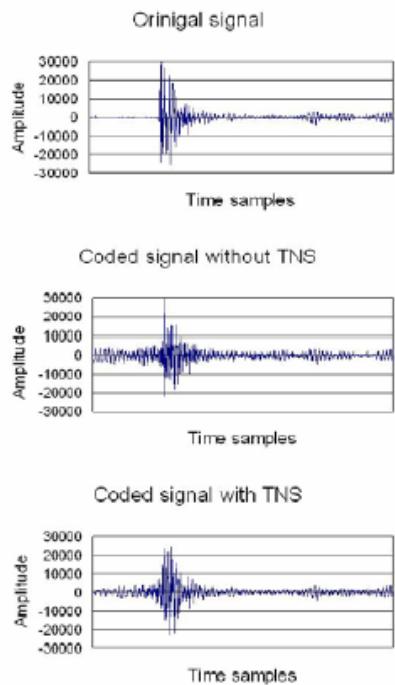
下面是 MPEG-2 AAC 编码/解码器模块的一些说明，其中和 MPEG-2 Audio (ISO13818-2) 最主要的不同是增加了瞬时噪声定形技术和预测技术。

(a) 改进的滤波器组 (Filter Bank)

改进离散余弦变换 MDCT 可以提供 2048 个频率分量的频率分辨率，几乎是 MP3 的 4 倍。

(b) 瞬时噪声定形

在感知声音编码中，瞬时噪声定形 (TNS, temporal noise shaping) 模块是用来控制量化噪声的瞬时形状的一种方法，解决掩蔽域值和量化噪声的错误匹配问题。这种技术的基本思想是，在时域中的音调声信号在频域中有一个瞬时尖峰，TNS 使用这种双重性来扩展已知的预测编码技术，把量化噪声置于实际的信号之下以避免错误匹配。



(c) 预测

预测(Prediction)是在语音编码系统中普遍使用的一种技术，它主要用来减少平稳(stationary)信号的冗余度。

### (2) MPEG-2 AAC 的轮廓

AAC 标准定义了三种配置：基本配置、低复杂性配置和可变采样率配置：

#### 1. 基本配置(Main Profile)：

在这种配置中，除了“增益控制(Gain Control)”模块之外，AAC 系统使用了图中所示的所有模块，在三种配置中提供最好的声音质量，而且 AAC 的解码器可以对低复杂性配置编码的声音数据进行解码，但对计算机的存储器和处理能力的要求方面，基本配置比低复杂性配置的要求高。

#### 2. 低复杂性配置(Low Complexity Profile)：

在这种配置中，不使用预测模块和预处理模块，瞬时噪声定形(temporal noise shaping, TNS)滤波器的级数也有限，这就使声音质量比基本配置的声音质量低，但对计算机的存储器和处理能力的要求可明显减少。

#### 3. 可变采样率配置(Scalable Sampling Rate Profile)：

在这种配置中，使用增益控制对信号作预处理，不使用预测模块，TNS 滤波器的级数和带宽也都有限制，因此它比基本配置和低复杂性配置更简单，可用来提供可变采样频率信号。

## 7.3 MPEG-4 Audio

### 7.3.1 MPEG-4 AAC (ISO14496-3)

MPEG-4 音频的通用音频编码(General Audio Coding)覆盖了从每通道 16kbit/s 到每通道 64kbit/s 的位速范围。使用 MPEG-4 通用音频能获得优于 AM 到透明声音质量的质量水准。MPEG-4 通用音频支持 4 种音频对象类型。其中，主 AAC、AAC LC 和 AAC SSR 来自 MPEG-2 的 AAC，但加入了一些新的功能以便进一步提高位速效率。第 4 种音频对象类型 AAC LTP 是 MPEG-4 独有的，不向下兼容。

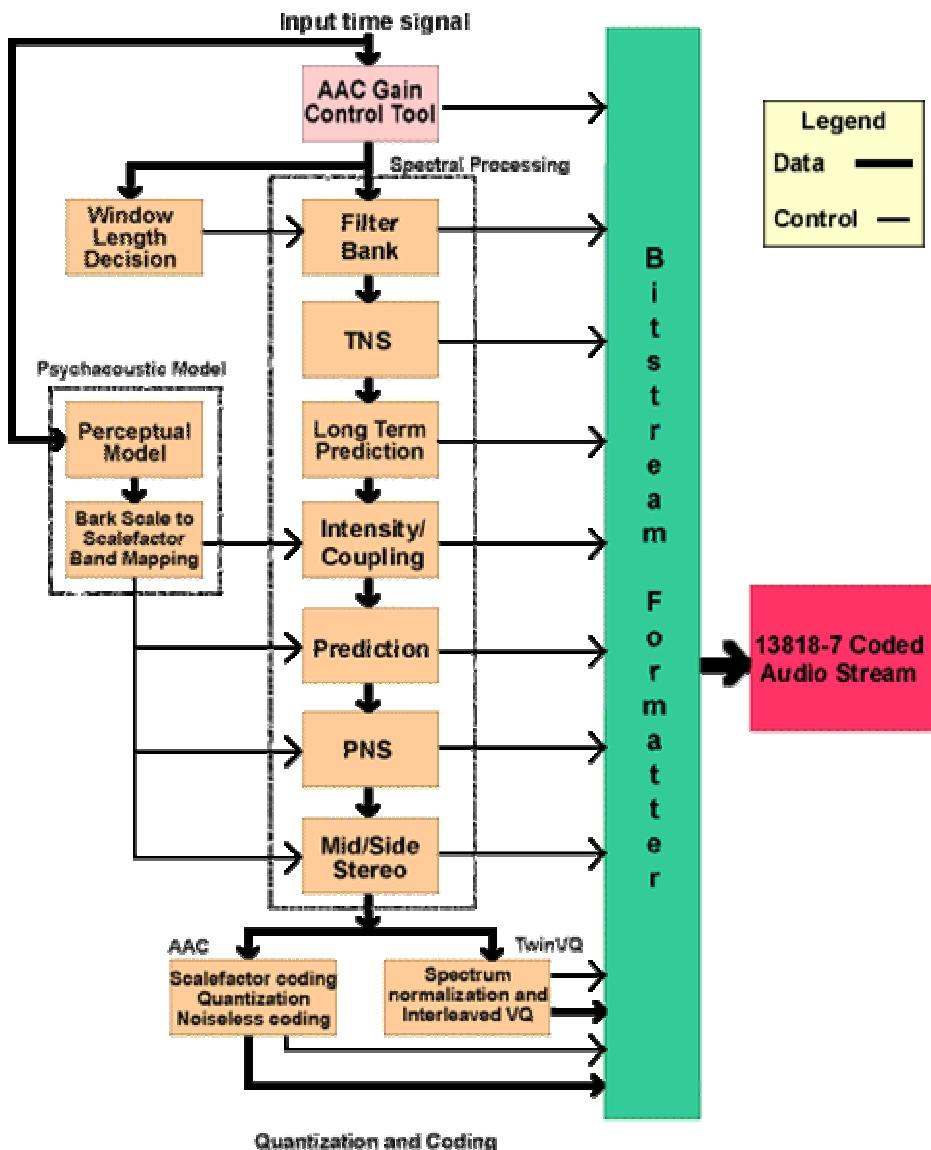


图 05-05-2 MPEG-4 普通声音编码器的组成模块

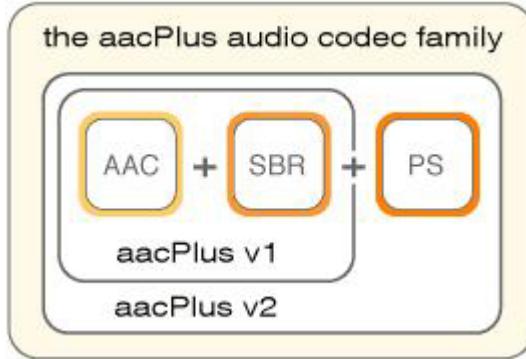
在 MPEG-4 GA 中引入了一种 MPEG-2 AAC 没有的新特性，即知觉噪音替换 (PNS, Perceptual noise substitution)。它的目的是进一步优化低位速中 AAC 的位率效率。

知觉噪音替换技术是基于这样一种观察结果，即一种噪音听起来像另一种噪音。这说明一个噪音信号实际的良好结构对于此信号的主观感受是不重要的。因而，不传输噪音信号的实际谱成分，而是用位流告诉说这一频率范围有一个类似某种噪音的信号，并在该频段内的总功率上给予一些附加的信息。PNS 能在标度因子基上切换，所以即使只有一些谱范围带有一个噪音结构，PNS 也能用于节约位速。在解码器中，将根据位流中标志的功率等级把一个随机生成的噪音插入相应的谱区域。

### 7.3.2 AACplus

MPEG-4 aacPlus 由三大 MPEG 技术组合而成，包括 AAC (高级音频编码) 技术、SBR (频段复制) 技术和 PS (变量立体声) 技术。SBR 是一种独特的带宽扩展技术，使用该技术进行音频编码，在同等质量的情况下可以节省一半的码率。而 PS 则能显著地提高低码率立体声信号的编解码效率。SBR 和 PS 都是采用向前或向后兼容的方法来提高音频编码效率的。

aacPlus v1 由 AAC 和 SBR 组合而成, 作为 MPEG-4 第 10 部分的高效编解码标准(HE AAC)。aacPlus v2 是建立在 aacPlus v1 巨大的成功经历基础上的, 由于可以对复杂立体声信号进行高效编码, 因此在所有领域里增加了很高的应用价值。同时, aacPlus v2 是 aacPlus v1 的一个真正扩展集, 就如同 aacPlus v1 是 AAC 的一个真正扩展集一样。由于 MPEG 中变量立体声技术的增加, aacPlus v2 已经成为最完美的、开放的低码率音频编码标准。



aacPlus 已经广泛地被多个国际标准组织所采用。2004 年 9 月, aacPlus v2 作为高质量音频编码标准, 已经被 3GPP (第三代移动通讯合作方案) 所采纳, 而且它的所有组成构件成为 MPEG-4 的一部分。aacPlus v1 已被 3GPP2、ISMA (国际流媒体联盟)、DVB (数字视频广播)、DVD 论坛、DRM (世界性数字广播) 等组织所采纳。作为 MPEG-4 音频标准的一个完整的部分, aacPlus 已经成为最新视频编码标准 MPEG-4 第 10 部分 (H.264/AVC) 的完美搭配。

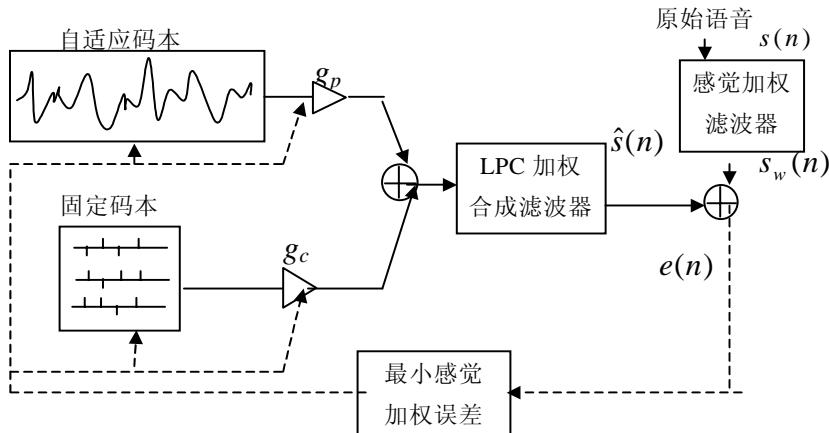
## 7.4 自适应多码率语音传输编解码器 (AMR)

AMR 作为第四代 GSM 语音编解码标准, 提供了一种自适应的解决方法来跟踪快速变化的无线信道情况和本地流量情况。现今的 GSM 语音和信道编码器工作在固定码率上, 这些码率在设计阶段就已经选定, 是理想信道性能和信道误码鲁棒性的一个折衷。而 AMR 编码器实时根据信道类型 (全速率或半速率) 选择多种码率中的一种, 从而达到语音编码和信道编码的最优组合以满足瞬时的无线信道条件和本地容量需求。

AMR 基于带宽的考虑可分为 AMR-NB (AMR Narrowband) 和 AMR-WB (AMR Wideband)。对于 AMR-NB, 语音通道带宽限制为 3.7 MHz, 8000 Hz 的采样频率, 定义了 8 种码率 12.2, 10.2, 7.95, 7.40, 6.70, 5.90, 5.15 and 4.75 kbit/s。而 AMR-WB 为 7MHz 的带宽, 采样频率 16000, 定义了 9 种码率 23.85, 23.05, 19.85, 18.25, 15.85, 14.25, 12.65, 8.85 和 6.6 kbit/s。考虑到语音的短时相关性, 每帧长度均为 20ms。

### 7.4.1 AMR 编码器原理

此编码器是基于代数码本线性预测 (ACELP) 这种混合编码算法。基本原理是原始语音按帧输入, 根据使合成语音与原始语音的加权均方误差最小的准则, 从随机码本和固定码本中挑选合适的码矢以代替残差信号, 并将码矢地址和增益及各滤波器的参数量化编码后传送到接收端; 接收端恢复各滤波器时, 采用与发送端相同的码本, 按照码矢地址找到该码矢乘上增益, 激励合成滤波器, 得到合成语音。编码器提取 ACELP 模型参数, 解码器再根据这些参数构成的激励信号合成、重建语音信号。在编码部分需要抽取下列典型参数: 线性预测滤波器系数 (LP), 自适应码本 (ACB) 和固定码本 (FCB) 索引以及 2 种码本的增益。原理如图二所示。下面将分别从编码和解码的角度概述 AMR 编解码方案。



图二 AMR 编码原理

#### 7.4.2 非连续传输编码

AMR 用于语音通讯时，为了进一步缩小码率，还应用了 3 种技术：话音活动检测(VAD: Voice Activity Detection)，不连续发射(DTX: discontinuous transmission)和舒适噪音生成技术(CNG: Comfort Noise Generation)。

在典型的谈话过程中，语音仅占总时间的大约 40%。因此可以采用非连续传输(DTX)，即有语音信号时才进行传输。具体方法是通过 VAD 判断当前是否有语音信号输入，当 VAD 确定没有语音时，在空中接口上不进行传输。经过一段预定时间间隔后，再发送一个包含一组参数的 SID 帧，这些参数用于接收器舒适噪音产生功能(CNG)。尽管在理论上 VAD 是实现 DTX 所需的全部，但来自接收器的完全静寂降低整体感觉质量。为解决此问题，接收器采用了“舒适噪音”功能，即利用 SID 帧参数的逐渐衰减来产生类似发射器背景噪音的模拟自然环境的后台音。

#### 7.5 WMA (Windows Media Audio)

WMA 为 Windows Media Audio 的缩写，是微软公司制定的音乐文件格式。WMA Codec 是 Microsoft 音频技术的首要 Codec。据微软自身发表的声明，目前最新的版本 WMA 9.0 相对于 WMA 8，在压缩率上有着 20% 的提升。

WMA 类似于 MP3，同样是一种失真压缩，损失了声音中人耳极不敏感的甚高、甚低音部分。但与 MP3 相比较起来，仍然具有不少优势。

- 1) 它具有与 MP3 相当的音质，但容量更小。
- 2) 更先进的压缩算法在给定速率下可获得更好的质量。
- 3) 特别适合于低速率传输。
- 4) 除了损失了的音频成份外，WMA 比起 MP3 在频谱结构上更接近于原始音频，因而相对起来具有更好的声音保真度。

#### 7.6 乐器数字接口 (MIDI)

MIDI (Musical Instrument Digital Interface) 是乐器数字接口的英文缩写。MIDI 本身并不能发出声音，它是一个协议，只包含用于产生特定声音的指令，而这些指令则包括调用何种 MIDI 设备的声音，声音的强弱及持续的时间等，然后由 MIDI 播放器解释这些指令去合成相应的声音。MIDI 文件实质上是指示电子音乐合成器要做什么、怎么做（如演奏某个音符、加大音量、生成音响效果）

的一套标准指令序列

### 7.6.1 MIDI 的 3 个标准

由于早期的 MIDI 设备在乐器的音色排列上没有统一的标准，造成不同型号的设备回放同一首乐曲时也会出现音色偏差。为了弥补这一不足，便出现了 GS、GM 和 XG 这类音色排列方式的标准。GS 排在第一位是由于它最早出台，并且是由业界大名鼎鼎的 ROLAND 公司制定并推出的。ROLAND 是日本非常出名的电子乐器厂商，其生产开发的电子键盘、MIDI 音源以及软波表都享有盛誉。所以 GS 颇具权威性，它完整的定义了 128 种乐器的统一排列方式，并规定了 MIDI 设备的最大复音数不可少于 24 个等详尽的规范。

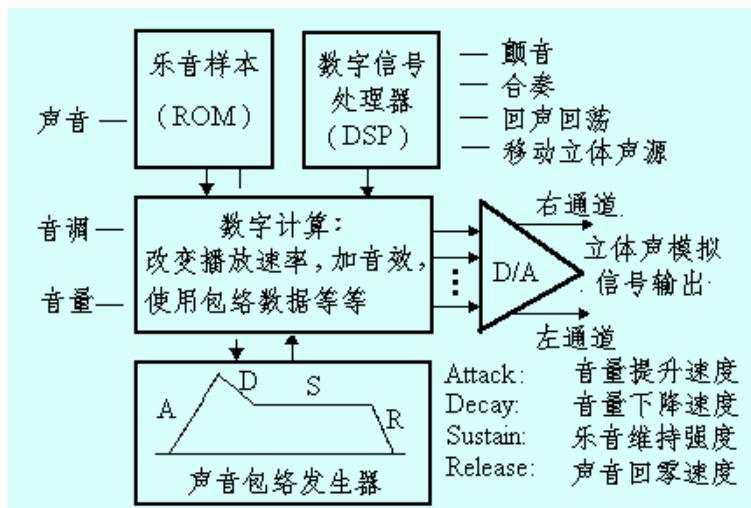
GM 标准则是在 GS 的基础上，加以适当简化而成的。由于它比较符合众多中小厂商的口味，成为了业界广泛接受的标准。

在电子乐器方面唯一可与 ROLAND 相匹敌的 YAMAHA 公司也不甘示弱，于 94 年推出自己的标准——XG。与 GM、GS 相比 XG 提供了更为强劲的功能和一流的扩展能力，并且完全兼容以上两大标准。而且凭借 YAMAHA 公司在电脑声卡方面的优势，使得 XG 在 PC 上有着广阔的用户群。

### 7.6.2 波表合成器

播放 MIDI 文件时，有两种方法合成声音；FM 合成和波表(wavetable)合成。现在常用的方法是波表合成。FM 合成是通过多个频率的声音混合来模拟乐器的声音，波表合成是将乐器的声音样本存储在声卡波形表中，播放时从波形表中取出来，产生声音。

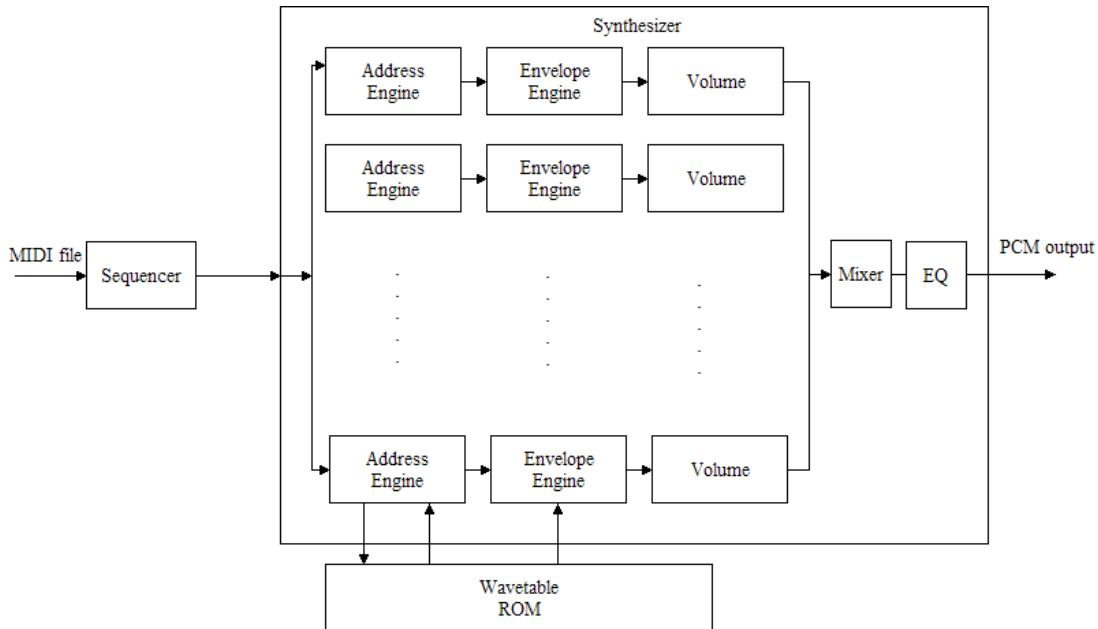
Wavetable 合成器所需要的输入控制参数比较少，可控的数字音效也不多，大多数采用这种合成方法的声音设备都可以控制声音包络的 ADSR 参数，产生的声音质量比 FM 合成方法产生的声音质量要高



### 7.6.3 MIDI 播放器结构

MIDI 解码器包括 2 个部分，序列器(sequencer)和合成器(Synthesizer)。Sequencer 负责解析出 MIDI 文件中的每一条指令，并且在正确的时间(Delta\_time)把相应的指令发给 synthesizer。波表合成器在接收到 sequencer 发送过来的指令后，Address engine 会按照指令中的声音在相应的波表(wave table)中取出对应的乐器声音样本，Envelope engine 会根据指令中的音调，音量的信息对乐器声音样本进行相应处理。混音器(Mixer)负责混合各个 Envelop engine 输出的 PCM 数据并进行均

衡(equalizer)处理，输出最后PCM样本。



#### 7.6.4 MIDI 播放器中 3 种效果

- Chorus

Chorus 直译就是和声，齐唱。简单来说，它可以把单一的声音复制、扩张、改变音色，这些复制声音会稍微延迟并尾随原音播放出来，可以造成好像同时有 2 种乐器在演奏的感觉。

- Reverb

演奏音乐时，声音一定会被环境周围的某些物体（如墙壁）吸收、反弹、被改变波形，进而形成回音。在不同的空间中，回音也有不同的程度。Reverb 效果既是模拟出不同的空间感，让输入的声音有回音的感觉，可以让声音听起来更有实际感和立体感，也可以提高声音的饱和度。

- 3D 音效

“3D 音效”既不是单声道，也不是双声道，它是一种音频的处理技术，使聆听者在非实际的环境下，感觉到发出声音的地点，这就必须非常讲究扬声器(喇叭)的放置位置与数目。但是在手机与个人数字助理中，无法放置如此多的扬声器，因此发展出以两个扬声器加上运用硬件或软件的方式来模拟“3D 音效”，就是所谓的“3D 增强立体声音效”(3D Enhancement)。

声音在不同位置传至左右耳朵时，会产生不同相位差。利用此相位差原理和硬件方法，便可以仿真出 3D 增强立体声音效。即使系统在体积或设备上受到限制，而必需将左右喇叭摆放得很近时，仍然可以改善立体声各个高低声部的定位的种种问题。

#### 8. 各种音频标准对比

Compression	Advantages	Disadvantages
MP3	The best portability, software and hardware support	Quality is not enough for demanding people

	Encoder and decoder source code available	Many MP3s available for download are far from optimal quality, because bad quality encoders exists
	Quality is good with high bitrate and with high bitrate VBR	Many different quality encoders (some are very bad quality)
	Future hardware support is pretty much certain	
<b>AAC</b>	High quality AAC encoder produces much better quality than MP3	Heavily patented
	MPEG ISO compatible streams (Psytel AAC encoder)	Licence royalties
	Hardware support is emerging: future looks good	Encoding is CPU-intensive
	Streaming of MPEG2 and MPEG4 AAC is free of charge	
	Very flexible (multichannel, 8–96 kHz, various bit depths)	
	Can achieve good quality at lower bitrates	
	Still in development (improvements to come)	
<b>ADPCM</b>	Fix bitrate and compression ratio	Low compression ratio with low quality
	Good portability, software and hardware support	bad quality at lower bitrates
	Encoder and decoder source code available	
<b>AMR</b>	Good quality at lower bitrate	only for lower bitrates
	High compression ratio	
<b>WMA, RealAudio, VQF, MP3PRO</b>	WMA, RealAudio and MP3pro should be used for very low bitrate music only, and high quality should not be expected	We don't consider these formats good enough quality wise, to compete with the above formats, except WMA and MP3pro can compete at 64 kbps and lower bitrates

Microsoft's claims that WMA8–64 kbps is CD-quality, is very far from the truth, no format can achieve even near transparent results at under 128 kbps

## 9. 文件格式

(wav, adpcm, midi, MPEG1 system, mpeg2 system, mp3, AVI, 3GP, VMI, VMD, VMDMovie...)

### 9.1 WAVE File Formats

For audio stream playback, WAVE file is always the most popular file format. The WAVE file format is a subset of Microsoft's RIFF specification for the storage of multimedia files. A RIFF file starts out with a file header followed by a sequence of data chunks. A WAVE file is often just a RIFF file with a single "WAVE" chunk which consists of two sub-chunks -- a "fmt" chunk specifying the data format and a "data" chunk containing the actual sample data. Below is an example: when "AudioFormat" is "wave\_format\_pcm".

### *The Canonical WAVE file format*

endian	File offset (bytes)	field name	Field Size (bytes)	
big	0	ChunkID	4	The "RIFF" chunk descriptor
little	4	ChunkSize	4	
big	8	Format	4	
big	12	Subchunk1ID	4	
little	16	Subchunk1 Size	4	
little	20	AudioFormat	2	
little	22	NumChannels	2	
little	24	SampleRate	4	
little	28	ByteRate	4	
little	32	BlockAlign	2	
little	34	BitsPerSample	2	The "fmt" sub-chunk
big	36	Subchunk2ID	4	
little	40	Subchunk2 Size	4	
little	44	Subchunk2Size		
		data		Indicates the size of the sound information and contains the raw sound data

Figure 0-1 Canonical wave file format

The format (fmt) chunk describes fundamental parameters of the waveform data such as sample rate, bit resolution, and how many channels of digital audio are stored in the wave. RIFF wave files of

AudioFormat “wave\_format\_pcm” need not have the extra chunk or the extended wave format description.

If compression is used (i.e. RIFF wave files of AudioFormat is some value other than “wave\_format\_pcm”), then there will be additional fields appended to the format chunk (thus extended format chunk is produced) which give needed information for a program wishing to retrieve and decompress that stored data. Furthermore, compressed formats must have a fact chunk.

The fact chunk contains chunk id, chunk size which are the same as other chunks, and an unsigned long indicating the size (in sample points) of the waveform after it has been decompressed.

The extended format chunk (except chunk id and chunk size) is described as follows:

```
/* general extended waveform format structure */
/* use this for all non pcm formats */
/* (information common to all formats) */
typedef struct waveform_extended_tag {
    word wformattag;           /* format type */
    word nchannels;            /* number of channels (i.e. mono, stereo...) */
    dword nsamplespersec;      /* sample rate */
    dword navgbytespersec;     /* for buffer estimation */
    word nblockalign;           /* block size of data */
    word wbitpersample;         /* number of bits per sample of mono data */
    word cbsize;                /* the count in bytes of the extra size */
} waveformtex;
```

Table 0-1 Description of Extended Waveform Format

wformattag	Defines the type of wave file.
nchannels	number of channels in the wave, 1 for mono, 2 for stereo
nsamplespersec	Frequency of the sample rate of the wave file. This should be 11025, 22050, or 44100. Other sample rates are allowed, but not encouraged. This rate is also used by the sample size entry in the fact chunk to determine the length in time of the data.
navgbytespersec	Average data rate. Playback software can estimate the buffer size using the <avgbytespersec> value.
nblockalign	The block alignment (in bytes) of the data in <data-ck>. Playback software needs to process a multiple of <nblockalign> bytes of data at a time, so that the value of <nblockalign> can be used for buffer alignment.
wbitpersample	This is the number of bits per sample per channel data. Each channel is assumed to have the same sample resolution. If this field is not needed, then it should be set to zero.
cbsize	the size in bytes of the extra information in the wave format header not including the size of the waveformtex structure. As an example, in the ima adpcm format cbsize is calculated as sizeof(imaadpcmwaveformat) - sizeof(waveformtex) which yields two.

Table 0-2 Defined wformattags

expr1	Wave form registration no. (hex)	expr2
#define wave_format_g723_adpcm	0x0014	/* antex electronics corporation */
#define wave_format_antex_adpcme	0x0033	/* antex electronics corporation */
#define wave_format_g721_adpcm	0x0040	/* antex electronics corporation */
#define wave_format_aptx	0x0025	/* audio processing technology */

#define wave_format_audiofile_af36	0x0024	/* audiofile, inc. */
#define wave_format_audiofile_af10	0x0026	/* audiofile, inc. */
#define wave_format_control_res_vqlpc	0x0034	/* control resources limited */
#define wave_format_control_res_cr10	0x0037	/* control resources limited */
#define wave_format_creative_adpcm	0x0200	/* creative labs, inc */
#define wave_format_dolby_ac2	0x0030	/* dolby laboratories */
#define wave_format_dspgroup_truespeech	0x0022	/* dsp group, inc */
#define wave_format_digistd	0x0015	/* dsp solutions, inc. */
#define wave_format_digifix	0x0016	/* dsp solutions, inc. */
#define wave_format_digireal	0x0035	/* dsp solutions, inc. */
#define wave_format_digiadpcm	0x0036	/* dsp solutions, inc. */
#define wave_format_echos1	0x0023	/* echo speech corporation */
#define wave_format_fm_towns_snd	0x0300	/* fujitsu corp. */
#define wave_format_ibm_cvsd	0x0005	/* ibm corporation */
#define wave_format_oligsm	0x1000	/* ing c. olivetti & c., s.p.a. */
#define wave_format_oliadpcm	0x1001	/* ing c. olivetti & c., s.p.a. */
#define wave_format_olicelp	0x1002	/* ing c. olivetti & c., s.p.a. */
#define wave_format_olisbc	0x1003	/* ing c. olivetti & c., s.p.a. */
#define wave_format_oliopr	0x1004	/* ing c. olivetti & c., s.p.a. */
#define wave_format_ima_adpcm	(wave_format_dvi_adpcm)	/* intel corporation */
#define wave_format_dvi_adpcm	0x0011	/* intel corporation */
#define wave_format_unknown	0x0000	/* microsoft corporation */
#define wave_format_pcm	0x0001	/* microsoft corporation */
#define wave_format_adpcm	0x0002	/* microsoft corporation */
#define wave_format_alaw	0x0006	/* microsoft corporation */
#define wave_format_mulaw	0x0007	/* microsoft corporation */
#define wave_format_gsm610	0x0031	/* microsoft corporation */
#define wave_format_mpeg	0x0050	/* microsoft corporation */

#define wave_format_nms_vbxadpcm	0x0038	/* natural microsystems */
#define wave_format_oki_adpcm	0x0010	/* oki */
#define wave_format_sierra_adpcm	0x0013	/* sierra semiconductor corp */
#define wave_format_sonarc	0x0021	/* speech compression */
#define wave_format_mediaspace_adpcm	0x0012	/* videologic */
#define wave_format_yamaha_adpcm	0x0020	/* yamaha corporation of america */

There are three type of ADPCM format commonly in use, i.e. 4bit IMA-ADPCM, 4bit MS-ADPCM, and 2bit ADPCM.

## 9.2 IMA ADPCM

### 9.2.1 Fact Chunk

This chunk is required for all wave formats other than wave\_format\_pcm. It stores file dependent information about the contents of the wave data. It currently specifies the time length of the data in samples.

### 9.2.2 Wave Format Header

```
# define wave_format_dvi_adpcm (0x0011)
typedef struct dvi_adpcmwaveformat_tag {
    waveformatex wfx;
    word wsamplesperblock;
} dviadpcmwaveformat;
```

Table 0-3 Description of IMA ADPCM Wave Format

wformattag	This must be set to wave_format_dvi_adpcm.	
nchannels	Number of channels in the wave, 1 for mono, 2 for stereo...	
nsamplespersec	Sample rate of the wave file. This should be 8000, 11025, 22050 or 44100. Other sample rates are allowed.	
navgbytespersec	Total average data rate. Playback software can estimate the buffer size for a selected amount of time by using the <avgbytespersec> value.	
nblockalign	This is dependent upon the number of bits per sample.	
	wbitspersample	nblockalign
	3	(( n * 3 ) + 1 ) * 4 * nchannels
	4	(n + 1) * 4 * nchannels
		where n = 0, 1, 2, 3 . . .
	The recommended block size for coding is 256 * <nchannels> bytes* min(1, (/ 11 khz)) Smaller values cause the block header to become a more significant storage overhead. But, it is up to the implementation of the coding portion of the algorithm to decide the optimal value for <nblockalign> within the given constraints (see above). The decoding portion of the algorithm must be able to handle any valid block size. Playback software needs to process a multiple of	

	<nblockalign> bytes of data at a time, so the value of <nblockalign> can be used for allocating buffers.
wbitspersample	This is the number of bits per sample of data. dvi adpcm supports 3 or 4 bits per sample.
cbsize	The size in bytes of the extra information in the extended wave 'fmt' header. This should be 2.
wsamplesperblock	Count of the number of samples per channel per block.

### 9.2.3 Block

The block is defined to be <nblockalign> bytes in length. For dvi adpcm this must be a multiple of 4 bytes since all information in the block is divided on 32 bit word boundaries.

The block has two parts, the header and the data. The two together are <nblockalign> bytes in length. See below take stereo stream as an example:

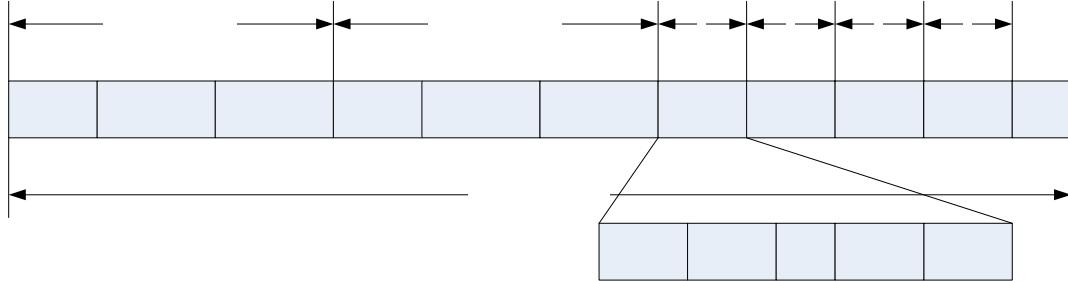


Figure 0-2 IMA ADPCM Block Structure

This is a c structure that defines the dvi adpcm block header.

```
typedef struct dvi_adpcmblockheader_tag {
    short isamp0;
    byte bsteptableindex;
    byte breserved;
} dvi_adpcmblockheader;
```

Table 0-4 Description of IMA ADPCM Header

field	description
isamp0	The first sample value of the block. When decoding, this will be used as the previous sample to start decoding with.
bsteptableindex	The current index into the step table array. (0 - 88)
breserved	L Channel Header This byte is reserved for future use.

R Channel Head

A block contains an array of <nchannels> header structures as defined above.

### 9.2.4 Data

The data words are interpreted differently depending on the number of bits per sample selected. For 4 bit dvi adpcm (where <wbitspersample> is equal to four) each data word contains eight sample codes.

Sample 0 is always included in the block header for the channel.

Each sample is 4 bits in length. Each block contains a total of <wsamplesperblock> samples for each channel.

Bsteptable index (8)

Nblockalig

### 9.2.5 Decoding Algorithm

This section describes the algorithm used for decoding the 4 bit dvi adpcm. This procedure must be followed for each block for each channel.

Each channel of the dvi adpcm file can be encoded/decoded independently. Since the channels are encoded/decoded independently, this document is written as if only one channel is being decoded. Since

the channels are interleaved, multiple channels may be encoded/decoded in parallel using independent local storage and temporaries.

Note that the process for encoding/decoding one block is independent from the process for the next block. Therefore the process is described for one block only, and may be repeated for other blocks.

The dvi adpcm algorithm relies on two tables to encode and decode audio samples. These are the step table and the index table. The contents of these tables are fixed for this algorithm. The 3 and 4 bit versions of the dvi adpcm algorithm use the same step table, which is:

```
const int steptab[ 89 ] = {
    7, 8, 9, 10, 11, 12, 13, 14,
    16, 17, 19, 21, 23, 25, 28, 31,
    34, 37, 41, 45, 50, 55, 60, 66,
    73, 80, 88, 97, 107, 118, 130, 143,
    157, 173, 190, 209, 230, 253, 279, 307,
    337, 371, 408, 449, 494, 544, 598, 658,
    724, 796, 876, 963, 1060, 1166, 1282, 1411,
    1552, 1707, 1878, 2066, 2272, 2499, 2749, 3024,
    3327, 3660, 4026, 4428, 4871, 5358, 5894, 6484,
    7132, 7845, 8630, 9493, 10442, 11487, 12635, 13899,
    15289, 16818, 18500, 20350, 22385, 24623, 27086, 29794,
    32767}
```

But, the index table is different for the different bit rates. For the 4 bit dvi adpcm the contents of index table is:

```
const int indextab[ 16 ] = { -1, -1, -1, -1, 2, 4, 6, 8, -1, -1, -1, -1, 2, 4, 6, 8};
```

The processes for decoding are discussed below.

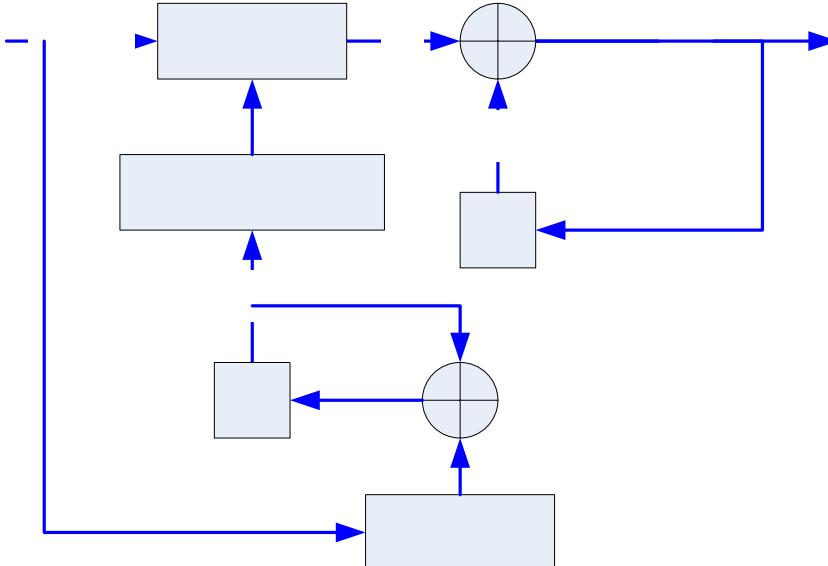


Figure 0-3 IMA ADPCM

get the first sample, samp0, from the block header

set the initial step table index, index, from the block header

output the first sample, samp0

set the previous sample value:

sampx-1 = samp0

while there are still samples to decode

1. get the next sample code, sampx code

2. calculate the new sample:

1) calculate the difference:

diff = 0

if ( sampx code & 4 ) sampx code

diff = diff + steptab[ index ] (4)

if ( sampx code & 2 )

diff = diff + ( steptab[ index ] >> 1 )

if ( sampx code & 1 )

diff = diff + ( steptab[ index ] >> 2 )

diff = diff + ( steptab[ index ] >> 3 )

2) check the sign bit:

if ( sampx code & 8 )

diff = -diff

Calculate  
difference

diff  
(16)

Step table  
89(16)

Sampx-  
(16)

```

3)sampx = sampx-1 + diff
4) check for overflow and underflow errors:
if sampx too large, make it the maximum allowable size (32767)
if sampx too small, make it the minimum allowable size (-32768)
5) output the new sample, sampx
3. adjust the step table index:
index = index + indextab[ sampx code ]
check for step table index overflow and underflow:
if index too large, make it the maximum allowable size (88)
if index too small, make it the minimum allowable size (0)
save the previous sample value:
sampx-1 = sampx

```

## 9.3 MS ADPCM

### 9.3.1 Fact Chunk

This chunk is required for all wave formats other than wave\_format\_pcm. It stores file dependent information about the contents of the wave data. It currently specifies the time length of the data in samples.

### 9.3.2 Wave Format Header

```

#define wave_format_adpcm (0x0002)
typedef struct adpcmcoef_tag {
    int icoef1;
    int icoef2;
} adpcmcoefset;

typedef struct adpcmwaveformat_tag {
    waveformatex wfxx;
    word wsamplesperblock;
    word wnumcoef;
    adpcmcoefset acoeff[wnumcoef];
} adpcmwaveformat;

```

Table 0-5 Description of MS ADPCM Wave Format

wformattag	This must be set to wave_format_adpcm.	
nchannels	Number of channels in the wave, 1 for mono, 2 for stereo.	
nsamplespersec	Frequency of the sample rate of the wave file. This should be 11025, 22050, or 44100. Other sample rates are allowed, but not encouraged.	
navgbytespersec	Average data rate. ((nsamplespersec / nsamplesperblock) * nblockalign). Playback software can estimate the buffer size using the value.	
nblockalign	The block alignment (in bytes) of the data in .	
	nsamplespersec x channels	nblockalign
	8k	256
	11k	256
	22k	512
	44k	1024
	Playback software needs to process a multiple of <nblockalign> bytes of data at a time, so that the value of <nblockalign> can be used for buffer alignment.	

wbitspersample	This is the number of bits per sample of adpcm. Currently only 4 bits per sample is defined. Other values are reserved.		
cbsize	<p>The size in bytes of the extended information after the waveformtex structure.</p> <p>For the standard wave_format_adpcm using the standard seven coefficient pairs, this is 32. If extra coefficients are added, then this value will increase.</p>		
nsamplesperblock	<p>Count of number of samples per block.</p> $((\text{nblockalign} - (7 * \text{nchannels}) * 8) / (\text{wbitspersample} * \text{nchannels})) + 2.$		
nnumcoef	Count of the number of coefficient sets defined in acoef.		
acoef	These are the coefficients used by the wave to play. They may be interpreted as fixed point 8.8 signed values. Currently there are 7 preset coefficient sets. They must appear in the following order.		
	coef set	coef1	coef2
	0	256	0
	1	512	-256
	2	0	0
	3	192	64
	4	240	0
	5	460	-208
	6	392	-232
	Note that if even only 1 coefficient set was used to encode the file then all coefficient sets are still included. More coefficients may be added by the encoding software, but the first 7 must always be the same.		

Note: 8.8 signed values can be divided by 256 to obtain the integer portion of the value.

### 9.3.3 Block

The block has three parts, the header, data, and padding. The three together are <nblockalign> bytes. Below is an example of stereo stream:

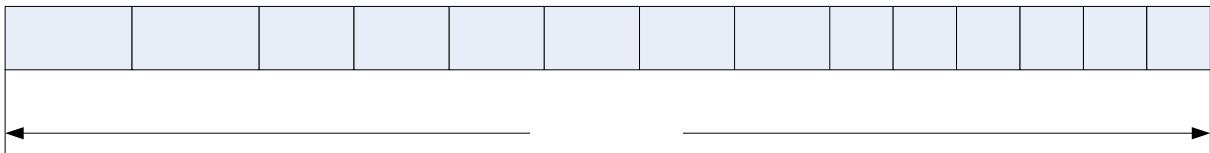


Figure 0-4 MS ADPCM Block Structure

```
typedef struct adpcmblockheader_tag {
    byte bpredictor[nchannels];
    int idelta[nchannels];
    int isamp1[nchannels];
    int isamp2[nchannels];
} adpcmblockheader;
```

Table 0-6 Description of MS ADPCM Header

field	description
bpredictor	Index into the acoef array to define the predictor used to encode this block.
idelta	Initial delta value to use.
isamp1	The second sample value of the block. When decoding this will be used as the previous sample to start decoding with.
isamp2	The first sample value of the block. When decoding this will be used as the previous' previous sample to start decoding with.

### 9.3.4 Data

The data is a bit string parsed in groups of (wbitspersample \* nchannels).

For the case of mono voice adpcm (wbitspersample = 4, nchannels = 1) we have:

$$\begin{aligned} & \langle \text{sample } 2n + 2 \rangle (\text{sample } 2n + 3) \\ &= ((4 \text{ bit error delta for sample } (2 * n) + 2) \ll 4) | (4 \text{ bit error delta for sample } (2 * n) + 3) \end{aligned}$$

For the case of stereo voice adpcm (wbitspersample = 4, nchannels = 2) we have:

$$\begin{aligned} & \langle \text{(left channel of sample } n + 2) \rangle (\text{right channel of sample } n + 2) \\ &= ((4 \text{ bit error delta for left channel of sample } n + 2) \ll 4) | (4 \text{ bit error delta for right channel of sample } n + 2) \end{aligned}$$

### 9.3.5 Padding

Bit padding is used to round off the block to an exact byte length.

The size of the padding (in bits):

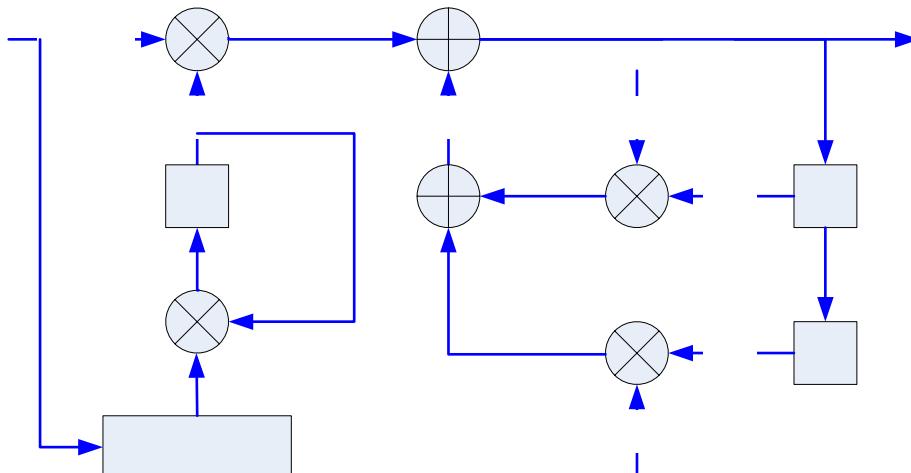
$$((\text{nblockalign} - (7 * \text{nchannels})) * 8) - (((\text{nsamplesperblock} - 2) * \text{nchannels}) * \text{wbitspersample})$$

The padding does not store any data and should be made zero.

### 9.3.6 Decoding Algorithm

Each channel of the adpcm file can be encoded/decoded independently. However this should not destroy phase and amplitude information since each channel will track the original. Since the channels are encoded/decoded independently, this document is written as if only one channel is being decoded. Since the channels are interleaved, multiple channels may be encoded/decoded in parallel using independent local storage and temporaries.

Note that the process for encoding/decoding one block is independent from the process for the next block. Therefore the process is described for one block only, and may be repeated for other blocks. While some optimizations may relate the process for one block to another, in theory they are still independent. Note that in the description below the number designation appended to isamp (i.e. isamp1 and isamp2) refers to the placement of the sample in relation to the current one being decoded. Thus when you are decoding sample n, isamp1 would be sample n - 1 and isamp2 would be sample n - 2. coef1 is the coefficient for isamp1 and coef2 is the coefficient for isamp2. This numbering is identical to that used in the block and format descriptions above.



First the predictor coefficients are determined by using the bpredictor field of block header. This value is an index into the acoef array in the file header.

```
bpredictor = getbyte
```

The initial idelta is also taken from the block header.

```
idelta = getword
```

Then the first two samples are taken from block header. (they are stored as 16 bit pcm data as isamp1 and isamp2. isamp2 is the first sample of the block, isamp1 is the second sample.)

```
isamp1= getint
```

```
isamp2 = getint
```

After taking this initial data from the block header, the process of decoding the rest of the block may begin. it can be done in the following manner:

while there are more samples in the block to decode:

```
predict the next sample from the previous two samples.
```

```
lpredsamp = ((isamp1 * icoef1) + (isamp2 * icoef2)) >> 8
```

```
get the 4 bit signed error delta.
```

```
(ierordelta = getnibble)
```

```
add the 'error in prediction' to the predicted next sample and prevent over/underflow errors.
```

```
(lnewsamp = lpredsamp + (idelta * ierordelta))
```

```
if lnewsamp too large, make it the maximum allowable size.
```

```
if lnewsamp too small, make it the minimum allowable size.
```

```
output the new sample.
```

```
output( lnewsamp )
```

```
adjust the quantization step size used to calculate the 'error in prediction'.
```

```
idelta = (idelta * adaptontable[ ierordelta]) >> 8
```

```
algorithm use the adaptontable, which is:
```

```
int AdaptationTable[] =
```

```
{230, 230, 230, 230, 307, 409, 512, 614,  
768, 614, 512, 409, 307, 230, 230};
```

```
if idelta too small, make it the minimum allowable size.
```

```
update the record of previous samples.
```

```
isamp2 = isamp1;
```

```
isamp1 = lnewsamp.
```

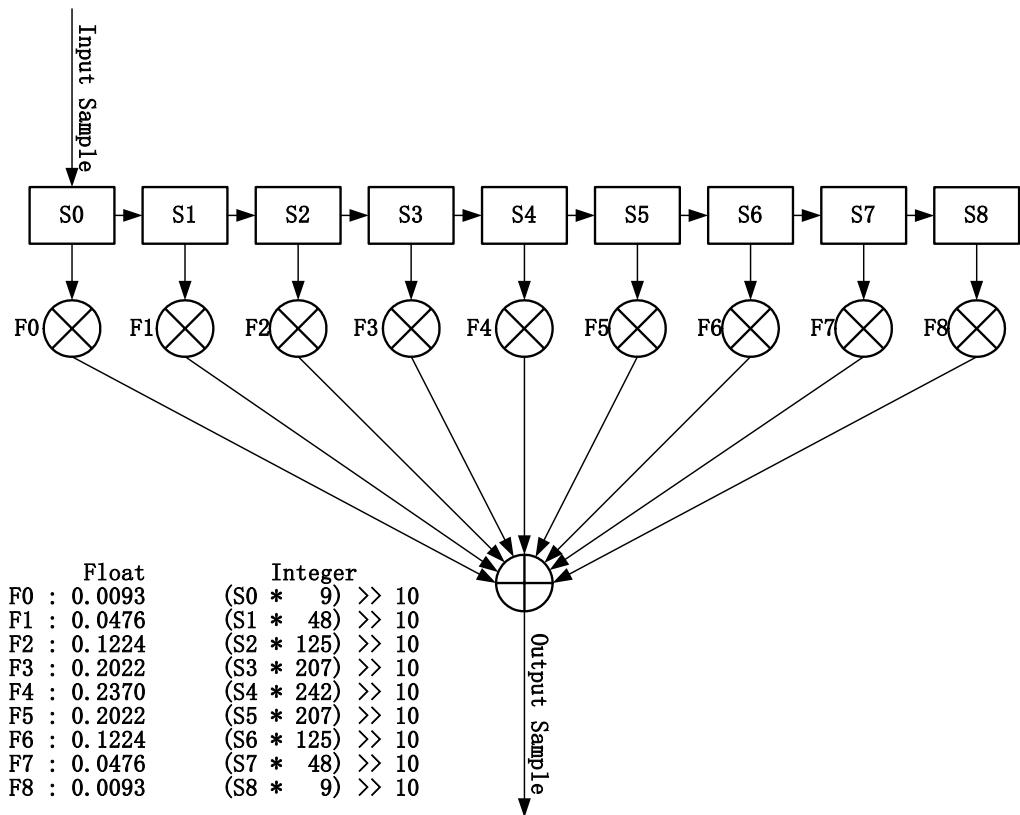
#### 9.4 2Bit IMA ADPCM Algorithm

2 bit ima adpcm compression and decompression algorithm use same step table as 4 bit. Index table have only 2 coefficients: const int indextab = {-1, 1};

Every 16 bit sample is compressed to 2 bit: low bit is delta and high bit is sign bit. Value range is form -1 to 1: b00 == 0;b01 == 1; b10 == 0; b11 == -1;

Every encoded byte(8bit) contain 4 sample(2bit per sample), sample order is form low bit to high bit, lower bits sample is previous to higher bits sample.

After decoding, sample data should be filtered with a FIR Filter. The FIR Filter is defined as follow:



## 9.5 ADPCM DATA Format

In order to simply HW operation, ADPCM data in memory should be placed as following format:

### IMA 4bit stereo

31	24	16	8	0
R Breserved	L Breserved	R Bsteptable index	L Bsteptable index	
R Isamp0				L Isamp0
R	L	R	L	R
• • •    • • •				
R	L	R	L	R
R	L	R	L	R

### IMA 2bit mono

31	24	16	8	0
Breserved				Bsteptable index
Isamp0 (Unfiltered)				Isamp0 (Filtered)
SMP 7	SMP 6	SMP 5	SMP 4	SMP 3
SMP 2	SMP 1	SMP 0	• • •    • • •	
SMP n	SMP	SMP	SMP	SMP
SMP	SMP	SMP	SMP	SMP

### MS 4bit stereo

31	24	16	8	0
Reserved		L Idelta		
Reserved		R Idelta		
L Isamp2				R Isamp2
L Isamp1				R Isamp1
L	R	L	R	L
R	L	R	L	R
• • •    • • •				
L	R	L	R	L
R	L	R	L	R

## 9.6 MPEG Audio

### 9.6.1 MPEG Audio Format

An MPEG audio file consists out of frames. Each frame contains a header at its beginning, after that follows the data. This audio data always contains a **fixed number of samples**. There currently exists three layers of MPEG audio, which differ in how the audio data is encoded in the frame (although they all have the same header format).

Additionally, there are also three different versions of MPEG audio, which only differ in the sampling rate they can handle (see table 3.2). MPEG 1(ISO/IEC 13818-3) and MPEG 2(ISO/IEC 11172-3) are ISO standards. MPEG 2.5 is an extension of MPEG 2 to support even lower sampling rates. MPEG 2/2.5 are also known under the abbreviation **LSF**, which stands for Lower Sampling Frequencies.

A file can be encoded either with constant bitrate (**CBR**) or with variable bitrate (**VBR**), which means that each frame can have a different bitrate. Therefore, the quality of those files is often higher than files encoded in constant bitrate mode, because it can use higher bitrates where the music needs it.

### 9.6.2 MPEG Audio Frame Header

The header at the beginning of each frame is 32 bits long and has the following format. (Note that the position is zero-based; position, length and example are each in bit-format)

Position	Length	Meaning	Example
0	11	Frame sync to find the header (all bits are always set)	1111 1111 111
11	2	Audio version ID (see table 3.2 also)  00 - MPEG Version 2.5 (later extension of MPEG 2) 01 - reserved 10 - MPEG Version 2 (ISO/IEC 13818-3) 11 - MPEG Version 1 (ISO/IEC 11172-3)	11
13	2	Layer index  00 - reserved 01 - Layer III 10 - Layer II 11 - Layer I	01
15	1	Protection bit  0 - protected by 16bit CRC following header 1 - no CRC	1
16	4	Bitrate index (see table 3.3)	1001
20	2	Sampling rate index (see table 3.2)	11
22	1	Padding bit  If it is set, data is padded with 4 bytes (Layer1) or 1 byte (Layer2/3), which is important for frame size calculation.	0
23	1	Private bit (only informative)	1
24	2	Channel mode  00 - Stereo 01 - Joint Stereo (Stereo) 10 - Dual channel (2 Mono channels) 11 - Single channel (Mono)  Note: Dual channel files are made of two independent mono channels. Each one uses exactly half the bitrate of the file. Most decoders output them as stereo, but it might not always be the case.	01
26	2	Mode extension (Only used in Joint Stereo)  (see table 3.5)	00
28	1	Copyright bit (only informative)	1

29	1	Original bit (only informative)	1
		Emphasis	
30	2	00 - none 01 - 50/15 ms 10 - reserved 11 - CCIT J.17  The emphasis indication is here to tell the decoder that the file must be de-emphasized, that means the decoder must 're-equalize' the sound after a Dolby-like noise suppression. It is rarely used.	00

3.1 MPEG Audio Frame Header

The sampling rates specify how many samples per second are stored in this file. Each MPEG version can handle different sampling rates.

Sampling Rate Index	MPEG 1	MPEG 2 (LSF)	MPEG 2.5 (unofficial extension)
00	44100 Hz	22050 Hz	11025 Hz
01	48000 Hz	24000 Hz	12000 Hz
10	32100 Hz	16000 Hz	8000 Hz
11	reserved		

3.2 MPEG Versions and Sampling Rates

The bitrates are always displayed in kilobits per second. Note that the prefix kilo (abbreviated with the small 'k') doesn't mean 1024 but 1000 bits per second! The bitrate index 1111 is reserved and should never be used. In the MPEG audio standard, there is a free format described. This free format means that the file is encoded with a constant bitrate, which is not one of the predefined bitrates. Only very few decoders can handle those files.

Bitrate Index	MPEG 1			MPEG 2, 2.5	
	Layer I	Layer II	Layer III	Layer I	Layer II & III
0000	free				
0001	32	32	32	32	8
0010	64	48	40	48	16
0011	96	56	48	56	24
0100	128	64	56	64	32
0101	160	80	64	80	40
0110	192	96	80	96	48
0111	224	112	96	112	56
1000	256	128	112	128	64
1001	288	160	128	144	80
1010	320	192	160	160	96
1011	352	224	192	176	112
1100	384	256	224	192	128
1101	416	320	256	224	144
1110	448	384	320	256	160
1111	reserved				

3.3 Bitrates (in kilobits per second)

For calculation of the frame size, you sometimes need the number of samples per MPEG audio frame. Therefore, you can use the following table:

	MPEG 1	MPEG 2 (LSF)	MPEG 2.5 (unofficial extension)

<b>Layer I</b>	384	384	384
<b>Layer II</b>	1152	1152	1152
<b>Layer III</b>	1152	576	576

### 3.4 Samples Per Frame

Then you can calculate the frame size like this:

$$\text{Frame Size} = (\text{Samples Per Frame} / 8 * \text{Bitrate}) / \text{Sampling Rate} + \text{Padding Size}$$

Note that the bitrate must be in bits (not kilobits) per second for this formula.

You get the length of the file in seconds by the following formula:

$$\text{Length} = \text{File Size} / \text{Bitrate} * 8$$

The method of getting the first frame header in the file and then calculating the length by the above formula works only for CBR files correctly.

The mode extension is used to join information that are of no use for stereo effect, thus reducing needed bits. These bits are dynamically determined by an encoder in Joint Stereo mode, and Joint Stereo can be changed from one frame to another, or even switched on or off.

Complete frequency range of MPEG file is divided in subbands. There are 32 subbands. For Layers I & II, these two bits determine frequency range (bands) where intensity stereo is applied. For Layer III, these two bits determine which type of joint stereo is used (intensity stereo or m/s stereo). Frequency range is determined within decompression algorithm.

Value	Layer I&II	Layer III	
		M/S stereo	Intensity stereo
00	bands 4 to 31	off	off
01	bands 8 to 31	off	on
10	bands 12 to 31	on	off
11	bands 16 to 31	on	on

### 3.5 Mode Extension

#### 9.6.3 VBR Headers

Some files are encoded with variable bitrate mode (VBR). To estimate the length of those files, you have to know the **average bitrate** of the whole file. It often differs a lot from the bitrate of the first frame, because the lowest bitrate available is used for silence in music titles (especially at the beginning). To get this average bitrate, you must go through all the frames in the file and calculate it, by summarizing the bitrates of each frame and dividing it through the number of frames. Because this isn't a good practice (very slow), there exists additional VBR headers within the data section of the first frame (after the frame header). They contain the total number of frames in the file from which you can calculate the length in seconds with the following formula:

$$\text{Length} = \text{Number of Frames} * \text{Samples Per Frame} / \text{Sampling Rate}$$

#### 9.6.4 XING Header

This header is often (but unfortunately not always) added to files which are encoded with variable bitrate mode. This header stands after the first MPEG audio header at a specific position.

	<b>MPEG 1</b>	<b>MPEG 2/2.5 (LSF)</b>
--	---------------	-------------------------

Stereo, Joint Stereo, Dual Channel	32	17
Mono	17	9
4.1.1 XING Header Offset (in bytes)		

This offset is relative to the end of the first MPEG audio header. So for reading out this header, you first have to find the first MPEG audio header and then go to this specific position from the end of the header. The XING header itself has the following format. (Note that the position is zero-based; position, length and example are each in byte-format)

Position	Length	Meaning	Example
0	4	VBR header ID in 4 ASCII chars, either 'Xing' or 'Info', not NULL-terminated	'Xing'
4	4	Flags which indicate what fields are valid, flags are combined with a logical OR. Field is mandatory.	0x0007 (means Frames, Bytes & TOC valid)
8	4	Number of Frames as Big-Endian DWORD	7344
8 or 12	4	Number of Bytes in file as Big-Endian DWORD	45000
8, 12 or 16	100	100 TOC entries for seeking as integral BYTE	
8, 12, 16, 108, 112 or 116	4	Quality indicator as Big-Endian DWORD from 0 - best quality to 100 - worst quality	0

4.1.2 XING Header

Sometimes, this header is also added to CBR files. It then often has the ID 'Info'.

#### 9.6.5 VBRI Header

This header is only used by MPEG audio files encoded with the Fraunhofer Encoder as far as I know. It is different from the XING header. You find it exactly **32** bytes after the end of the first MPEG audio header in the file. (Note that the position is zero-based; position, length and example are each in byte-format.)

Position	Length	Meaning	Example
0	4	VBR header ID in 4 ASCII chars, always 'VBRI', not NULL-terminated,	'VBRI'
4	2	Version ID as Big-Endian WORD	1
6	2	Delay as Big-Endian float	7344
8	2	Quality indicator	75
10	4	Number of <b>Bytes</b> as Big-Endian DWORD	45000
14	4	Number of <b>Frames</b> as Big-Endian DWORD	7344
18	2	Number of entries within TOC table as Big-Endian WORD	100
20	2	Scale factor of TOC table entries as Big-Endian DWORD	1
22	2	Size per table entry in bytes (max 4) as Big-Endian WORD	2
24	2	Frames per table entry as Big-Endian WORD	845
26		<b>TOC</b> entries for seeking as Big-Endian integral. From size per table entry and number of entries, you can calculate the length of this field.	

4.2.1 VBRI Header

#### 9.6.6 Additional Tags

Please consider that at the end of the file might be the following data: ID3V1 Tags, Lyrics Tags, Musicmatch Tag. At the beginning of the file might be an ID3V2 Tag. You must consider this, because only the MP3 data count for the length estimation.

#### 9.6.6.1 ID3V1

ID3V1 比较简单，它是存放在 MP3 文件的末尾，用 16 进制的编辑器打开一个 MP3 文件，查看其末尾的 128 个顺序存放字节，数据结构定义如下：

```
typedef struct tagID3V1
{
    char Header[3]; /*标签头必须是"TAG"否则认为没有标签*/
    char Title[30]; /*标题*/
    char Artist[30]; /*作者*/
    char Album[30]; /*专集*/
    char Year[4]; /*出品年代*/
    char Comment[28]; /*备注*/
    char reserve; /*保留*/
    char track;; /*音轨*/
    char Genre; /*类型*/
}ID3V1,*pID3V1;
```

ID3V1 的各项信息都是顺序存放，没有任何标识将其分开，比如标题信息不足 30 个字节，则使用'\0'补足，否则将造成信息错误。Genre 使用原码表示，对照表如下：

```
/* Standard genres */
0="Blues";
1="ClassicRock";
2="Country";
3="Dance";
4="Disco";
5="Funk";
6="Grunge";
7="Hip-Hop";
8="Jazz";
9="Metal";
10="NewAge";
11="Oldies";
12="Other";
13="Pop";
14="R&B";
15="Rap";
16="Reggae";
17="Rock";
18="Techno";
19="Industrial";
20="Alternative";
21="Ska";
22="DeathMetal";
23="Pranks";
```

24="Soundtrack";  
25="Euro-Techno";  
26="Ambient";  
27="Trip-Hop";  
28="Vocal";  
29="Jazz+Funk";  
30="Fusion";  
31="Trance";  
32="Classical";  
33="Instrumental";  
34="Acid";  
35="House";  
36="Game";  
37="SoundClip";  
38="Gospel";  
39="Noise";  
40="AlternRock";  
41="Bass";  
42="Soul";  
43="Punk";  
44="Space";  
45="Meditative";  
46="InstrumentalPop";  
47="InstrumentalRock";  
48="Ethnic";  
49="Gothic";  
50="Darkwave";  
51="Techno-Industrial";  
52="Electronic";  
53="Pop-Folk";  
54="Eurodance";  
55="Dream";  
56="SouthernRock";  
57="Comedy";  
58="Cult";  
59="Gangsta";  
60="Top40";  
61="ChristianRap";  
62="Pop/Funk";  
63="Jungle";  
64="NativeAmerican";  
65="Cabaret";  
66="NewWave";  
67="Psychadelic";

```
68="Rave";
69="Showtunes";
70="Trailer";
71="Lo-Fi";
72="Tribal";
73="AcidPunk";
74="AcidJazz";
75="Polka";
76="Retro";
77="Musical";
78="Rock&Roll";
79="HardRock";
/* Extended genres */
80="Folk";
81="Folk-Rock";
82="NationalFolk";
83="Swing";
84="FastFusion";
85="Bebob";
86="Latin";
87="Revival";
88="Celtic";
89="Bluegrass";
90="Avantgarde";
91="GothicRock";
92="ProgesiveRock";
93="PsychedelicRock";
94="SymphonicRock";
95="SlowRock";
96="BigBand";
97="Chorus";
98="EasyListening";
99="Acoustic";
100="Humour";
101="Speech";
102="Chanson";
103="Opera";
104="ChamberMusic";
105="Sonata";
106="Symphony";
107="BootyBass";
108="Primus";
109="PornGroove";
110="Satire";
```

```
111="SlowJam";
112="Club";
113="Tango";
114="Samba";
115="Folklore";
116="Ballad";
117="PowerBallad";
118="RhythmicSoul";
119="Freestyle";
120="Duet";
121="PunkRock";
122="DrumSolo";
123="Acapella";
124="Euro-House";
125="DanceHall";
126="Goa";
127="Drum&Bass";
128="Club-House";
129="Hardcore";
130="Terror";
131="Indie";
132="BritPop";
133="Negerpunk";
134="PolskPunk";
135="Beat";
136="ChristianGangstaRap";
137="HeavyMetal";
138="BlackMetal";
139="Crossover";
140="ContemporaryChristian";
141="ChristianRock";
142="Merengue";
143="Salsa";
144="TrashMetal";
145="Anime";
146="JPop";
147="Synthpop";
```

#### 9. 6. 6. 2 ID3V2

ID3V2 到现在一共有 4 个版本，但流行的播放软件一般只支持第 3 版，既 ID3v2.3。由于 ID3V1 记录在 MP3 文件的末尾，ID3V2 就只好记录在 MP3 文件的首部了(如果有一天发布 ID3V3，真不知道该记录在哪里)。也正是由于这个原因，对 ID3V2 的操作比 ID3V1 要慢。

而且 ID3V2 结构比 ID3V1 的结构要复杂得多，但比前者全面且可以伸缩和扩展。

下面就介绍一下 ID3V2.3。

每个 ID3V2.3 的标签都一个标签头和若干个标签帧或一个扩展标签头组成。关于曲目的信息如标题、作者等都存放在不同的标签帧中，扩展标签头和标签帧并不是必要的，但每个标签至少要有一个标签帧。标签头和标签帧一起顺序存放在 MP3 文件的首部。

## 1、标签头

在文件的首部顺序记录 10 个字节的 ID3V2.3 的头部。数据结构如下：

```
char Header[3]; /*必须为"ID3"否则认为标签不存在*/  
char Ver; /*版本号 ID3V2.3 就记录 3*/  
char Revision; /*副版本号此版本记录为 0*/  
char Flag; /*存放标志的字节，这个版本只定义了三位，稍后详细解说*/  
char Size[4]; /*标签大小，包括除了标签头的 10 个字节外的所有标签帧的大小*/
```

### 1) .标志字节

标志字节一般为 0， 定义如下：

abc00000

a -- 表示是否使用 **Unsynchronisation**(这个单词不知道是什么意思，字典里也没有找到，一般不设置)

b -- 表示是否有扩展头部，一般没有(至少 Winamp 没有记录)，所以一般也不设置

c -- 表示是否为测试标签(99.99%的标签都不是测试用的啦，所以一般也不设置)

### 2) .标签大小

一共四个字节，但每个字节只用 7 位，最高位不使用恒为 0。所以格式如下

0xxxxxxxxx 0xxxxxxxxx 0xxxxxxxxx 0xxxxxxxxx

计算大小时要将 0 去掉，得到一个 28 位的二进制数，就是标签大小(不懂为什么要这样做)，计算公式如下：

```
int total_size;  
total_size = (Size[0]&0x7F)*0x200000  
+(Size[1]&0x7F)*0x400  
+(Size[2]&0x7F)*0x80  
+(Size[3]&0x7F)
```

## 2、标签帧

每个标签帧都有一个 10 个字节的帧头和至少一个字节的不固定长度的内容组成。它们也是顺序存放在文件中，和标签头和其他的标签帧也没有特殊的字符分隔。得到一个完整的帧的内容只有从帧头中的到内容大小后才能读出，读取时要注意大小，不要将其他帧的内容或帧头读入。

帧头的定义如下：

```
char FrameID[4]; /*用四个字符标识一个帧，说明其内容，稍后有常用的标识对照表*/  
char Size[4]; /*帧内容的大小，不包括帧头，不得小于 1，可变*/
```

```
char Flags[2]; /*存放标志，只定义了 6 位，稍后详细解说*/
```

### 1) .帧标识

用四个字符标识一个帧，说明一个帧的内容含义，常用的对照如下：

TIT2=标题 表示内容为这首歌的标题，下同

TPE1=作者

TALB=专集

TRCK=音轨 格式：N/M 其中 N 为专集中的第 N 首，M 为专集中共 M 首，N 和 M 为 ASCII 码表示的数字

TYER=年代 是用 ASCII 码表示的数字

TCON=类型 直接用字符串表示

COMM=备注 格式："eng\0 备注内容"，其中 eng 表示备注所使用的自然语言

### 2) .大小

这个可没有标签头的算法那么麻烦，每个字节的 8 位全用，格式如下

xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx

算法如下：

```
int FSize;  
FSize = Size[0]*0x100000000  
+Size[1]*0x10000  
+Size[2]*0x100  
+Size[3];
```

### 3) .标志

只定义了 6 位，另外的 10 位为 0，但大部分的情况下 16 位都为 0 就可以了。格式如下：

abc00000 ijk00000

a -- 标签保护标志，设置时认为此帧作废

b -- 文件保护标志，设置时认为此帧作废

c -- 只读标志，设置时认为此帧不能修改(但我没有找到一个软件理会这个标志)

i -- 压缩标志，设置时一个字节存放两个 BCD 码表示数字

j -- 加密标志(没有见过哪个 MP3 文件的标签用了加密)

k -- 组标志，设置时说明此帧和其他的某帧是一组

值得一提的是 winamp 在保存和读取帧内容的时候会在内容前面加个'\0'，并把这个字节计算在帧内容的大小中。

详细的情况可以到 <http://www.id3.org/> 查询，对于 ID3V1 和 ID3V2 的读写，我用 DELPHI 写了两个类来实现，可以写信给我索取 [q.d.zhang@sohu.com](mailto:q.d.zhang@sohu.com)

附：帧标识的含义

### 4) . Declared ID3v2 frames

The following frames are declared in this draft.

AENC Audio encryption  
APIC Attached picture

COMM Comments  
COMR Commercial frame

ENCR Encryption method registration  
EQUA Equalization  
ETCO Event timing codes

GEOB General encapsulated object  
GRID Group identification registration

IPLS Involved people list

LINK Linked information

MCDI Music CD identifier  
MLLT MPEG location lookup table

OWNE Ownership frame

PRIV Private frame  
PCNT Play counter  
POPM Popularimeter  
POSS Position synchronisation frame

RBUF Recommended buffer size  
RVAD Relative volume adjustment  
RVRB Reverb

SYLT Synchronized lyric/text  
SYTC Synchronized tempo codes

TALB Album/Movie>Show title  
TBPM BPM (beats per minute)  
TCOM Composer  
TCON Content type  
TCOP Copyright message  
TDAT Date  
TDLY Playlist delay  
TENC Encoded by  
TEXT Lyricist/Text writer  
TFLT File type  
TIME Time

TIT1 Content group description  
TIT2 Title/songname/content description  
TIT3 Subtitle/Description refinement  
TKEY Initial key  
TLAN Language(s)  
TLEN Length  
TMED Media type  
TOAL Original album/movie/show title  
TOFN Original filename  
TOLY Original lyricist(s)/text writer(s)  
TOPE Original artist(s)/performer(s)  
TORY Original release year  
TOWN File owner/licensee  
TPE1 Lead performer(s)/Soloist(s)  
TPE2 Band/orchestra/accompaniment  
TPE3 Conductor/performer refinement  
TPE4 Interpreted, remixed, or otherwise modified by  
TPOS Part of a set  
TPUB Publisher  
TRCK Track number/Position in set  
TRDA Recording dates  
TRSN Internet radio station name  
TRSO Internet radio station owner  
TSIZ Size  
TSRC ISRC (international standard recording code)  
TSSE Software/Hardware and settings used for encoding  
TYER Year  
TXXX User defined text information frame

UFID Unique file identifier  
USER Terms of use  
USLT Unsynchronized lyric/text transcription

WCOM Commercial information  
WCOP Copyright/Legal information  
WOAF Official audio file webpage  
WOAR Official artist/performer webpage  
WOAS Official audio source webpage  
WORS Official internet radio station homepage  
WPAY Payment  
WPUB Publishers official webpage  
WXXX User defined URL link frame

## 9.7 MIDI file format

### 1.概述:

一个MIDI文件基本上由两个部分组成，头块和轨道块。第二节讲述头块，第三节讲述轨道块。一个MIDI文件有一个头块用来描述文件的格式、许多的轨道块等内容。一个轨道可以想象为像一个大型多音轨录音机那样，你可以为某种声音、某种乐谱、某种乐器或者你需要的任何东西分配一个轨道。

### 2.头块:

头块出现在文件的开头，有三种方式来描述文件。头块看起来一直是这样的：

4D 54 68 64 00 00 00 06 ff ff nn nn dd dd

前4个字节等同于ASCII码MThd，接着MThd之后的4个字节是头的大小。它将一直是00 00 00 00 06，因为现行的头信息将一直是6字节。

ff ff 是文件的格式，有3种格式：

0—单轨

1—多规，同步

2—多规，异步

单轨，很显然就只有一个轨道。同步多轨意味着所有轨道都是垂直同步的，或者其他措辞为他们都在同一时间开始，并且可以表现一首歌的不同部分。异步多轨没有必要同时开始，而且可以完全的不同步。

nn nn 是MIDI文件中的轨道数。

dd dd 是每个4分音符delta-time节奏数（这之后将做详细介绍）。

### 3.轨道块:

头块之后剩下的文件部分是轨道块。每一个轨道包含一个头，并且可以包含你所希望的许多MIDI命令。轨道头与文件头及其相似：

4D 54 72 6B xx xx xx xx

与头一致，前4个字节是ASCII吗，这个是MTrk，紧跟MTrk的4个字节给出了以字节为单位的轨道的长度（不包括轨道头）。

在头之下是MIDI事件，这些事件同现行的可以被带有累加的MIDI合成器端口接受和发送的数据是相同的。一个MIDI事件先于一个delta-time。一个delta-time是一个MIDI事件被执行后的节奏数，每个四分之一音符的节奏数先前已经定义在了文件的头块中。这个delta-time是一个可变长度的编码值。这种格式虽然混乱，可是允许根据需要利用多位表示较大的数值，这不会因为需求小的数值情况下以添零的方式浪费掉一些字节！数值被转换为7位的字节，并且除了最后一个字节以最高有效位是0外，各个字节最有意义的一位是1。这就允许一个数值被一次一个字节地读取，你如果发现

最高有效位是 0，则这就是这个数值的最后一位(意义比较小)。依照 MIDI 说明，全部 delta-time 的长度最多超过 4 字节。

delta-time 之后就是 MIDI 事件，每个 MIDI 事件（除了正在运行的事件外）带有一个最高有效位总是 1 的命令字节（值将>128）。大部分命令的列表在附录 A 中。每个命令都有不同的参数和长度，但是接下来的数据将是最高有效位为零（值将<128）。这里有个例外就是 meta-event，最高有效位可以是 1。然而，meta-events 需要一个长的参数以区分。

微小失误就可以导致混乱的是运行模式，这是现行 MIDI 命令所忽略的地方，并且最终发行的 MIDI 命令是假定的。这就意味这如果包含了命令，那么 MIDI 事件就是由 delta-time 与参数组成而转换的。

## 附录 A

### 1.MIDI 事件命令

每个命令字节有两部分，左 nybble(4 位) 包含现行的命令，右 nybble 包含将被执行的命令的通道号，这里有 16 各 MIDI 通道 8 个 MIDI 命令（命令 nybble 必须最高有效位是 1 的）。在下表中，X 表示 MIDI 通道号。所有的音符即数据字节都<128（最高有效位是 0）。

#### 十六进制 二进制 数据 描述

8x	1000xxxx	nn vv	音符关闭 (释放键盘)
		nn=音符号	
		vv=速度	
9x	1001xxxx	nn vv	音符打开 (按下键盘)
		nn=音符号	
		vv=速度	
Ax	1010xxxx	nn vv	触摸键盘以后
		nn=音符号	
		vv=速度	
Bx	1011xxxx	cc vv	调换控制
		cc=控制号	
		vv=新值	
Cx	1100xxxx	pp	改变程序 (片断)
		pp=新的程序号	
Dx	1101xxxx	cc	在通道后接触
		cc=管道号	
Ex	1110xxxx	bb tt	改变互相咬合的齿轮 (2000H 表明缺省或没有改变)(什么意思搞不懂:)
		bb=值的低 7 位(least sig)	
		tt=值的高 7 位 (most sig)	

下表是没有通道的 meta-events 列表，他们的格式是：

FF xx nn dd

所有的 meta-events 是以 FF 开头的命令 (xx), 长度, 或者含在数据的字节数 (nn), 现行的数据 (dd)

#### 十六进制 二进制 数据 描述

00	00000000	nn ssss	设定轨道的序号 nn=02 (两字节长度的序号) ssss=序号
01	00000001	nn tt ..	你需要的所有文本事件 nn=以字节为单位的文本长度 tt=文本字符
02	00000010	nn tt ..	同文本的事件, 但是用于版权信息 nn tt=同文本事件
03	00000011	nn tt ..	序列或者轨道名 nn tt=同文本事件
04	00000100	nn tt ..	轨道乐器名 nn tt=同文本事件
05	00000101	nn tt ..	歌词 nn tt=同文本事件
06	00000110	nn tt ..	标签 nn tt=同文本事件
07	00000111	nn tt ..	浮点音符 nn tt=同文本事件
2F	00101111	00	这个事件一定在每个轨道的结尾出现
51	01010001	03 tttttt	设定拍子 tttttt=微秒/四分音符
58	01011000	04 nn dd cc bb	拍子记号 nn=拍子记号分子 dd=拍子记号分母 2=四分之一 3=八分拍, 等等. cc=节拍器的节奏 bb=对四分之一音符标注的第 32 号数字
59	01011001	02 sf mi	音调符号 sf=升调/降调(-7=降调, 0=基准 C 调, 7=升调) mi=大调/小调(0=大调, 1=小调)
7F	01111111	xx dd ..	音序器的详细信息 xx=被发送的字节数 dd=数据

下表列出了控制整个系统的系统消息。这里没有 MIDI 通道数 (这些一般仅应用于 MIDI 键盘等。)

十六进制	二进制	数据	描述
------	-----	----	----

F8	11111000		同步所必须的计时器
FA	11111010		开始当前的队列
FB	11111011		从停止的地方继续一个队列

FC 11111100 停止一个队列

下表列出的是与音符相对应的命令标记。

八度音阶	音符号
#	##
C	C#
D	D#
E	F
F#	G
G#	A
A#	B
-----	-----
0    0   1   2   3   4   5   6   7   8   9   10   11	
1    12   13   14   15   16   17   18   19   20   21   22   23	
2    24   25   26   27   28   29   30   31   32   33   34   35	
3    36   37   38   39   40   41   42   43   44   45   46   47	
4    48   49   50   51   52   53   54   55   56   57   58   59	
5    60   61   62   63   64   65   66   67   68   69   70   71	
6    72   73   74   75   76   77   78   79   80   81   82   83	
7    84   85   86   87   88   89   90   91   92   93   94   95	
8    96   97   98   99   100   101   102   103   104   105   106   107	
9    108   109   110   111   112   113   114   115   116   117   118   119	
10    120   121   122   123   124   125   126   127	

## 9.8 MPEG file format

### 9.8.1 MPEG-1 system bitstream

The system bitstream multiplexes the audio and video bitstreams into a single bitstream, and formats it with control information into a specific protocol as defined by MPEG 1. Packet data may contain either audio or video information. Up to 32 audio and 16 video streams may be multiplexed together. Two types of private data streams are also supported. One type is completely private; the other is used to support synchronization and buffer management.

Maximum packet sizes usually are about 2,048 bytes, although much larger sizes are supported. When stored on CDROM, the length of the packs coincides with the sectors. Typically, there is one audio packet for every six or seven video packets.

Figure 12.6 illustrates the system bitstream, a hierarchical structure with three layers.

From top to bottom the layers are:

ISO/IEC 11172 Layer

Pack

Packet

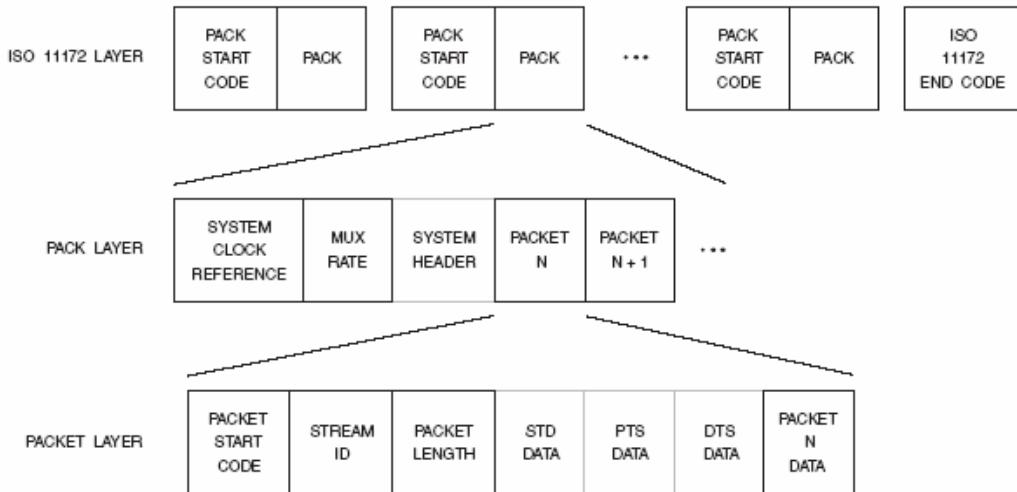


Figure 12.6. MPEG 1 System Bitstream Layer Structures. Marker and reserved bits not shown.

ISO_11172_end_code	0x000001B9
Pack_start_code	0x000001BA
System_header_start_code	0x000001BB
Reserved_stream	0x000001BC
Private_stream_1	0x000001BD
Padding_stream	0x000001BE
Private_stream_2	0x000001BF
Audio_stream	0x000001C0~0x000001DF
Video_stream	0x000001E0~0x000001EF
Reserved_data_stream	0x000001F0~0x000001FF

### 9.8.2 MPEG-1 video bitstream

Figure 12.5 illustrates the video bitstream, a hierarchical structure with seven layers. From top to bottom the layers are:

- Video Sequence
- Sequence Header
- Group of Pictures (GOP)
- Picture
- Slice
- Macroblock (MB)
- Block

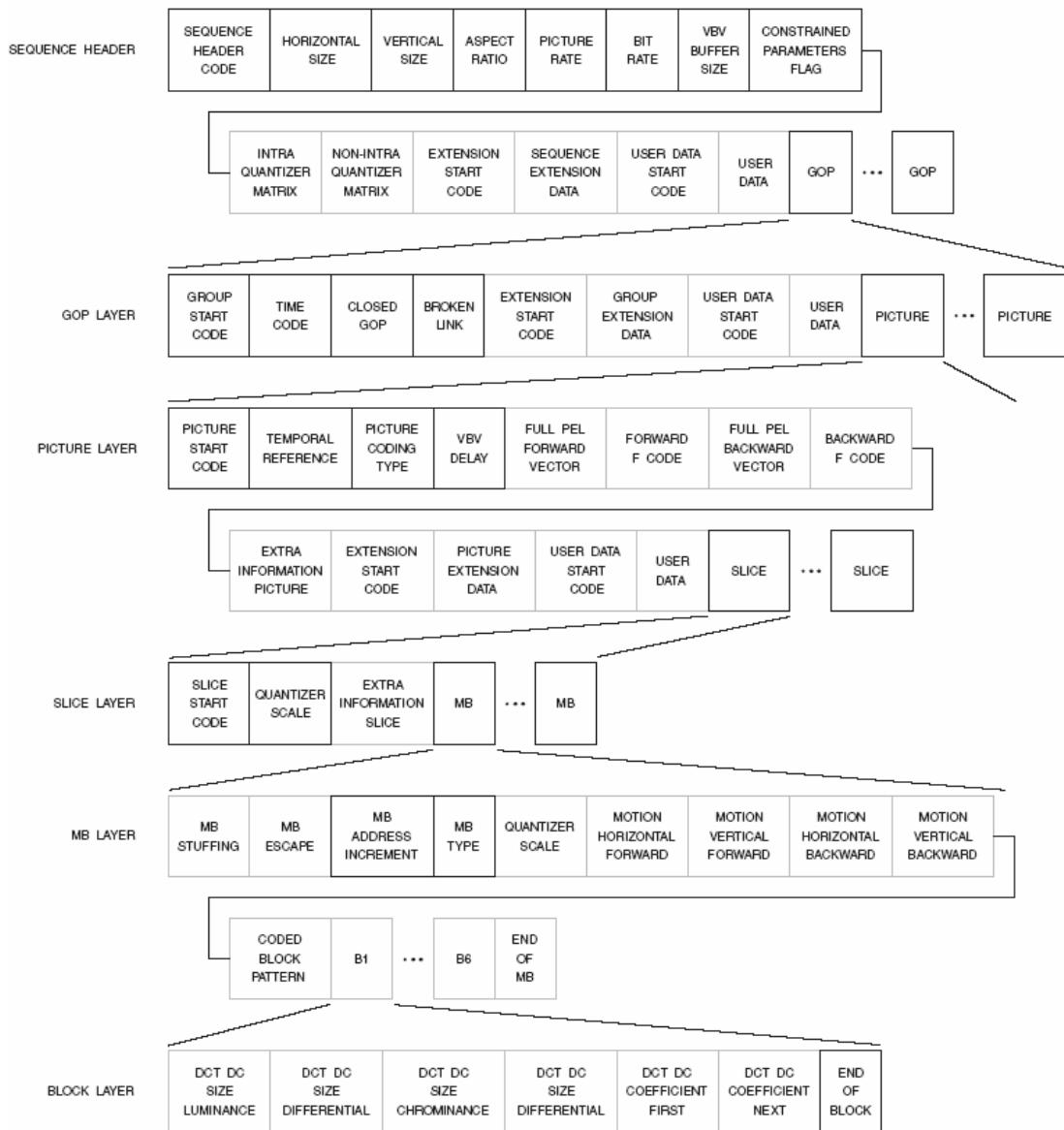
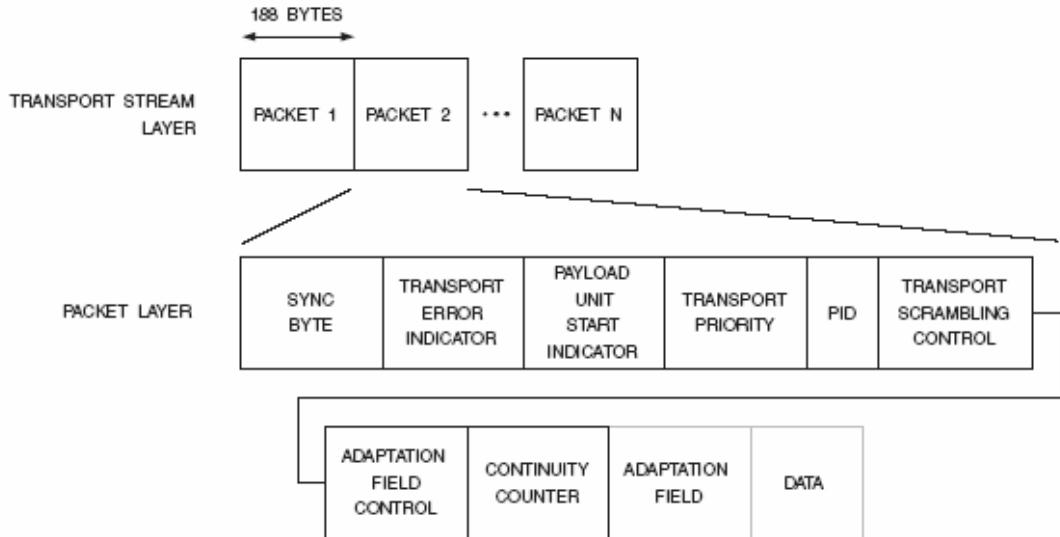


Figure 12.5. MPEG 1 Video Bitstream Layer Structures. Marker and reserved bits not shown.

Picture_start_code	0x00000100
Slice_start_code	0x00000101~0x000001AF
Reserved	0x000001B0
Reserved	0x000001B1
User_data_start_code	0x000001B2
Sequence_header_code	0x000001B3
Sequence_error_code	0x000001B4
Extension_start_code	0x000001B5
Reserved	0x000001B6
Sequence_end_code	0x000001B7
Group_start_code	0x000001B8

### 9.8.3 MPEG-2 transport stream

The transport stream consists of one or more 188-byte packets. The data for each packet is from PES packets, PSI (Program Specific Information) sections, stuffing bytes, or private data. The general format of the transport stream is shown in Figure 13.22.



#### Sync\_byte

This 8-bit string has a value of “0100 0111.”

### 9.8.4 MPEG-2 program stream

The program stream consists of one or more audio and/or video streams multiplexed together. Data from each audio and video stream is multiplexed and coded with data that allows them to be decoded in synchronization.

Data from audio and video streams is stored in PES (Packetized Elementary Stream) packets. PES packets are then organized in packs. The general format of the program stream is shown in Figure 13.21.

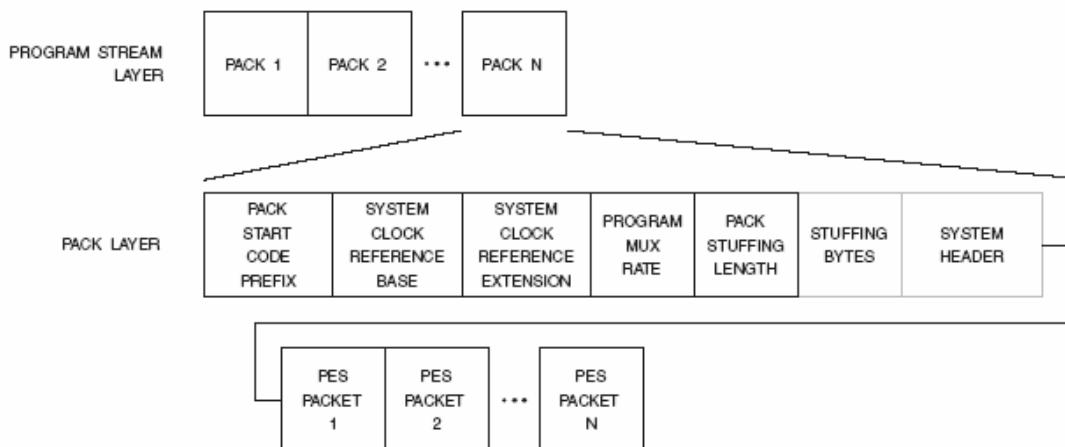


Figure 13.21. MPEG 2 Program Stream Structure. Marker and reserved bits not shown.

Pack_start_code	0x0000001BA
System_header_start_code	0x0000001BB

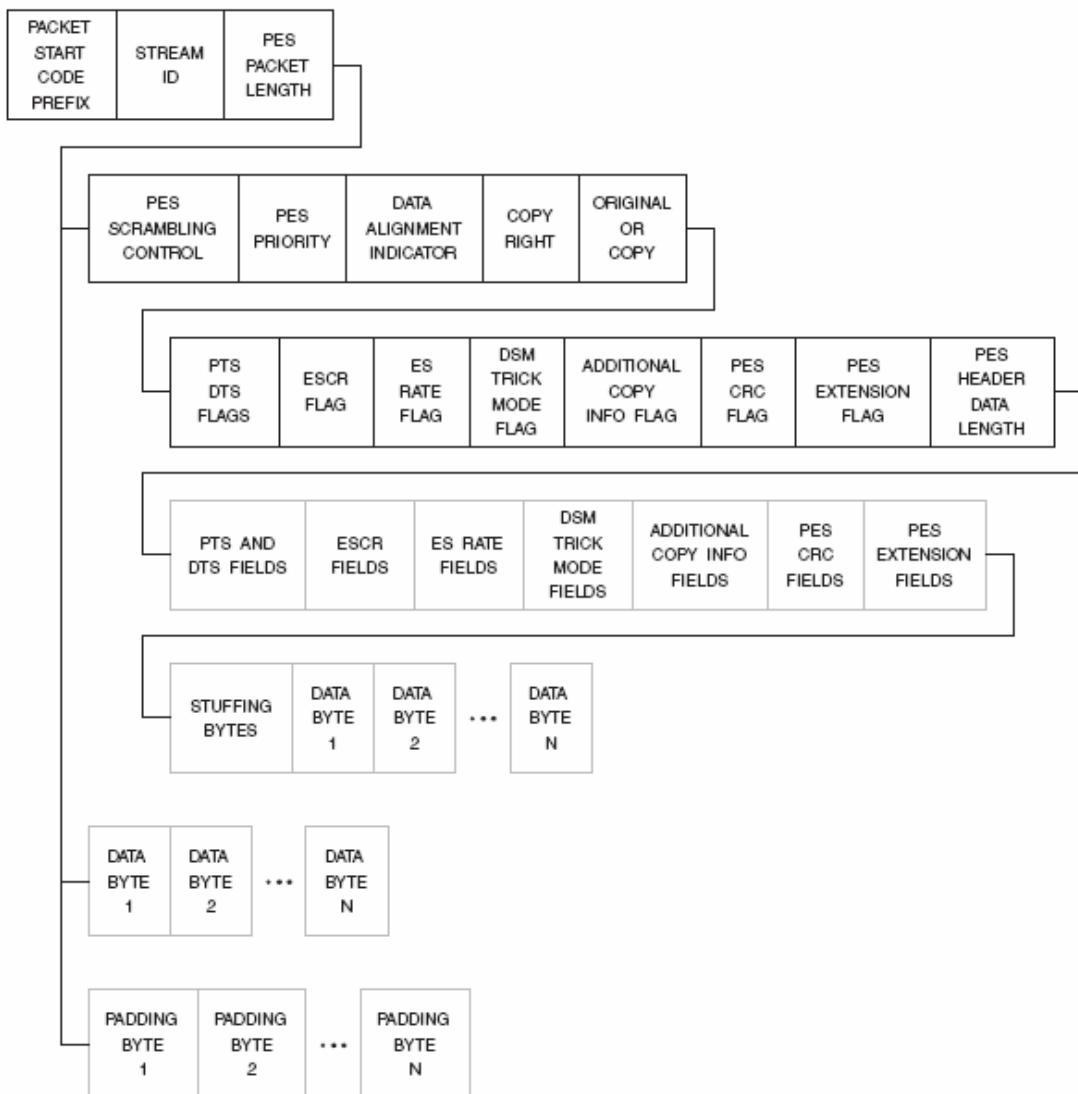


Figure 13.23. MPEG 2 PES Packet Structure. Marker and reserved bits not shown.

Program_stream_map	0x0000001BC
Private_stream_1	0x0000001BD
Padding_stream	0x0000001BE
Private_stream_2	0x0000001BF
Audio_stream	0x0000001C0~0x0000001DF
Video_stream	0x0000001E0~0x0000001EF
ECM_stream	0x0000001F0
EMM_stream	0x0000001F1
DSM_CC_stream	0x0000001F2
ISO/IEC_13552_stream	0x0000001F3

H. 222. 1 type A	0x000001F4
H. 222. 1 type B	0x000001F5
H. 222. 1 type C	0x000001F6
H. 222. 1 type D	0x000001F7
H. 222. 1 type E	0x000001F8
Ancillary_stream	0x000001F9
Reserved	0x000001FA ~ 0x000001FE
Program_stream_directory	0x000001FF

Table 13.41. MPEG 2 *stream\_ID* Codewords.

#### 9.8.5 MPEG-2 video bitstream

Figure 13.5 illustrates the video bitstream, a hierarchical structure with seven layers. From top to bottom the layers are:

- Video Sequence
- Sequence Header
- Group of Pictures (GOP)
- Picture
- Slice
- Macroblock (MB)
- Block

Several extensions may be used to support various levels of capability. These extensions are:

- Sequence Extension
- Sequence Display Extension
- Sequence Scalable Extension
- Picture Coding Extension
- Quant Matrix Extension
- Picture Display Extension
- Picture Temporal Scalable Extension
- Picture Spatial Scalable Extension

If the first sequence header of a video sequence is not followed by an extension start code (000001B5H), then the video bitstream must conform to the MPEG 1 video bitstream. For MPEG 2 video bitstreams, an extension start code (000001B5H) and a sequence extension must follow each sequence header.

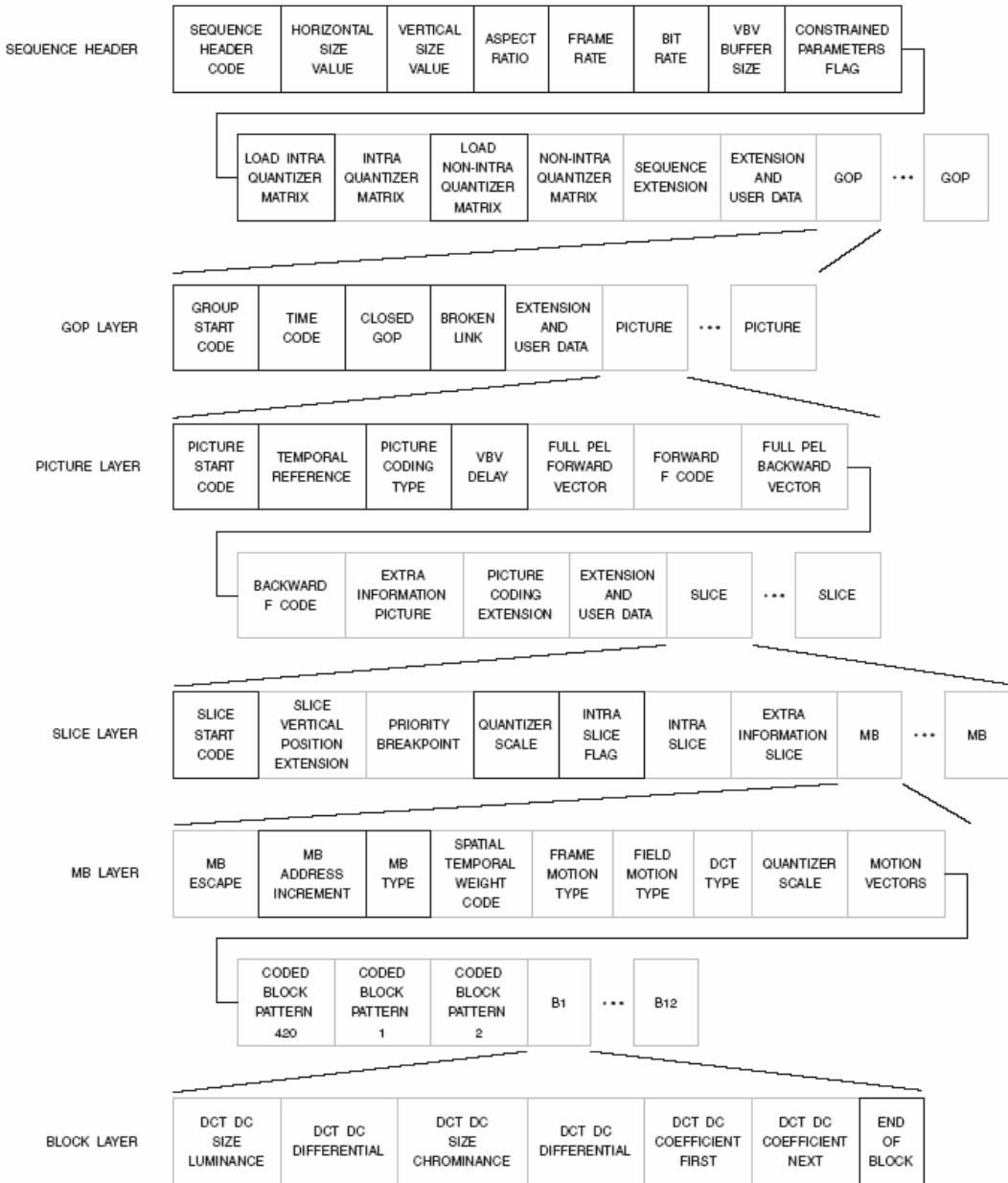


Figure 13.5. MPEG 2 Video Bitstream Layer Structures. Marker and reserved bits not shown.

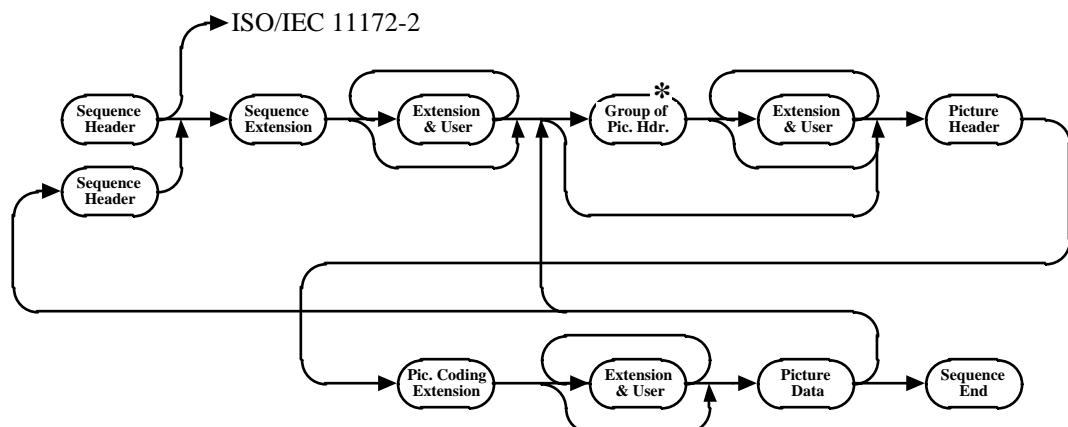


Figure 6-15. High level bitstream organisation

**Table 6-2. extension\_start\_code\_identifier codes.**

extension_start_code_identifier	Name
0000	reserved
0001	Sequence Extension ID
0010	Sequence Display Extension ID
0011	Quant Matrix Extension ID
0100	Copyright Extension ID
0101	Sequence Scalable Extension ID
0110	reserved
0111	Picture Display Extension ID
1000	Picture Coding Extension ID
1001	Picture Spatial Scalable Extension ID
1010	Picture Temporal Scalable Extension ID
1011	reserved
1100	reserved
...	...
1111	reserved

#### 9.8.7 How to distinguish MPEG-1/2 bitstream without additional information

##### 1. 区分 MPEG-1/2 系统层码流

MPEG-1 系统流中，在 pack\_start\_code (0x0000001BA) 后面 4bit 为固定值 “0010”。在 MPEG-2 Program Stream 系统流中，在 pack\_start\_code (0x0000001BA) 后面 4bit 为固定值 “01xx”。而对 MPEG-2 Transport Stream 来说，每个包固定都是 188byte，每个包的开头第一个 byte 均为 0x47。

##### 2. 区分 MPEG-1/2 视频流

MPEG-2 的每一个序列(sequence)中，第一个 sequence\_header (sequence\_header\_start\_code 0x0000001B3) 后面一定会紧跟一个扩展数据包(extension start code 0x0000001B5)。如果跟的不是扩展数据包，那么这个视频流是 MPEG-1 编码的。

#### 9.9 AVI file format

AVI (Audio Video Interleaved 的缩写) 是一种 RIFF (Resource Interchange File Format 的缩写) 文件格式，多用于音视频捕捉、编辑、回放等应用程序中。通常情况下，一个 AVI 文件可以包含多个不同类型的媒体流（典型的情况下有一个音频流和一个视频流），不过含有单一音频流或单一视频流的 AVI 文件也是合法的。AVI 可以算是 Windows 操作系统上最基本的、也是最常用的一种媒体文件格式。

先来介绍 RIFF 文件格式。RIFF 文件使用四字符码 FOURCC (four-character code) 来表征数据类型，比如 ‘RIFF’、‘AVI’、‘LIST’ 等。注意，Windows 操作系统使用的字节顺序是 little-endian，因此

一个四字符码 ‘abcd’ 实际的 DWORD 值应为 0x64636261。另外，四字符码中像 ‘AVI’ 一样含有空格也是合法的。

RIFF 文件最开始的 4 个字节是一个四字符码 ‘RIFF’，表示这是一个 RIFF 文件；紧跟着后面用 4 个字节表示此 RIFF 文件的大小；然后又是一个四字符码说明文件的具体类型（比如 AVI、WAVE 等）；最后就是实际的数据。注意文件大小值的计算方法为：实际数据长度 + 4（文件类型域的大小）；也就是说，文件大小的值不包括 ‘RIFF’ 域和“文件大小”域本身的大小。

RIFF 文件的实际数据中，通常还使用了列表（List）和块（Chunk）的形式来组织。列表可以嵌套子列表和块。其中，列表的结构为：‘LIST’ listSize listType listData —— ‘LIST’ 是一个四字符码，表示这是一个列表；listSize 占用 4 字节，记录了整个列表的大小；listType 也是一个四字符码，表示本列表的具体类型；listData 就是实际的列表数据。注意 listSize 值的计算方法为：实际的列表数据长度 + 4（listType 域的大小）；也就是说 listSize 值不包括 ‘LIST’ 域和 listSize 域本身的大小。再来看块的结构：ckID ckSize ckData —— ckID 是一个表示块类型的四字符码；ckSize 占用 4 字节，记录了整个块的大小；ckData 为实际的块数据。注意 ckSize 值指的是实际的块数据长度，而不包括 ckID 域和 ckSize 域本身的大小。（注意：在下面的内容中，将以 LIST ( listType ( listData ) ) 的形式来表示一个列表，以 ckID ( ckData ) 的形式来表示一个块，如[ optional element ] 中括号中的元素表示为可选项。）

接下来介绍 AVI 文件格式。AVI 文件类型用一个四字符码 ‘AVI’ 来表示。整个 AVI 文件的结构为：一个 RIFF 头 + 两个列表（一个用于描述媒体流格式、一个用于保存媒体流数据）+ 一个可选的索引块。AVI 文件的展开结构大致如下：

```
RIFF ('AVI')
  LIST ('hdrl'
    'avih' (主 AVI 信息头数据)
    LIST ('strl'
      'strh' (流的头信息数据)
      'strf' (流的格式信息数据)
      [ 'strd' (可选的额外的头信息数据) ]
      [ 'strn' (可选的流的名字) ]
      ...
    )
    ...
  )
  LIST ('movi'
    { SubChunk | LIST ('rec'
      SubChunk1
      SubChunk2
      ...
    )
    ...
  )
  ...
}
```

```
[ ‘idx1’ (可选的 AVI 索引块数据)
)
```

首先，RIFF(‘AVI’,...)表征了 AVI 文件类型。然后就是 AVI 文件必需的第一个列表——‘hdrl’ 列表，用于描述 AVI 文件中各个流的格式信息（AVI 文件中的每一路媒体数据都称为一个流）。「hdrl」列表嵌套了一系列块和子列表——首先是一个‘avih’块，用于记录 AVI 文件的全局信息，比如流的数量、视频图像的宽和高等，可以使用一个 AVIMAINHEADER 数据结构来操作：

```
typedef struct _avimainheader {
    FOURCC fcc; // 必须为 ‘avih’
    DWORD cb; // 本数据结构的大小，不包括最初的 8 个字节 (fcc 和 cb 两个域)
    DWORD dwMicroSecPerFrame; // 视频帧间隔时间 (以毫秒为单位)
    DWORD dwMaxBytesPerSec; // 这个 AVI 文件的最大数据率
    DWORD dwPaddingGranularity; // 数据填充的粒度
    DWORD dwFlags; // AVI 文件的全局标记，比如是否含有索引块等
    DWORD dwTotalFrames; // 总帧数
    DWORD dwInitialFrames; // 为交互格式指定初始帧数 (非交互格式应该指定为 0)
    DWORD dwStreams; // 本文件包含的流的个数
    DWORD dwSuggestedBufferSize; // 建议读取本文件的缓存大小 (应能容纳最大的块)
    DWORD dwWidth; // 视频图像的宽 (以像素为单位)
    DWORD dwHeight; // 视频图像的高 (以像素为单位)
    DWORD dwReserved[4]; // 保留
} AVIMAINHEADER;
```

然后，就是一个或多个‘strl’子列表。（文件中有多少个流，这里就对应有多少个‘strl’子列表。）每个‘strl’子列表至少包含一个‘strh’块和一个‘strf’块，而‘strd’块（保存编解码器需要的一些配置信息）和‘strn’块（保存流的名字）是可选的。首先是‘strh’块，用于说明这个流的头信息，可以使用一个 AVISTREAMHEADER 数据结构来操作：

```
typedef struct _avistreamheader {
    FOURCC fcc; // 必须为 ‘strh’
    DWORD cb; // 本数据结构的大小，不包括最初的 8 个字节 (fcc 和 cb 两个域)
    FOURCC fccType; // 流的类型：‘auds’ (音频流)、‘vids’ (视频流)、
                    // ‘mids’ (MIDI 流)、‘txts’ (文字流)
    FOURCC fccHandler; // 指定流的处理器，对于音视频来说就是解码器
    DWORD dwFlags; // 标记：是否允许这个流输出？调色板是否变化？
    WORD wPriority; // 流的优先级 (当有多个相同类型的流时优先级最高的为默认流)
    WORD wLanguage;
    DWORD dwInitialFrames; // 为交互格式指定初始帧数
    DWORD dwScale; // 这个流使用的时间尺度
    DWORD dwRate;
    DWORD dwStart; // 流的开始时间
    DWORD dwLength; // 流的长度 (单位与 dwScale 和 dwRate 的定义有关)
    DWORD dwSuggestedBufferSize; // 读取这个流数据建议使用的缓存大小
}
```

```

DWORD dwQuality; // 流数据的质量指标 (0 ~ 10,000)
DWORD dwSampleSize; // Sample 的大小
struct {
    short int left;
    short int top;
    short int right;
    short int bottom;
} rcFrame; // 指定这个流 (视频流或文字流) 在视频主窗口中的显示位置
// 视频主窗口由 AVIMAINHEADER 结构中的 dwWidth 和 dwHeight 决定
} AVISTREAMHEADER;

```

然后是 ‘strf’ 块，用于说明流的具体格式。如果是视频流，则使用一个 BITMAPINFO 数据结构来描述；如果是音频流，则使用一个 WAVEFORMATEX 数据结构来描述。

```

typedef struct {
    WORD wFormatTag;
    WORD nChannels;
    DWORD nSamplesPerSec;
    DWORD nAvgBytesPerSec;
    WORD nBlockAlign;
    WORD wBitsPerSample;
    WORD cbSize;
} WAVEFORMATEX;

```

当 AVI 文件中的所有流都使用一个 ‘strl’ 子列表说明了以后（注意：‘strl’ 子列表出现的顺序与媒体流的编号是对应的，比如第一个 ‘strl’ 子列表说明的是第一个流（Stream 0），第二个 ‘strl’ 子列表说明的是第二个流（Stream 1），以此类推），‘hdrl’ 列表的任务也就完成了，随后跟着的就是 AVI 文件必需的第二个列表——‘movi’ 列表，用于保存真正的媒体流数据（视频图像帧数据或音频采样数据等）。那么，怎么来组织这些数据呢？可以将数据块直接嵌在 ‘movi’ 列表里面，也可以将几个数据块分组成一个 ‘rec’ 列表后再编排进 ‘movi’ 列表。（注意：在读取 AVI 文件内容时，建议将一个 ‘rec’ 列表中的所有数据块一次性读出。）但是，当 AVI 文件中包含有多个流的时候，数据块与数据块之间如何来区别呢？于是数据块使用了一个四字符码来表征它的类型，这个四字符码由 2 个字节的类型码和 2 个字节的流编号组成。标准的类型码定义如下：‘db’（非压缩视频帧）、‘dc’（压缩视频帧）、‘pc’（改用新的调色板）、‘wb’（音频帧）。比如第一个流（Stream 0）是音频，则表征音频数据块的四字符码为 ‘00wb’；第二个流（Stream 1）是视频，则表征视频数据块的四字符码为 ‘00db’ 或 ‘00dc’。对于视频数据来说，在 AVI 数据序列中间还可以定义一个新的调色板，每个改变的调色板数据块用 ‘xxpc’ 来表征，新的调色板使用一个数据结构 AVIPALCHANGE 来定义。（注意：如果一个流的调色板中途可能改变，则应在这个流格式的描述中，也就是 AVISTREAMHEADER 结构的 dwFlags 中包含一个 AVISF\_VIDEO\_PALCHANGES 标记。）另外，文字流数据块可以使用随意的类型码表征。

最后，紧跟在 ‘hdrl’ 列表和 ‘movi’ 列表之后的，就是 AVI 文件可选的索引块。这个索引块为 AVI 文件中每一个媒体数据块进行索引，并且记录它们在文件中的偏移（可能相对于 ‘movi’ 列表，也可能相对于 AVI 文件开头）。索引块使用一个四字符码 ‘idx1’ 来表征，索引信息使用一个数据结构来

AVIOLDINDEX 定义。

```

typedef struct _avioldindex {
    FOURCC  fcc; // 必须为 ‘idx1’
    DWORD    cb;   // 本数据结构的大小，不包括最初的 8 个字节 (fcc 和 cb 两个域)
    struct _avioldindex_entry {
        DWORD    dwChunkId; // 表征本数据块的四字符码
        DWORD    dwFlags;    // 说明本数据块是不是关键帧、是不是 ‘rec’ 列表等信息
        DWORD    dwOffset;   // 本数据块在文件中的偏移量
        DWORD    dwSize;    // 本数据块的大小
    } aIndex[]; // 这是一个数组！为每个媒体数据块都定义一个索引信息
} AVIOLDINDEX;

```

注意：如果一个 AVI 文件包含有索引块，则应在主 AVI 信息头的描述中，也就是 AVIMAINHEADER 结构的 dwFlags 中包含一个 AVIF\_HASINDEX 标记。

还有一种特殊的数据块，用一个四字符码 ‘JUNK’ 来表征，它用于内部数据的队齐（填充），应用程序应该忽略这些数据块的实际意义。

## 9.10 3GP file format

### 9.10.1 Evolution of 3GPP standard

	Release 4	Release 5	Release 6
Capability Exchange	NONE	UAProf	UAProf
Video Codecs	<ul style="list-style-type: none"> <li>• H.263 P0L10 (M), P3L10 (O)</li> <li>• MPEG-4 VSP L0 (O)</li> </ul>	<ul style="list-style-type: none"> <li>• As in Release 4</li> </ul>	<ul style="list-style-type: none"> <li>• As in Release 4</li> <li>• H.263 P0 L45 (O)</li> <li>• MPEG-4 VSP L0b (O)</li> <li>• H.264 Full Baseline (O)</li> </ul>
Audio & Speech Codecs	<ul style="list-style-type: none"> <li>• AMR-NB &amp; WB (M)</li> <li>• MPEG-4 AAC LC, LTP (O)</li> </ul>	<ul style="list-style-type: none"> <li>• As in Release 4</li> </ul>	<ul style="list-style-type: none"> <li>• As in Release 4</li> <li>• AMR-WB+ or AACPlus (O)</li> </ul>
Media File Format	<ul style="list-style-type: none"> <li>• 3GPP File Format (.3gp)</li> <li>• .amr</li> </ul>	<ul style="list-style-type: none"> <li>• As in Release 4</li> <li>• ISO Base Format Conformance (M)</li> <li>• Timed-text (O)</li> </ul>	<ul style="list-style-type: none"> <li>• As in Release 5</li> <li>• Different 3GP file profiles (server, MMS, program, downloadable, generic)(O)</li> <li>• DRM (O)</li> </ul>
Session Establishment	<ul style="list-style-type: none"> <li>• RTSP (M)</li> <li>• SDP (M)</li> <li>• HTTP (O)</li> </ul>	<ul style="list-style-type: none"> <li>• As in Release 4</li> </ul>	<ul style="list-style-type: none"> <li>• As in Release 4</li> <li>• Media Alternatives in SDP</li> <li>• Metadata signalling in SDP (O)</li> <li>• MBMS - FLUTE</li> </ul>

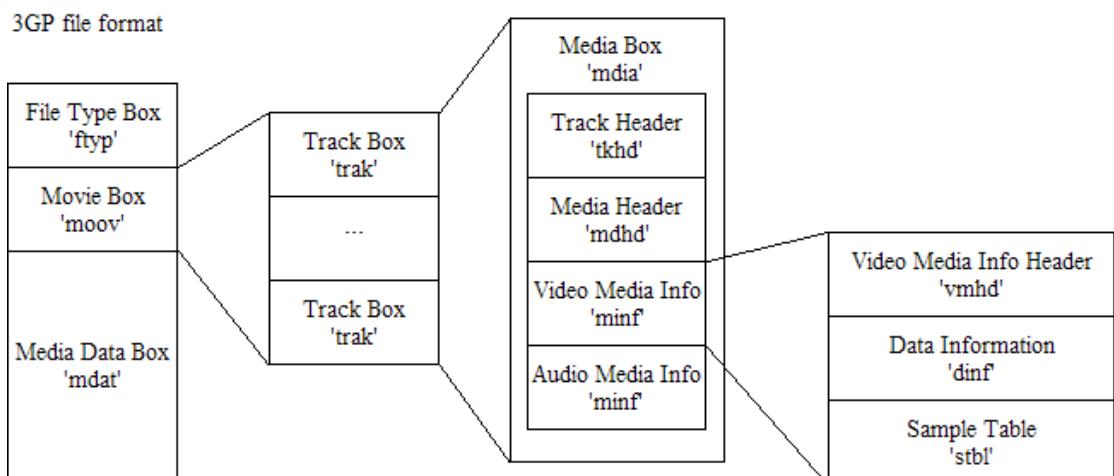
			(M)
Data Transport	<ul style="list-style-type: none"> <li>RTP/RTCP (M)</li> </ul>	<ul style="list-style-type: none"> <li>As in Release 4</li> <li>Progressive Download (O)</li> </ul>	<ul style="list-style-type: none"> <li>As in Release 5</li> <li>MBMS Download (M)</li> <li>DRM (O)</li> <li>SRTP(O)</li> </ul>
QoS	NONE (only if network provided QoS)	<ul style="list-style-type: none"> <li>As in Release 4</li> </ul>	<ul style="list-style-type: none"> <li>Additional RTSP &amp; SDP level signalling (O)</li> <li>QoE Protocol (O)</li> <li>RTCP extensions (O)</li> </ul>
Rate Control	NONE	<ul style="list-style-type: none"> <li>Annex.G (video only)</li> </ul>	<ul style="list-style-type: none"> <li>As in Release 5</li> <li>3GPP Rate Adaptation (O)</li> </ul>

#### 9.10.2 3GP file format standard (3GPP TS 26.244)

Each 3GP file consists of "Boxes". In general, a 3GP file contains the File Type Box (ftyp), the Movie Box (moov), and the Media Data Box (mdat). The File Type Box identifies the type and properties of the 3GP file itself. The Movie Box and the Media Data Box, serving as containers, include own boxes for each media. Boxes start with a header, which indicates both size and type (these fields are called namely "size" and "type"). Additionally, each box type may include a number of boxes.

In the following, only those boxes are mentioned, which are useful for the purposes of this payload format.

The Movie Box (moov) contains one or more Track Boxes (trak), which include information about each track. A Track Box contains, among others, the Track Header Box (tkhd), the Media Header Box (mdhd) and the Media Information Box (minf).



### 9.10.2.1 Track Atoms

Track atoms define a single track of a movie. A movie may consist of one or more tracks. Each track is independent of the other tracks in the movie and carries its own temporal and spatial information. Each track atom contains its associated media atom.

Tracks are used specifically for the following purposes:

- To contain media data references and descriptions (media tracks).
- To contain modifier tracks (tweens, and so forth).
- To contain packetization information for streaming protocols (hint tracks). Hint tracks may contain references to media sample data or copies of media sample data. For more information about hint tracks, refer to “Hint Media”.

Figure 2-6 shows the layout of a track atom. Track atoms have an atom type value of 'trak'. The track atom requires the track header atom ('tkhd') and the media atom ('mdia'). Other child atoms are optional, and may include a track clipping atom ('clip'), a track matte atom ('matt'), an edit atom ('edts'), a track reference atom ('tref'), a track load settings atom ('load'), a track input map atom ('imap'), and a user data atom ('udta').

Figure 2-6 The layout of a track atom

Track atom		
Atom size Type = 'trak'		
Track header atom	'tkhd'	‡
Clipping atom	'clip'	
Track matte atom	'matt'	
Edit atom	'edts'	
Track reference atom	'tref'	
Track loading settings atom	'load'	
Track input map atom	'imap'	
Media atom	'mdia'	‡
User-defined data atom	'udta'	

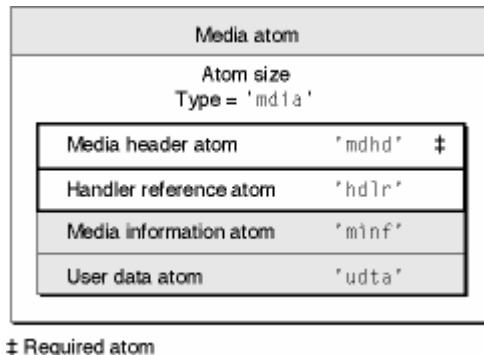
‡ Required atom

### 9.10.2.2 Media Atoms

Media atoms describe and define a track's media type and sample data. The media atom contains information that specifies the media type, such as sound or video, the media handler component used to interpret the sample data, the media timescale and track duration, and media-and-track-specific information such as sound volume or graphics mode. It also contains the media data references, which typically specify the file where the sample data is stored, and the sample table atoms, which specify the sample description, duration, and byte offset from the data reference for each media sample.

The media atom has an atom type of 'mdia'. It must contain a media header atom ('mdhd'), and it can contain a handler reference ('hdlr'), media information ('minf'), and user data ('udta').

Figure 2-15 shows the layout of a media atom.



### 9.10.2.3 Track Header

The track header atom specifies the characteristics of a single track within a movie. A track header atom contains a size field that specifies the number of bytes and a type field that indicates the format of the data (defined by the atom type 'tkhd').

Figure 2-7 The layout of a track header atom

	Bytes
Track header atom	
Atom size	4
Type = 'tkhd'	4
Version	1
Flags	3
Creation time	4
Modification time	4
Track ID	4
Reserved	4
Duration	4
Reserved	8
Layer	2
Alternate group	2
Volume	2
Reserved	2
Matrix structure	36
Track width	4
Track height	4

#### 9.10.2.4 Media Header Atoms

The media header atom specifies the characteristics of a media, including time scale and duration. The media header atom has an atom type of 'mdhd'.

Figure 2-16 The layout of a media header atom

	Bytes
Media header atom	
Atom size	4
Type = 'mdhd'	4
Version	1
Flags	3
Creation time	4
Modification time	4
Time scale	4
Duration	4
Language	2
Quality	2

#### 9.10.2.5 Video Media Information Atoms

Video media information atoms are the highest-level atoms in video media. These atoms contain a number of other atoms that define specific characteristics of the video media data. Figure 2-18 shows the layout of a video media information atom.

Figure 2-18 The layout of a media information atom for video

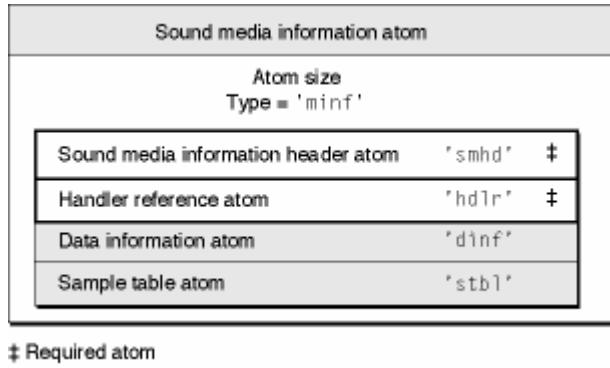
Video media information atom		
Atom size		
Type = 'minf'		
Video media information header atom	'vmhd'	‡
Handler reference atom	'hdlr'	‡
Data information atom	'dinf'	
Sample table atom	'stbl'	

‡ Required atom

#### 9.10.2.6 Sound Media Information Atoms

Sound media information atoms are the highest-level atoms in sound media. These atoms contain a number of other atoms that define specific characteristics of the sound media data.

Figure 2-20 the layout of a sound media information atom.



#### 9.10.2.7 Sample Table Atoms

The sample table atom contains information for converting from media time to sample number to sample location. This atom also indicates how to interpret the sample (for example, whether to decompress the video data and, if so, how). This section describes the format and content of the sample table atom.

The sample table atom has an atom type of 'stbl'. It can contain the sample description atom, the time-to-sample atom, the sync sample atom, the sample-to-chunk atom, the sample size atom, the chunk offset atom, and the shadow sync atom.

The sample table atom contains all the time and data indexing of the media samples in a track. Using tables, it is possible to locate samples in time, determine their type, and determine their size, container, and offset into that container.

If the track that contains the sample table atom references no data, then the sample table atom does not need to contain any child atoms (not a very useful media track).

If the track that the sample table atom is contained in does reference data, then the following child atoms are required: sample description, sample size, sample to chunk, and chunk offset. All of the subtables of the sample table use the same total sample count.

The sample description atom must contain at least one entry. A sample description atom is required because it contains the data reference index field that indicates which data reference atom to use to retrieve the media samples. Without the sample description, it is not possible to determine where the media samples are stored. The sync sample atom is optional. If the sync sample atom is not present, all samples are implicitly sync samples.

Figure 2-26 The layout of a sample table atom

Sample table atom	
Atom size	
Type = 'stbl'	
Sample description atom	'stsd'
Time-to-sample atom	'stts'
Sync sample atom	'stss'
Sample-to-chunk atom	'stsc'
Sample size atom	'stsz'
Chunk offset atom	'stco'
Shadow sync atom	'stsh'

### 9.11 Real Media file format

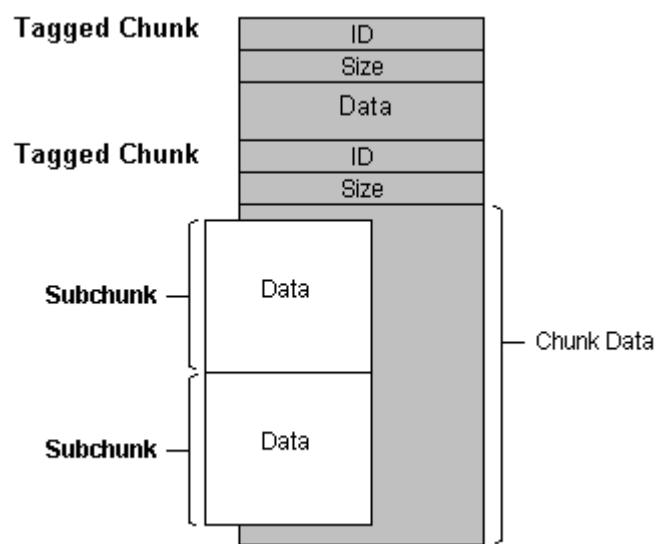
RealMedia File Format is a standard tagged file format that uses four-character codes to identify file elements. These codes are 32-bit, represented by a sequence of one to four ASCII alphanumeric characters, padded on the right with space characters. The data type for four-character codes is FOURCC.

The basic building block of a RealMedia File is a chunk , which is a logical unit of data, such as a stream header or a packet of data. Each chunk contains the following fields:

- Four-character code specifying the chunk identifier
- 32-bit value specifying the size of the data member in the chunk
- Blob of opaque chunk data

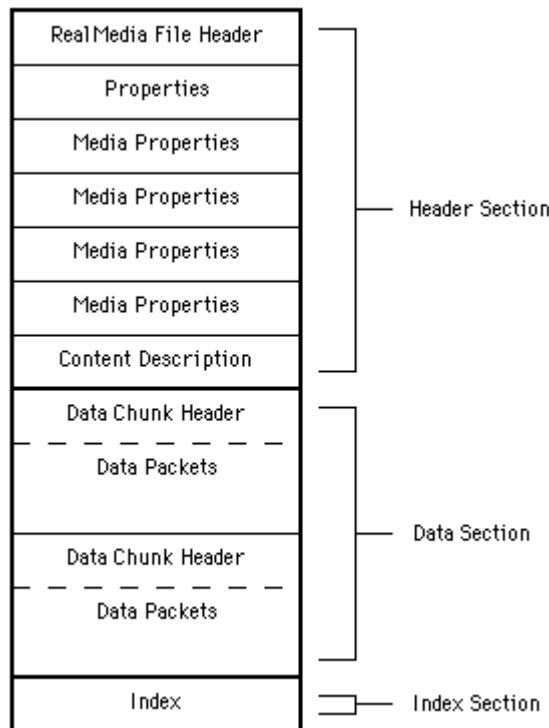
Depending on its type, a top-level chunk can contain subobjects. This document describes the tagged chunks contained in RMFF, as well as the format of the data stored in each type of tagged chunk.

### 9.11.1 Tagged File Formats



RealMedia File Format organizes tagged chunks into a header section, a data section, and an index section. The organization of these tagged chunks is shown in the following figure.

### 9.11.2 Sections of a RealMedia File



### 9.11.3 Header Section

Because RMFF is a tagged file format, the order of the chunks is not explicit, except that the RealMedia File

Header must be the first chunk in the file. However, most applications write the standard headers into the file's header section. The following chunks are typically found in the header section of RMFF:

- RealMedia File Header (this must be the first chunk of the file)
- Properties Header
- Media Properties Header
- Content Description Header

After the RealMedia File Header object, the other headers may appear in any order. All headers are required except the Index Header. The following sections describe the individual header objects.

#### 9.11.4 RealMedia File Header

Each RealMedia file begins with the RealMedia File Header, which identifies the file as RMFF. There is only one RealMedia File Header in a RealMedia file. Because the contents of the RealMedia File Header may change with different versions of RMFF, the header structure supports an object version field for determining what additional fields exists. The following pseudo-structure describes the RealMedia File Header:

```
RealMedia_File_Header
{
    UINT32    object_id;
    UINT32    size;
    UINT16    object_version
;

    if ((object_version == 0) || (object_version == 1))
    {
        UINT32    file_version;
        UINT32    num_headers;
    }
}
```

The RealMedia File Header contains the following fields:

##### **object\_id**

The unique object ID for a RealMedia File (.RMF). All RealMedia files begin with this identifier. The size of this member is 32 bits.

##### **size**

The size of the RealMedia header section in bytes. The size of this member is 32 bits.

##### **object\_version**

The version of the RealMedia File Header object. All files created according to this specification have an **object\_version** number of 0 (zero) or 1. The size of this member is 16 bits.

##### **file\_version**

The version of the RealMedia file. The Helix Client and Server SDK only covers files with a file version of either 0 (zero) or 1. This member is present on all **RealMedia\_File\_Header** objects with an **object\_version** of 0 (zero) or

1. The size of this member is 32 bits.

### **num\_headers**

The number of headers in the header section that follow the RealMedia File Header. This member is present on all `RealMedia_File_Header` objects with an `object_version` of 0 (zero) or 1. The size of this member is 32 bits.

### **9.11.5 Properties Header**

The Properties Header describes the general media properties of the RealMedia File. Components of the RealMedia system use this object to configure themselves for handling the data in the RealMedia file or stream. There is only one Properties Header in a RealMedia file. The following pseudo-structure describes the Properties header:

```
Properties
{
    UINT32    object_id;
    UINT32    size;
    UINT16    object_version;

    if (object_version == 0)
    {
        UINT32    max_bit_rate;
        UINT32    avg_bit_rate;
        UINT32    max_packet_size;
        UINT32    avg_packet_size;
        UINT32    num_packets;
        UINT32    duration;
        UINT32    preroll;
        UINT32    index_offset;
        UINT32    data_offset;
        UINT16    num_streams;
        UINT16    flags;
    }
}
```

The Properties Header contains the following fields:

#### **object\_id**

The unique object ID for a Properties Header ('PROP'). The size of this member is 32 bits.

#### **size**

The 32-bit size of the Properties Header in bytes. The size of this member is 32 bits.

#### **object\_version**

The version of the RealMedia File Header object. All files created according to this specification have an `object_version` number of 0 (zero). The size of this member is 16 bits.

#### **max\_bit\_rate**

The maximum bit rate required to deliver this file over a network. This member is present on all Properties objects with an `object_version` of 0 (zero). The size of this

member is 32 bits.

#### **avg\_bit\_rate**

The average bit rate required to deliver this file over a network. This member is present on all Properties objects with an `object_version` of 0 (zero). The size of this member is 32 bits.

#### **max\_packet\_size**

The largest packet size (in bytes) in the media data. This member is present on all Properties objects with an `object_version` of 0 (zero). The size of this member is 32 bits.

#### **avg\_packet\_size**

The average packet size (in bytes) in the media data. This member is present on all Properties objects with an `object_version` of 0 (zero). The size of this member is 32 bits.

#### **num\_packets**

The number of packets in the media data. This member is present on all Properties objects with an `object_version` of 0 (zero). The size of this member is 32 bits.

#### **duration**

The duration of the file in milliseconds. This member is present on all Properties objects with an `object_version` of 0 (zero). The size of this member is 32 bits.

#### **preroll**

The number of milliseconds to prebuffer before starting playback. This member is present on all Properties objects with an `object_version` of 0 (zero). The size of this member is 32 bits.

#### **index\_offset**

The offset in bytes from the start of the file to the start of the index header object. This value can be 0 (zero), which indicates that no index chunks are present in this file. This member is present on all Properties objects with an `object_version` of 0 (zero). The size of this member is 32 bits.

#### **data\_offset**

The offset in bytes from the start of the file to the start of the Data Section. This member is present on all Properties objects with an `object_version` of 0 (zero). The size of this member is 32 bits.



**Note:** There can be a number of Data\_Chunk\_Headers in a RealMedia file. The `data_offset` value specifies the offset in bytes to the first Data\_Chunk\_Header. The offsets to the other Data\_Chunk\_Headers can be derived from the `next_data_header` field in a Data\_Chunk\_Header.

#### **num\_streams**

The total number of media properties headers in the main headers section. This member is present on all Properties objects with an `object_version` of 0 (zero). The size of this member is 16 bits.

#### **flags**

Bit mask containing information about this file. The following bits carry information—all of the rest should be zero:

Bit	Flag	Description
0	Save_Enabled	If 1, clients are allowed to save this file to disk.
1	Perfect_Play	If 1, clients are instructed to use extra buffering.
2	LIve	If 1, these streams are from a live broadcast.

The size of this member is 16 bits.

#### 9.11.6 Media Properties Header

The Media Properties Header describes the specific media properties of each stream in a RealMedia file. Components of the RealMedia system use this object to configure themselves for handling the media data in each stream. There is one Media Properties Header for each media stream in a RealMedia file. The following pseudo-structure describes the Media Properties header:

```
Media_Properties
{
    UINT32      object_id;
    UINT32      size;
    UINT16      object_version;

    if (object_version == 0)
    {
        UINT16                  stream_number;
        UINT32                  max_bit_rate;
        UINT32                  avg_bit_rate;
        UINT32                  max_packet_size;
        UINT32                  avg_packet_size;
        UINT32                  start_time;
        UINT32                  preroll;
        UINT32                  duration;
        UINT8                   stream_name_size;
        UINT8[stream_name_size]   stream_name;
        UINT8                   mime_type_size;
        UINT8[mime_type_size]     mime_type;
        UINT32                  type_specific_len;
        UINT8[type_specific_len]  type_specific_data;
    }
}
```

The Media Properties Header contains the following members:

##### **object\_id**

The unique object ID for a Media Properties Header ("MDPR"). The size of this member is 32 bits.

##### **size**

The size of the Media Properties Header in bytes. The size of this member is 32 bits.

##### **object\_version**

The version of the Media Properties Header object. The size of this member is 16 bits.

#### **stream\_number**

The `stream_number` (synchronization source identifier) is a unique value that identifies a physical stream. Every data packet that belongs to a physical stream contains the same `STREAM_NUMBER`. The `STREAM_NUMBER` enables a receiver of multiple physical streams to distinguish which packets belong to each physical stream. This member is present on all `MediaProperties` objects with an `object_version` of 0 (zero). The size of this member is 32 bits.

#### **max\_bit\_rate**

The maximum bit rate required to deliver this stream over a network. This member is present on all `MediaProperties` objects with an `object_version` of 0 (zero). The size of this member is 32 bits.

#### **avg\_bit\_rate**

The average bit rate required to deliver this stream over a network. This member is present on all `MediaProperties` objects with an `object_version` of 0 (zero). The size of this member is 32 bits.

#### **max\_packet\_size**

The largest packet size (in bytes) in the stream of media data. This member is present on all `MediaProperties` objects with an `object_version` of 0 (zero). The size of this member is 32 bits.

#### **avg\_packet\_size**

The average packet size (in bytes) in the stream of media data. This member is present on all `MediaProperties` objects with an `object_version` of 0 (zero). The size of this member is 32 bits.

#### **start\_time**

The time offset in milliseconds to add to the time stamp of each packet in a physical stream. This member is present on all `MediaProperties` objects with an `object_version` of 0 (zero). The size of this member is 32 bits.

#### **preroll**

The time offset in milliseconds to subtract from the time stamp of each packet in a physical stream. This member is present on all `MediaProperties` objects with an `object_version` of 0 (zero). The size of this member is 32 bits.

#### **duration**

The duration of the stream in milliseconds. This member is present on all `MediaProperties` objects with an `object_version` of 0 (zero). The size of this member is 32 bits.

#### **stream\_name\_size**

The length of the following `stream_name` member in bytes. This member is present on all `MediaProperties` objects with an `object_version` of 0 (zero). The size of this member is 8 bits.

#### **stream\_name**

A nonunique alias or name for the stream. This member is present on all `MediaProperties` objects with an `object_version` of 0 (zero). This size of this member is variable.

#### **mime\_type\_size**

The length of the following `mime_type` field in bytes. This member is present on all `MediaProperties` objects with an `object_version` of 0 (zero). This size of this member is 8 bits.

#### **mime\_type**

A nonunique MIME style type/subtype string for data associated with the stream. This member is present on all `MediaProperties` objects with an `object_version` of 0 (zero). This size of this member is variable.

#### **type\_specific\_len**

The length of the following `type_specific_data` in bytes. The `type_specific_data` is typically used by the data type renderer to initialize itself in order to process the physical stream. This member is present on all `MediaProperties` objects with an `object_version` of 0 (zero). The size of this member is 32 bits.

#### **type\_specific\_data**

The `type_specific_data` is typically used by the data type renderer to initialize itself in order to process the physical stream. This member is present on all `MediaProperties` objects with an `object_version` of 0 (zero). The size of this member is variable.

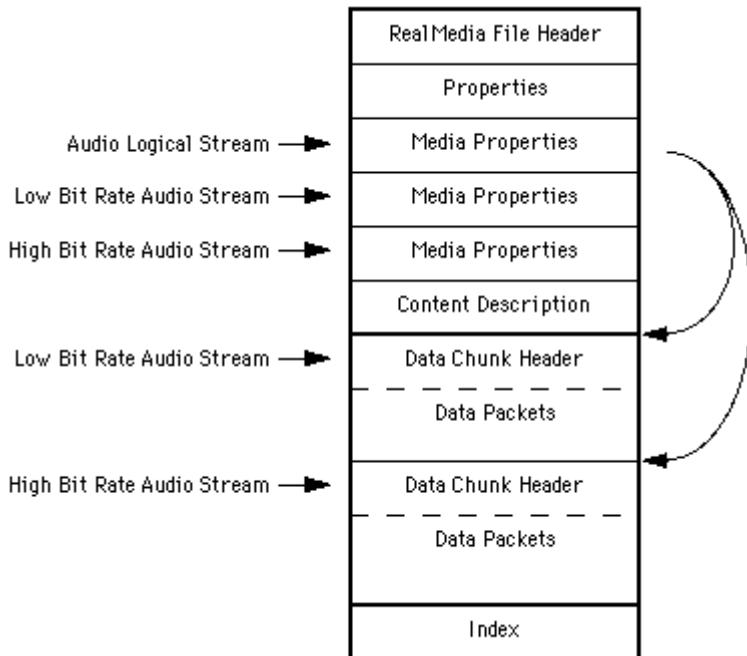
### **9.11.7 Logical Stream Organization**

A RealMedia file can contain a higher-level grouping of physical streams. This grouping is called a logical stream. Logical streams contain the following information:

- Identifies which physical streams are grouped together into a logical stream.
- Contains name value properties that can be used to identify properties of the logical stream. (such as language, packet grouping, and so on.)

A logical stream is represented with a Media Properties Header. The mime type of the physical stream is preceded with "logical- ". For example, the mime type for an ASM-compatible RealAudio stream is `audio/x-pn-multirate-realaudio`. A logical stream consisting of a set of RealAudio physical streams would therefore have the mime type `logical-audio/x-pn-multirate-realaudio`. An example of a logical stream is shown in the following figure.

### 9.11.8 Logical Stream Organization



In this example there is one logical stream, one low bit rate audio stream and one high bit rate audio stream. This results in a RealMedia file with three Media Property Headers and three data sections. The type\_specific\_data field of the logical stream's Media Property Header contains a LogicalStream structure. This structure contains all of the information required to interpret the logical stream and its collection of physical streams. The structure refers to the low bit rate and high bit rate audio streams. The LogicalStream structure also contains the data\_offset s to the start of the data section for each physical stream.

The logical stream number assigned to the logical stream is determined from the stream\_number field in the Media Properties Header.

There is also one special logical stream of MIME type "logical-fileinfo" containing information about the entire file. There should only be one media header with this type. Behavior of players and editing tools is undefined if you have more than one.

The ASM rules contained in the logical-fileinfo stream are used to define precisely how bandwidth will be divided between the streams in the file. The logical-fileinfo may also contain a name-value pair that specifies which stream combinations should be served to older players.

### 9.11.9 LogicalStream Structure

The following sample shows the LogicalStream structure:

```
LogicalStream
{
    ULONG32          size;
    UINT16 object_version;
```

```

if (object_version == 0)
{
    UINT16      num_physical_streams;
    UINT16      physical_stream_numbers[num_physical_streams];
    ULONG32     data_offsets[num_physical_streams];
    UINT16      num_rules;
    UINT16      rule_to_physical_stream_number_map[num_rules];
    UINT16      num_properties;
    NameValueProperty properties[num_properties];
}
};

```

The LogicalStream structure contains the following fields:

#### **size**

The size of the **LogicalStream** structure in bytes. The size of this structure member is 32 bits.

#### **object\_version**

The version of the **LogicalStream** structure. The size of this structure member is 16 bits.

#### **num\_physical\_streams**

The number of physical streams that make up this logical stream. The physical stream numbers are stored in a list immediately following this field. These physical stream numbers refer to the **stream\_number** field found in the Media Properties Object for each physical stream belonging to this logical stream. The size of this structure member is 16 bits

#### **physical\_stream\_numbers[]**

The list of physical stream numbers that comprise this logical stream. The size of this structure member is variable.

#### **data\_offsets[]**

The list of data offsets indicating the start of the data section for each physical stream. The size of this structure member is variable.

#### **num\_rules**

The number of ASM rules for the logical stream. Each physical stream in the logical stream has at least one ASM rule associated with it or it will never get played. The mapping of ASM rule numbers to physical stream numbers is stored in a list immediately following this member. These physical stream numbers refer to the **stream\_number** field found in the Media Properties Object for each physical stream belonging to this logical stream. The size of this structure member is 16 bits.

#### **rule\_to\_physical\_stream\_map[]**

The list of physical stream numbers that map to each rule. Each entry in the map corresponds to a 0-based rule number. The value in each entry is set to the physical stream number for the rule. For example:

```
rule_to_physical_stream_map[0] = 5
```

This example means physical stream 5 corresponds to rule 0. All of the ASM rules referenced by this array are stored in the first name-value pair of this logical stream which must be called

"ASMRuleBook" and be of type "string". Each rule is separated by a semicolon.

The size of this structure member is variable.

### **num\_properties**

The number of `NameValueProperty` structures contained in this structure. These name/value structures can be used to identify properties of this logical stream (for example, language). The size of this structure member is 16 bits.

### **properties[]**

The list of `NameValueProperty` structures (see [NameValuePair Structure](#) below for more details). As mentioned above, it is required that the first name-value pair be a string named "ASMRuleBook" and contain the ASM rules for this logical stream. The size of this structure member is variable.

## **9.11.10 NameValueProperty Structure**

The following sample shows the `NameValueProperty` structure:

```
NameValueProperty
{
    ULONG32          size;
    UINT16           object_version;

    if (object_version == 0)
    {
        UINT8 name_length;
        UINT8 name[name_length];
        INT32   type;
        UINT16  value_length;
        UINT8 value_data[value_length];
    }
}
```

The `NameValueProperty` structure contains the following fields:

### **size**

The size of the `NameValueProperty` structure in bytes. The size of this structure member is 32 bits.

### **object\_version**

The version of the `NameValueProperty` structure. The size of this structure member is 16 bits.

### **name\_length**

The length of the name data. The size of this structure member is 8 bits.

### **name[]**

The name string data. The size of this structure member is 8 bits.

### **type**

The type of the value data. This member can take on one of three values (any other value is undefined), as shown in the following table:

type	Description	value_length
------	-------------	--------------

0	32-bit unsigned integer property	4
1	buffer	variable
2	string	variable

The size of this structure member is 32 bits.

#### **value\_length**

The length of the value data. The size of this structure member is 16 bits.

#### **value\_data[]**

The value data. The size of this structure member is 8 bits.

### 9.11.11 Content Description Header

The Content Description Header contains the title, author, copyright, and comments information for the RealMedia file. All text data is in ASCII format. The following pseudo-structure describes the Content Description Header:

```
Content_Description
{
    UINT32      object_id;
    UINT32      size;
    UINT16      object_version
;

    if (object_version == 0)
    {
        UINT16      title_len;
        UINT8[title_len] title;
        UINT16      author_len;
        UINT8[author_len] author;
        UINT16      copyright_len;
        UINT8[copyright_len] copyright;
        UINT16      comment_len;
        UINT8[comment_len] comment;
    }
}
```

The Content Description Header contains the following fields:

#### **object\_id**

The unique object ID for the Content Description Header ("CONT"). The size of this member is 32 bits.

#### **size**

The size of the Content Description Header in bytes. The size of this member is 32 bits.

#### **object\_version**

The version of the Content Description Header object. The size of this member is 16 bits.

#### **title\_len**

The length of the title data in bytes. Note that the title data is not null-terminated. This member is present on all Content Description Header objects with an

`object_version` of 0 (zero). The size of this member is 16 bits.

#### **title**

An array of ASCII characters that represents the title information for the RealMedia file. This member is present on all Content Description Header objects with an `object_version` of 0 (zero). The size of this member is variable.

#### **author\_len**

The length of the author data in bytes. Note that the author data is not null-terminated. This member is present on all Content Description Header objects with an `object_version` of 0 (zero). The size of this member is 16 bits.

#### **author**

An array of ASCII characters that represents the author information for the RealMedia file. This member is present on all Content Description Header objects with an `object_version` of 0 (zero). The size of this member is variable.

#### **copyright\_len**

The length of the copyright data in bytes. Note that the copyright data is not null-terminated. This member is present on all Content Description Header objects with an `object_version` of 0 (zero). The size of this member is 16 bits

#### **copyright**

An array of ASCII characters that represents the copyright information for the RealMedia file. This member is present on all Content Description Header objects with an `object_version` of 0 (zero). The size of this member is variable.

#### **comment\_len**

The length of the comment data in bytes. Note that the comment data is not null-terminated. This member is present on all Content Description Header objects with an `object_version` of 0 (zero). The size of this member is 16 bits.

#### **comment**

An array of ASCII characters that represents the comment information for the RealMedia file. This member is present on all Content Description Header objects with an `object_version` of 0 (zero). The size of this member is variable.

### **9.11.12 Data Section**

The data section of the RealMedia file consists of a Data Section Header that describes the contents of the data section, followed by a series of interleaved media data packets. Note that the size field of the Data Chunk Header is the size of the entire data chunk, including the media data packets.

### **9.11.13 Data Chunk Header**

The Data Chunk Header marks the start of the data chunk. There is usually only one data chunk in a RealMedia file; however, for extremely large files, there may be multiple data chunks. The following pseudostructure describes the Data chunk header:

```
Data_Chunk_Header
{
    UINT32    object_id;
    UINT32    size;
    UINT16    object_version;
```

```

if (object_version == 0)
{
    UINT32    num_packets;
    UINT32    next_data_header;
}
}

```

The Data Chunk Header contains the following fields:

#### **object\_id**

The unique object ID for the Data Chunk Header ('DATA'). The size of this member is 32 bits.

#### **size**

The size of the Data Chunk in bytes. The size includes the size of the header plus the size of all the packets in the data chunk. The size of this member is 32 bits.

#### **object\_version**

The version of the Data Chunk Header object. The size of this member is 16 bits.

#### **num\_packets**

Number of packets in the data chunk. This member is present on all Data Chunk Header objects with an `object_version` of 0 (zero). The size of this member is 32 bits.

#### **next\_data\_header**

Offset from start of file to the next data chunk. A non-zero value refers to the file offset of the next data chunk. A value of zero means there are no more data chunks in this file. This field is not typically used. This member is present on all Data Chunk Header objects with an `object_version` of 0 (zero). The size of this member is 32 bits.

### 9.11.14 Data Packet Header

Following a data chunk header is `num_packet` data packets. These packets can all be from the same stream, or packets from different streams can follow one another. These packets, whether from the same stream or from different streams, should have an increasing value of timestamp. That is, the timestamp of a packet should be greater than or equal to the timestamp of the previous packet in the file.

The following pseudo-structure describes the details of the packet:

```

Media_Packet_Header
{
    UINT16          object_version;

    if ((object_version == 0) || (object_version == 1))
    {
        UINT16      length;
        UINT16      stream_number;
        UINT32      timestamp;
        if (object_version == 0)
        {
            UINT8      packet_group;
        }
    }
}

```

```

    UINT8      flags;
}

else if (object_version == 1)
{
    UINT16      asm_rule;
    UINT8      asm_flags;
}

UINT8[length]      data;
}
else
{
    StreamDone();
}
}

```

The Media Packet Header contains the following fields:

#### **object\_version**

The version of the Media Packet Header object. The size of this member is 16 bits.

#### **length**

The length of the packet in bytes. This member is present on all Media Packet Header objects with an `object_version` of 0 (zero) or 1. The size of this member is 16 bits.

#### **stream\_number**

The 16-bit alias used to associate data packets with their associated Media Properties Header. This member is present on all Media Packet Header objects with an `object_version` of 0 (zero) or 1. The size of this member is 16 bits.

#### **timeStamp**

The time stamp of the packet in milliseconds. This member is present on all Media Packet Header objects with an `object_version` of 0 (zero) or 1. The size of this member is 32 bits.

#### **packet\_group**

The packet group to which the packet belongs. If packet grouping is not used, set this field to 0 (zero). This member is present on all Media Packet Header objects with an `object_version` of 0 (zero). The size of this member is 8 bits.

#### **flags**

Flags describing the properties of the packet. The following flags are defined:

- HX\_RELIABLE\_FLAG

If this flag is set, the packet is delivered reliably.

- HX\_KEYFRAME\_FLAG

If this flag is set, the packet is part of a key frame or in some way marks a boundary in your data stream.

This member is present on all Media Packet Header objects with an `object_version` of 0 (zero). The size of this member is 8 bits.

#### **asm\_rule**

The ASM rule assigned to this packet. Only present if `object_version` equals 1. The size of this member is 16 bits.

#### **asm\_flags**

Contains `HX_` flags that dictate stream switching points. Only present if `object_version` equals 1. The size of this member is 8 bits.

#### **data**

The application-specific media data. This member is present on all Media Packet Header objects with an `object_version` of 0 (zero) or 1. The size of this member is variable.

### **9.11.15 Index Section**

The index section of the RealMedia file consists of a Index Chunk Header that describes the contents of the index section, followed by a series of index records. Note that the size field of the Index Chunk Header is the size of the entire index chunk, including the index records.

### **9.11.16 Index Section Header**

The Index Chunk Header marks the start of the index chunk. There is usually one index chunk per stream in a RealMedia file. The following pseudo-structure describes the Index chunk header.

```
Index_Chunk_Header
{
    u_int32    object_id;
    u_int32    size;
    u_int16    object_version
;

    if (object_version == 0)
    {
        u_int32    num_indices;
        u_int16    stream_number;
        u_int32    next_index_header;
    }
}
```

The Index Chunk Header contains the following fields:

#### **object\_id**

The unique object ID for the Index Chunk Header ("INDX"). The size of this member is 32 bits.

#### **size**

The size of the Index Chunk in bytes. The size of this member is 32 bits.

#### **object\_version**

The version of the Index Chunk Header object. The size of this member is 16 bits.

#### **num\_indices**

Number of index records in the index chunk. This member is present on all Index Chunk Header objects with an `object_version` of 0 (zero). The size of this member is 32 bits

#### **stream\_number**

The stream number for which the index records in this index chunk are associated. This member is present on all Index Chunk Header objects with an `object_version` of 0 (zero). The size of this member is 16 bits.

#### **next\_index\_header**

Offset from start of file to the next index chunk. This member enables RealMedia file format readers to find all the index chunks quickly. A value of zero for this member indicates there are no more index headers in this file. This member is present on all Index Chunk Header objects with an `object_version` of 0 (zero). The size of this member is 32 bits.

### 9.11.17 Index Record

The index section of a RealMedia file consists of a series of index record objects. Each index record contains information for quickly finding a packet of a particular time stamp for a physical stream. The following pseudo-structure describes the details of each index record:

```
IndexRecord
{
    UINT16    object_version;

    if (object_version == 0)
    {
        u_int32  timestamp;
        u_int32  offset;
        u_int32  packet_count_for_this_packet;
    }
}
```

An Index Record contains the following fields:

#### **object\_version**

The version of the Index Record object. The size of this member is 16 bits.

#### **timestamp**

The time stamp (in milliseconds) associated with this record. This member is present on all Index Record objects with an `object_version` of 0 (zero). The size of this member is 32 bits.

#### **offset**

The offset from the start of the file at which this packet can be found. This member is present on all Index Record objects with an `object_version` of 0 (zero). The size of this member is 32 bits.

### **packet\_count\_for\_this\_packet**

The packet number of the packet for this record. This is the same number of packets that would have been seen had the file been played from the beginning to this point. This member is present on all Index Record objects with an `object_version` of 0 (zero). The size of this member is 32 bits.

## 9.11.18 Metadata Section

The metadata section of the RealMedia file consists of a tag containing a set of named metadata properties that describe the media file. These properties can be text, integers, or any binary data. The tag is preceded by a header that identifies the size of the entire metadata section. Following the tag, the footer identifies the size of the tag. Since the metadata section is found at the end of the file, the footer can be used to expedite seeking backwards. At the end of the metadata section, and the file itself, is an ID3v1 tag.

### **9.11.19 Metadata Section Header**

The Metadata Section Header marks the start of the metadata section. There is one metadata section in a RealMedia file and it is expected to be at the end of the file. The following structure describes the Metadata section header:

```
MetadataSectionHeader
{
    u_int32      object_id;
    u_int32      size;
}
```

The Metadata Section Header contains the following fields:

#### **object\_id**

The unique object ID for the Metadata Section Header ("RMMD"). The size of this member is 32 bits.

#### **size**

The size of the full metadata section in bytes. The size of this member is 32 bits.

## 9.11.20 Metadata Tag

The metadata tag of a RealMedia file consists of a series of properties. The properties are represented as a tree hierarchy with one unnamed root property. Each property contains a type and value, as well as multiple (optional) sub-properties. The following structure describes the details of the metadata tag:

```
MetadataTag
{
    u_int32      object_id;
    u_int32      object_version;
    u_int8[]     properties;
}
```

The Metadata Tag contains the following fields:

#### **object\_id**

The unique object ID for the Metadata Tag ("RJMD"). The size of this member is 32 bits.

#### **object\_version**

The version of the Metadata Tag. The size of this member is 32 bits.

## **properties[]**

The `MetadataProperty` structure that makes up the metadata tag (see "["Metadata Property Structure"](#) for more details). As mentioned above, the properties will be represented as one unnamed root metadata property with multiple sub-properties, each with their own optional sub-properties. These will be nested, as in a tree.

### **9.11.21 Metadata Property Structure**

The following sample describes the details of the `MetadataProperty` structure:

```
MetadataProperty
{
    u_int32      size;
    u_int32      type;
    u_int32      flags;
    u_int32      value_offset;
    u_int32      subproperties_offset;
    u_int32      num_subproperties;
    u_int32      name_length;
    u_int8[name_length]  name;
    u_int32      value_length;
    u_int8[value_length]  value;
    PropListEntry[num_subproperties]  subproperties_list;
    MetadataProperty[num_subproperties]  subproperties;
}
```

The `MetadataProperty` structure contains the following fields:

#### **size**

The size of the `MetadataProperty` structure in bytes. The size of this member is 32 bits.

#### **type**

The type of the value data. The data in the value array can be one of the following types:

- MPT\_TEXT

The value is string data.

- MPT\_TEXTLIST

The value is a separated list of strings, separator specified as sub-property/type descriptor.

- MPT\_FLAG

The value is a boolean flag—either 1 byte or 4 bytes, check size value.

- MPT ULONG

The value is a four-byte integer.

- MPT\_BINARY

The value is a byte stream.

- MPT\_URL

The value is string data.

- MPT\_DATE

The value is a string representation of the date in the form:  
YYYYmmDDHHMMSS (m = month, M = minutes).

- MPT\_FILENAME

The value is string data.

- MPT\_GROUPING

This property has subproperties, but its own value is empty.

- MPT\_REFERENCE

The value is a large buffer of data, use sub-properties/type  
descriptors to identify mime-type.

The size of this member is 32 bits.

### flags

Flags describing the property. The following flags are defined these can be used in combination:

- MPT\_READONLY

Read only, cannot be modified.

- MPT\_PRIVATE

Private, do not expose to users.

- MPT\_TYPE\_DESCRIPTOR

Type descriptor used to further define type of value.

The size of this member is 32 bits.

#### **value\_offset**

The offset to the `value_length`, relative to the beginning of the `MetadataProperty` structure. The size of this member is 32 bits.

#### **subproperties\_offset**

The offset to the `subproperties_list`, relative to the beginning of the `MetadataProperty` structure. The size of this member is 32 bits.

#### **num\_subproperties**

The number of subproperties for this `MetadataProperty` structure. The size of this member is 32 bits.

#### **name\_length**

The length of the name data, including the null-terminator. The size of this member is 32 bits.

#### **name[]**

The name of the property (string data). The size of this member is designated by `name_length`.

#### **value\_length**

The length of the value data. The size of this member is 32 bits.

#### **value[]**

The value of the property (data depends on the type specified for the property). The size of this member is designated by `value_length`.

#### **subproperties\_list[]**

The list of `PropListEntry` structures. The `PropListEntry` structure identifies the offset for each property (see "["PropListEntry Structure"](#)" for more details. The size of this member is `num_subproperties * sizeof(PropListEntry)`.

#### **subproperties[]**

The sub-properties. Each sub-property is a `MetadataProperty` structure with its own size, name, value, sub-properties, and so on. The size of this member is variable.

### **9.11.22 PropListEntry Structure**

The following sample describes the details of the `PropListEntry` structure:

```
PropListEntry
{
    u_int32      offset;
    u_int32      num_props_for_name;
}
```

The `PropListEntry` structure contains the following fields:

#### **offset**

The offset for this indexed sub-property, relative to the beginning of the containing `MetadataProperty`. The size of this member is 32 bits.

#### **num\_props\_for\_name**

The number of sub-properties that share the same name. For example, a lyrics property could have multiple versions as differentiated by the language sub-property type

descriptor. The size of this member is 32 bits.

#### 9.11.23 Metadata Section Footer

The metadata section footer marks the end of the metadata section of a RealMedia file. The metadata section footer contains the size of the metadata tag. Since the metadata section is at the end of the file, the section footer lies a fixed offset of 140 bytes from the end of the file. The size of the metadata tag enables a file reader to quickly locate the beginning of the metadata tag relative to the end of the file. The following structure describes the Metadata section footer.

```
MetadataSectionFooter
{
    u_int32      object_id;
    u_int32      object_version;
    u_int32      size;
}
```

The MetadataSectionFooter contains the following fields:

##### **object\_id**

The unique object ID for the Metadata Section Footer ("RMJE"). The size of this member is 32 bits.

##### **object\_version**

The version of the metadata tag. The size of this member is 32 bits.

##### **size**

The size of the preceding metadata tag. The size of this member is 32 bits.

#### 9.11.24 ID3v1 Tag

The ID3v1 Tag is at the end of the metadata section and is expected to be at the end of the entire file. It is a fixed size—128 bytes—and begins with the characters "TAG". Further information about the informal ID3v1 standard can be found at <http://id3.org/id3v1.html>.

#### 9.12 Real Media file format

##### **DivX**



**Maintainer:** DivX, Inc.

**Latest release:** 6.3.2 / [September 19, 2006](#)

**OS:** Cross-platform

**Use:** [Media player](#) / [Codec](#) / [Media format](#)

**License:** proprietary

Website: [www.divx.com](http://www.divx.com)

**DivX** is a video [codec](#) created by [DivX, Inc.](#) (formerly DivXNetworks, Inc.), which has become popular due to its ability to [compress](#) lengthy video segments into small sizes while maintaining relatively high visual quality. DivX uses [lossy MPEG-4 Part 2](#) compression, where quality is balanced against [file size](#) for utility. It is one of several codecs commonly associated with [ripping](#), where [audio](#) and [video](#) multimedia are transferred to a [hard disk](#) and [transcoded](#). As a result, DivX has been a center of controversy because of its use in the replication and distribution of copyrighted [DVDs](#).

Many newer "DivX Certified" DVD players are able to play DivX encoded movies. However, "DivX" should not be confused with "DIVX", an unrelated attempt at a new [DVD](#) rental system employed by the US retailer [Circuit City](#). The winking [emoticon](#) in the early "DivX ;)" codec name was a [tongue-in-cheek](#) reference to the failed DIVX system.

## Contents

- [1History](#)
  - [1.1DivX and Spyware](#)
- [2Current Version](#)
- [3DivX Profiles](#)
- [4Quality, alternative MPEG-4 ASP implementations](#)

### 9.12.1 History

DivX ;-) 3.22 and earlier versions generally refer to a [hacked](#) version of the [Microsoft](#) MPEG-4 Version 3 video codec, extracted around [1998](#) by French hacker [Jerome Rota](#) (also known as Gej). The Microsoft codec, which originally required that the compressed output be put in an [ASF](#) file, was altered to allow other [containers](#) such as [AVI](#). From 1998 through 2002, independent enthusiasts within the DVD-ripping community created software tools which dramatically enhanced the quality of video files that the DivX ;-) 3.11 Alpha codec could produce. One notable tool is [Nandub](#), a modification of the open-source [VirtualDub](#), which features two-pass encoding (termed "[Smart Bitrate Control](#)" or SBC) as well as access to internal codec features.

In early 2000, Rota created a company (originally called DivXNetworks, Inc., renamed to DivX, Inc. in 2005) to improve DivX and steward its development. This effort resulted first in the release of the "OpenDivX" codec and source code on January 15, 2001. OpenDivX was hosted as an open source

project on the Project Mayo web site ([projectmayo.com](http://projectmayo.com)). The company's internal developers and some external developers worked jointly on OpenDivX for the next several months, but the project eventually stagnated. In early 2001, DivX employee "Sparky" wrote a new and improved version of the codec's encoding algorithm known as "encore2." This code was included in the OpenDivX public source repository for a brief time, but then was abruptly removed. The explanation from DivX at the time was that "the community really wants a Winamp, not a Linux." It was at this point that the project [forked](#). DivX took the encore2 code and developed it into DivX 4.0, initially released in July 2001. Other developers who had participated in OpenDivX took encore2 and started a new project—[XviD](#)—that started with the same encoding core. The company released a [clean room](#) version of the codec as DivX 4.0 in July 2001. DivX, Inc has since continued to develop the DivX codec, releasing DivX 5.0 in March 2002.



The latest generation, DivX 6, was released on [June 15, 2005](#) and expands the scope of DivX from being just a codec to including a full [media container format](#). DivX 6 introduces a new file format called "[DivX Media Format](#)" (with a [.divx](#) extension) that includes support for the following DVD-like features:

- Interactive video menus
- Multiple subtitles
- Multiple audio tracks
- Chapter points
- Other metadata
- Multiple format

While in previous generations video encoded with DivX was analogous to video formats such as MPEG-2, in its 6.0 generation the new DivX Media Format is analogous to media container formats such as Apple's QuickTime. In much the same way that media formats such as DVD specify MPEG-2 video as a part of their specification, the DivX Media Format specifies MPEG-4-compatible video as a part of its specification. However, despite the use of the ".divx" extension, this format is simply the [AVI](#) file format by another name. The methods of including multiple audio and even subtitle tracks involve storing the data in RIFF headers and other such AVI hacks which have been known for quite a while, such that even

[VirtualDubMod](#) supports them. Of course, the traditional method of creating standard AVI files is still supported.

### 9.12.1.1 DivX and [Spyware](#)

At one point, DivX Networks offered for download an "ad supported" version of their DivX Professional product free of charge to users who were willing to view advertisements. The ads in question were delivered by the notorious [Gator](#) adware software. While this attracted much criticism at the time, it should be noted that users had to manually select the "ad supported" download rather than the for-pay professional version or the free version. Additionally, users were informed during installation of the ad-supported version that the Gator software would be installed on their PC and were presented with a license agreement to which they had to consent in order to continue the installation. Unfortunately, the Gator software would still install parts of itself without the user agreeing to this installation, and was notoriously difficult to remove after installation; this raised considerable consternation amongst DivX users, causing many to turn to its [Open Source](#) rival, [XviD](#). The latter is freely available without installing adware and has been demonstrated in independent comparisons to produce better quality output (see section on Quality below).

Due to the generally hostile opinion towards spyware on the Internet, DivX Networks announced on the [DivX.com](#) web site that, from [July 15, 2004](#), no further DivX software would incorporate any adware [\[2\]](#). Free versions of DivX Pro before 5.2 typically contained spyware. From 5.2 onwards, including version 6, no spyware was included. When accessed in March 2006, the Professional version of DivX was only available in the form of a paid release or a 6-month free trial with no adware included.

### 9.12.2 Current Version

The current version of the DivX Community Codec, which is the same for all platforms, is version 6.4, available from [DivX.com](#). The latest version of the DivX package for [Windows 2000/XP](#) (which contains DivX Player 6.4.?, DivX Community Codec 6.2.5, and DivX Web Player 1.0) is version 6.4, and the latest version of the DivX package for [Mac OS X](#) (which contains DivX Player 1.0.3, DivX Community Codec 6.2.5, and DivX Web Player 1.0.2) is version 6.5, released [May 25, 2006](#). In addition, an unofficial DivX for [Linux](#) codec update has also been [released at version 6.1.1](#). The DivX codec and [DivX Player](#) are available for free at the DivX website. Paying customers can access additional features of the DivX codec in the registered version, known as [DivX Pro](#), and can also use [DivX Converter](#), a

one-click encoding application as a revamp of [Dr. DivX](#) and associated encoding tools (such as the Electrokompressiongraph™, or EKG, which helped increase the viewability of highly compressed high-motion scenes). The latest version of DivX Converter for Windows is 6.2, and the latest version of DivX Converter for Mac is 1.1.1.

Recently DivX have also released the [DivX Web Player 1.0.1](#) (formerly known as the DivX Browser Plug-In Beta) via the [DivX Labs](#) website, demonstrating 720p HD playback live inside major browsers for Windows and Mac OS. [Dr DivX 2 OSS](#), an Open Source DivX transcoding application, is available from [SourceForge](#).

An open source version of the codec—called [OpenDivX®](#)—was released by DivX in early 2001, and this version served as the basis for the open source [XviD](#) codec, the source code of which is maintained by an independent group.

The main competitors in the for-license video compression software market are Adobe's Flash 8 (using ON2's VP6), Microsoft's [Windows Media Video](#) series, [Apple Computer's QuickTime](#), and the [RealNetworks RealVideo](#) series.

### 9.12.3 DivX Profiles

To group the various MPEG-4 options in a different way than what is specified in the MPEG-4 standard itself to create a DivX-specific device certification process for device manufactureres, DivX has defined many profiles. These are sets of MPEG-4 features as determined by DivX. [1] DivX's profiles differ from the standardized profiles of the ISO/IEC MPEG-4 international standard.

		Profiles		
		Handheld	Portable	Home Theater
<b>Supports</b>	<b>all DivX 5.xx only</b>	<b>DivX 4.xx &amp; 5.xx</b>	<b>720 x 480</b>	<b>DivX 3.11, 4.xx &amp;</b>
<b>resolutions</b>	<b>up to 176 x 144 @ maximum of 15 fps</b>	<b>@ 30 fps; 720 x 576 @ 25 fps</b>	<b>5.xx 720 x 480 @ 30 fps</b>	<b>DivX 4.xx &amp; 5.xx 1280 x 720 @ 30 fps</b>
<b>Macroblocks per second</b>	<b>1485</b>	<b>DivX 4.xx &amp; 5.xx 40500</b>	<b>DivX 3.11 9900</b>	<b>40500 108000</b>
<b>Maximum average bitrate</b>	<b>200 kbps</b>	<b>DivX 4.xx &amp; 5.xx 4000 kbps</b>	<b>DivX 3.11 768 kbps</b>	<b>4000 kbps</b>

<b>Maximum peak bitrate during any 1 second of video</b>	400 kbps	<b>DivX 4.xx &amp; 5.xx</b>	8000 kbps	8000 kbps	20000 kbps
<b>DivX 3.11</b>	2000 kbps				
<b>Minimum VBV buffer size (Kilobytes)</b>	32k bytes	<b>DivX 4.xx &amp; 5.xx</b>	384k bytes	384k bytes	768k bytes
		<b>DivX 3.11</b>	128k bytes		

## 9.12.4 Quality, alternative MPEG-4 ASP implementations

While DivX has long been renowned for its excellent video quality, its open-source equivalent [XviD](#), also based on MPEG-4 Part 2, now offers comparable quality. In a series of [subjective quality tests](#) at [Doom9.org](#), the DivX codec was beaten by XviD in the [2003](#), [2004](#), and [2005](#) tests.

The open source library [libavcodec](#) can decode MPEG-4 video encoded with DivX (and other MPEG-4 codecs, such as XviD or libavcodec MPEG-4). Combined with image postprocessing code from the [MPlayer](#) project, it has been packaged into a [DirectShow](#) decoding filter called [ffdshow](#), which can be used with most Windows [video players](#) and reportedly achieves higher image quality while generating less CPU load than the DivX codec<sup>[2]</sup>.

Since the standardization of [H.264/MPEG-4 AVC](#), also known as MPEG-4 Part 10, a new generation of codecs has arisen, such as [x264](#). Despite being at a relatively early stage of development, these codecs out-performed DivX in Doom9's 2005 quality test, thanks to the more advanced features of MPEG-4 Part 10. Part 10's advanced features come at a cost: they are two to three times more CPU intensive than the relatively lightweight algorithms used in the DivX codec. It remains to be seen whether DivX will release a new codec based on the newer specification, like the XviD team have with their XviD AVC codec.

## 10 文本编码

### 10.1 ASCII

#### 概述

ASCII 是 American Standard Code for Information Interchange (美国标准信息交换代码) 的简称, 是目前计算机中用得最广泛的字符集及其编码, 是由美国国家标准局(ANSI)制定的 ASCII 码, 它已被国际标准化组织 (ISO) 定为国际标准, 称为 ISO 646 标准。适用于所有拉丁文字字母, ASCII 码有 7 位码和 8 位码两种形式。

## 基本原理

因为 1 位二进制数可以表示 2 种状态：0、1；而 2 位二进制数可以表示 4 种状态：00、01、10、11；依次类推，7 位二进制数可以表示 128 种状态，每种状态都唯一地编为一个 7 位的二进制码，对应一个字符（或控制码），这些码可以排列成一个十进制序号 0~127。所以 7 位 ASCII 码是用七位二进制数进行编码的，可以表示 128 个字符。

第 0~32 号及第 127 号（共 34 个）是控制字符或通讯专用字符，如控制符：LF（换行）、CR（回车）、FF（换页）、DEL（删除）、BEL（振铃）等；和通讯专用字符：SOH（文头）、EOT（文尾）、ACK（确认）等；

第 33~126 号（共 94 个）是字符，其中第 48~57 号为 0~9 十个阿拉伯数字；65~90 号为 26 个大写英文字母，97~122 号为 26 个小写英文字母，其余为一些标点符号、运算符号等。

注意：在计算机的存储单元中，一个 ASCII 码值占一个字节（8 个二进制位），其最高位（b7）用作奇偶校验位。所谓奇偶校验，是指在代码传送过程中用来检验是否出现错误的一种方法，一般分奇校验和偶校验两种。奇校验规定：正确的代码一个字节中 1 的个数必须是奇数，若非奇数，则在最高位 b7 添 1；偶校验规定：正确的代码一个字节中 1 的个数必须是偶数，若非偶数，则在最高位 b7 添 1。

## 优缺点

### 优点

- 支持广泛，实现简单
- 字符集比较小。

### 缺点

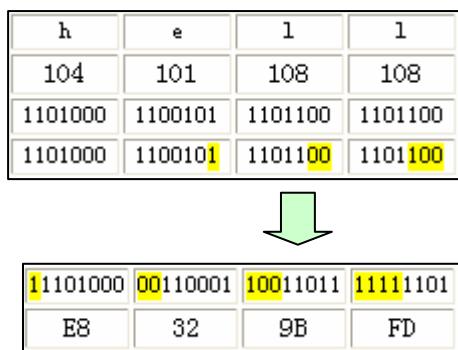
- 不支持汉字。

## 10.2 GSM7bit

### 概述

GSM 03.38 7-bit，是由 ETSI (European Telecommunications Standards Institute) 欧洲通讯规格机关/欧洲电子通讯标准化委员会制定。通讯标准化团体规定为 SMS/SIM 的编码。

### 基本原理



首先将字符转换为 7 位的二进制，然后，将后面字符的位调用到前面，补齐前面的差别。例如：h 翻译成 1101000，e 翻译成 1100101，显然 h 的二进制编码不足八位，那么就将 e 的最后一位补足到 h 的前面。那么就成了 11101000（E8）。

## 优缺点

优点:

- GSM 的短消息业务中, 在使用 7bit 编码时, 140 个字节的长度可包含 160 个字符

## 10.3 UCS-4 和 ISO 10646

### 概述

为容纳全世界各种语言的字符和符号, ISO 的一些会员国于 1984 年发起制定新的国际字符集编码标准。新标准由工作小组 JTC1/SC2/WG2 负责拟订 (以下简称 WG2) 最后定案的标准, 命名为 “Universal Multiple-Octet Coded Character Set” 简称 UCS, 其编号则订为 ISO/IEC 10646。UCS-4 是 ISO10646 字符码的正规形式。

### 原理

ISO10646 字符码的正规形式 (可简称为 UCS-4) 为 32 个位, 划分成 4 个八位。这 4 个八位, 由左而右命名为群八位 (G-octet)、面八位 (P-octet)、列八位 (R-octet) 和格八位 (C-octet), 分别代表编码结构中的群组 (group)、字面 (plane)、列 (row) 与格 (cell)。ISO10646 规定其字符码的 b32 必须为 0, 因而整个编码空间可区分为 128 个群组 (群八位的值为 00~7Fh), 每一群组由 256 个字面所组成 (面八位元为 00~FFh), 每一个字面由 256 列所组成 (列八位为 00~FFh), 每一列则包含 256 格 (格八位为 00~FFh), 为一个编码位置。除此之外, ISO10646 还规定每一个字面的最后两个编码位置 FFFEh 和 FFFFh, 保留不用。所以, ISO10646 整个编码空间总共  $256 \times 128 = 32,768$  个字面, 每个字面为  $256 \times 256 - 2 = 65,534$  个编码位置, 合计  $65534 \times 32768 = 2,147,418,112$  个编码位置。ISO10646 的第 0 群组第 0 字面 (群八位和面八位的值都为 00h) 称为 “基本多语文字面” (Basic Multi-lingual Plane, BMP), 其编码字符与 Unicode 相同。BMP 之外的 32,767 个字面区分为辅助字面 (supplementary planes) 和专用字面 (private use planes)。辅助字面用以收容 WG2 陆续收集、整理和编码的各国语文字元; 专用字面的内容 WG2 不予规定, 保留供使用者自行添加 ISO10646 未收容的字符。专用字面共 8,226 个, 包括 00h 群组的 0Fh、10h 和 E0~FFh 共计 34 个字面, 以及 60~7Fh 共 32 个群组的 8,192 个字面。除了这 8,226 个专用字面之外, 其余的 24,541 个字面都是辅助字面。

## 优缺点

优点:

- UCS-4 是所有其他字符集标准的一个超集, 它保证与其他字符集是双向兼容的。UCS-4 包含了用于表达所有已知语言的字符。

缺点

- 过分侵占存储空间并影响信息传输的效率

## 10.4 Unicode 和 UCS-2

### 概述

1988 年初, 美国 Xerox 公司的 Joe Becker 倡议以新的编码结构另外编订世界性字符编码标准: 将计算机字符集编码的基本单位由现行的 7 或 8 个位一举扩充为 16 个位, 并且充分利用 65,536 个编码位置以容纳全世界各种语言的字符和常用符号。新的字符集编码标准被命名为 “Unicode” (Universal Multiple Octet Coded Character Set)。一些来自 Xerox 公司和 Apple 公司的工程师组成工作小组, 负责 Unicode 的原始设计工作。1991 年元月, 十多家计算机硬软件、网络和信息服务业者, 包括: IBM、DEC、Sun、Xerox、Apple、MicroSoft、Novell 名公司, 共同出资成立 Unicode

协会 (The Unicode Consortium)，并由协会设立非营利的 Unicode 公司。

Unicode 草案第一版于 1989 年 9 月发表，历经多次修订后，分别于 1991、92 年出版了 Unicode 标准第一版 (The Unicode Standard, version 1.0) 的第一、第二册。由于 Unicode 协会持续的游说和施压，WG2 终于放弃原先选择的 ISO2022 八位延伸编码结构，改采 Unicode 的编码方式。1991 年 10 月，历经几个月的协商之后，WG2 和 Unicode 协会达成协议，将 Unicode 并入 ISO10646 成为第 0 字面。

## 基本原理

当计算机系统只使用 BMP 的字符码时，可以省略群八位和面八位元，因而可将字符码由 32 个位缩短为 16 个位，称为 ISO10646 字符码的基本面形式(可简称为 UCS-2)，其实也可视同于 Unicode。

ISO10646 所有字面中，目前仅有第 0、第 1 和第 2 字面真正收容编码字符。ISO10646 的 BMP 和 Unicode 的编码字符，依其 UCS-2 编码序介绍如下：

(1) 0000~007Fh：基本拉丁字母区。其中 0000~001Fh 为 C0 控制码，0020h 为空格 (space)，0021~007Eh 为 ASCII 图形字符，007Fh 为句柄 DEL。事实上，这 128 个字符只要把前 8 个位去掉就可变成习见的 8 位形式的 ASCII 码。

(2) 0080~00A0h：句柄区。其中 0080~009Fh 为 C1 句柄，00A0h 为不中断空格 (no-break space)。

(3) 00A1~1FFFh：拼音文字区。收容除基本拉丁字母以外的各种 拼音文字字符，包括欧洲各国语言、希腊文、斯拉夫语文、希伯来文、阿拉伯文、亚美尼亚文、印度各地方言、马来文、泰文、寮文、柬埔寨文、满文、蒙文、藏文、印地安语文等。

(4) 2000~28FFh：符号区。收容各种符号，包括标点符号、上下 标、钱币符号、数字、箭头、数学符号、工程符号、光学辨识符号、带圈或带括符的文数字、表格绘制符号、地理图标、盲用点字、装饰图形等。

(5) 2E80~33FFh：中日韩符号区。收容康熙字典部首、中日韩辅 助部首、注音符号、日本假名、韩文音符，中日韩的符号、标点、带圈或带括符文数字、月份，以及日本的假名组合、单位、年号、月份、日期、时间等。

(6) 3400~4DFFh：中日韩认同表意文字扩充 A 区，总计收容 6,582 个中日韩汉字。

(7) 4E00~9FFFh：中日韩认同表意文字区，总计收容 20,902 个中 日韩汉字。

(8) A000~A4FFh：彝族文字区，收容中国南方彝族文字和字根。

(9) AC00~D7FFh：韩文拼音组合字区，收容以韩文音符拼成的文 字。

(10) D800~DFFFh：S 区，专用于 UTF-16，详情后叙。

(11) E000~F8FFh：专用字区，其内容 WG2 不予规定，保留供使用者自行添加 ISO10646 未收容的字符。

(12) F900~FAFFh：中日韩兼容表意文字区，总计收容 302 个中日 韩汉字。何谓兼容表意文字，留待后叙。

(13) FB00~FFFFh：文字表现形式区，收容组合拉丁文字、希伯来 文、阿拉伯文、中日韩直式标点、小符号、半角符号、全角符号等。

## 优缺点

### 优点

- ◆ UCS2 的一个优点是它仅用两个字节表示。
- ◆ Unicode 可表示所有语言，在 XML、WML、Java 和 CORBA 等技术中广泛使用，为许多操作系统和浏览器所支持。

## 10.5 UTF-8

### 概述

由于历史原因，当某个以 Unicode 为内码的操作系统或应用程序将 Unicode 编码文件传出之后，马上会被网络设备切成两个 8 位的字节，这可能导致至少数千个 Unicode 将因此变得残缺不全。为了能让 Unicode 和 ISO10646 字符码能安然通行，Unicode 协会特别提出 UTF-8 以解决问题。UTF-8 意即把原为 32 位的 ISO10646 元码或原为 16 位的 Unicode 转换为多个 8 位的字节。

### 基本原理

利用 UTF-8 转码规则可将一个 Unicode 或 ISO10646 字符码转换成 1~4 个字节的编码。UTF-8 的转换规则很简单：若原始字符码位于 0000~007Fh（或 0000-0000~0000-007Fh）的范围内，则直接截取最右 8 位即可，转换结果其实就是 ASCII 码。若原始字符码大于 007Fh（或 0000-007Fh），亦即超出 ASCII 范围时，就必须转换成 2~4 个字节。UTF-8 规定，以转换后第 1 字节起头连续设为“1”的标记位的数目表示转换成几个字节：110 表示转换结果为 2 个位元组，1110 表示 3 个字节，而 11110 则表示 4 个字节。至于跟随在标记位之后设为 0 的位，其在分隔标记位和字符码位。第 2~第 4 字节起头两个位被设为 10 当做识别，剩下的 6 个位才做为字符码位使用。总计，2 字节 UTF-8 码剩下 11 个字符码位，可用以转换 0080~07FFh 的原始字符码。3 字节剩下 16 个字符码位，可用以转换 0800~FFFFh 的原始字符码。而 4 字节则剩下 21 个字符码位，可用来转换 0001-0000~001F-FFFh（即 ISO10646 第 1~第 31 字面）的原始字元码。请注意，虽然 4 个字节的 UTF-8 编码可包容 1~3 个字节的码，3 个字节的编码可包容 1~2 个字节的码，以及 2 个字节的编码可包容 1 个字节的码，但是 UTF-8 规定转码时必须选择最短者。换言之，ASCII 区只能转换成单一位，0080~07FFh 的原始字符码只能转换成 2 个位组长度，依此类推。

特点如下：

- UCS 字符 U+0000 到 U+007F (ASCII) 被编码为字节 0x00 到 0x7F (ASCII 兼容). 这意味着只包含 7 位 ASCII 字符的文件在 ASCII 和 UTF-8 两种编码方式下是一样的.
- 所有 >U+007F 的 UCS 字符被编码为一个多个字节的串，每个字节都有标记位集. 因此，ASCII 字节 (0x00-0x7F) 不可能作为任何其他字符的一部分.
- 表示非 ASCII 字符的多字节串的第一个字节总是在 0xC0 到 0xFD 的范围里，并指出这个字符包含多少个字节. 多字节串的其余字节都在 0x80 到 0xBF 范围里. 这使得重新同步非常容易，并使编码无国界，且很少受丢失字节的影响.
- 可以编入所有可能的 231 个 UCS 代码
- UTF-8 编码字符理论上可以最多到 6 个字节长，然而 16 位 BMP 字符最多只用到 3 字节长.

下列字节串用来表示一个字符. 用到哪个串取决于该字符在 Unicode 中的序号.

U-00000000 ~ U-0000007F:	0xxxxxxx
U-00000080 ~ U-000007FF:	110xxxxx 10xxxxxx
U-00000800 ~ U-0000FFFF:	1110xxxx 10xxxxxx 10xxxxxx
U-00010000 ~ U-001FFFFF:	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
U-00200000 ~ U-03FFFFFF:	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
U-04000000 ~ U-7FFFFFFF:	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

xxx 的位置由字符编码数的二进制表示的位填入. 越靠右的 x 具有越少的特殊意义. 只用最短

的那个足够表达一个字符编码数的多字节串。注意在多字节串中，第一个字节的开头“1”的数目就是整个串中字节的数目。

例如：Unicode 字符 U+00A9 = 1010 1001（版权符号）在 UTF-8 里的编码为：11000010 10101001 = 0xC2 0xA9 而字符 U+2260 = 0010 0010 0110 0000（不等于）编码为：11100010 10001001 10100000 = 0xE2 0x89 0xA0 这种编码的官方名字拼写为 UTF-8，其中 UTF 代表 UCS Transformation Format。请勿在任何文档中用其他名字（比如 utf8 或 UTF\_8）来表示 UTF-8，当然除非你指的是一个变量名而不是这种编码本身。

## 优缺点

优点：

- UTF8 很适用于 HTML 等协议和网络上须慢速传送的文件。
- UTF-8 是 Unicode 的一种可变长度编码形式，它透明地保留了 ASCII 字符代码值。
- UTF-8 与 ASCII 完全兼容，现有软件不经重写可处理用 UTF-8 表示的与 ASCII 兼容的 Unicode 字符。

缺点

- 需要在支持 Unicode 的平台上才能够采用

## 10.6 UTF-16

### 概述

ISO10646 的编码空间足以容纳古今人类使用过的所有文字和符号，但目前真正被使用的文字或符号，绝大多数都已编入 BMP，它们的使用频率可能超过 99%，甚至 99.99%。换言之，就 99%以上的使用者或使用场合而言，16 位的 Unicode 已是足敷需求，32 位的 ISO10646 正规编码则显得有些冗余。32 位编码不要比 16 位编码占用多一倍的储存空间，而且在网络传输和信息处理上也需花费比较长的时间。就经济效益而言，将来计算机和网络选择使用 Unicode 的可能性，很明显地要高于选择使用 32 位的 ISO10646 正规编码。问题是在 Unicode 的世界里，遇到必须使用第 1、第 2 字面甚至更后方字面的字符时，怎么办？Unicode 协会提出的解决方式称为 UTF-16 或代理法（surrogates）。UTF 为“a UCS (or Unicode) Transformation Format”的缩写，UTF-16 意即把原为 32 位的 ISO10646 字符码转换为 2 或多个 16 位的 Unicode。

### 基本原理

目前的作法是组合两个 Unicode 字符码来代表一个 ISO10646 字符，所以又称为代表法。两个做为代表的 Unicode，位于前方（左方）者称为高半字符，限定只能选用 D800~DBFFh 当中之一，位于后方（右方）者称为低半字符，限制只可从 DC00~DFFFh 当中选择。高低半字符的编码位置各为  $1,024=4\times256$ ，因此 UTF-16 总计可提供  $4\times256\times(4\times256)=16\times65536$  个编码位置，亦即 16 个字面。对 BMP 而言，当然无需使用 UTF-16 转码，所以 UTF-16 主要应用于 ISO10646 的第 1~第 14 字面（因为第 15 字面为专用字面，WG2 不予编码）。将 ISO10646 字符（编码范围 0001~0000~000E~FFFFh）以 UTF-16 转换成 Unicode 组合形式的规则方式不难，将原来的 32 位 ISO10646 字符码，从右往左取出 10 个位前头附加上特定的 6 个位 110111 即成为低半字符。接着往左取出 6 个位（即图中的 Y16~Y11）做为高半字符的最低 6 个位，然后再往左取出 4 个位，将其值减 1，置于刚才那 6 个位的左方，最后在最前方附加上特定的 6 个位 110110，即成为高半字符。

### 优缺点

优点

- UTF-16 编码紧凑
  - 常用字符可用单 16 位单元表示
- 缺点
- 需要在支持 Unicode 的平台上才能够采用

## 10.7 GB2312

### 概述

全称是 GB2312-80《信息交换用汉字编码字符集 基本集》，1980 年发布是中文信息处理的国家标准，在大陆及海外使用简体中文的地区（如新加坡等）是强制使用的唯一中文编码。P-Windows3.2 和苹果 OS 就是以 GB2312 为基本汉字编码，Windows 95/98 则以 GBK 为基本汉字编码、但兼容支持 GB2312。该标准的制定和应用为规范、推动中文信息化进程起了很大作用。

### 原理

- 为高低两字节编码方式，每个字节 7 位编码。  
范围：A1A1~FEFE  
A1-A9：符号区，包含 682 个符号  
B0-F7：汉字区，包含 6763 个汉字
- 遵循 ISO 2022 的 2Byte 标准
- 收录简化汉字及一般符号、序号、数字、拉丁字母、日文假名、希腊字母、俄文字母、汉语拼音符号、汉语注音字母，共 7445 个图形字符。

### 优缺点

- 优点
- 编码简单，占用空间小
  - 比较通用
- 缺点
- 收集汉字和其他符号数量少，无繁体字

## 10.8 GB 12345

### 概述

1990 年制定了繁体字的编码标准 GB12345-90《信息交换用汉字编码字符集 第一辅助集》，目的在于规范必须使用繁体字的各种场合，以及古籍整理等。

### 基本原理

双字节编码，范围：A1A1~FEFE，A1-A9：增加竖排符号，包含 6866 个汉字。原则上，本字符集是将 GB 2312 中的简化字用相应的繁体字替换而成，纯繁体的字大概有 2200 余个，比 GB2312 多 103 个字，其它厂商的字库大多不包括这些字。

### 优缺点

- 优点
- 编码简单，占用空间小

### 缺点

- 通用性差，应用范围窄

## 10.9 GBK

### 概述

GBK (Guojia Biaozhun Kuozhan Chinese Internal Code Specification), 《汉字内码扩展规范》由全国信标委制定发布，等同于 UCS 的新的中文编码扩展国家标准。GBK 工作小组于 1995 年 12 月完成 GBK 规范。该编码标准兼容 GB2312，共收录汉字 21003 个、符号 883 个，并提供 1894 个造字码位，简、繁体字融于一库。

Windows95/98 简体中文版的字库表层编码就采用的是 GBK，通过 GBK 与 UCS 之间一一对应的码表与底层字库联系。

### 基本原理

GBK 采用了 2Byte 编码。在 Unicode/GB13000.1 发布之前，GB2312 已被广泛的应用。GBK 维持了 GB13000.1 和 GB2312 的兼容性。GBK 规范收录了 ISO 10646.1 中的全部中日韩汉字和符号，并有所补充。

### 优缺点

#### 优点

- 简、繁体字融于一库。
- Windows95/98 简体中文版的字库表层编码就采用的是 GBK。

#### 缺点

- 一些中文搜索引擎不支持，将被 GB/T18030 替代。

## 10.10 GB 18030

### 概述

GB 18030-2000，《信息技术 信息交换用汉字编码字符集》由国家技术监督局于 2000 年 3 月 17 日发布。与国家标准 GB 2312 信息处理交换码所对应的事上的内码标准兼容。

#### 原理

对现行的 GB2312 和 GB13000.1 标准的扩充，GB18030 收录了与金融服务行业及互联网有关的汉字。

采用单字节、双字节和四字节三种方式对字符编码。收录了 27484 个汉字，总编码空间超过 150 万个码位，在字汇上支持 GB 13000.1 的全部中、日、韩(CJK)统一汉字字符和全部 CJK 扩充的字符。

### 优缺点

#### 优点

- 全面兼容 GB2312，兼容性、扩展性、前瞻性好。
- 在技术上是 GBK 的超集，与其兼容，将全面替代 GBK。

## 10.11 BIG5

### 概述

参考标准为《通用汉字标准交换码》CISCII，1983年10月，台湾“国家科学委员会”等单位共同制定了《通用汉字标准交换码》(Chinese Ideographic Standard Code for Information Interchange, CISCII 码)，BIG-5 码是 1984 年台湾资讯工业策进会根据《通用汉字标准交换码》制订的编码方案。

### 基本原理

双字节编码方案，其第一字节的值在 16 进制的 A0~FE 之间，第二字节在 40~7E 和 A1~FE 之间，其第一字节的最高位是 1，第二字节的最高位则可能是 1，也可能是 0。现在流行使用的 BIG-5 一般收录 13468 个汉字和符号。例：F9DA 为“恒”。

### 优缺点

#### 优点

- 事实上的中文繁体业界标准

## 10.12 总结和比较表格

名称	优点	缺点
ASCII	它已被国际标准化组织 (ISO) 定为国际标准，称为 ISO 646 标准 支持广泛，实现简单 字符集比较小。	-
GSM-7bit	通讯标准化团体规定为 SMS/SIM 的编码。 GSM 的短消息业务中，在使用 7bit 编码时，140 个字节的长度可包含 160 个字符	不支持汉字
UCS-4 和 ISO 10646	UCS-4 是所有其他字符集标准的一个超集，它保证与其他字符集是双向兼容的。UCS-4 包含了用于表达所有已知语言的字符。	过分侵占存储空间并影响信息传输的效率
UCS-2 和 Unicode	UCS2 的一个优点是它仅用两个字节表示。 Unicode 可表示所有语言，在 XML、WML、Java 和 CORBA 等技术中广泛使用，为许多操作系统和浏览器所支持。	-
UTF-8	UTF8 很适用于 HTML 等协议和网络上须慢速传送的文件。 UTF-8 是 Unicode 的一种可变长度编码形式，它透明地保留了 ASCII 字符代码值 UTF-8 与 ASCII 完全兼容，现有软件不经重写可处理用 UTF-8 表示的与 ASCII 兼容的 Unicode 字符。	需要在支持 Unicode 的平台上才能够采用
UTF-16	UTF-16 编码紧凑，常用字符可用单 16 位单元表示	需要在支持 Unicode 的平台上才能够采用

GB2312	比较通用 编码简单，占用空间小	收集汉字和其他符号数量少，无繁体字
GB12345	编码简单，占用空间小	通用性差，应用范围窄
GBK	简、繁体字融于一库。 Windows95/98 简体中文版的字库表层编码就采用的是 GBK	一些中文搜索引擎不支持，将被 GB/T18030 替代
GB 18030	全面兼容 GB2312，可平滑切换至 GB13000.1，兼容性、扩展性、前瞻性好。 在技术上是 GBK 的超集，与其兼容，将结束 GBK 的历史使命。	-
BIG5	事实上的中文繁体业界标准	-