数字媒体技术揭秘

一、史话

如果算上模拟时代，多媒体传输也算不上是多么新鲜的事情。实际上，早在上世纪三十年代，人们便可以在家观赏奥运赛事：来自柏林现场的活动画面连同声音通过电缆或者无线电波被传送到世界各地[1)]，虽然图像还不是彩色的，但就质量来说并不见得就比YouTube上NBC的北京2008差。从某种意义上讲，数字技术的突飞猛进对多媒体通信的推动并非它能够在多大程度上提高媒体内容的质量——这方面艺术家们所起到的作用可能会更大——而是它可以令媒体的传播更便捷、更便宜，于是，如果愿意的话，现今每一个人都可以演一出好戏发到世界各个角落里去吸引眼球，就如上世纪三十年代那位元首所做的一样。

在模拟时代，最大的一个大难题是如何有效利用带宽。这里的带宽可以理解为传输媒介，对于以电磁信号为介质的传输方式，通常指的是一个频率范围。一般情况下每一路电视节目要占据6~8MHz的无线频谱或者电缆频谱，而且即使是在空闲的时候，这些被占据的频谱也无法被释放，有些电视台只好传送几根呆滞的彩条（三十年前出生的人几乎都有过观赏彩条的经历）。相对而言，引入交换机制的电话网看上去会好一点：电信局确然可以保证成千上万对用户同时进行通话，可是分给每对用户的只有可怜兮兮的4KHz带宽，而且，在双方沉默无语的时候，这些带宽并不能够被挪作他用，结果，人们在讲电话的时候总是竭力地寻找话题填满所有的通话时间，以免因产生的空闲而支付昂贵的电话费用。

上世纪四十年代末，香农提出了关于信息及其传送的关键理论 [http://en.wikipedia.org/wiki/A_Mathematical_Theory_of_Communication]，于是人们发现，对于给定的带宽，其理论上的传输能力要比当时通信系统实际所实现的大得多。然而，在模拟的范畴内，无论是从信源的角度还是从信道的角度，靠近香农极限都是一件相当困难的事情。

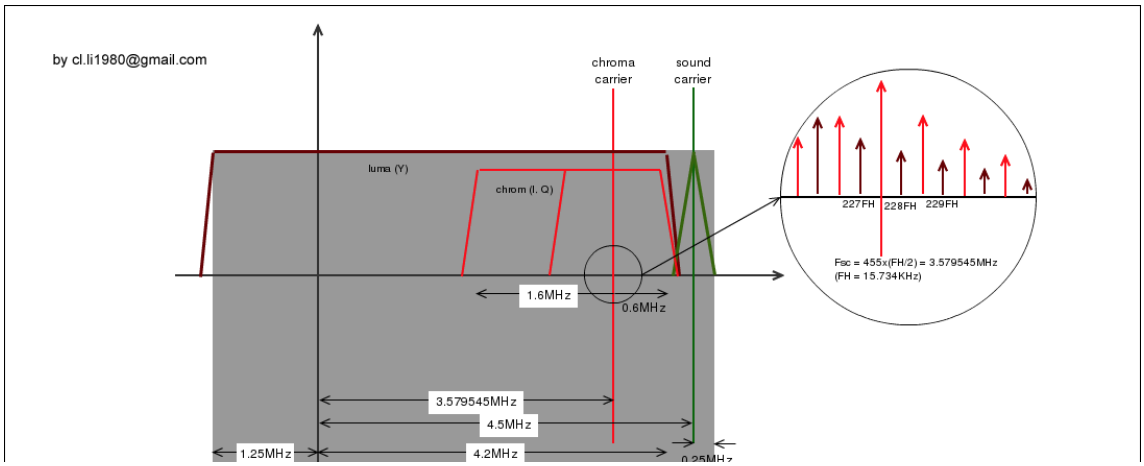
不妨以广播电视为例，回顾一下先前的人们为节省带宽而付出的努力：

1. 残留边带调制 [http://en.wikipedia.org/wiki/Vestigial_sideband#Vestigial_sideband_.28VSB.29]：图像信号的下边带只保留了少部分带宽（N制中是1.25MHz）。
2. 亮色混合：由于扫描产生的图像信号具有某种程度上的周期性，在频谱上则表现为以行扫描频率为间隔的众多脉冲，这使得亮度信号和色差信号在同一带宽范围内进行叠加成为可能，条件是它们在频域相互错开半个行扫描频率。换句话说，人们利用一种巧妙的方法在不增加带宽的前提下把黑白电视升级为彩色电视。
3. 隔行扫描：老式的CRT显示器要求刷新频率大于50Hz，否则会让人感觉到闪烁，但人眼感知运动画面的最低要求仅为每秒钟二十余帧左右，为了在不增加冗余的带宽（亦即不增加帧频）的前提下消除闪烁现象，人们引入了隔行扫描技术，即通过将一幅图像拆作相互交错的两场来实现二倍于帧频的刷新频率。（利用 frame buffer将同一副图像连续显示多次其实亦可解决这个问题，可惜在模拟电视时代该方法的实现具有一定的技术难度）。

计算机专业出身的同学可能对上述历史并不很熟悉，然而不可否认的是，尽管多媒体技术通常被认为是计算机学科的一个分支，其源起却在通信领域。在ITU的学僚们开始筹划将电信网络改造为能够传送"综合业务"的多媒体网络时，计算机还仅仅被认为是一种计算机器。

最先被数字化的是语音。早在上世纪六十年代，贝尔实验室的PCM技术便开始被应用于电话网。通过简单的采样和量化，这种技术将4KHz带宽的语音转换为64Kbps的数字信息，使之能够在采用时分复用的电话干线中传输。有趣的是，当时，普通的电话双绞线反而无法传输这种数字语音——64Kbps的数据率对于它们来说太大了。难道数字媒体要求更多的带宽？并非如此，从香农的理论出发很容易找出问题所在：其一，PCM技术所产生的信息速率大大超出了语音信息本身的熵率——这是种很粗糙的编码技术；其二，33.6kbps的调制技术远远没有达到双绞线可以提供的传输极限。结论就是，数字化并非就是简单的采样和量化，它是一个复杂的信息表示过程：既包括信源编码——在保证信息可恢复的前提下产生尽量少的数据速率；也包括信道编码——在给定带宽的前提下如何承载更大的数据速率。对于多媒体信息来说，前者更是备受关注。譬如，同样是4KHz的语音，如果采用新的编码技术如G.723.1，产生的数据速率只有5~6kbps，对于双绞线来说绰绰有余。

图1. NTSC频谱结构：



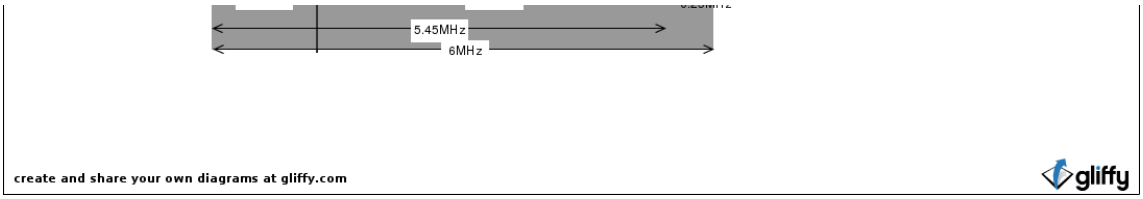


图2. 一个L1载波 [http://en.wikipedia.org/wiki/L-carrier]的频谱分配：

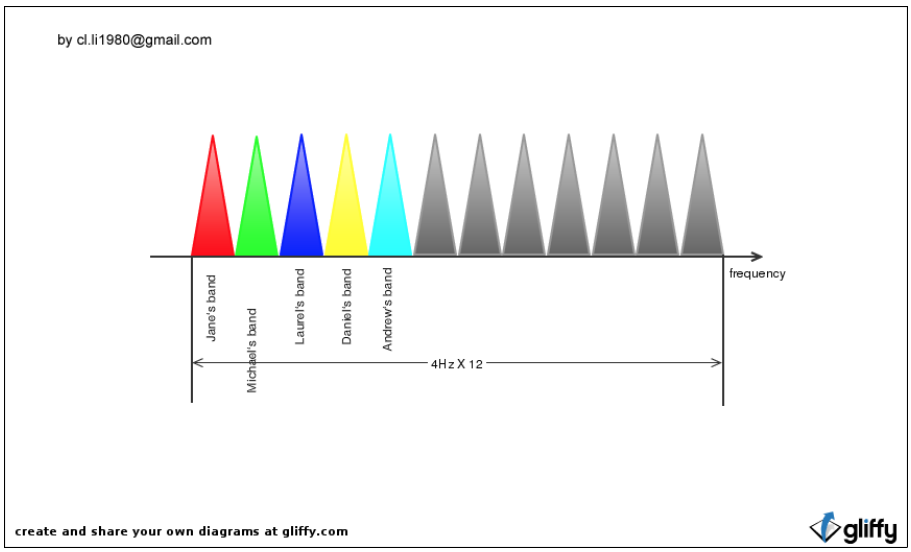
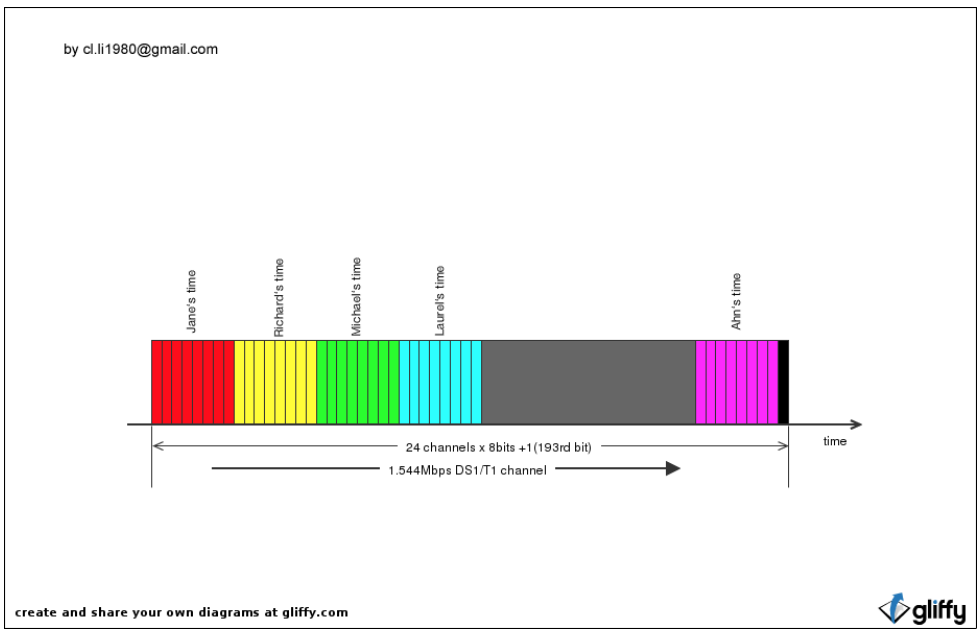


图3. 一个DS载波 [http://en.wikipedia.org/wiki/Digital_Signal_1]的时隙分配：



视频面临的挑战更大一些，按照4:4:4采样的N制彩色电视信号产生的数据率高达60.8Mbps，即使采用目前极先进的调制手段——如号称接近香农极限的DVB-C2@1024QAM，也会将8MHz的带宽全部吃掉。因此，早期的数字视频传输研究都集中于内容简单的低分辨率图像（CIF及QCIF）。ITU是最初的推动者，上世纪八十年代，该组织一直致力于一项彻底改造老式电话网的工程，企图实现包括用户接入在内的全数字化网络，以提供包括数字语音、数字视频和数据的"综合业务"。不幸的是，由于新的数字调制方式的出现以及因特网的冲击，这种被称作ISDN的网络很快就过时了，但由其产生的视频压缩标准H.261却开了数字视频传输的先河，成为后来一系列技术诸如H.263、H.264、RM以及MPEG系列的鼻祖。

开会和制定标准是ITU学僚们的特长，但要坐等他们引导世界进入多媒体时代，恐怕需要相当大的耐心。幸运的是，PC的面世使个人处理多媒体数据成为可能，哪怕是一台装有DOS的286，相信也会比一台ISDN电话机能给人带来更大的遐想空间。

1991年制定了第一个多媒体PC的标准：

- 16 MHz 386SX CPU
- 2 MB RAM
- 30 MB hard disk
- 256-color, 640×480 VGA video card
- 1x (single speed) CD-ROM drive using no more than 40% of CPU to read, with < 1 second seek time

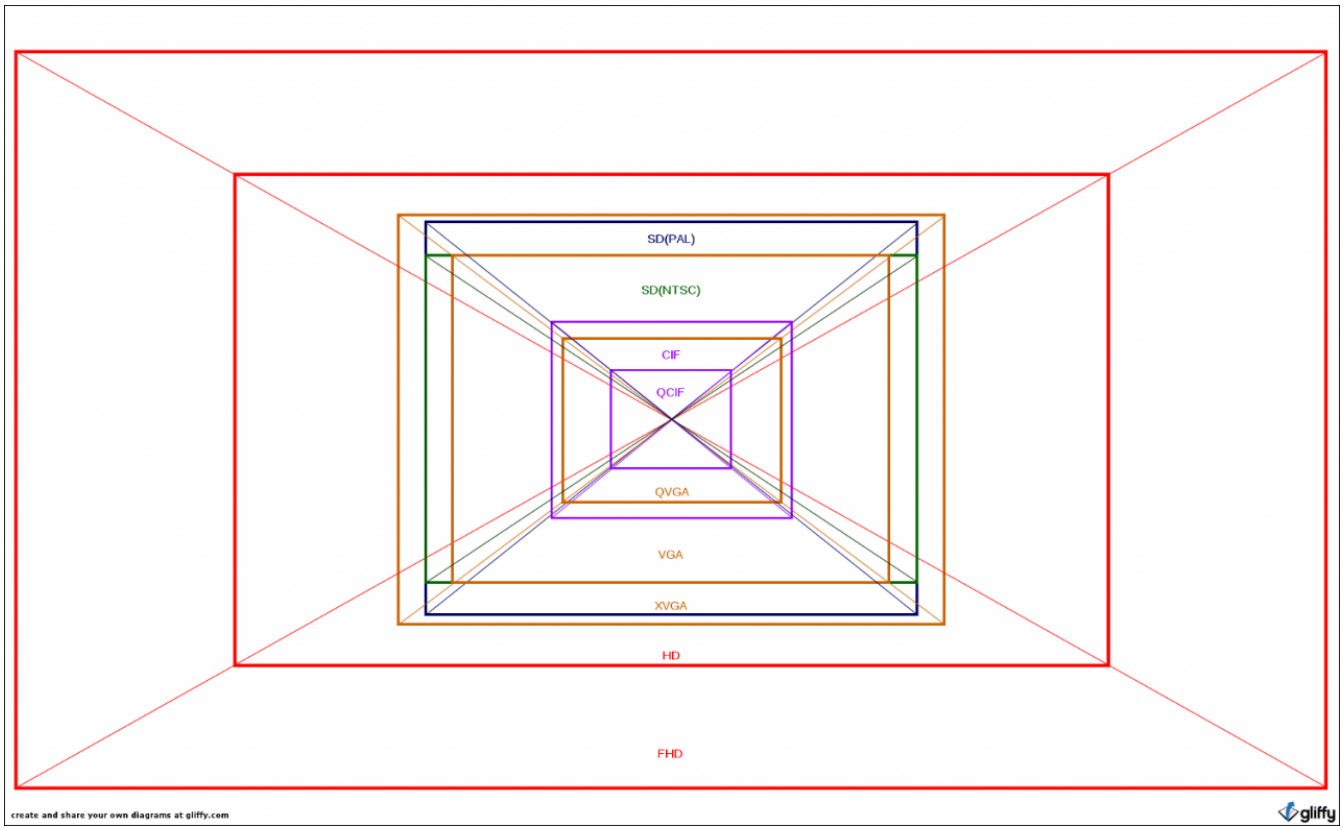
- 1x (single speed) CD-ROM drive using no more than 40% of CPU to read, with < 1 second seek time
- Sound card outputting 22 kHz, 8-bit sound; and inputting 11 kHz, 8-bit sound
- Windows 3.0 with Multimedia Extensions.

看上去它似乎还做不了什么，但它将来能。

世界总是如此，多少风光无限者其实早已薄暮西山，多少貌不惊人者却是在暗自酝酿着爆发的能量，又有多少明眼人可以看得出来?

二、挑战

通过对模拟音视频数据采样、量化得到的原始数字音视频的数据量庞大无比：



RGB图像分辨率	数据量
QCIF(176×144)	76,032 Byte
CIF(352×288)	304,128 Byte
QVGA(320×240)	230,400 Byte
VGA(640×480)	921,600 Byte
SVGA(800×600)	1,440,000 Byte
SD-PAL(720×576)	1,244,160 Byte
SD_NTSC(720×480)	1,036,800 Byte
HD(1280×720)	2,8764,800 Byte
FHD(1920×1080)	6,220,800 Byte
4K(4096×2160)	26, 542, 080 Byte

而当前的数字存储媒介和传输信道所能承载的数据速率相当有限：

媒介/传输方式	容量/速率
CD-ROM	650M Byte
DVD-ROM	4.7G Byte
Blueray	25G/50G Byte
Voice Modem	33.4 kbps
ISDN	64 kbps
T1	1.544 Mbps
GSM	15 kbps
UTMS	2.8 Mbps

为了弥补二者的差距，我们需要在竭力降低传输代价的前提下，提供给受众主观感受上尽可能良好的音视频信息。这里面对人类的听觉和视觉感受系统的研究是非常重要的，以视觉系统（HVS）为例：人眼能够识别的分辨率是有限的，而且对水平和垂直方向较其他方向更加敏感，对灰度信息较颜色信息更加敏感，对静止画面较活动画面更加敏感；人眼还会在主观上放大边缘区域的对比度；更关注感兴趣区域等。

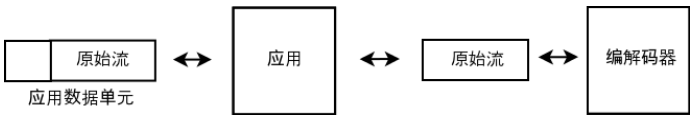
三、传输与存储

3.1 数字媒体流

二进制比特是数字媒体传输和存储的基本形式，每个比特非0即1，任何数字媒体信息必然由若干连续的0或1形成的比特序列来表示。之所以称之为流，是由于声音和视频等媒体信息总是在时间线上展开的，譬如DVD光盘上的一个电影片段、手机通话过程中的一段语音、或者电脑中的一个MP3文件。

3.1.1 多媒体原始流

原始流的概念来源于ISO的MPEG-2标准[2)]，通常指音频、视频或数据编码器输出的二进制比特流，也即可以直接作为解码器输入的比特流。原始流是数字媒体传输和存储系统中最基本、最底层的数据单元，而更高级的、面向应用的数据单元都是在原始流的基础上按照特定协议层层封装而来的。

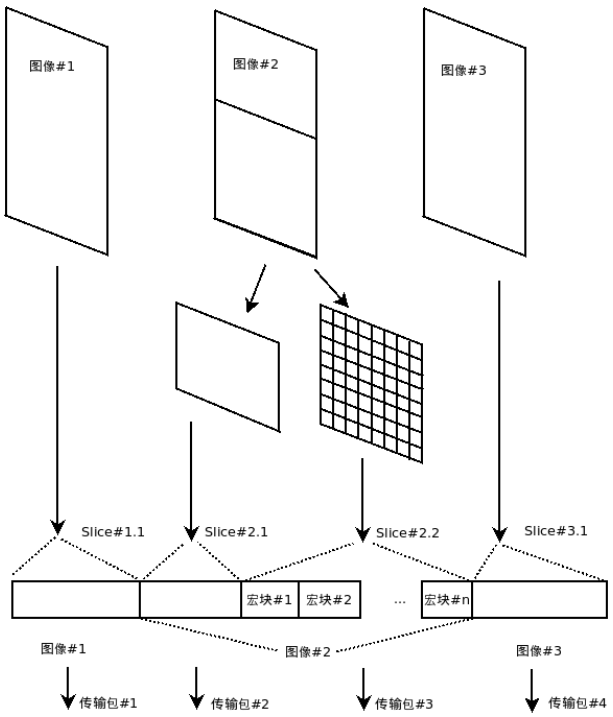


3.1.1.1 音频原始流的结构

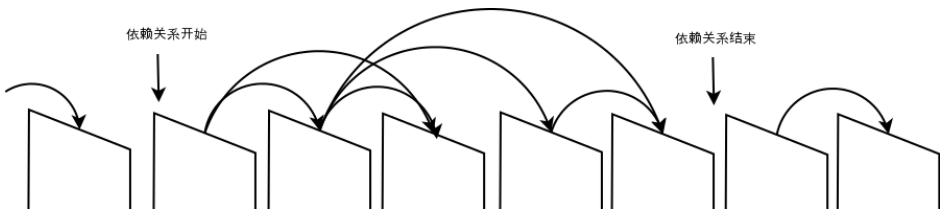
3.1.1.2 视频原始流的结构

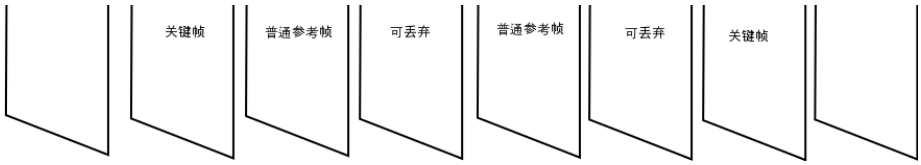
目前，常见的视频编码基本基于混合编码框架（详见4.3.1），因此视频原始流的结构也大都类似。

视频等价于时间轴上的一个图像序列，根据混合编码方案，每一副图像都将被切割为大小相等的方块，除了某些特殊的应用场景之外，以此为基本编码单位的编码过程按照扫描顺序在图像中自左至右、自上而下进行。在H.265之前的视频标准中，这一基本编码单位的大小为16像素见方，称为宏块；H.265中这一大小得到修正，最大可达到64像素见方，以实现对超高分辨率的最佳支持。为了适应包传输网络，有些标准允许一副图像被划分成多个Slice，每个Slice由图像中若干连续的基本编码单位组成。通过引入Slice，视频数据包的大小得到了控制。此外，由于Slice之间一般不允许存在编解码的依赖关系，即使发生丢包也不会出现错误蔓延。



由于混合编码方案使用了基于时间轴的预测技术，在解码过程中，图像数据之间往往是存在依赖关系的，跟据依赖关系可以将图像分为三种类型：完全不依赖于其他图像的图像称为关键帧，完全不为其它图像依赖的称为可丢弃帧，其余图像属于普通的参考帧，除了可丢弃帧之外，关键帧或参考帧的缺失或错误都会导致依赖于该图像的其他图像发生错误，且这种错误会随着依赖关系蔓延，直至找到下一个关键帧结束当前的依赖关系，如下图所示。对于的多媒体应用来说，事先获取图像的类型信息是非常有意义的，某些特定的操作需要这些信息，如随机访问和丢帧——前者要求尽快找到一个关键帧，而后者可以在需要的时候提高处理速度，但需要识别一副图像是不是可丢弃。类型信息通常会包含在图像头中，有时也会作为元数据附加到原始流之外。





在原始流中，图像的顺序与显示顺序不一定相同，这是由于编码器在编码过程中可能会对图像顺序进行了重排，因此，解码器需要负责恢复图像的显示顺序。

3.1.2 13818-1传输流

13818-1主要用于解决数字广播系统中音频、视频及数据信息的复用问题，实际上就是给出一种同时传输音频、视频及数据的有效途径。如前所述，在模拟时代，复用的问题可以通过频分来解决，比如模拟电视通过将音频数据调制到视频信息带宽之外来实现图像和声音的的同时传输。针对数字化的信息，13818-1定义了一个基于数据包的时分复用系统。这是一个面向比特流的传输系统，定义了传输流和节目流两种不同形式的比特流：其中，传输流由固定长度的TS包组成，主要应用于数字广播；节目流则以数据组为单位，应用于数字存储系统，数据组长度是可变的，通常也比TS包要大得多。

13818-1系统还必须保证在数据接收端重建音视频及数据信息的同步，这要求在传输的数据中插入足够多的时间信息。此外，多组使用不同时间基准的音视频信息也可以按照13818-1规定的方式同时传输，这使得同时传输多个多媒体*业务*节目成为可能，不过，只有传输流支持这种方式。

传输流的组成：

3.1.2.1 TS包

标准的TS包包含188个字节，除去4个字节的头信息，还能传输184个字节的数据。头信息重，13位的PID值是最重要的部分，那是一个类似子信道号的值，标识着TS包的数据"身份"。PID值为0x1FFF的TS包为空包，插入空包到传输流中只是为了调整复用结构。payload_unit_start_indicator是头信息中另一个比较重要的位段，对于以PES数据为载荷的TS包，该位设1表示这个TS包中传输的是PES包的起首部分；对于以PSI数据为载荷的TS包，该位设1表示这个TS包中传输的是PSI段的起首部分，而TS载荷的第一个字节是表示PSI数据在载荷中的位置的pointer字段。此外，transport_error_indicator字段表示包内的数据是否有比特错误；transport_scrambling_control表示包内的数据是否加密。

```
transport_packet(){
        sync_byte:8
        transport_error_indicator:1
        payload_unit_start_indicator:1
        transport_priority:1
        PID:13
        transport_scrambling_control:2
        adaptation_field_control:2
        continuity_counter:4
        if(adaptation_field_control=='10'  || adaptation_field_control=='11'){
                adaptation_field()
        }
        if(adaptation_field_control=='01' || adaptation_field_control=='11') {
                for (i=0;i<N;i++){
                        data_byte
                }
        }
}
```

除了实际的PES或PSI载荷之外，跟在TS包头后面的还可能是adaption域，这由TS头中的另一个2比特的字段adaptation_field_control标志决定。 adaption域包含的内容有：

- discontinuity_indicator用于指示系统时钟的不连续事件
- random_access_indicator用于指示随机访问点
- elementary_stream_priority_indicator用于指示ES数据优先度
- PCR字段包含系统时钟的采样值，以27MHz时钟周期为单位，在PID为特定值的TS包中传输
- OPCR字段包含原始系统时钟的采样值，在PID为特定值的TS包中传输。
- splice_countdown字段用于指示音视频载荷的衔接点
- private_data包含13818-1规定外的数据。

3.1.2.2 PES包

在13818系统中，通过为每一个TS包指定特定的PID可以实现多路数据的复用传输，PID不同的TS包可能承载着不同类型的数据，来自不同的编码器或数据发生单元，需要接收端送至不同的接收处理单元。

PES包是由TS包承载传输的标准载荷之一，其内容包括MPEG定义的音、视频信息（11172/13818/14496）、ECM/EMM信息、DSMCC信息以及各种自定义数据等。在传输之前，PES包必须被分割并分配到若干（至少一个）TS包的数据载荷部分。

一个PES包序列形成一路13818系统的数据流，其承载的数据被称作原始流数据，它们可能是某个视频编码器的输出、某个音频编码器的输出，某个数据发生器的输出或者某种控制信息，为了使接收端能够无缝地获取到这些数据并将其送至特定的处理单元如视频解码器、音频解码器、数据处理单元或控制单元，传输这一路PES数据的TS包必须使用相同的PID（即占用同一个子信道），并且保证在传输过程中不会发生乱序。

对于所有的PES包，首部有三个字段是必需的： packet_start_code_prefix(0x000001) stream_id(一个字节的流标志字段，指示PES的载荷类型) PES_packet_length(两个字节的长度字段)，这个数值可以设为零。

对于载荷类型除program_stream_map、padding_stream、private_stream_2、ECM、EMM、program_stream_directory、DSMCC_stream及ITU-T Rec. H.222.1 type E_stream外的PES包，13818-1还规定了更多的标准字段作为补充性的首部信息：

- PES_scrambling_control指示PES载荷是否被加扰；
- PES_priority指示PES载荷的优先级；
- copyright指示PES载荷是否存在版权保护；
- original_or_copy指示PES载荷属于原始信息还是拷贝信息；
- PTS和DTS是以90kHz时钟周期为单位的时间标签，分别表示显示时间和解码时间；
- ESCR字段包含ES系统时钟的采样值，以27MHz时钟的周期为单位
- ES_rate指示ES流速率。

…

3.1.2.3 辅助信息

承载辅助信息的数据在13818-1系统中被称作表（table），每一种表由一个8比特的table id标识。其中，预定义的表有三种：节目关联表（PAT）、条件接收表（CAT）和节目映射表（PMT），它们占用的table id分别为0、1和2，此外，0x03~0x3f范围内的table id为13818-1保留，0x40~0xFE范围内的table id可用作私有扩展（DVB、ATSC等）。

在13818-1系统中，这些表需要按照规定的时间间隔重复传输，以使得接收端在任意的时间点都可以尽快地拿到所需要的全部辅助信息。每一种表会占用一个PID，表中的数据以section的方式来组织，每个section的长度不超过4096字节，如果表的长度大于4096字节，则被分为多个section来传输。 section的组织方式如下：

- 方式一：

```
section() {
        table_id                8
        '1'                     1
        private_indicator       1
        reserved                2
        section_length          12
        table_id_extension      16
        reserved                2
        version_number          5
        current_next_indicator  1
        section_number          8
        last_section_number     8
        for (i=0;i<section_length-9;i++) {
                data_byte
        }
        CRC_32                  32
}
```

- 方式二：

```
section() {
```

```
        table_id                8
        '0'                     1
        private_indicator       1
        reserved                2
        section_length          12
        for (i=0;i<N;i++) {
                data_byte
        }
        CRC_32                  32
}
```

其中：

- table_id标识表的类型。
- private_indicator表示该表是否为私有信息。
- section_length为紧随该域的所有数据的长度（含4字节的CRC校验）。
- table_id_extension是对table id的一个16比特的扩充，对于不同的表有不同的含义。
- version_number是个5比特的版本号。在重复轮播的过程中，表的内容也会发生变化，每次发生变化之后，要求这个版本号加一。
- current_next_indicator是一个标志位，标志该表的内容即可生效还是在不久的将来生效，若该位为0，则表示这个表是使用与将来的，起到一个通知接收端表内容即将发生变化的作用。
- 如果一个表由多个section传送，则第一个section的序号为0，section_number表示当前section的序号，last_section_number表示最后一个section的序号，也就是传送这个表的section的总数减去1。

最后，每一个section的数据会被分配到对应某一个PID的TS包载荷中。

2.1.2.4 复用的实现和PSI表

13818-1系统通过将不同的数据分配到PID不同的TS包内传输实现复用，而且，通过节目专用信息（PSI）表，该系统还可以实现多路多媒体~~业务~~节目的同时传输。

13818-1内定义的节目专用信息表有四种，分别为节目关联表（PAT），节目映射表（PMT），网络信息表（NIT）和条件访问表（CAT）。

PAT给出了一个传输流中各路多媒体~~业务~~节目的信息，其table id为0，在pid为0的TS包中传输，使用如下的语法结构：

```
program_association_section() {
        table_id                        8
        section_syntax_indicator        1
        '0'                             1
        reserved                        2
        section_length                  12
        transport_stream_id             16
        reserved                        2
        version_number                  5
        current_next_indicator          1
        section_number                  8
        last_section_number             8
        for (i=0; i<N;i++) {
                program_number          16
                reserved                3
                if(program_number == '0') {
                        network_PID     13
                }
                else {
                        program_map_PID 13
                }
        }
        CRC_32  32
}
```

除去标准的section字段之外，需要关注的是：

- transport_stream_id给出该PAT所属的传输流的id，以与网络中的其他传输流区分开来。
- program_number是本传输流内的某路节目的编号。
- program_map_PID是用于传输PMT的pid。

其中，N循环中包含了多组program_number和program_map_PID的结对，描述了用以传输各路节目的PMT section所使用的pid。（program_number为0除外，它对应的program_map_PID为传输NIT section所使用的pid）

PMT给出了某路多媒体~~业务~~节目的业务信息，即该多媒体节目中包含哪些媒体，分别为何种类型，用于传输各个媒体数据的pid为多少等，其table id为2，传输使用的pid在PAT中指定，section格式如下：

```
TS_program_map_section() {
        table_id                        8
        section_syntax_indicator        1
        '0'                             1
        reserved                        2
        section_length                  12
        program_number                  16
        reserved                        2
        version_number                  5
        current_next_indicator          1
        section_number                  8
        last_section_number             8
        reserved                        3
        PCR_PID                         13
        reserved                        4
        program_info_length             12
        for (i=0; i<N; i++) {
                descriptor()
        }
        for (i=0;i<N1;i++) {
                stream_type             8
                reserved                3
                elementary_PID          13
                reserved                4
                ES_info_length          12
                for (i=0; i<N2; i++) {
                        descriptor()
                }
        }
        CRC_32                          32
}
```

其中，program_number为该PMT所描述的节目的编号，和PAT中的program_number相同。N1循环中给出节目中各个媒体的信息：

stream_type为媒体类型，如0x01为11172-1视频格式、0x02为13818-2视频格式、0x03为11172-1音频格式、0x04为13818-2音频格式、0x0f为13818-7音频格式、0x10为14496-2视频格式、0x11为14496-3音频格式、0x1b为14496-10视频格式等。而elementary_PID为传输该媒体数据使用的pid值。

3.1.2.5 时钟信息

13818-1传输流中的每一路节目都有一个独立的27MHz的时钟，该时钟的采样值被封装到TS包的adaption域内定期传送给接收端（至少100毫秒一次），使接收端的时钟得以保持和发送端的同步，这些采样值被称为节目时钟参考（PCR），是一个42比特的值，表示0.037微妙的时钟嘀嗒，极限值大约是45个小时。对于特定的某一路节目，用以传输PCR的pid是固定的，需要在该节目的PMT中指明。当PCR发生突变的时候，发送端需要预先通知接收端，这同样要使用到adaption域，时钟突变发生之前，传输PCR的adaption域的discontinuity_indicator需要被置1。

音视频等媒体的时间标签封装在PES头中，有PTS和DTS两种，分别表示媒体的播放时间和解码时间，它们都是以27M的系统时钟为基准的，但时间的颗粒度更粗一些，为90KHz。PTS和DTS的位长都是33比特，表示11微妙的时钟嘀嗒，极限值大约是26个小时。

3.1.2.6 同步

13818-1系统是强同步的，这意味着，接收端不仅要保证特定节目内的每一种媒体（音频、视频及字幕等）按照给定的速率播放、媒体间的同步关系得以保持（唇音同步、字幕同步等），还要保证与发送端使用一致的时钟，也就是说所有接收端在播放同一个广播节目时的表现必须完全相同。

因此，接收端必须根据传输流中的PCR值内建一个本地时钟，而且，还必须依据收到的PCR值对这个本地时钟不断校正，以弥补编码和传输过程中引入的PCR抖动。而所有媒体的播放控制都依赖于这个本地时钟，目标就是令每个媒体包的播放时刻与它的时间戳完全一致。

3.2 容器

在数字媒体领域，容器专指数字媒体数据的封装格式。音频、视频及其他数据借助于某个特定的容器可以被组织、复用到同一个文件之中，从而满足某些特定的操作要求（主要是针对播放器），如播放、暂停、快进、后退、跳转等。 容器不涉及多媒体数据的具体编码方式。

3.2.1 AVI

AVI是微软使用的一种多媒体文件格式，由于迄今为止Windows一直是PC的主流操作系统，AVI也成为PC中最流行的多媒体文件格式。不过这种格式并非微软的原创，它最早是由电子艺界提出的，其特征是使用一种称为chunk的数据块来存储多媒体数据及其附加信息。每个chunk只有八字节的头，前四字节是四个ASCII码，作为该chunk的标识；后四字节为一个整数，表示紧跟在头信息后面的数据的长度，结构非常简单。此外，还有分别以"RIFF"和"LIST"为标识的两种复合chunk，它们的数据内容为多个chunk组成的序列，从而使数据的层次化得以实现。

3.2.1.1 结构

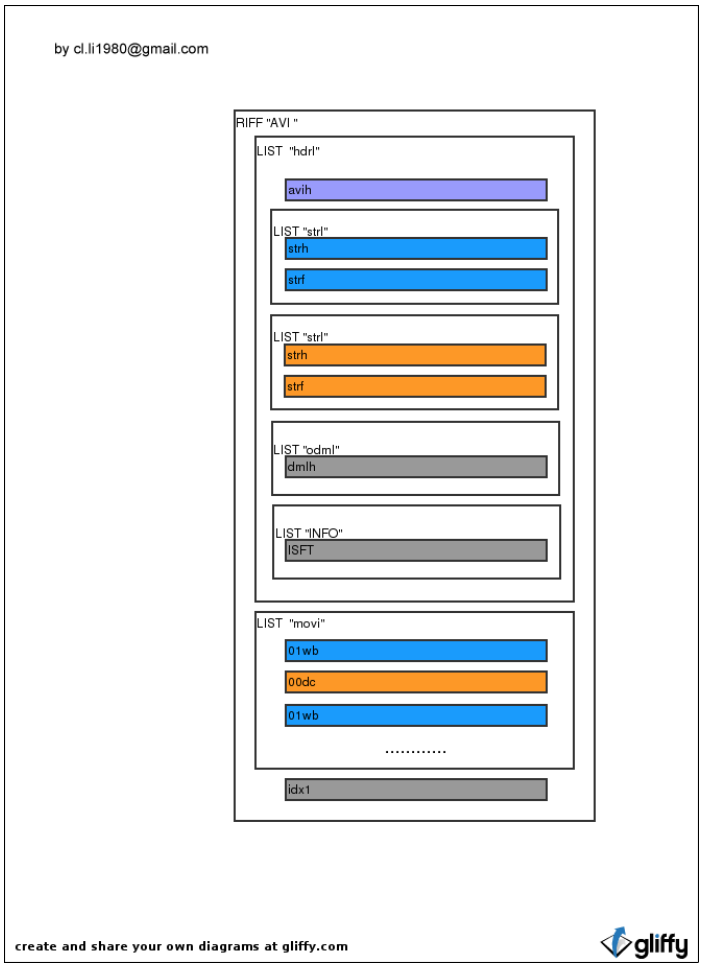
一个avi文件在结构上由一个RIFF复合chunk组成，因此，首四个字节 "52494646"是RIFF的ASCII码，接下来的四个字节表示该复合chunk的长度，亦即该avi文件的长度减8，至于数据部分，则首先是四个字节"41564920"，即ASCII的"AVI "，表示这个复合chunk的具体名称，然后才是组成该avi的各个chunk。通常，一个avi文件包括一个名为"hdrl"的LIST，存放所有头相关信息；一个名为"movi"的LIST，存放所有媒体数据；然后是一个标识为"idx1"的chunk，存放相关索引信息。索引信息描述了各个数据块在 LIST中的位置，有助于提高SEEK操作的速度。

需要注意的是，chunk中的数据是要求双字节对齐的，如果某个chunk的长度是奇数，那么其后要填一个零。

AVI RIFF File Reference [http://msdn.microsoft.com/en-us/library/ms779636.aspx] from the Microsoft site:

> "The data is always padded to nearest WORD boundary. ckSize gives the size of the valid data in the chunk; it does not include the padding, the size of ckID, or the size of ckSize."

图3. AVI文件的结构:



一个实例：

```
RIFF[AVI ]+60337434B
    LIST[hdrl]+8830B
        avih+56B
        {FPS:1000000/41667, 0bps, 0 Byte Aligned, /HASINDEX/INTERLEAVED 2758 frames, Inital 0, 2 streams, Buffer:0 Bytes, 1280x720}
        LIST[strl]+4244B
            strh+56B
            {[vids], [divx], Initial 0, 24/1, 0+2758, Buffer:368324 Bytes, quality:0x2710, Size of Sample:0, }
            strf+40B
            {1280x720, 24 Bits, DX50}
            JUNK+4120B
        LIST[strl]+4234B
            strh+56B
            {[auds], [], Initial 1, 24000/1, 0+2753491, Buffer:12000 Bytes, quality:0x0, Size of Sample:1, }
            strf+30B
            {id=0x55, 2 ch, Sample rate:44100, Bit rate:192000, 1 block align, 0 bits, }
            JUNK+4120B
        LIST[odml]+260B
            dmlh+248B
        LIST[INFO]+56B
            ISFT+44B
            {VirtualDubMod 1.5.10.1 (build 2439/release)}
        JUNK+1318B
    LIST[movi]+60239170B
        01wb+12000B
        00dc+70699B
        01wb+1000B
        00dc+465B
        01wb+1000B
        00dc+466B
        01wb+1000B
        ......
    idx1+88016B
       {fourcc=01wb, flag=0x10, pos=4, len=12000}
       {fourcc=00dc, flag=0x10, pos=12012, len=70699}
       {fourcc=01wb, flag=0x10, pos=82720, len=1000}
       {fourcc=00dc, flag=0x0, pos=83728, len=465}
       {fourcc=01wb, flag=0x10, pos=84202, len=1000}
       {fourcc=00dc, flag=0x0, pos=85210, len=466}
       {fourcc=01wb, flag=0x10, pos=85684, len=1000}
```

```
        ......
```

以上结构可以通过一系列简单的C函数来理解，主要涉及到五个Microsoft数据结构：

```
typedef struct _avimainheader {
    FOURCC fcc;                 // 'avih'
    DWORD  cb;                  // size of the header, initial 8 bytes excluded
    DWORD  dwMicroSecPerFrame;  // frame period in microsecond
    DWORD  dwMaxBytesPerSec;    // maximum bitrate
    DWORD  dwPaddingGranularity; // alignment for data, in bytes
    DWORD  dwFlags;
    DWORD  dwTotalFrames;       // total number of frames of data in the file
    DWORD  dwInitialFrames;     // initial frame for interleaved files. Noninterleaved files should specify zero
    DWORD  dwStreams;           // the number of streams in the file
    DWORD  dwSuggestedBufferSize;// suggested buffer size for reading the file
    DWORD  dwWidth;             // width of the AVI file in pixels
    DWORD  dwHeight;            // height of the AVI file in pixels
    DWORD  dwReserved[4];       // reserved. Set this array to zero.
} AVIMAINHEADER;

typedef struct _avistreamheader {
    FOURCC fcc;                 // 'strh'
    DWORD  cb;                  // size of the header, initial 8 bytes excluded
    FOURCC fccType;             // data type of the stream
    FOURCC fccHandler;          // data handler, preferred codec for audio and video
    DWORD  dwFlags;
    WORD   wPriority;           // highest priority might be the default stream
    WORD   wLanguage;           // Language tag
    DWORD  dwInitialFrames;
    DWORD  dwScale;             // dividing dwRate by dwScale gives the number of samples per second
    DWORD  dwRate;
    DWORD  dwStart;             // starting time for this stream
    DWORD  dwLength;            // length of this stream
    DWORD  dwSuggestedBufferSize;   // how large a buffer should be used to read this stream
    DWORD  dwQuality;           // an indicator of the quality of the data
    DWORD  dwSampleSize;        // the size of a single sample of data, 0 for video
    struct {
        short int left;
        short int top;
        short int right;
        short int bottom;
    }  rcFrame;                // destination rectangle for display
} AVISTREAMHEADER;

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

typedef struct tagBITMAPINFO {
  BITMAPINFOHEADER bmiHeader;
  RGBQUAD          bmiColors[1];
}BITMAPINFO, *PBITMAPINFO;

typedef struct tagBITMAPINFOHEADER {
  DWORD biSize;          // The number of bytes required by the structure
  LONG  biWidth;         // The width of the bitmap, in pixels
  LONG  biHeight;        // The height of the bitmap, in pixels
  WORD  biPlanes;        // must be set to 1
  WORD  biBitCount;      //  The number of bits-per-pixel
  DWORD biCompression;   // The type of compression, codec indicator
  DWORD biSizeImage;     // The size of image buffer
  LONG  biXPelsPerMeter; // The horizontal resolution, in pixels-per-meter
  LONG  biYPelsPerMeter; // The virtical resolution, in pixels-per-meter
  DWORD biClrUsed;       // The number of color indexes in the color table that are actually used by the bitmap
  DWORD biClrImportant;  // The number of color indexes that are required for displaying the bitmap
}BITMAPINFOHEADER, *PBITMAPINFOHEADER;
```

AVI文件的头信息存放在以"avih"为标识的chunk中，其数据格式由AVIMAINHEADER结构描述，但实际上里面的多数信息都属于冗余信息，因为诸如视频帧率、比特率、视频的宽度和高度等参数都会在其后的媒体流头信息和媒体流格式信息中给出，而且，许多编码器还会将这些信息硬编码到媒体流内部，因此，一般不必它们进行处理。参考以下来自FFMPEG的一个处理avi的代码片断：

```
        case MKTAG('a', 'v', 'i', 'h'):
            /* AVI header */
            /* using frame_period is bad idea */
            frame_period = get_le32(pb);
            bit_rate = get_le32(pb) * 8;
            get_le32(pb);
            avi->non_interleaved |= get_le32(pb) & AVIF_MUSTUSEINDEX;

            url_fskip(pb, 2 * 4);
            get_le32(pb);
            get_le32(pb);
            avih_width=get_le32(pb);
            avih_height=get_le32(pb);

            url_fskip(pb, size - 10 * 4);
            break;
```

接下来是若干名为"strl"的LIST Chunk，用以传输媒体流的头信息，每一个媒体流对应一个LIST，譬如：对于一路音频加一路视频的AVI文件，则分别有一个描述音频流的"strl" LIST和一个描述视频流的"strl" LIST。整个LIST由两个chunk组成：第一个chunk的ascii标识是"strh"，其数据格式由AVISTREAMHEADER描述，给出了若干通用的媒体信息；第二个chunk的ascii标识是"strf"，其数据长度和数据格式因媒体的不同而不同，对于视频流，它的数据是一个BITMAPINFO的结构，而对于音频，则是一个WAVEFORMATEX的结构。

从AVI头和"strl" LIST可以获得如下信息：

- 媒体流的数目

AVIMAINHEADER的dwStreams成员会给出文件中包含的媒体流数目，不过，一般会依据文件中包含的"strl" LIST的实际个数来判断。

- 媒体编码格式

AVISTREAMHEADER的fccHandler会给出一个对应的媒体流所使用的codec的ascii码标识，此外，在描述视频流的BITMAPINFO结构中有一个biCompression成员，在描述音频流的WAVEFORMATEX结构中有一个wFormatTag，它们也会出对应音视频流的codec信息，其中大部分codec的定义来自RFC2361 [http://www.faqs.org/rfcs/rfc2361.html]。

- 文件长度

AVISTREAMHEADER的dwTotalFrames给出了媒体流的总帧数。

- Buffer的大小

AVIMAINHEADER中有dwSuggestedBufferSize，AVISTREAMHEADER还有dwSuggestedBufferSize。

- Sample Size

AVISTREAMHEADER中有dwSampleSize字段，微软在MSDN中是如是解释的：

> 该字段给出一个数据采样的大小（字节）。若各数据采样大小不同则设该字段为0。该字段不为0时，多个数据采样可以被组合到一个chunk中。该字段为0（比如视频）时，每个数据采样必须占有一个独立的chunk。对于视频流，这个值通常被设为0（虽然各视频帧大小相同时也可设为非零）。对于音频流，这个数值应该同描述音频的WAVEFORMATEX结构中nBlockAlign成员相等。(Specifies the size of a single sample of data. This is set to zero if the samples can vary in size. If this number is nonzero, then multiple samples of data can be grouped into a single chunk within the file. If it is zero, each sample of data (such as a video frame) must be in a separate chunk. For video streams, this number is typically zero, although it can be nonzero if all video frames are the same size. For audio streams, this number should be the same as the nBlockAlign member of the WAVEFORMATEX structure describing the audio.)

而实际应用中有如下几种情况：对于视频来说，一个数据采样就是一个视频帧，除非是非压缩数据，不然很难令每个视频帧的字节数相等，因此，视频数据的Sample Size往往是0，

每个视频帧占用一个chunk。对于音频来说，情况略复杂：

1. 0值，表示变比特率，一个数据采样表示一个音频帧，和视频流一样，每个音频帧也占用一个chunk，此时nBlockAlign给出音频帧的采样点个数，nSamplesPerSec/nBlockAlign可获取帧率。
2. 1值，此时dwSampleSize是一个没有意义的值，每个数据Chunck存放一个音频帧，固定比特率（CBR）的音频压缩数据往往这么做。
3. 以采样点记的帧长度，通常只适用于固定比特率的音频。在这种情况下，dwSampleSize等于nBlockAlign，表示一个音频帧的长度，但这个长度指的不是字节数，而是采样点数，一个音频帧的字节数为：(dwSampleSize*bitrate)/(sample_rate*8)。比如，采样率48000、比特率63952、帧长1152的mp3格式，每帧数据的字节数为(1152*63952)/(48000*8))=192。在这种情况下，一个Chunck中可能会存放多个音频帧，则解包时需要计算每一帧数据的字节数。
4. 以字节记的采样点长度，音频数据。往往针对PCM。有这种情况?

- 时间信息

AVISTREAMHEADER中有两个成员dwScale和dwRate，以dwRate/dwScale的形式给出对应媒体流每秒钟的样本个数。对于视频数据，dwRate/dwScale即是帧率，如果不考虑B帧的影响，第n帧的时间信息即为n*dwScale/dwRate；对于音频数据，其意义因dwSampleSize的不同而不同：

1. 对应dwSampleSize情况1，dwRate/dwScale给出音频的帧率，且每个chunk包含一个音频帧，时间信息的计算方法与视频流相同。
2. 对应dwSampleSize情况2，dwRate/dwScale给出的是音频数据每秒钟的字节数，即比特率除以8，可得到第n个字节的时间信息为n*dwScale/dwRate。
3. 对应dwSampleSize情况3，dwRate/dwScale给出音频的帧率，但需要事先计算出每个音频帧的字节数，然后才能确认每帧的时间信息。
4. 对应dwSampleSize情况4，dwRate/dwScale给出音频的采样率，第n个采样点的时间信息为n*dwScale/dwRate。有这种情况？？？

- 视频信息

帧率和分辨率是两个对于播放器来说极其重要的数据，它们给出了视频数据的基本时间信息和空间信息。AVIMAINHEADER有描述帧时长的dwMicroSecPerFrame，这个值并非很可靠，更多情况下会参考6）中的提到的dwScale和dwRate成员，但事实上，许多视频编码标准会将帧率的信息直接编码到视频流中，相对于前二者，这个帧率才是最可靠的。与帧率类似，AVIMAINHEADER、BITMAPINFO以及原始视频流中都会有视频高度和宽度的信息，仍旧是以原始视频流中的信息最为可靠。

- 音频信息

音频流的信息定义在一个WAVEFORMATEX结构中。起首的两字节字段wFormatTag定义了音频的格式，如：

- 0x0001：PCM格式。
- 0x0003：IEEE Float格式
- 0x0006：G.711 A律格式
- 0x0007：G.711 μ律格式

而通道数、采样率、比特率、块长度和采样位长分别由nChannels、nSamplesPerSec、nAvgBytesPerSec/8、nBlockAlign和wBitsPerSample给出。最后2字节的cbSize则是接下来的扩展长度，可以为0。

3.2.1.2 媒体数据

媒体数据以Chunk的形式存放在"movi" 每个Chunk的FOURCC对应媒体流的id，其中前两个字节表示媒体的类型，后两个字节表示媒体流的序号，通常，"wb"表示音频，"dc"表示压缩视频，"db"表示非压缩视频，“sb"表示字幕。一般情况下，每个Chunk存放一个视频帧或音频帧。

3.2.1.3 Meta Data

AVI允许使用一个名为“INFO"的LIST存放一些metadata，以下是FFMPEG中给出的AVI metadata的定义，其中较为常见的是ISFT，描述生成AVI文件的应用程序：

```
const AVMetadataConv ff_avi_metadata_conv[] = {
    { "IART", "artist"    },
    { "ICMT", "comment"   },
    { "ICOP", "copyright" },
    { "ICRD", "date"      },
    { "IGNR", "genre"     },
    { "ILNG", "language"  },
    { "INAM", "title"     },
    { "IPRD", "album"     },
    { "IPRT", "track"     },
    { "ISFT", "encoder"   },
    { "ITCH", "encoded_by"},
    { "strn", "title"     },
    { 0 },
};
```

3.2.1.4 索引和随机访问

随机访问是播放器必不可少的一项功能，它允许用户可以直接访问某个时间点上的媒体内容。但是，文件系统中往往以字节偏移为单位进行随机访问，因此，一种多媒体容器必须提供一种字节偏移和时间的对应表，即索引，它的作用和我们平时读书的目录是相同的，假如没有目录，我们不得不从头一页一页地翻书，直到找到需要的内容。 AVI的索引数据在文件的末尾，是一个名为"idx1"的数据块，包含了"movi" List中所有数据块的索引信息。每个索引项的结构如下：

```
typedef struct {
    DWORD  ckid;
    DWORD  dwFlags;
    DWORD  dwChunkOffset;
    DWORD  dwChunkLength;
} AVIINDEXENTRY;
```

ckid是该索引对应的数据块的id，如"00dc"，"01wb"等； dwFlags给出几个标志，主要是用于表明对应数据块是否为关键帧（0x10表示关键帧）；接下来两个32位值分别表示对应数据块在文件中相对于movi List的字节偏移及数据块的长度。 由上可见，AVI的索引项中没有时间信息，第n个索引项对应的即是第n个数据chunk的信息。 一个示例：

```
example.avi
  video stream:   00dc, 29.97 fps;
  audio stream 1: 01wb, 448000 bps;
  audio stream 2: 02wb, 192000 bps

original index information:
01wb 0x10 4      28000
02wb 0x10 28012  12000
00dc 0x10 40020  3356
01wb 0x10 43384  1869
02wb 0x10 45262  801
00dc 0x00 46072  159
01wb 0x10 46240  1869
02wb 0x10 48118  801
00dc 0x00 48928  159
......

parsed index information:
video stream:
t = 0        s; offset = 40020 B; length = 3356 B; [KEY]
t = 100/2997 s; offset = 46072 B; length = 159 B;
t = 200/2997 s; offset = 48929 B; legnth = 159 B;
...
audio stream 1:
t = 0          s; offset = 4     B; length = 28000 B;
t = 28000/56000 s; offset = 43384 B; length = 1869 B;
t = 29869/56000 s; offset = 46240 B; length = 1869 B;
...
audio stream 2:
t = 0          s; offset = 28012 B; length = 12000 B;
t = 12000/24000 s; offset = 45262 B; length = 801 B;
t = 12801/56000 s; offset = 48118 B; length = 801 B;
```

索引数据的缺失或破损严重影响AVI文件的随机访问，即使我们可以粗略地估计一个位置，但由于读到的AVI数据Chunk中不包含时间标签，也会导致音视频之间失去同步关系。毕竟AVI是一种为本地播放文件而设计的格式，它不会为每个数据包提供显式的时间信息，其时间关系完全隐含在数据的存储组织关系内。

3.2.1.5 交织

顾名思义，AVI格式的一个初衷即是实现音频和视频数据的交织存储，这有助于播放器在顺序读取数据的情况下实现音视频的同步，因为某些设备如硬盘、光驱等并不适合频繁的随机访问，对于播放器来说，只需要关注合适的缓冲策略和同步策略即可。

3.2.1.6 打包和解包

视频数据和变比特率音频数据的打包通常以帧为单位，一个Chunk存放一帧压缩图像或声音；

对于固定比特率的音频数据，允许多个压缩的帧存放在一个chunk中，但要求首部信息给出帧长度（通常原始音频单帧采样点个数的形式给出，比如对于mp3，这个数值一般是1152），否则无法将Chunk中的多个包分割开来。

3.2.1.7 一些结论

AVI文件语法结构简单，易于解析，媒体数据采样存储在独立的chunk中，有简单的同步标记和长度信息，即使文件的索引部分和头部分发生损坏，媒体数据也依然能够被读取。然而，AVI的缺陷也确实不少，一个致命的缺点是缺少时间标签，只能根据全局帧率或比特率累积推测时间信息，对于变帧率视频和变比特率音频的支持有限；其次，每个流没有独立的索引信息，索引信息的位置也没有在头部给出；此外，AVI的可扩展性也不好，不支持超级大文件，不支持媒体数据打包时的分割和分组，不支持数字版权管理（Divx公司通过定义新的chunk实现了其专有的DRM策略），也无法满足日益迫切的流媒体需求。这些缺陷在之后出现的类似媒体格式（包括微软推出的ASF和Real推出的realmedia等）中得到了修正。

目前AVI主要用于本地文件回放，来源包括一些不合法的DVD转码拷贝软件以及Divx公司 [http://www.divx.com]的软件。

3.2.1.8 ODML扩展

ODML扩展修正了许多AVI固有的缺陷，如不支持大文件（> 2G Bytes）、多个媒体流共用一个索引数据块等。

ODML扩展的AVI文件提供一种方法允许文件中含有多个RIFF，以突破2G长度限制。第一个RIFF为主RIFF（RIFF 'AVI '），内含"hdrl"信息，而其他的RIFF为扩展RIFF（RIFF 'AVIX'），仅包含"movi" LIST。主RIFF的"hdrl"中会包含一个"odml" LIST，其中的"dmlh" Chunk包含有整个AVI文件的长度。

ODML扩展的AVI文件还提供了新的索引结构，其一、索引信息位于"strl"中，以"indx"为FOURCC，各个媒体流有自己的索引数组；其二、支持分层索引，即"indx"中的 索引项可以是一系列指向二级索引表的指针，而FOURCC为"ix00"形式的Chunk作为二级索引表散布"movi"中。这里是一个解析ODML索引的示范程序。

3.2.1.9 DivX

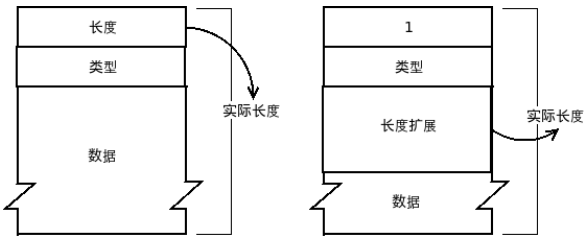
DivX文件格式是Divx公司在AVI的基础上开发的一种多媒体容器，主要是提供了一些类似与DVD的功能，如交互式菜单、多字幕、多音轨、章节支持、数字版权保护等。一般来说，divx文件是与AVI保持兼容的，一个支持AVI的播放器可能不会支持DivX的某些扩展功能，但至少可以播放其中的音视频。

3.2.2 ISO标准及其衍生

3.2.2.1 QTFF

QTFF(Quick Time File Format)是苹果公司推出的多媒体文件格式，第一个版本于1991年随着Quick Time多媒体框架一起问世，通常以"MOV"为扩展名。

QTFF的基本组成单位是Atom，最基本的Atom是由长度、类型和数据三部分组成的，其中长度和类型分别为4字节长，长度部分能够支持扩展为8个字节。某些复杂一点的Atom规定在长度和类型之后增加1个字节的版本号和3个字节的标记位，以支持更多扩展。类型字段则往往是四个ASIC字符，这在多媒体文件格式中及其常见。一个Atom的数据允许包含子Atom，因此，在理论上Atom是可以自由嵌套的。



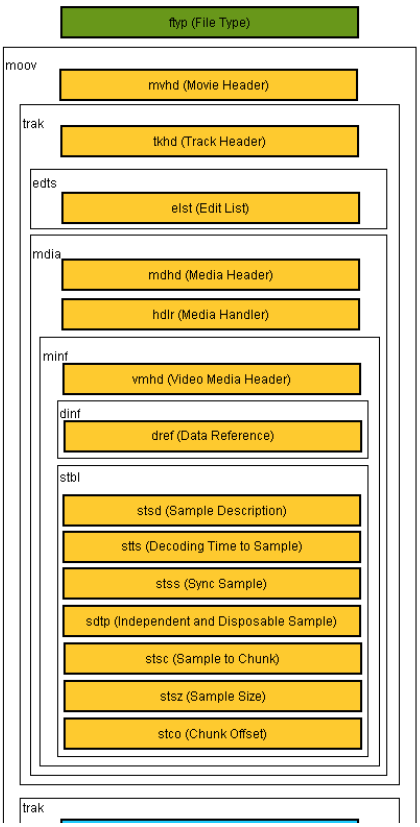
3.2.2.2 ISO媒体文件标准

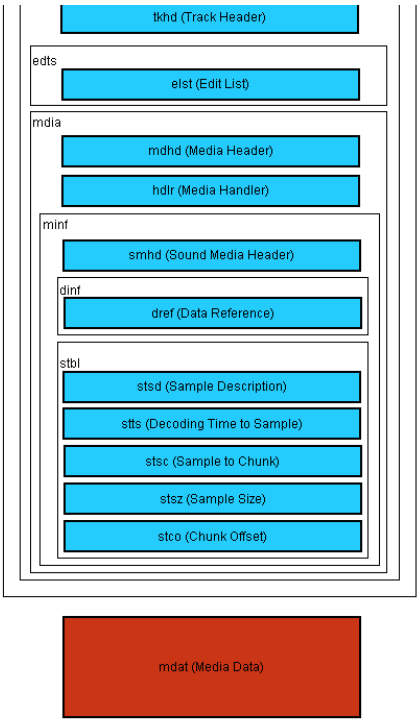
2001年，ISO在QTFF的基础上制定了一个多媒体文件标准（ISO/IEC 14496-12），由于具备非常好的功能性和扩展性，该标准逐渐得到了业界的认可。在网站http://www.mp4ra.org/atoms.html [http://www.mp4ra.org/atoms.html]列出了目前经过认证的ISO文件扩展，其中包含了3GPP、MP4以及QTFF格式。

ISO媒体文件将QTFF中的Atom重定义为盒子（Box），构造方式并无变化，而且基本的盒子的定义也与QTFF保持一致。虽然QTFF的出现先于ISO，但仍可以将QTFF看作是ISO标准的某种扩展。

图4. 一个典型的ISO格式文件

by cl.li1980@gmail.com



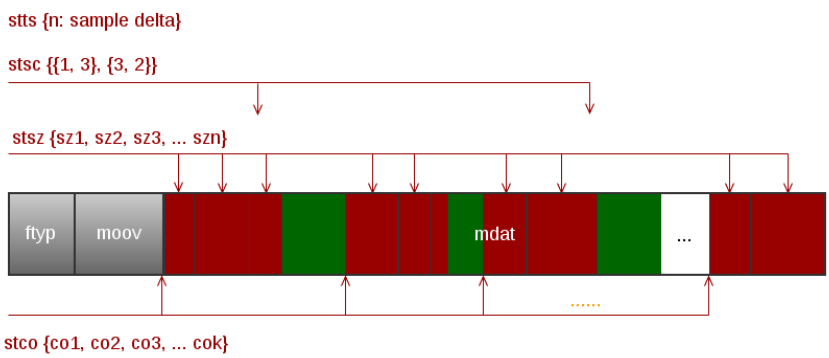


这里有一段C++代码，用于实现ISO媒体文件的解析。

盒子的定义和包含关系大致如下：

- ftyp：文件类型，主要包括当前格式的版本号，兼容性等信息。
- mdat：数据块，以Chunk为单位交错存放的媒体数据。
- moov：影片信息，含有子字段。
  - mvhd：影片信息头，含有影片的时间粒度及此粒度下的影片时长等字段。
  - trak：媒体轨道信息，含有子字段。
    - tkhd：媒体轨道信息头，含有媒体轨道的标识号、以影片时间粒度度量的时长等信息。
    - mdia：媒体信息，含有子字段。
      - mdhd：媒体头，该媒体的时间粒度及此粒度下的时长。
      - hdlr：媒体类型，含有一个FourCC标识的媒体类型，如vide、soun、subt等。
      - minf：媒体详细信息，含有子字段。
        - vmhd/smhd/……：视频、音频及其他媒体的基本描述
        - dinf：数据信息。
        - stbl：样本表，内含媒体信息的详细信息。

其中，moov→trak→mdia→minf中的stbl是一个比较重要的box，其中包含的stsd box内有解码器需要的媒体描述信息；stss内有关键帧信息；stts、stco、stsc、stsz用于构建索引，其中stts给出每个数据帧的时间信息、stco给出每个数据Chunk在文件中的偏移、stsc给出个各个数据Chunk中包含的数据帧、stsz给出各个数据帧的长度；所有媒体数据则统一存放在mdat box中，没有同步字，没有分隔符，只能根据索引进行访问。mdat的位置比较灵活，可以位于moov之前，也可以位于moov之后，但必须和stbl中的信息保持一致。但是，如果mdat的位置在moov之前，通过流的方式播放文件会出现问题，因为没有办法在一开始就获得文件的媒体信息和索引。



Box的扩展通过uuid实现。用户可以使用类型为'uuid'的box，以16个特定的字节作为标识，定义自己的数据格式。

目前，各种类ISO 14496-12格式如MOV、F4V、3GP等在数码相机、互联网视频、移动视频等领域应用相当广泛，然而由于HTML5的问世，其主导地位受到基于MKV的WebM格式的威胁。

3.2.2.3 媒体信息的封装

3.2.3 ASF

1995年起微软着手开发新的媒体格式ASF，这是一种适合网络传输的流媒体格式。相对于AVI而言，ASF引入了很多改进，包括：

- 使用128比特的GUID代替四个字节的fourCC；
- 使用64比特的长度域，支持超大文件；
- 定义了三个基本的顶层对象：Header，Data和Index，这些对象可以独立存储、传输；
- 数据打包允许分割（一个原始媒体数据分成多个包）和分组（多个原始媒体数据打入一个包）；
- 数据包的首部包含有时间信息。
- 支持基于绝对时间、相对偏移的索引信息，支持多个媒体流的独立索引。

3.2.3.1 结构

一个ASF文件由三种最基本的顶层对象组成：Header, Data和Index，亦即头、数据和索引，其中，索引是非强制的。头对象相当于AVI的"hdrl" List，内部可以包含多个子对象；数据对象相当于AVI的"movi" List，由多个封装媒体数据的包组成；而索引则相当于"idx1" Chunk，给出各个媒体的数据在数据对象内的偏移与时间的对应关系。通常，一个典型的ASF文件的结构如下：

```
ASF Object Found with Size 144957266
      Header Object Found with Size 5326
          Codec List Object Found with Size 302
          File Properties Object Found with Size 104
          Header Extension Object Found with Size 4268
          Extended Content Description Object Found with Size 328
          Stream Properties Object Found with Size 122
          Stream Properties Object Found with Size 134
          Unknown Object Found with Size 38
      { 7 headers contained. }
      Data Object Found with Size 144948546
      { 8114 packets contained. }
      Index Object Found with Size 1934
      Simple Index Object Found with Size 1460
```

ASF不支持嵌套结构，头对象是唯一可以包含子对象的对象，内含有各种与文件结构和媒体流相关的信息。

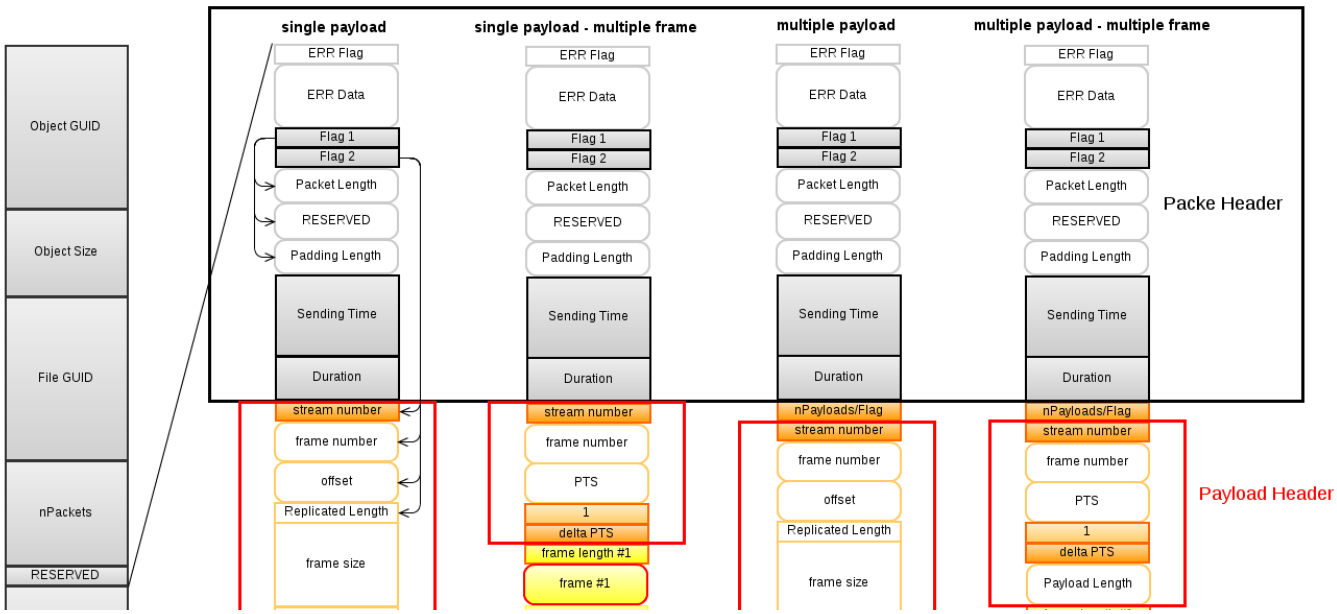
File Properties对象相当于AVI中的"avih" Chunk，给出了媒体文件的时长、数据包的总数、预缓冲时长、比特率以及一些标志等信息。其中，如果预缓冲时长不为零，则说明媒体文件时长以及数据包的时间戳已经加上了这个预缓冲时长。

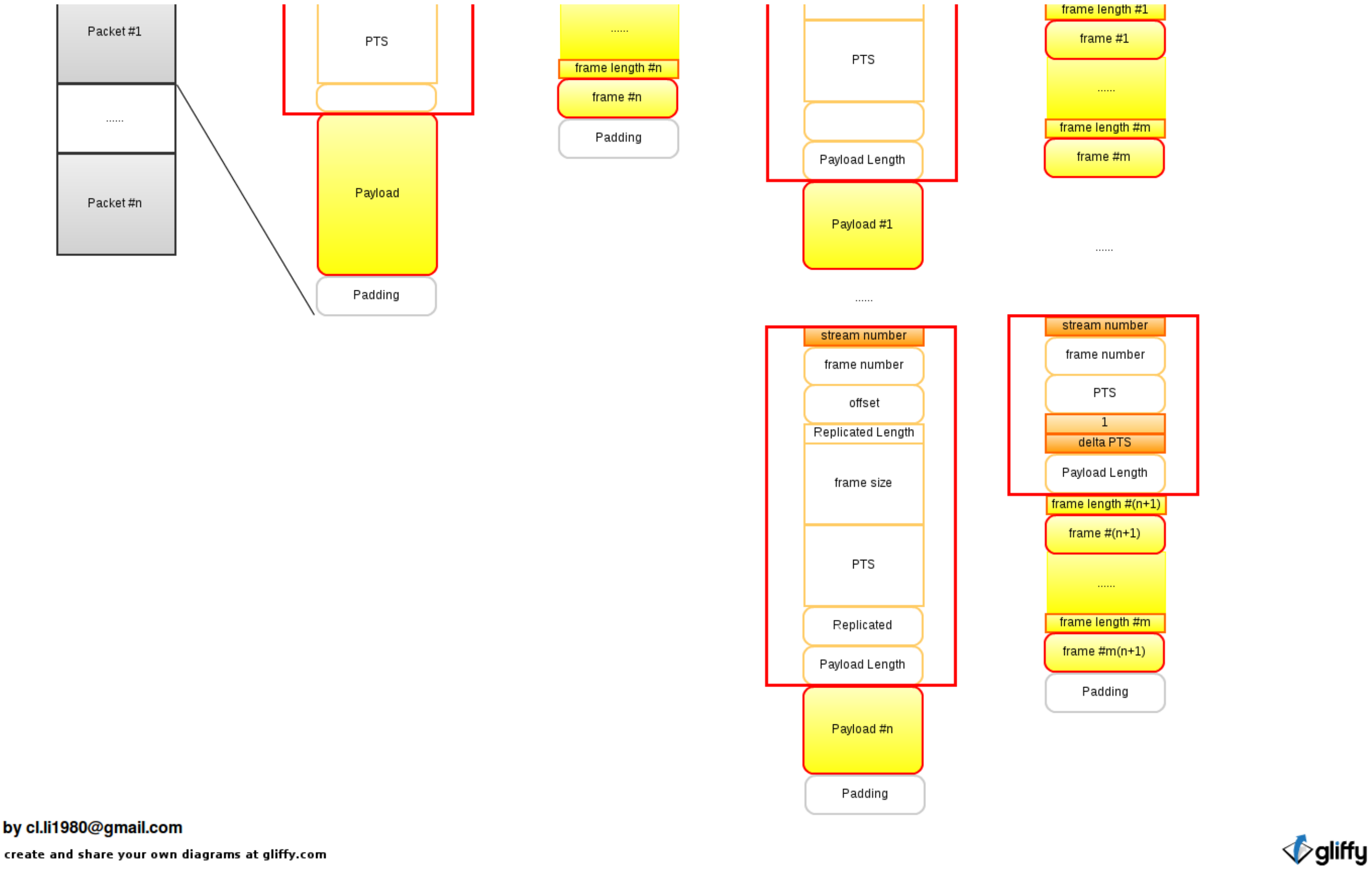
Stream Properties对象相当于AVI中的"strl" List，基本上是将AVI中"strh"和"strh"中信息统一到一个数据结构中。其中，Stream Type以GUID的形式给出，Stream Number限定为1~127。而与媒体类型相关的信息结构也因媒体类型的不同而不同，与AVI一样，音频信息使用WAVFORMATEX结构，而视频信息使用BITMAPINFOHEADER结构。Stream Properties对象的个数取决于文件中包含的媒体流个数，每一个媒体流都需要一个Stream Properties对象（由于File Properties对象中并不象AVI文件的"avih"一样包含媒体流个数，需要通过统计Stream Properties对象的个数来得到文件包含的媒体流个数）。

Header Extension对象允许通过嵌入子对象定义扩展的头信息。 这里有一个简单的解析ASF文件的JAVA程序。

3.2.3.2 打包和解包

ASF是一种流媒体格式，基本的媒体数据封装单位为包，封装方式非常灵活：包长可以是固定的，也可以是可变的；包内可以含有一个载荷，也可含有多个载荷（最多64个）；每个载荷可以是部分媒体数据帧，也可以是整个媒体数据帧，还可以是多个媒体数据帧。下图给出各种封装方式的示意：





当采用固定包长的方式封装时，包头中的Packet Length通常可以省略，实际的包长在文件头对象中给出，如果采用可变包长的方式进行封装，Packet Length字段必须出现在包头中，可以是8位、16位或32位，最大的包长可以达到4GB。包头中的时间戳表示该包的发送时间。

载荷紧跟包头，由首部和数据组成，由于载荷中承载的不一定是完整的数据帧，其首部需要给出完整的数据帧的长度和当前载荷在数据帧中的偏移量，以便解包程序进行组帧。当一个包内含有多个载荷时，不同的载荷可以来自不同的媒体流，也就是说在一个包内就可以实现媒体的交错。

在某些情况下，一个载荷中可以包含多个数据帧，但要求：

1. 所有的数据帧来自同一个媒体流
2. 所有的数据帧都是关键帧或都不是关键帧
3. 长度在256个字节之内
4. 帧率固定

3.2.3.3 索引

ASF Index对象提供索引信息，由若干索引块组成，每个索引块包含多个索引项，各索引项对应的的时间点的间隔给定，索引项内包含各个媒体流的数据在该时间点上对应的偏移，这个偏移需要加上索引块中给出的一个基础偏移，最终的结果是相对于数据对象中第一个数据包的偏移。

索引信息有三种：

1. 指向最近的包含给定时间点上的媒体数据帧的数据包；
2. 指向包含给定时间点上的媒体数据帧的起始部分的数据包；
3. 指向包含距给定时间点上的媒体数据帧最近的关键帧的起始部分的数据包。

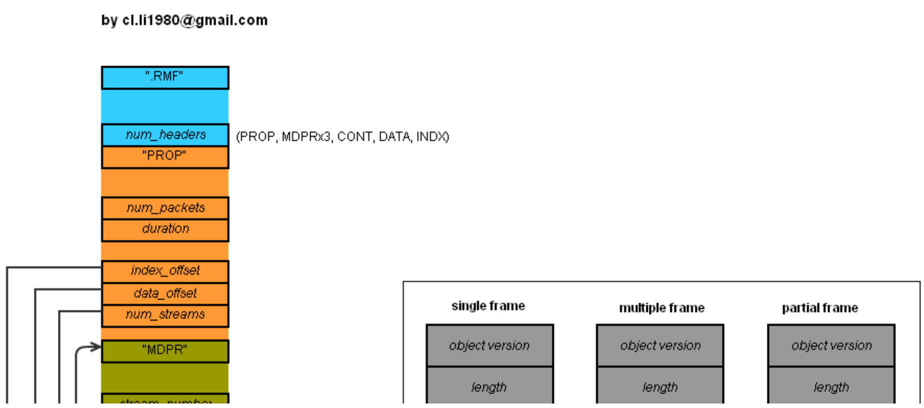
```
sending time:  |  1000  |  2000  |  3000  |  4000  |  5000  |  6000  |  7000  |
frame number:  |  1     |  1     |  2     |  2     |  3     |  3     |  3     |
Key:           |  Y     |  Y     |  N     |  N     |  N     |  N     |  N     |
               ^                                   ^                 ^
time=7750      |                                   |                 |
          index type #3                     index type #2  index type #1
```

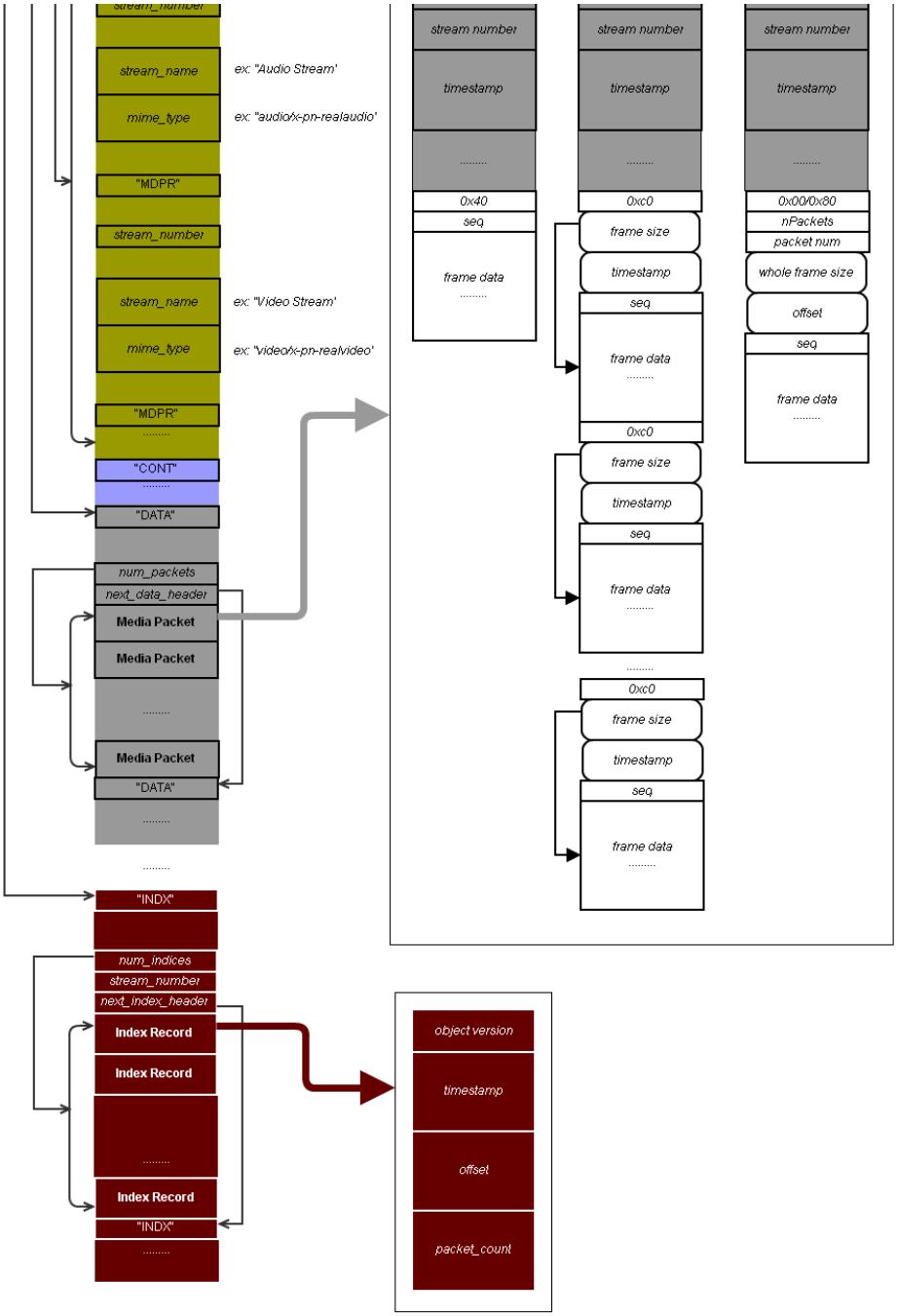
3.2.4 RealMedia

这是RealNetworks开发的一种多媒体文件格式，早期只支持固定比特率(CBR)，扩展名为rm，属于最早的互联网流媒体格式之一，曾经红极一时。然而，虽然CBR支持有限带宽下的高效数据传输，却无法保证视频质量，这使得rm一度成为低质视频的代名词，有时某些画面几乎是惨不忍睹。后来，RM逐渐退出了互联网视频舞台，取而代之的是FLV和基于ISO格式的F4V，虽然后来RealNetworks在新的编码器中增加了变比特率(VBR)的支持（以rmvb为文件扩展名），仍旧无力回天，目前，Real格式仅在中国地区比较流行。

3.2.4.1 文件结构

RealMedia 文 件 由 首 部 、 数 据 和 索 引 三 大 部 分 组 成 ， 数 据 组 织 方 式 与 ASF 非 常 相 似 。





create and share your own diagrams at gliffy.com

Chunk是基本的数据构成单位，每个Chunk起首为一个32位的FourCC和一个32位的长度字段，数据紧随其后。FourCC是四个ASIC字符，标识Chunk的类型，长度字段则标明净数据的字节数。

首部包含四种Chunk：File Header, Properties, Media Properties和Content Description，FOURCC分别为：“.RMF”，“PROP”，“MDPR”和“CONT”，详细的数据结构如下所示：

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

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

在File Header中，num_headers给出接下来文件中Chunk的个数，这个数目不仅仅包含了各种首部Chunk，也包含了DATA Chunk和INDX Chunk（但对于多INDX Chunk，似乎只算做一个）。

```
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;
  }
}
```

Properties结构给出了比特率、包长、数据包的总数、时长、索引块的偏移、数据块的偏移以及媒体流的数目。

```
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;
  }
}
```

Media Properties给出特定媒体流的比特率、包长、起始时间、时长、媒体名称、MIME类型等信息。此外，这个结构的最后可以是一个依赖于具体流类型的黑盒结构，内含与媒体流的相关的特定信息，如编解码参数等，从而提高这一容器的可扩展性，使其能够支持各种不同的媒体流。

对于视频流来说，主要的信息包括：

Field Name	bits
长度	32
类型标记（VIDO）	32
子类型标记	32
宽度	16
高度	16
比特位数	16
填充宽度	16
填充高度	16
帧率	32

```
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;
  }
}
```

Content Description中包含了一些简单的影片说明信息（相当简单）。

通常，数据部分紧接着首部，由一个或多个以"DATA"为标识的Chunk组成，每个数据Chunk包含了一系列交织的媒体包以及一个指向下一个数据Chunk的指针：

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

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

以上是Chunk的头结构：num_packets给出了本Chunk内数据包的个数，next_data_header如果不为0的话表示下一个数据Chunk相对于文件首的偏移量，不过Real的SDK文档中说："This field is not typically used"。

索引Chunk以"INDX"为标识，每一个媒体流有自己对应的索引Chunk

```
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;
  }
}
```

num_indices是索引项的个数，stream_number指示对应的媒体流，next_index_header为下一个索引Chunk基于文件首的偏移量。

```
IndexRecord
{
  UINT16   object_version;

  if (object_version == 0)
  {
    u_int32  timestamp;
    u_int32  offset;
    u_int32   packet_count_for_this_packet;
  }
}
```

这里有一段用于解析Real Media文件的代码。

3.2.4.2 打包和解包

作为一种流媒体格式，与ASF相似，Real Media同样是以包作为基本的数据封装单位，包头附有时间戳，支持媒体帧的分割打包和组合打包。Real Media规定的包长最大不能超过65536字节，而且打在一个包内的媒体数据必须来自同一个媒体流。对于组合打包模式，Real Media要求每帧媒体数据之前附上的数据帧长度和时间戳；对于分割打包模式，则要求附上整个数据帧的长度和当前数据在整个数据帧中的偏移。

```
Media_Packet_Header
```

```
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();
  }
}
```

3.2.4.3 逻辑流

RealMedia格式支持逻辑媒体流的概念，以实现对多个物理媒体流进行分组。一个逻辑媒体流拥有自己的Media Properties，其中会给出该逻辑流所包含的物理流的编号、数据位置等信息。假如一个RealMedia文件包含一个视频流和一个逻辑流，该逻辑流由两个不同比特率的音频流组成，则该文件要包含四个Media Properties头，分别对应一个视频流、两个音频流和一个逻辑流，但只包含一个视频流和两个音频流的数据包。

逻辑流Media Properties中的type_specific_data是如下一个结构：

```
LogicalStream
{
        ULONG32                                         size;
        UINT16                                          object_version;

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

3.2.5 MKV

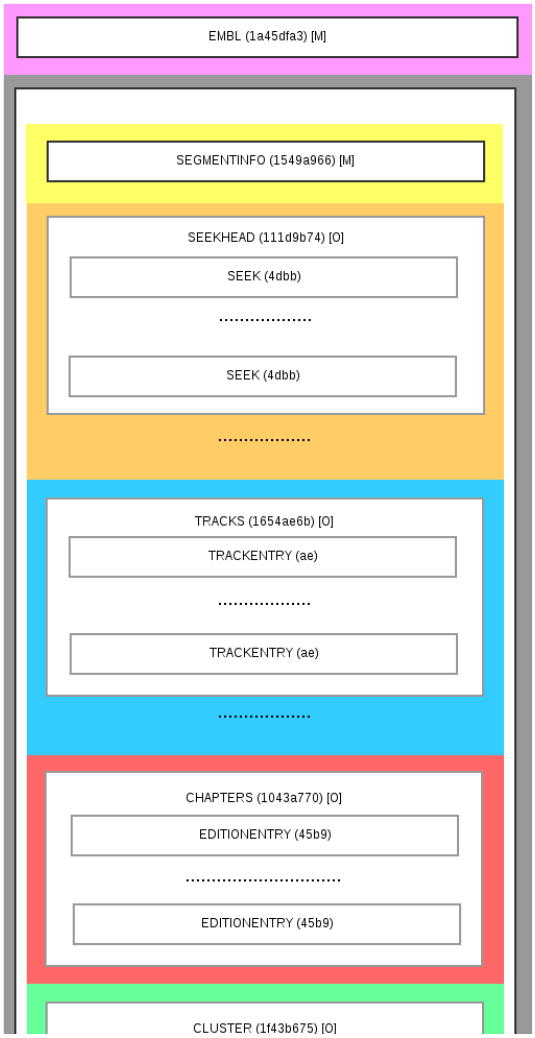
MKV是俄罗斯人于2002年发起的一个开放标准，2010年成为WebM格式的基础，借助HTML5的兴起，有望成为目前流行格式FLV/F4V的有力竞争对手。

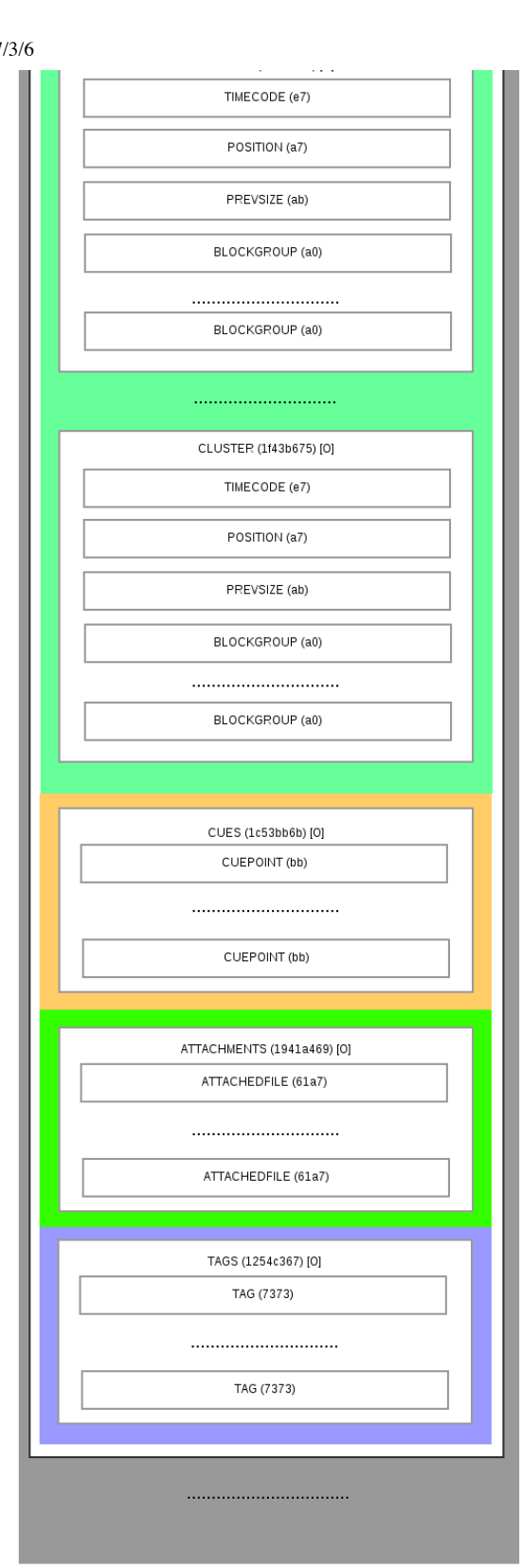
3.2.5.1 文件结构

MKV文件的基本数据单元叫做Element，每个Element依然以ID/Size的形式开始，不同的是ID/Size采用可变长度的EBML编码。

和一般的多媒体容器类似，MKV定义了SEGMENTINFO来承载文件信息，定义了TRACKS/TRACKENTRY来承载媒体流的信息，定义了CUES来承载索引信息。除此之外，MKV还定义了CHAPTERS来支持类似于DVD的章节功能，定义了ATTACHMENTS允许将文件作为附件。

by cl.li1980@gmail.com





MKV采用二级结构存储媒体数据，首先，MKV文件中包含有多个CLUSTER，而每个CLUSTER包含若干BLOCKGROUP，BLOCKGROUP内的BLOCK Element存储一个或多个媒体数据帧以及某些附加信息。承载不同媒体流的数据的BLOCKGROUP在CLUSTER中交错存放，CLUSTER首部会给出一个时间戳，作为其内部各BLOCKGROUP中的媒体数据的时间信息的基准。为了减少数据量，也可以不使用BLOCKGROUP而将BLOCK直接存放在CLUSTER中，这种情况下的BLOCK称为SIMPLEBLOCK。

3.2.5.2 索引和随机访问

在MKV文件中，每个索引项由一个CUEPOINT表示，其中包含一个CUETIME和多个CUETRACKPOSITIONS，CUETIME表示当前索引项对应的时间点，CUETRACKPOSITIONS则给出该时间点上某个媒体流对应的媒体数据的在文件中位置，每个CUETRACKPOSITIONS包含track号，目标CLUSTER相对于文件的偏移量，目标BLOCKGROUP在CLUSTER中的编号。通常，MKV只对关键帧作索引。

3.2.5.3 章节信息

3.2.5.4 打包和解包

通常，MKV文件的媒体数据经由CLUSTER和BLOCKGROUP二级封装，如下：

```
CLUSTER
    Timecode(uint)
    Postion(uint)
    Prevsize(unit)
    BLOCKGROUP
        Reference(int)
        Duration(int)
        BLOCK
            Tracknumber(vint)
```

```
            Timecode(sint16)
            Flags(int8)
            Framedata
```

CLUSTER首部的Postion给出了该CLUSTER相对于SEGMENT数据起始的偏移量，Prevsize给出了上一个CLUSTER的字节数（含ID/Size部分），为文件损坏的情况下进行重同步提供了有效的线索。

BLOCKGROUP中的Reference以相对时间的形式给出了当前媒体帧对其它帧的依赖关系，Duration为媒体数据的持续时间（一般用于字幕）。

BLOCK中的Tracknumber标识媒体数据所属的媒体流；Timecode是媒体数据的时间标签，相对于CLUSTER中的Timecode。Flags各位元定义如下：

```
Bit 0x80: keyframe:
          No frame after this frame can reference any frame before
          this frame and vice versa (in AVC-words: this frame is an
          IDR frame). The frame itself doesn't reference any other
          frames.
Bits 0x06: lace type
           00 - no lacing
           01 - Xiph lacing
           11 - EBML lacing
           10 - fixed-size lacing
Bit 0x08 : invisible: duration of this block is 0

Bit 0x01 : discardable: this frame can be discarded if the decoder
           is slow
```

其中，Lace允许将多个媒体帧封装到一个BLOCK中。如果使用Lace，则紧跟BLOCK首部信息的是一个标识总帧数的字节，然后是一系列帧长度的信息，语法因Lace方法不同而不同，然后才是媒体帧数据。

MKV CLUSTER也有简化的语法，使用SIMPLEBLOCK代替BLOCKGROUP：

```
CLUSTER
    Timecode(uint)
    Postion(uint)
    Prevsize(unit)
    BLOCK
        Tracknumber(vint)
        Timecode(sint16)
        Flags(int8)
        Framedata
```

3.2.5.5 Divx HD

3.2.5.6 WebM

3.3 同步问题

媒体同步主要涵盖了三个方面的内容，其一是媒体自身的同步，即保持媒体样本在时间轴上的相对关系，以满足受众的感知要求，譬如需要按照正确的采样率播放一段声音，如果采样率不对，受众听到的声音会走样；其二是媒体之间的同步，这是为了保证媒体之间的时间关系，譬如唇音同步：声音和口型要对得上，也即声音和图像的播放在时间轴上要保持一致。最后就是发送端和解收端的同步，这是一种最强烈的同步，要求接收端各媒体在时间轴上的分布与发送端保持完全一致，譬如电视直播应用中，各个接收端播放某个媒体样本的时刻必须一致。除直播和实时通信以外的多媒体系统通常只需要满足前两种同步关系既可。

3.3.1 同步机制的实现

首先探讨如何实现媒体自身的同步要求，主要是声音和视频。由于两种媒体的播放设备存在着差异，其同步机制也不尽相同。声音数据的同步控制和播放通常完全由硬件实现，音频芯片有内置的时钟，只要采样率设置正确，数字样本可以被准确地转换为模拟音频。为了保持一定的播放速度，音频芯片需要源源不断地读取数据到其内置缓冲中，如果不能及时拿到数据，会出现缓冲下溢，而如果缓冲满的情况下向音频芯片写入数据，则会出现缓冲上溢。所以，音频芯片驱动通常会提供回调机制，应用程序利用这种回调机制发送数据给硬件，这也是一种典型的“拉”的数据传输方式，可以有效避免上下溢的发生。视频则不然，视频设备只提供一个用于更新图像的帧缓冲，图像数据需要在一个外部定时器的控制下写入帧缓冲中，写入的时刻也需要由外部控制。

- 音频同步机制一（回调方式）：

当音频播放设备的数据缓冲快要空的时候会激活某种回调机制，从而使上层注册的回调函数得以调用。回调函数负责从本地数据缓冲拷贝数据到音频设备缓冲，如果本地数据缓冲空则启动解码过程，以获取更多的数据。在这一过程中，音频设备控制着音频播放的同步，假如回调函数因某种原因没有及时拷贝数据，则音频设备发生缓冲下溢，体现为声音输出的停顿。

- 音频同步机制二（轮询方式）：

在这种方式下，音频数据的写入由独立的线程来实现，该线程定时检查本地数据缓冲和设备缓冲的水平，然后根据情况从解码队列中获取音频数据或将本地数据写入到设备缓冲。在这种方式下，如果不能及时获取解码数据，则会导致音频设备缓冲下溢，体现为声音输出的停顿。

- 视频同步机制（轮询方式）：

视频的同步方式略有不同，完全依赖于外部时钟。这种方式也需要启动一个定时器，定时检查解码队列，如果解码队列中有图像，则比较该图像的时间戳和当下的系统时钟，如果系统时钟已经到了播放的时间，则拷贝该图像到帧缓冲中。而如果系统时钟远远超过了图像的时间戳所指示的时间，说明发生了“图像迟到”，体现为屏幕上的画面停顿。

再看音视频间的同步。音视频间的同步机制建立在音频同步机制和视频同步机制的基础上，通常有以下三种策略：

- 基于音频时钟的策略

这种策略下，音频的同步方式可以采用以上的两种方式的任意一种，视频同步以音频播放的时间为准，即根据音频播放的时间确定当前的图像是否“迟到”。如果发生了“图像迟到”，首先要将该图像丢弃，同时启动跳帧策略，通知解码器以适当的频率在解码后直接丢弃图像，直至重新恢复同步。如果“图像迟到”是因某种突发情况产生的（解码使用的CPU被突然大量占用、媒体文件中暂时无法获得视频数据等），突发情况消失后调帧策略会加快解码输出速度，在一定的时间内重新恢复同步。如果音频发生缓冲下溢，音频时钟会变慢，由于音频时钟同时是参考时钟，会导致视频的播放受到影响，如果缓冲下溢持续的时间很长，视频播放会发生停顿，但下溢消失之后同步可以立即恢复。

- 基于视频时钟的策略

这种策略下音频同步以视频播放的时间为准，若本地缓冲中音频数据的时间戳远大于视频时钟，则暂停向音频设备缓冲注入数据；若本地缓冲中音频数据的时间戳远小于视频时钟，则丢弃该数据，这两种情况均会导致音频设备缓冲下溢。第一种情况往往是由视频方面获取解码数据时间过长引起的，因为那会导致视频时钟变慢，第二种情况则是由于获取音频解码数据时间过长引起的，这种情况下大量的音频数据会被丢弃，需要较长的时间恢复同步。

- 基于独立时钟的策略

这种策略采用一个独立的时钟作为标准时钟。对于视频，如果发生了“图像迟到”，丢弃该图像之后立即调慢系统时钟，虽然同时会导致声音的缓冲下溢，但引发“图像迟到”的因素消失之后同步迅速恢复。同样，对于音频，如果获取音频数据时间太长导致缓冲下溢发生，也可以立即调慢系统时钟，虽然同时会导致图像停顿，但恢复同步的时间大大缩小。

音频设备缓冲下溢现象和视频的“图像迟到”现象都属于媒体失步现象，一个好的多媒体系统首先要竭力避免这种情况发生，通常认为，可以觉察到的视频延迟范围为-125ms到45ms，而能忍受的范围为-185ms到90ms[3)]。其次，系统要具备发生失步之后尽可能快地恢复同步的能力，为达到此目的，需要综合传输协议设计、媒体容器设计、缓冲控制及同步策略等多个方面来设计系统。

3.3.2 失步的预防

上一节给出的策略不是为了避免失步，而是为了在发生失步的情况下恢复尽快同步。避免失步必须从协议、容器和缓冲策略几个方面来考虑。 首先，在设计传输协议和媒体容器时，要尽可能保证音视频数据适当的交错，避免长时间无法获取某种媒体数据的情况发生。这一条对于实时媒体的传输尤其重要，因为缓冲策略会增加延时，无法应用于实时通信。其次，要根据传输协议构造一定大小的传输缓冲，以防止网络抖动和音视频交错不好导致的媒体数据迟到；此外最好构造适当长度的解码缓冲来保存解码后的音视频数据，以防止解码抖动导致的媒体数据迟到。

3.4 流媒体技术

流媒体的概念仍旧可以追溯到模拟时代，但其内涵随着通信传输技术的演进也在不断变化。简单讲，流媒体技术的核心即是将多媒体数据从发端源源不断地传送到接收端并使其能够在接收端持续播放，所以，模拟时代的电视、广播都属于流媒体的范畴。数字化革命发生之后，通信网络开始承载数字信息，于是，发送端采集的数字音视频得以经编码器压缩之后同控制信息一起复用到调制解调器所支持的数字信道中，经由传统电话网发送到接收端，最终由接收端解码并播放，这便是ITU-H.324建议中描绘的可视电话及视频会议应用场景，也算是数字流媒体的一个先例。然而，当通信网络泛IP化之后，流媒体技术也随之演进为一种在IP网络的基础上实现数字媒体自发端到终端的持续传输的解决方案，通常体现为一系列基于IP的传输协议和控制协议的组合。由于多媒体信息具有强烈的时间相关性，因此传输过程必须保证媒体的连续性，延时抖动能够得到严格的控制。为了适应IP传输，流媒体格

式出现了，首先是微软的ASF，然后是RealNetworks的RM，它们共有的特性是将媒体数据如编码的音频帧或视频帧封装到包中，包头附有时间标签，以便实现媒体数据在IP网络中的无缝传输以及接收端的乱序重组和丢包统计，同时保证一定的实时性要求。彼时，流媒体应用尚集中在通信领域如视频通话、远程监控等，但随着传输带宽的飞速提升，娱乐逐渐成为流媒体的主要应用领域，于是，实时性的优先级逐渐让位于内容质量——毕竟，为了观看高品质的电影，用户可以忍受一段并不算太长的缓冲时间，而内容提供商也获得了一个不可多得的广告投放机会，此时，更为简单而成熟的HTTP协议变成了主流。

3.4.1 HTTP与RTSP

HTTP是IP网络中占据统治地位的应用层协议，这一点在今天来讲是毋庸置疑的。数十年来，当初基于不同目的而设计的各种应用层协议不断地被这一基于简单的请求/应答机制的无状态协议所替代。HTTP很简单，但它可以实现文件传输、信息检索、即时消息传送，用户会话、电子邮件等各种其他专有协议所负责的功能，虽然在性能上HTTP并非最优，但一个通用的应用协议对于构造一个富应用的下一代互联网来说是非常必要的，这不仅可以降低网络基础架构及软件平台的成本，也能够简化应用开发的模型。

为流媒体传输设计的RTSP便是这一泛HTTP化过程中的牺牲品。目前，互联网上大部分视频都是通过HTTP来进行传输的，自从HTTP 1.1增加Accept-Ranges支持文件的部分传输以克服其致命缺点之后，其剩余缺陷如较大的头开销、重传引起的延时、长时间缓冲造成的带宽浪费等随着基础网络的飞速发展变得不足为道了。

RTSP采取的则是另一种策略：首先，它是个有状态的，RTSP允许客户端通过发送一系列初始化请求与服务器建立一个多媒体会话，会话建立之后，客户端能够通过发送播放或暂停命令启动或停止服务器端的媒体推送。媒体的信息由服务器以SDP包的形式提供给客户端，而媒体的传输由RTP/RTCP系列协议实现，由于RTP只是对UDP的一个简单封装，因此客户端要负责RTP包的排序及丢包统计；媒体流的推送在服务器端完成，RTSP服务器负责解析目标媒体文件、抽取媒体流并打包发送，同时还需要根据以RTCP包回传的统计信息调整推送策略。基于RTSP的流媒体传输具备带宽开销低，实时性好，延迟抖动小等优点，适用于各种专业的流媒体推送应用，尤其是基于IP网的流媒体直播。只是一直以来互联网对RTSP并不友好，各种防火墙、代理服务器的存在为RTSP的部署制造了重重障碍，再加上高性能的RTSP服务器成本高，构建复杂，RTSP并未得到广泛的应用。而随着HTTP对自适应流传输的支持逐步增强，RTSP的前景越发不被看好。

HTTP的另一个影响是流格式的式微：由于HTTP具备强大的传输控制能力，包的概念在多媒体文件格式中变得不再重要，先前为了适应流传输而定义的包封装对于HTTP来说成为一种冗余。事实上，就目前基于互联网的多媒体应用来看，ISO文件格式远比ASF流行。

3.4.2 SDP

SDP用于描述一个多媒体会话，如发起者信息，媒体流的格式，网络连接信息等。协议的设计秉承了TCP/IP协议簇的一贯传统，简洁而有效。数据格式基于文本，以行为单位定义各个域的值。如：

```
m=video 1234 RTP/AVP 96
```

表示当前会话包含一个视频流，以RTP/AVP形式传输，端口为1234，格式为96，继而是对视频格式的补充：

```
a=rtpmap:96 H264/90000
```

即时钟为1/90000的H.264编码。而：

```
c=IN IP4 10.131.56.1
```

表示当前会话使用一个目的为10.131.56.1的IPv4连接

3.4.3 RTP/RTCP

RTP是对UDP的一个轻量级封装，在UDP的基础上增加了包序号和时间戳等信息，解决了多媒体传输时的同步和乱序问题。

3.4.4 基于HTTP的自适应流技术

欠实时性是HTTP流传输技术的致命缺陷，它限制了HTTP流传输在某些特定场合的使用，如直播、监控以及视频会议等。此外，剧烈的带宽波动常常致使网络的瞬时带宽降低至不足以承载正在传输的媒体的比特率，此时，大量的重传反过来又加重了带宽的负载，结果导致媒体播放发生频繁的中断，严重影响用户的主观感受。

苹果公司使用了一种叫做HLS（HTTP Live Streaming）的自适应流传输技术来解决带宽波动的问题。该技术要求服务器端提供内容相同但是比特率不同的多个备选媒体流，每一个媒体流都被分割为大量小的MPEG TS文件，如是，在传输过程中，客户端将不再请求一个大的媒体文件，而是不断地请求分割开来的片段文件，因此，当带宽发生变化时，客户端可以平滑切换至比特率更合适的流，以降低带宽的负载。通常，HLS自适应流的URL是一个播放列表，其中给出了媒体的基本信息以及各个流的描述，客户端可以根据该列表发起传输请求、进行流的切换。

另一种相似的技术DASH（Dynamic Adaptive Streaming over HTTP）则是由MPEG发起的国际标准。之前另外两个组织3GPP（3rd Generation Partnership Project）和OIPF（Open IPTV Forum）曾先后提出了Adaptive HTTP streaming （AHS）和HTTP Adaptive Streaming（HAS）技术，分别给出了针对手持设备和IPTV的自适应传输方案，这些方案成为了DASH的基础。正因如此，DASH同时支持MP4和MPEG2 TS两种文件格式，而作为国际标准，DASH也提供更多文件格式的扩展。

总的来说，自适应的HTTP传输基本上解决了带宽波动的环境中音频频点播和直播的用户体验问题，但仍旧无法满足强实时性应用的要求（如视频通话），毕竟，以秒级计算的延迟是无法接受的。

3.5 设备与接口

3.5.1 I2S

I2S使用三个信号来传输双声道的数字音频信息，分别为BCLK，LRCLK和DATA。准确地讲，I2S属于一种板上传输标准，因此没有统一的硬件接口规格。

BCLK是数据传输的基础时钟，每一个时钟周期传输一个比特音频数据，BCLK = samplerate × bitwidth × 2；LRCLK定义左右声道传输周期。多组I2S接口可支持多声道PCM的传输。

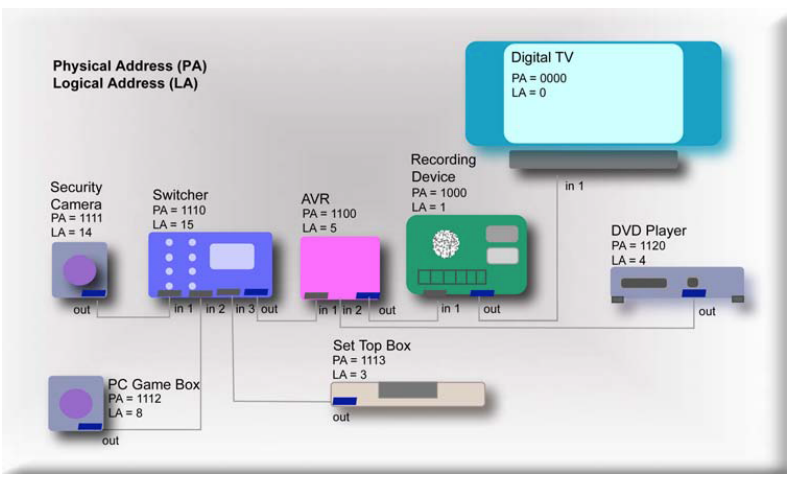
3.5.2 S/PDIF

S/PDIF是单纯的数字音频传输接口，通常用于连接播放器和功放，有两种物理接口：同轴和光纤。数据传输格式基于IEC标准60958和61973，分别支持PCM格式和某些压缩格式（杜比AC3和DTS等）。

尽管是数字传输，S/PDIF接口没有时钟线，接收设备需要从差分编码的音频信号中重建时钟。

3.5.3 HDMI

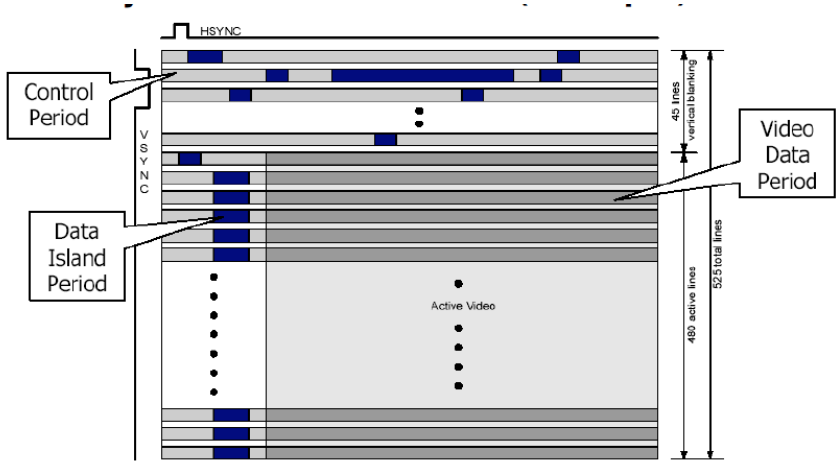
HDMI是较早出现的一种数字传输接口，用于在一定距离内同步传输高品质的多媒体数据，如高解析度的原始数字视频、打包的数字音频及辅助数据等。通过HDMI接口，可以建立音视频接收设备和显示设备及音箱之间的纯数字连接，此外，借助于HDMI中继器和交换器，还可以在家庭范围内搭建一个数字化视听系统，统一实现设备的控制和媒体的推送。



HDMI接口由3个TMDS数据通道，1个TMDS时钟通道，1个基于I2C的DDC通道，一根CEC连接线和一个热插拔检测脚组成。

3.5.3.1 TMDS

TMDS是以差分方式进行数字传输的高速信道，最新的2.0标准支持高达8Gbps的传输速率，以满足承载60Hz 4K原始视频的需求。



音频的传输与IEC 60958标准兼容，是以包的形式插入到视频信号的场、行间隙中的。音频数据没有独立的时钟，其实际时钟需要根据控制信息在视频时钟的基础上重建。最新标准将所支持的音频声道数由8增加至32，同时也提高了压缩音频（AC3， DTS及Dolby TrueHD等）的比特率上限。

3.5.3.2 HDCP

HDCP协议实现了HDMI传输过程中的数字版权保护，以防止版权保护的数字影音在传输的过程中被非法截获。相关的协议交互通过DDC通道完成，整个过程稍显复杂：

首先，在决定是否发送媒体数据之前，HDMI发送设备会对接收设备或转发设备进行认证，确认其为经HDCP机构认证过的设备，对于未经认证的设备，发送端可以拒绝发送数据或仅传送较低质量的视频；其次，发送一帧视频数据之前，发送端会对数据进行加密，并确认接收端能够正确解密该数据，因此，TMDS信号传送的实际上是加密的数据，理论上讲直接截获TMDS信号是无法重建原始视频的；最后，HDCP协议还引入了黑名单机制，持续更新的黑名单记录了被破解的设备，HDMI发送设备可以通过输入的媒体数据获取更新的黑名单。

协议的复杂导致实现的不便：哪怕是一个简单的HDMI转发器都需要至少一个微处理器用以实现协议交互，同时还需要具备一定的运算能力完成实时加密和解密。

3.5.3.3 CEC

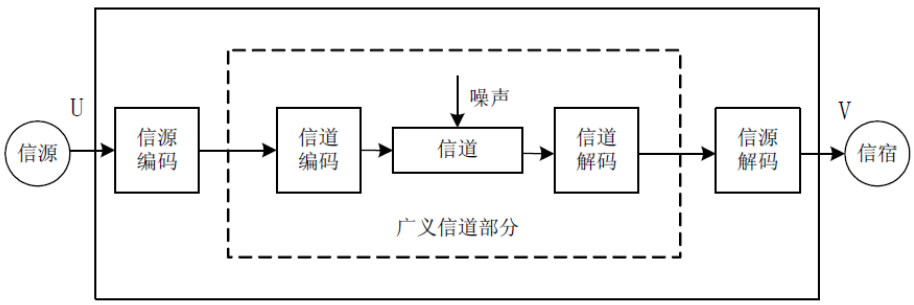
CEC连接允许在两个HDMI接口之间交换CEC控制信息，一个由HDMI连接构成的CEC系统能够实现家庭视听设备的统一管理和控制，包括下列功能：

- 一键播放：允许一个播放设备唤醒电视并将该播放设备设定为当前输入。
- 一键休眠：允许用户通过一键促使家庭视听系统中的所有设备都进入休眠状态。
- 一键录制：允许用户通过一键启动相应的录制设备开始录制当前电视播放的内容。
- 播放控制：允许用户通过TV控制播放设备的播放动作，如播放、暂停、快进、后退等。
- 调台控制：允许用户通过TV控制机顶盒的调台功能，如换台，切换模拟数字源等。
- 声音控制：允许音频放大器充当TV当前输入的声音输出设备，从而通过TV或STB的音量键实现音频放大器的音量控制。
- 遥控器转发：允许TV将某些遥控器事件转发给CEC系统中的某个特定设备。
- 菜单控制：当TV在显示某个输入设备的菜单时，允许TV将当前遥控器的事件转发给相应设备，从而实现利用TV的遥控器控制输入设备的菜单。
- 路径控制：要求HDMI交换器能够识别当前活跃设备并保证其与TV之间的连接。

四、压缩技术

4.1 理论基础

廿世纪中叶，为了从理论上证明对信息系统进行优化的可行性，Shannon引入了熵的概念，用来表示信息的不确定性，熵越大，信息的不确定性就越大[4]，而信息的不确定性越大，其对应的传输和存储成本就越高。换句话说，如果某种信息的熵不是那么大，则人们应该有信心使用有限的资源去承载它。举一个简单的例子，假设气象台负责预报明天是否天晴，而地震局负责预报明天是否地震，那么显然，来自气象台的信息要比来自地震局的信息具有更大的不确定性，也就是说气象信息的熵更大，如若使用喇叭来传递信息，对于气象台而言，以鸣喇叭来表示天晴或者表示天阴，对喇叭的使用寿命影响并不大，地震局则不然，如果以鸣喇叭来表示地震，那这喇叭的使用寿命远大于气象台的那一只。这说明，信息传输的成本是有下限的，这个下限由信源的熵决定，而如何达到或接近这个下限成为通信领域的主要研究内容，数据压缩便是其中的主题之一，在Shannon的通信模型中属于信源编码的范畴。



通过建立一个简化的信源模型可以算出熵的最大值，这是非常有意义的，基于这个最大熵可以得到传递信息的极限成本。离散平稳无记忆信源就是这样的一个简化模型，源自这种信源的信息统计特性相同，但相互独立，于是可以用一个概率空间[**M**, **P**]来抽象这些信息，其中**M**={M1, M2, ..., Mk}是概率分布为**P**={P1, P2, ..., Pk}的一个随机变量，那么**M**的熵由下面公式给出：

$$H(M)=-\sum(Pi*\log_2 Pi)$$

公式中$-\log_2 Pi$表示Mi的信息量：概率越大信息量越小。于是不难发现，H(**M**)不过是信息量的一个概率平均。对于离散平稳无记忆信源，H(**M**)也可以看作信源的熵，针对某种特定的分布，这个熵存在最大值，对应的分布叫做最大熵分布。离散无记忆信源的最大熵分布是均匀分布，此时其熵值为$\log_2(k)$，k是其可能值的个数。

如果信源是有记忆的，也就是说信源产生的信息相互并不独立，则需要引入联合熵的概念。以两个相关的随机变量表示信源产生的两个信息来构造一个最简单的模型，以下三个公式成立：

$$H(X,Y) = H(X)+H(Y)-I(X,Y) \quad (4-1)$$
$$I(X,Y) = H(X)-H(X|Y) = H(Y)-H(Y|X) \quad (4-2)$$
$$H(X|Y) = H(X,Y)-H(Y) \quad (4-3)$$

其中，H(X,Y)为联合熵，表示这两个信息整体上的不确定性；I(X,Y)为互信息量，表示两个信息的相关性，不相关的信息互信息量为0；H(X|Y)叫做相对熵，表示在Y已知的情况下X的不确定性。第一个公式说明，对于相关的信息，其各自熵的和会大于描述其整体不确定性的联合熵。第二个公式定义两个信息的互信息量为其中一个的熵减去其相对熵。第三个公式

表示，X在Y已知的情况下的相对熵等于两个信息的联合熵减去Y的熵。

对于有记忆信源，其熵值不再等于其产生的某一个的信息的熵，这种情况下要使用熵率来描述信源的不确定性，这是一个极限值，假设Hn是信源产生的n个信息的联合熵，则熵率就是n趋于无穷大时的Hn/n。

数据压缩就是对信源产生的信息进行编码的一个过程，即使用某个符号表，如0和1来表示要传输的信息。这里涉及到两种情形：无损编码和有损编码。对于无损编码来说，要求解码之后的信息和编码之前的信息完全相同，即编码过程不引入任何失真，在这种情况下，如果使用二进制符号来表示信源产生的某个信息，其平均长度不能小于信息的熵或信源的熵率。有损编码则会在编码过程中引入失真，因此，从根本上讲是一个信息率-失真最优化的问题。

假设编码过程引入的均方失真为D，则存在一个函数R(D)，表示不超过给定失真D的前提下对该信源编码所需要的最小的信息率，即所谓的率失真函数。如果信源的概率分布给定，平均失真D仅由信源编码前后的转移概率——亦即编码方式决定，则率失真函数给出的其实是一个信源编码的极限信息率，也就是说，对于既定信源，总可以找到一种编码方式，能够保证在既定失真的前提下达到率失真函数给出的最小信息率。率失真函数取决于信源的统计特性，一般不存在显式的表达式，但是对于某些特定分布的信源，率失真函数能够以明了的形式给出，比如高斯分布的连续无记忆信源的率失真函数为：$R(D) = ½ \log_2(\sigma^2/D)$ $(0 \le D \le \sigma^2)$。

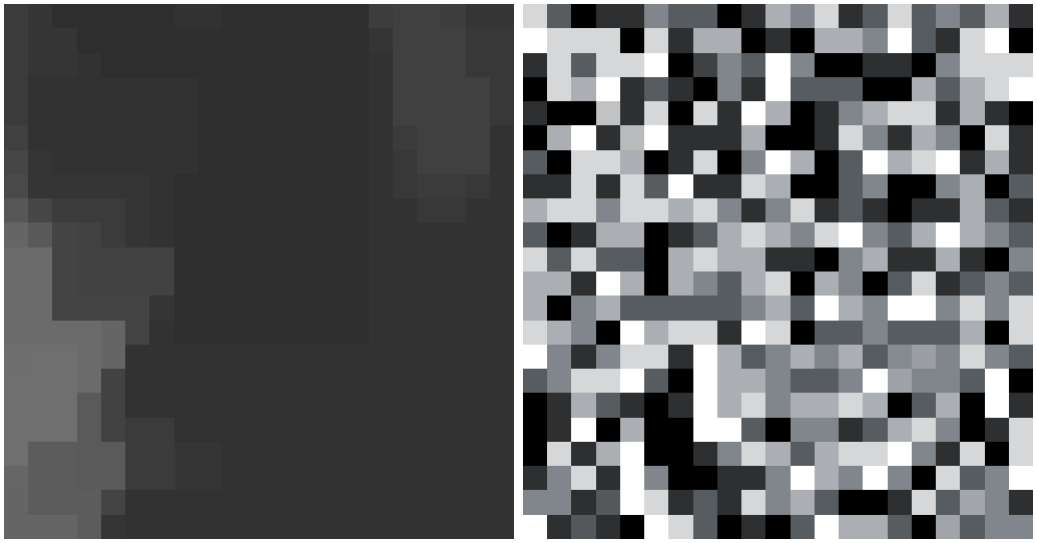
再举一个二值的离散无记忆信源X的例子：概率P分别为0.1、0.2、0.3和0.5的情况下其率失真函数如下图示：

可以发现，当P=0.5，即均匀分布的情况下，信息率失真函数最靠上，也就是说给定最小失真对应的极限信息率越大。当失真为零时，信息率的极限为1，亦即信源的熵。也就是说，有这样一个信源，它以50%的概率在产生符号0和符号1，则无失真地编码该信源产生的一个符号最少也需要1个比特，注意，这是传输成本最高的一种信源。此时，我们便不难理解气象台的喇叭为什么更容易损坏了。

4.1.1 变换编码

从理论上讲，变换的主要目的是去相关。由公式4-1可知，对于相关性很强的两个随机变量，其互信息非常大，导致两个信源的熵的和远大于其联合熵。如果将这两个随机变量看做为一个二维的随机向量，通过一个变换矩阵，可以将{X, Y}变换为 {X', Y'}，在这一过程中，H(X,Y)=H(X',Y')，如果变换矩阵选择适当，令I(X',Y')=0，则H(X')和H(Y')将远小于H(X)和H(Y)，从而对X'和Y'编码需要的比特数将大大减少。能够使X'和Y'相互独立的变换叫做KL变换，这是一种理论上的最佳变换，但由于相应的变换矩阵需要通过X和Y的统计特性来计算，在工程上很难应用起来。

可以从更直观的角度来理解这种说法：以图像数据为例，假设每个像素点的亮度范围为0~255，则在空间域独立地来看某个像素点的话，其统计特性是近似均匀的，也就是说，各阶像素值发生的概率大致都差不多，因此，至少需要8个比特编码一个像素值，而不能够给某些像素值多些的比特，而给某些值少一些。但实际上，对于一个来自某幅图像的像素矩阵采样，如果其中某个像素点为值0的话，其它点为255的概率也不会太大。如下图所示，左图是典型的图像数据（采样自Lenna），而右图则极为罕见，在编码的时候，左图数据应该给以多于右图数据的比特数，要达到这样的目标，在缺少完美的矢量量化方法之前，变换不失为一种很好的工具。



那么，变换域中各点取值的概率分布又如何呢？首先，各点的值域将发生变化，比如DCT变换域各点的取值范围为-2048~2048；其次，各点的概率分布更加独立。 还以上面两幅图像为例，与罕见的图像采样（右图）相比，经典的图像采样（左图）变换域右下方向的值更接近0。因此，对于图像数据来说，变换域右下角出现大量0值是比较常见的，可以使用更少的比特编码这些大概率的情况。换句话说，变换是为了从全局的角度抽取一组数据的特征，并将这些特征分割开来。

DCT是音视频压缩中的一种常用变换，它虽然不能保证使变换后的随机变量相互独立，但仍能大大减少它们的相关性，而且，DCT变换还能产生能量聚集的效果，即对于变换后的随机向量，能量相对集中在索引较小的分量上，更有利于量化。

4.1.2 差分编码

差分编码基于预测来实现，即不编码原始信源数据，而去编码原始信源数据和预测数据的差分数据，主要目的是在不引入失真的前提下减小原信号序列的动态范围。假设信源产生了一个随机序列：

```
S(0), ..., S(n-5), S(n-4), S(n-3), S(n-2), S(n-1), S(n)
```

设S'(n)为S(n)的预测值：

```
S'(n) = f(S(n-1), S(n-2), S(n-3), S(n-4), S(n-5))
```

则预测值序列为：

```
S'(0), ..., S'(n-5), S'(n-4), S'(n-3), S'(n-2), S''(n-1), S(n)
```

令d(n) = S(n)-S'(n)，则差分序列为：

```
d(0), ..., d(n-5), d(n-4), d(n-3), d(n-2), d(n-1), d(n)
```

预测的准则是均方误差最小，及找到一个合适的预测函数f，使$d^2(n)$最小。 与变换编码不同之处是，，即使找到了一个最优的预测函数f，差分编码也不一定会提高编码效率。如果随机序列中个分量不具备相关性甚至是负相关的，差分序列中个分量的均方差会变得很大，甚至大于原始序列中各分量的均方差，这种情况下编码效率会严重下降。

对于一个相关系数接近1的马尔可夫序列，S'(n)=S(n-1)是一个较优的预测函数，这种差分编码便是广泛使用的DPCM技术。

需要注意的是，在实际的编码过程中，由于解码端无法得到原始值，所以预测函数通常使用预测值来代替原始值，即：

```
S'(n) = f(S'(n-1), S'(n-2), S'(n-3), S'(n-4), S'(n-5))
```

对于DPCM，S'(n)=S'(n-1)。

4.1.3 熵编码

通过变换和预测等方法使信源的统计特性得到一定的改善之后（去相关性，降低均方差……），接下来需要进行的是熵编码，也是数据压缩的最后一步，其主要责任是将压缩视频的各种头信息、控制信息以及变换系数转换为二进制的比特流。有两种最基本的熵编码方法：定长编码和变长编码，前者对所有的待编码信息使用相同长度的码字，后者则使用不同长度的码字。假设某个待编码的信息元素A∈{A0, A1, …, An}，如果采用定长编码，需要的比特数为$\log_2(n)$取整，而使用变长编码，其平均码长的极限取决于该信息的熵H，除非是均匀分布，不然所需要的比特数必定小于$\log_2(n)$。因此，如果某个信息元素在概率分布上是极度不均匀的，通常都会采用变长编码的方式，即概率小的值使用长码字，概率大的值使用短码字。

指数哥伦布码、Huffman编码和算术编码都是常用的变长编码方法。

4.1.4 量化

量化[5]是一种多对一的映射，是引入失真的一个过程，也是限失真信源编码技术的基础。无论是对时间采样后的模拟信号进行数字化的过程，还是对数字序列进行有损压缩的过程，都需要完成一个由输入集合到输出集合的映射，这个映射是由量化来实现的。

最简单的量化方法是将单个样本的取值进行量化，因为被量化的变量是一维的，所以这种量化方法叫做标量量化。设n阶标量量化器的输入为连续随机变量x，输出为离散随机变量y，其中：

```
x∈(A0, An)， y∈{Y1, Y2, …, Yn}, A0≤ Y1 ≤ A1 ≤ Y2 ≤ …… An-1 ≤ Yn ≤ An。
```

则y 的取值由下式决定：

```
y = Yi 若Ai-1≤x<Ai
```

当量化阶数n一定时，选择合适的Ai 和Yi 可以使量化器的平均失真最小，这时的量化称为最佳标量量化。若输入变量x满足均匀分布，可以将(A0, An)均匀分割成n个小区间，每个小区间的中点作为量化值。这种量化方法叫做均匀量化，对于均匀分布的输入变量来说，均匀量化是最佳标量量化。当采用均方失真函数时，可以计算出其平均失真为$\Delta^2/12$，其中$\Delta = (An-A0)/n$。

然而，从率失真的角度来考虑，最佳标量量化并不能达到最佳率失真编码的要求，通常需要对量化后的数据进行继续进行处理，如熵编码等。

为了使量化后不再进行后处理而能逼近率失真函数的界，人们开始探讨根据多个连续信源符号联合编码的方法，即矢量量化技术。假设**X** = {X1, X2, …, Xn}是信源的一个n维矢量，它的取值范围是n维空间中的一个区域Rn，一个k级的矢量量化器就是X∈Rn到k个n维量化矢量Y1, Y2, …, Y的映射函数Q(X)。对于任意Yi，i = 1, 2, …, k，指定一个n维的区域Ai，对于所有X∈Ai，有Q(X) = Yi。其中Ai称为Yi的包腔，各量化矢量称为码字，它们的集合称为码书。如果选择的码书和各包腔可以使平均失真最小，这时的矢量量化称为最佳矢量量化。

4.1.5 率失真优化

率失真函数给出了信源编码的信息率极限，而率失真优化则研究如何达到该极限，即在给定信息率上限Rc的前提下，寻找一种编码方法使D最小化：

```
min{D(P)} s.t. R(P) ≤ Rc
```

其中，P表示一个信源编码前后的转移概率，代表某种编码方法。

这是一个典型的有约束的非线性规划问题，可以通过拉格朗日乘子法转化为一个无约束的求极小值的问题：

```
min{D(P)+λ*R(P)}
```

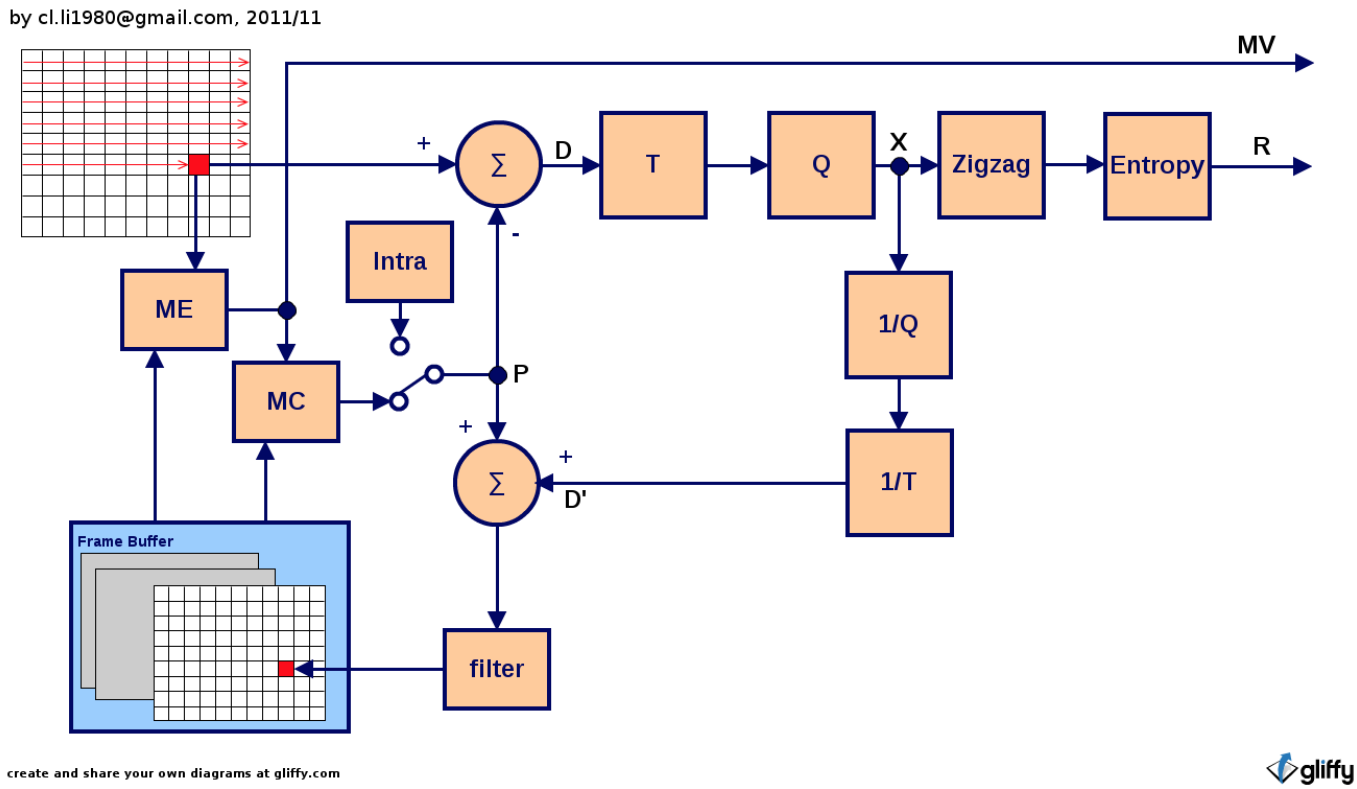
这里的λ与约束条件Rc息息相关，目标速率越大，则λ越小，当λ为零时，表示不限制目标速率，则只剩下min{D(P)}了。

实际的编码过程不是数学推理过程，无需对上面的方程求解，只要确定了λ（这是个关键点），通过穷举搜索即可找到最佳的编码方法。

4.2 图像

4.3 视频

4.3.1 混合编码系统



所谓混合指的是运动预测差分编码和变换编码的混合，其一般编码原理由上图给出，通常是以宏块为单位按照扫描顺序进行编码，当然也不排除基于某种大规模并行运算的编码方法改掉这一方式。整个处理过程中，涉及到的数据包括：

- P: 预测值
- D: 差分值
- D': 本地恢复的差分值
- X: 量化后的残差变换系数
- R: 经过熵编码的残差变换系数

主要的处理模块包括：

4.3.1.1 ME：运动估计

这里的运动估计指的是为待编码宏块中的各个像素点寻找最佳预测值的搜索过程，找到的预测值位于某一帧已经编码并重建的图像中，与被预测的像素点在位置上存在偏差，这些偏差就叫作运动矢量，它们和参考帧的位置共同作为运动估计过程的最终输出物。

4.3.2.2 MC: 运动补偿

运动补偿也叫做运动补偿预测，这一过程根据运动估计给出的运动矢量和参考帧位置生成待编码宏块各像素点的预测值。由待编码图像和经运动补偿的参考图像逐点相减可生成一副差分图像，相对与自然图像，差分图像的动态范围大大减小，从而更有力于后续的压缩

4.3.2.3 T: 变换

通过对待编码像素与预测值的差值进行一个二维变换，有效地去除空间冗余。

4.3.2.4 Q: 量化

变换系数的量化通过引入失真降低比特率。

4.2.3.5 1/T: 反变换

这一过程和下一过程的主要目的是生成本地重建图像。由于解码器端无法获取原始图像，为了防止误差积累，需要在编码器端复制解码过程，使用解码后的重建图像代替原始图像进行预测。

4.2.3.6 1/Q: 反量化

4.2.3.7 Zigzag: Z扫描

Z扫描可以将二维的残差变换系数转换为一维的序列，更有利于其后的熵编码。

4.2.3.8 Entropy: 熵编码

利用数据流内部的统计特性对一维的残差变换系数进行无损压缩。

4.2.3.9 Filter: 环内滤波

环内滤波有两个目的：其一是为了去除因变换产生的块效应；其二是通过改善重建图像，使预测过程更有效。

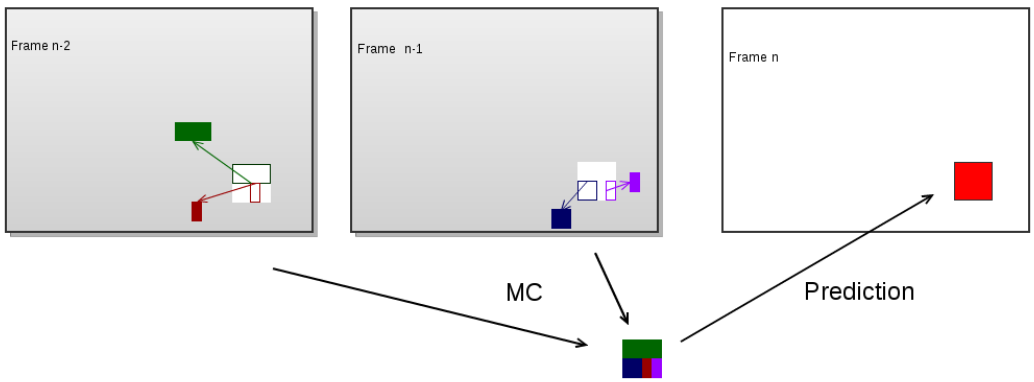
4.2.3.10 Intra: 帧内预测

在某些情况下，无法获得参考帧，或参考帧中的预测值与实际值的差距过大（比如视频序列发生场景切换），则不采用已编码重建图像中的像素值作为预测值，而以当前图像中已编码部分的像素值作为预测值。

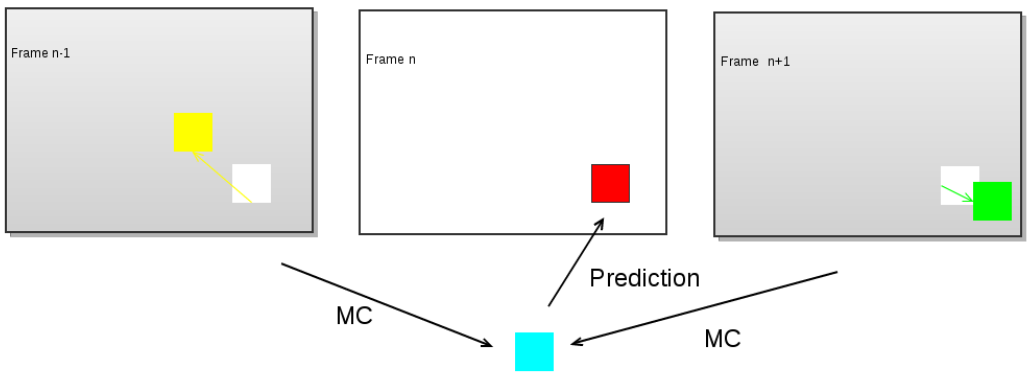
4.2.3.11 模式选择

此外，编码过程还隐含着一个模式选择的模块，体现在上图中就是决定帧内还是帧间的开关，而实际上除了这个的开关，其他诸如运动补偿、帧内预测都要涉及到模式选择，譬如基于多大的块进行运动补偿、使用哪一个参考帧等。

运动补偿预测可以用下图来表示：



红色部分为待编码的宏块，彩色部分为利用运动补偿由本地解码的重构图像生成的预测值，而最终编码的是红色部分和彩色部分的差值。可以看出，为了完成该宏块的预测，需要四个运动矢量和两个参考帧。某些编码技术如双向预测及MPEG2中的Dual-Prime会令预测过程更加复杂，预测值需要由两个经运动补偿的预测值加权平均得到：



4.3.2 MPEG-2视频

MPEG-2视频的官方代号为ISO 13818-2，是ISO针对数字存储和数字电视应用提出的视频压缩标准，ISO同时在13818-1中给出了相应的传输标准，通常，MPEG-2视频的传输都在13818-1规定的框架内进行。

MPEG-2的制定是在H.261的的基础上完成的，主要特征为：

1. 基于序列/图像组/图像/条/宏块/块(Sequence/GOP/Picture/Slice/MB/block)的组织方式；
2. 逐行序列和隔行序列(Progressive/Interlaced Sequence)的支持；
3. 帧图像和场图像(Frame Picture/Field Picture)的支持；
4. 更多的运动补偿方式
5. 半像素精度运动补偿的支持
6. I/P/B三种图像编码方式
7. 8×8 DCT

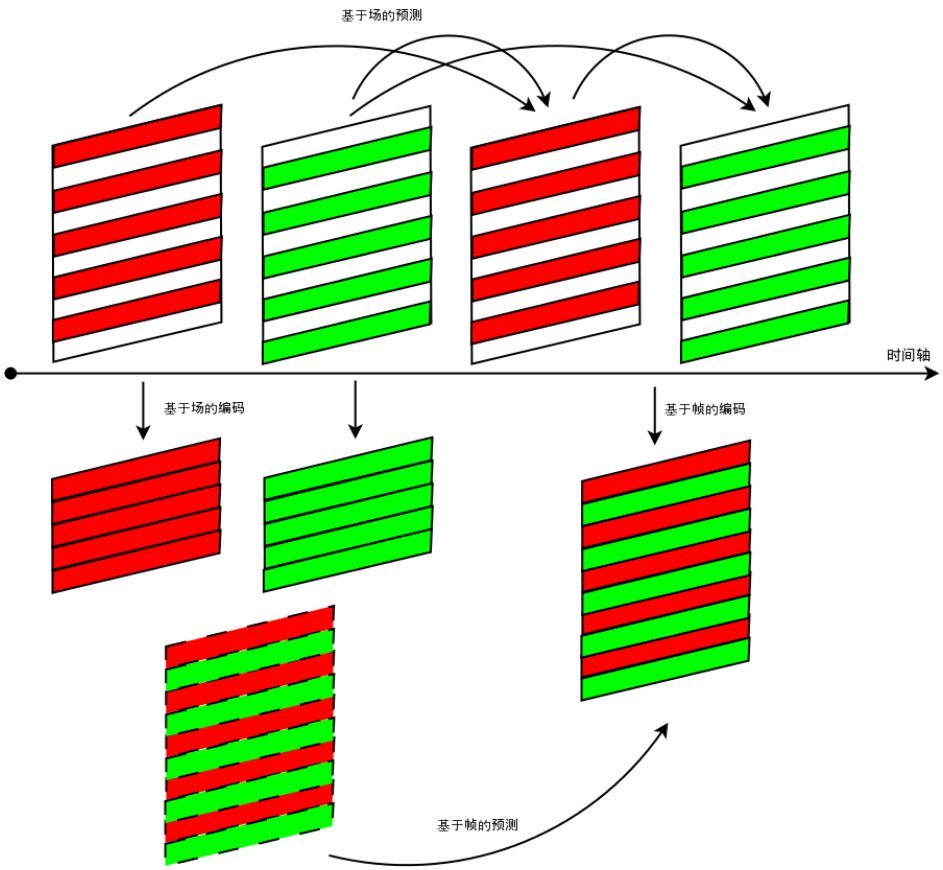
其中包括了一些新的技术点：

对隔行扫描的支持

为了兼容模拟时代不得以引入的隔行扫描技术，MPEG-2增加了对隔行扫描图像序列的支持。MPEG-2的序列头中有一个progressive_sequence标志，progressive_sequence为1表示逐行序列，progressive_sequence为0表示隔行序列。隔行序列就是采用隔行扫描的技术产生的图像序列，其中的每个扫描图像叫做一个场，它们或者扫描自奇数行，或者扫描自偶数行，分别叫作顶场和底场，整个序列中顶场和底场按照采样时间交错排列。

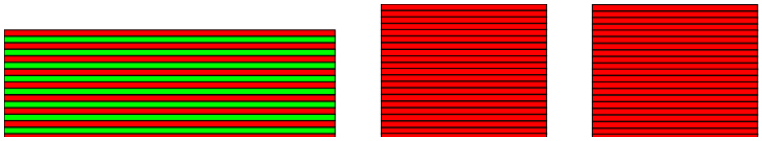
逐行序列的图像编码方式基本上同H.263相同，是以帧为单位的，图像头中的picture_structure字段永远为3；frame_pred_frame_dct字段永远为1。但对于隔行序列，MPEG-2既支持以帧为单位进行编码，也支持以场为单位进行编码。所谓以帧为单位进行编码是指将两个场图像合并为一个帧，解码输出时再将其拆作两个场，图像头中的top_field_first字段用以指示输出场的顺序。如果是以场为单位进行编码，一幅图像就是一个场，图像头中的picture_structure字段可以为1和2，分别表示顶场和底场。事实证明，对于某些近乎静止的视频片段，即使采用隔行扫描，以帧为单位进行编码也比以场为单位效率更高。

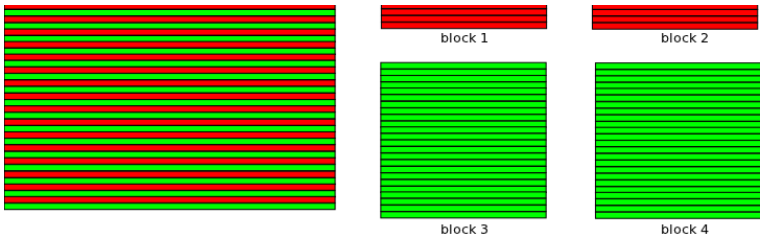
对隔行序列的支持也引入了更复杂的预测方式和DCT方式：



对于以场为单位进行编码的图像，其运动补偿预测都是基于场的，参考图像可以从已完成解码的两个场中选择，既可以是和当前场采样位置相同的场，也可以是和当前场采样位置相反的场。其次，新增加了16×8的运动补偿方式，此时一个宏块会包含两个运动矢量——B帧的话可能出现四个，而每个运动矢量都可以选择与当前场采样位置相同或相反的场作为参考。此外，基于场的预测还支持Dual-Prime的方式，这种方式允许同时使用两个采样位置相反的参考场进行平均预测，其中基于和当前场采样位置相同的参考场的运动矢量需要包含在码流中，而基于和当前场采样位置相反的参考场的运动矢量以在前者基础上的校正值的形式传输，校正的范围至多为1个像素。

对于以帧为单位进行编码的图像，每个宏块既可以采用帧预测方式实现运动补偿，也可以采用场预测方式进行运动补偿，如果采用场预测方式，顶场和底场各需要一个运动矢量（B帧要两个），而且每个运动矢量可以选择使用它们的各自的参考场。当然，以帧为单位进行编码的图像也可以使用Dual-Prime场预测，此时一个宏块只包含一个运动矢量和一个校正值，上下两个场都依据此运动矢量和校正值完成相应的Dual-Prime预测和运动补偿。DCT也类似，如果采用场方式DCT，亮度四个DCT块的组织方式就会变成：上半场左侧、上半场右侧、下半场左侧、下半场右侧。这里，预测方式和DCT方式的选择都是以宏块为单位选择的，分别由宏块头的frame_motion_type，和dct_type字段指定，前提条件是图像头扩展中的frame_pred_frame_dct设为了0。



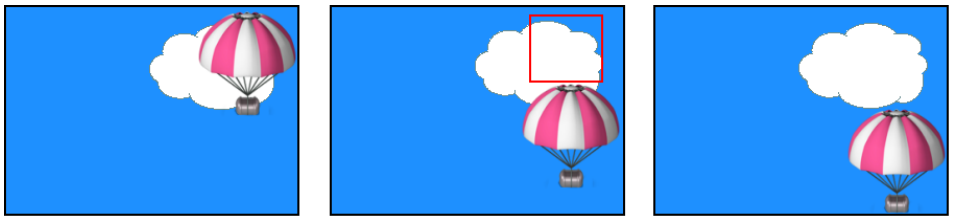


量化系数矩阵及步进控制

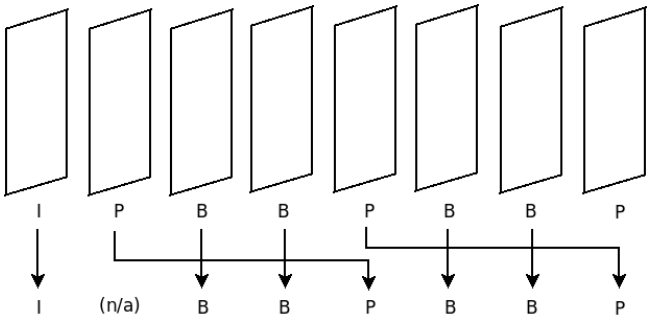
标准规定了缺省的量化矩阵，但也允许编码器发送自定义的量化矩阵，这些数据可以插入到序列头或Quant matrix extension中，YUV420格式的视频序列需要两个8×8的矩阵，分别用于帧内编码和帧间编码，YUV422和YUV444格式则需要四个，因为亮度信号和色差信号使用不同的矩阵。对于帧内的直流系数，量化因子几乎是固定的，只能取1、2、4、8四个值，在图像头扩充中指定，而且不允许步进变化。对于其他系数，允许在量化矩阵的基础上做伸缩，伸缩因子可以在slice层面完成控制，也可以在macroblock层面完成控制。

后向预测和B帧

后向预测允许使用当前编码图像在时间轴上向后的图像作为运动补偿预测的参考图像，这可以有效地解决运动掩盖问题——被运动物体掩盖的部分会在将来重现于画面，而这部分内容很难在过去的图像中找到对应的参考预测区域。于是，MPEG-2中首次提出了既可以使用前向预测也可以使用后向预测的B帧，编码一个B帧需要两幅参考图像，一幅取自过去，作为前向预测的参考图像，另一幅取自将来，作为后向预测的参考图像。此外，B帧还允许同时使用两幅参考图像，最终的预测值由分别来自两幅参考图像的预测值取均值得到，这种预测方式被称为双向预测，双向预测需要两个运动矢量来进行运动补偿，一个用于寻找前向预测值，另一个用于寻找后向预测值。MPEG-2标准规定B帧以宏块为单位选择预测方式，编码器根据待编码宏块的图像内容决定使用前向预测、后向预测还是双向预测。



B帧带来的好处主要有两方面：首先，前向预测和双向预测技术可以明显改善某些特定内容的编码效果；其次，MPEG-2中的B帧是不作为参考图像参与预测编码的，因此，B帧的损坏不会引起错误扩散，在某些特定的情形下——如出现传输和处理延迟——也可以通过丢弃B帧来加快处理过程。当然，引入B帧也要付出一些代价，除了增加运算复杂度之外，解码器端需要多的图像缓冲，为了调整解码图像的顺序使之与显示顺序保持一致，解码器会产生一定的输出延迟。



伸缩编码

MPEG-2是第一个引入伸缩编码的标准。伸缩编码亦可理解为分层编码，其目标是允许客户端根据自己的能力和网络环境选择接收适合自身的视频子流。伸缩编码通过在一个高质量的视频流中嵌入一个或多个较低质量的视频子流来实现，其中最基础的低质量视频子流可以有解码器独立解码，而高质量的子流则需要在低质量子流被解码的基础上实现解码。标准提供了三种实现方法：

- SNR伸缩

在编码过程中，通过将本地解码得到的重构图像与原始图像做差分，然后将差分数据再次进行DCT变换、可以得到一个增强的视频码流。在传输过程中，增强流和基础视频流同时传输，于是在解码器端，单纯解码基础流可以生成较低质量的视频数据，而增强流和基础流合并解码则可以生成较高质量的视频数据。

- 时间伸缩

由对原始输入视频进行帧率的降采样来实现。其中，基础视频流通过对降采样的视频序列进行标准编码得到，而增强流通过对帧率降采样过程中丢掉的那些图像进行增强编码得到，增强编码必须使用基础流的重建图像作为参考图像。

- 空间伸缩

由对原始输入视频进行空间的降采样来实现，其中，基础视频流通过对降采样后的低分辨率视频序列进行标准编码得到，而增强流通过对全分辨率的视频序列进行增强编码得到，增强编码必须使用基础流的重建图像作为参考图像。

可自定义的运动矢量范围

MPEG-2标准可以根据视频序列本身的特征确定运动矢量的范围，从最小的-8~7.5到最大的-2048~2047.5，由picture_coding_extension中的f_code标识。对于不同的运动矢量范围，运动矢量自身的分辨率也不同，范围越大粒度越粗。

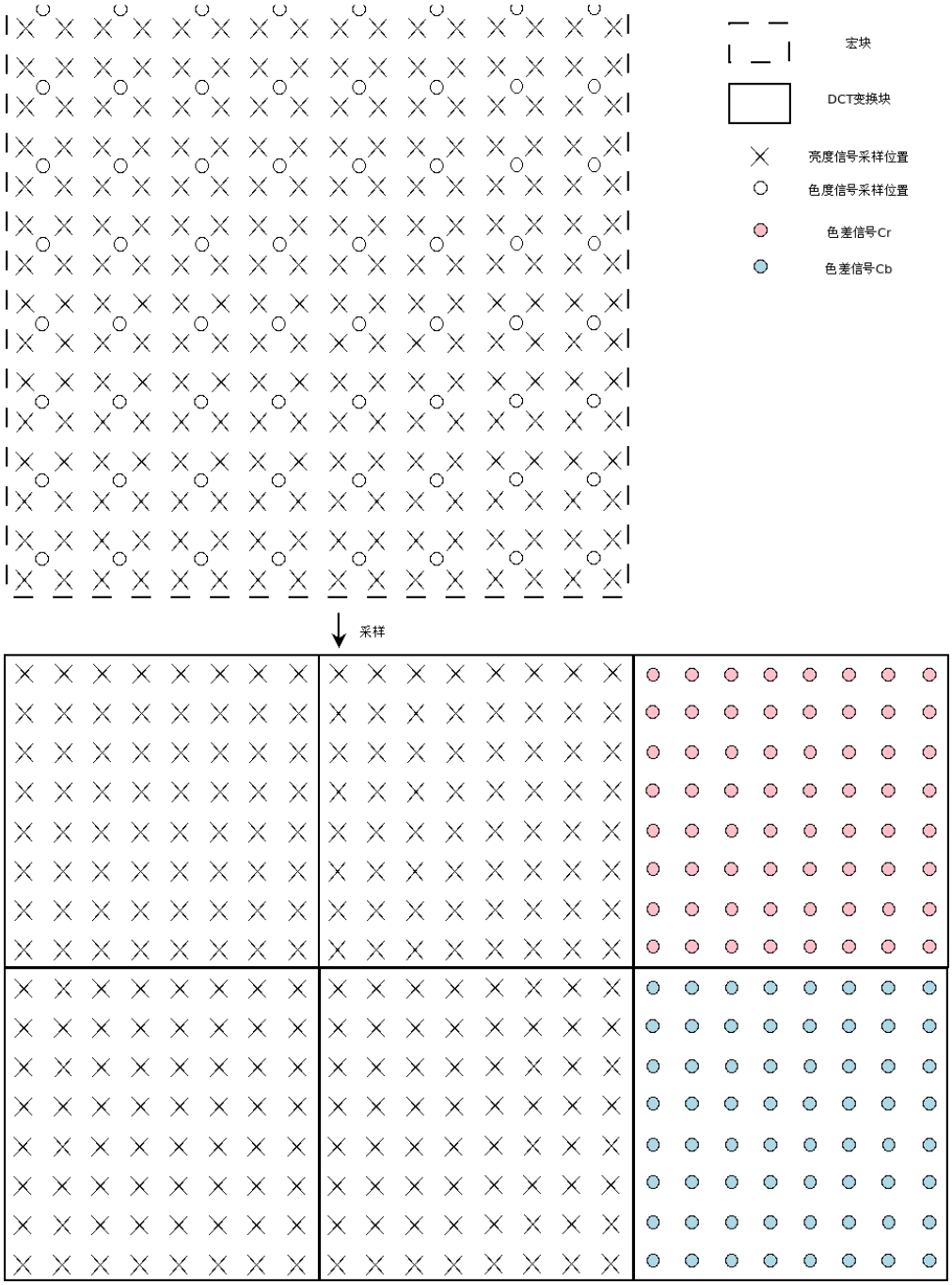
MPEG-2标准首次提出了profile和level的概念，profile一般被译为类或者档，每一种档都会规定一个可使用的编码技术的集合，同时对应一个确定的码流兼容标准，从而在保持标准兼容性的同时增强了灵活度；level通常译为级，来指示对解码器运算资源的要求，涵盖了对码流的比特率、图像的大小、运动矢量的范围等各种规定。MPEG-2定义了五个档和四个级别，它们之间可以自由组合，以表示解码器解码能力的要求。而作为一种通用的概念档和级别也自此被沿用到后续的标准中。

现在来看，MPEG-2算不上是很出色的视频压缩技术（在某些地方对比特的使用近乎奢侈：譬如换行时运动矢量的预测值并不重置等），却是一种在商业应用上非常成功的一种技术，同时也是很昂贵的一种技术，：2002年前每台设备的专利费为4到6美元，后来降至2.5美元，最近又降到2美元，由于其在DVD和数字电视中的广泛应用，非MPEG-LA [http://www.mpegla.com]专利持有者的设备商和内容提供商不得不为使用此技术支付巨额的专利费用。

4.3.3 H.263

H.263是其前身H.261——第一个真正意义上的视频压缩标准——的增强版，由ITU制定，侧重于低码率视频应用环境，如基于老式电话网络的视频会议和可视电话等。通常意义上的H.263指的是1995年定稿的版本，对应的文档按照ITU的惯例以建议的形式发布。这个版本的H.263除了定义编解码的主框架之外，还附带8个附件，其中有4个属于对主框架算法改进的选项。由于H.263在制定之初就具有很强的针对性，即服务于甚低带宽的老式电话网，其性能的优势也主要体现在低比特率的应用场景。直到后来，ITU不断以附件的形式对基础算法进行修订和增强，产生了H.263+、H.263++等诸多版本，才逐步使H.263系列技术的应用范围得到更大的扩展。

H.263最初的版本规定输入图像必须使用YUV色彩空间，即每幅图像由一组亮度信号采样和两组色差信号采样组成，考虑到人眼对颜色信号不甚敏感，色差信号的采样频率只有亮度信号的四分之一，这样从源端即可以减掉一部分数据量。编码过程则以宏块为基本单位，每个宏块包含一个16×16的亮度信号采样和两个8×8的色差信号采样。DCT变换以8×8的块为单位进行，主算法以16×16的块为单位对亮度信号预测进行运动补偿，以8×8的块为单位对色差信号预测进行运动补偿。



H.263的主算法中首次使用了小数精度的运动补偿技术，规定运动矢量的基本单位为半个像素。基于运动补偿的预测算法可以由一个基本公式来描述：

```
Xp(x, y) = Xr(x+dx, y+dy)
```

其中，Xr(i,j)和Xp(i,j)分别表示参考图像在某一特定位置(i,j)的像素值以及当前图像在某一特定位置(i,j)的像素的预测值，(dx,dy)描述了一个二维的运动矢量。显然，要在对应的参考图像中得到Xr(x+dx, y+dy)，必须要求dx和dy为整数，否则，Xr的值必须通过插值才能得到，则以上公式演化为如下形式，其中Dx和Dy为dx和dy的整数部分：

```
Xp(x, y) = Σ{a(i,j)*Xr(x+Dx+i, y+Dy+j)}
```

此时图像中某一个像素的预测值不再来自参考图像中经运动补偿的某一个点，而是由参考图像中多个像素点加权平均得到。显然，这是一种空间域的FIR形式维纳滤波器，半像素只是其中的一个特例：

```
Xp(x, y) = a*Xr(x+Dx, y+Dy) + b*Xr(X+Dx+1, y+Dy) + c*Xr(x+Dx, y+Dy+1) + d*Xr(x+Dx+1, y+Dy+1)
```

参数a、b、c、d的取值由运动矢量的小数部分决定：

(0, 0)	a=1, b=0, c=0, d=0
(0.5, 0)	a=0.5, b=0.5, c=0, d=0
(0, 0.5)	a=0.5, b=0, c=0.5, d=0
(0.5, 0.5)	a=0.25, b=0.25, c=0.25, d=0.25

当然，由于涉及整除后小数取整的问题，建议中规定的系数与上表略有不同。

一些建议制定之初尚处于实验阶段的算法被以附件的形式添加到草案中，其中对性能提高有显著帮助的主要有两项：

4MV

使用4MV模式编码的宏块使用四个运动矢量完成运动补偿，每一个运动矢量对应一个8×8的宏块。关于究竟采用多大的块来进行运动补偿，学术界曾进行过一段时间的探索。理论上讲，块越小，对运动的描述越精确，如果每个象素点使用一个运动矢量的话，几乎可以完美地描述两幅图像间的运动关系，可是，运动信息的编码是需要占用比特数的，因此需要在运动补偿的精确度和运动矢量的编码开销上进行一定的权衡，早期的论文认为最佳的块尺寸是16×16或32×32[6]。但这样的结论是很草率的，其前提是一个认为画面各区域运动情况基本一致的假设，而这一假设在大多数情况下都不成立：一组典型的运动画面往往存在几乎没有变化的背景，满足刚性运动的前景以及变化复杂的细节部分。因此，很难找出最佳的固定尺寸来定义实现运动补偿的块，而根据内容选择块的尺寸倒是一种最佳的解决办法，4MV是基于这一解决办法的最初探索。至于选择4MV还是基础的1MV来编码一个宏块，建议中没有给出任何信息，而在内容识别算法取得突破性进展之前，这一选择只能在后验的基础上做出，即分别使用4MV和1MV对一个宏块编码一次，哪种方式效果好就使用哪种。

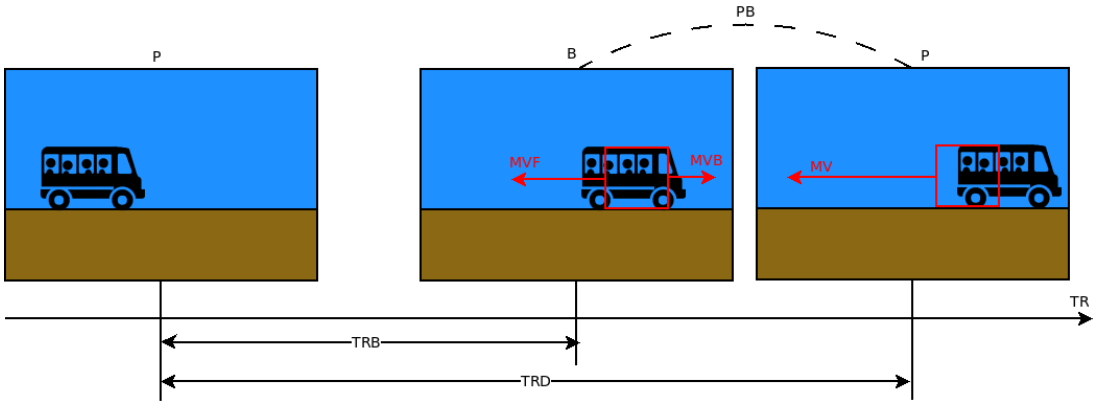
PB帧

PB帧借鉴了MPEG-2中B帧的概念。但B帧的使用会增加额外的传输负担，主要是更多的预测模式和运动矢量信息，这在低码率传输环境中并不划算，所以H.263对这一技术进行了简化。首先，只允许一种预测模式，即双向预测；其次，尽量不传输运动矢量，而是根据时间关系来利用P帧的运动矢量来推算B帧的运动矢量；最后，B帧没有自己的头信息，也就是

说让一个P帧和其后的B帧使用相同的头信息——包括图像头、宏块头等，但每一个宏块内包含12个块数据，分别属于P帧和B帧。最后一点是至关重要的，否则，仅仅使用双向预测得到的收益（这种双向预测还并非真正意义上的双向预测，因为其运动矢量不是最优的）远抵不上头信息的开销。

运动矢量的推算根据以下公式进行，显然，其基础是一个非常简单的匀速刚性运动模型，就现实的运动画面来讲，符合此模型的情况极少，因而单纯从预测的角度讲，PB帧的效果并不一定比P帧更好，因为至少使用全搜索的情况下P帧可以保证预测残差值的绝对值和最小。

```
MVF = (TRB*MV)/TRD
MVB = ((TRD-TRB)*MV)TRD
```



PB帧的使用定义在附件G中，但是后来附件M对附件G做了修订，进而将附件G废弃。修订的PB帧允许使用更多的预测模式，而此时的H.263也不再是一个纯粹的针对甚低码率应用的技术规范了。

4.3.4 MPEG-4

4.3.5 H.264

H.264的官方名称是AVC（先进视频编码），对应的标准文档是ISO 14496-10，因此也称作MPEG4-AVC，是由MPEG和ITU合作制定的视频编码标准。从原理上讲，H.264仍旧秉承传统混合编码架构，但由于在编码细节引入和诸多改进，将许多一直停留在纸上的技术如可变尺寸块的运动补偿、多参考帧预测等付诸实践，使编码效率得到了显著的提高，同时也大大增加了编解码器的复杂度和运算量。总的来说，在图像质量保持相同的条件下，H.264编码需要的数据率可以小于MPEG-4的一半。

H.264主要引入了如下技术点：

4x4小矩阵变换

小的变换矩阵有时可以产生更好的压缩效果，能够在不增加比特率的情况下改善图像质量，而且会减少方块效应；此外，小的变换矩阵也可以降低运算量：如果不使用快速算法，8×8的二维变换总共需要1024次乘法（8x64x2），但4个4×4的二维变换值需要512次（4x16x2x4），16个2×2的变换256次。

基于此，H.264引入了4×4小矩阵变换的编码选项，并对变换过程和量化过程进行了深度改造，尽可能摒弃其中的浮点数运算和乘除法操作，进一步降低了运算量，以至变换过程不再成为编解码运算的瓶颈。

假设H.264 4×4变换过程为**Y=FXF'**，其变换矩阵**F**简化为：

```
1  1  1  1
2  1 -1 -2
1 -1 -1  1
1 -2  2 -1
```

则整个变换的函数也可以简化至：

```
void dct4x4(short dct[16])
{
    int i;
    int s03, s12, d03, d12;
    short tmp[16];

    for (i = 0; i < 4; i++)
    {
        s03 = dct[i*4+0] + dct[i*4+3];
        s12 = dct[i*4+1] + dct[i*4+2];
        d03 = dct[i*4+0] - dct[i*4+3];
        d12 = dct[i*4+1] 1 dct[i*4+2];

        tmp[0*4+i] = s03 + s12;
        tmp[1*4+i] = (d03<<1) + d12;
        tmp[2*4+i] = s03 - s12;
        tmp[3*4+i] = d03 - (d12<<1);
    }

    for (i = 0; i < 4; i++)
    {
        s03 = tmp[i*4+0] + tmp[i*4+3];
        s12 = tmp[i*4+1] + tmp[i*4+2];
        d03 = tmp[i*4+0] - tmp[i*4+3];
        d12 = tmp[i*4+1] 1 tmp[i*4+2];

        dct[i*4+0] = s03 + s12;
        dct[i*4+1] = (d03<<1) + d12;
        dct[i*4+2] = s03 - s12;
        dct[i*4+3] = d03 - (d12<<1);
    }
}
```

显然，矩阵**F**不具备正交性，因此以上变换并非正交变换，它只能作为正交变换的一部分，整个正交变换其实为：**Y=[F●R]X[F'●R']=FXF'●[R●R']**，其中，R为：

```
0.5    0.5    0.5    0.5
0.3162 0.3162 0.3162 0.3162
0.5    0.5    0.5    0.5
0.3162 0.3162 0.3162 0.3162
```

R●R'为：

```
0.25   0.1581 0.25   0.1581
0.1581 0.1    0.1581 0.1
0.25   0.1581 0.25   0.1581
0.1581 0.1    0.1581 0.1
```

则真正的变换矩阵**F●R**为：

```
0.5     0.5     0.5     0.5
0.6325  0.3162  -0.3162 -0.6325
0.5     -0.5    -0.5    0.5
0.3162  -0.6325 0.6325  -0.3162
```

可以发现，这个矩阵已经接近正交——其实如果用sqr(10)/10代替0.3162、sqr(10)/5代替0.6325的话，该矩阵就是正交矩阵，再来看正宗的DCT变换矩阵：

```
0.5     0.5     0.5     0.5
0.6532  0.2706  -0.2706 -0.6532
0.5     -0.5    -0.5    0.5
0.2706  -0.6532 0.6532  -0.2706
```

显然，H.264使用的4×4变换矩阵实际上不过是DCT矩阵的一种近似——使用sqr(10)/10代替sqr(1/2)*cos(π/8)、使用sqr(10)/5代替sqr(1/2)*cos(3π/8)，然而，这种近似可以将变换过程分离成为一个子变换过程和一个系数抽取的过程，其中前者是一个整形化且剥离乘法的简单运算；后者则可以合并到其后的量化过程中。至此，小矩阵变换的运算过程实现了一个完美的简化。

逆变换**X=[I●S]X[I'●S']**的分解略有不同，其中**I**为：

```
1    1    1   0.5
1    0.5 -1  -1
1   -0.5 -1   1
1   -1    1  -0.5
```

S为：

```
0.5    0.6325 0.5    0.6325
0.5    0.6325 0.5    0.6325
0.5    0.6325 0.5    0.6325
0.5    0.6325 0.5    0.6325
```

S●S'为：

```
0.25   0.3162 0.25   0.3162
0.3162 0.4    0.3162 0.4
0.25   0.3162 0.25   0.3162
0.3162 0.4    0.3162 0.4
```

而逆变换矩阵**I●S**为：

```
0.5    0.6325  0.5  0.3162
0.5    0.3162 -0.5 -0.6325
0.5   -0.3162 -0.5  0.6325
0.5   -0.6325  0.5 -0.3162
```

不出所料，I●S是F●R的转置，这符合正交变换逆变换的要求，但需要注意的是，逆变换的伸缩矩阵**S●S'**却正变换中的**R●R'**存在差异，这导致H.264的量化和反量化过程相对之前的压缩标准也有所不同。

在H.264中，对变换系数矩阵进行量化的步长是统一的，但由于伸缩矩阵的存在，每个量化步长对应着一个伸缩过的量化矩阵。比如量化步长为1时的量化矩阵为：

```
1/0.25   1/0.1581 1/0.25   1/0.1581
1/0.1581 1/0.1    1/0.1581 1/0.1
1/0.25   1/0.1581 1/0.25   1/0.1581
1/0.1581 1/0.1    1/0.1581 1/0.1
```

变换系数需要除以量化矩阵中对应的量化系数以完成量化过程。为了剔出除法运算，H.264定义了整型量化矩阵，变换系数需要先乘以该整型矩阵对应的元素，最后统一除以32768，则以上量化矩阵对应的整型量化矩阵为：

```
8192 5243 8192 5243
5243 3355 5243 3355
8192 5243 8192 5243
5243 3355 5243 3355
```

同样，对于步长为1的反量化操作，反变换矩阵需要乘以伸缩矩阵，而由于伸缩矩阵中存在浮点数，则需要将该伸缩矩阵扩大64倍，反量化之后再将结果统一除以64。扩大后的反量化矩阵为：

```
16 20 16 20
20 25 20 25
16 20 16 20
20 25 20 25
```

这个反量化矩阵定义于H.264的标准中，对应量化因子为4。标准中共定义了6个基本的反量化矩阵，对应6个量化因子，而在这6个反量化矩阵的基础上通过为系数乘以2的倍数又可以形成更多的扩展反量化矩阵。标准中对量化矩阵没有做定义，但根据反量化矩阵不难计算出对应的量化矩阵。

帧内预测

H.264中使用的帧内预测方法有两个特征：其一是基于空间域中的预测，其二是基于方块的预测。被预测的方块区域有4×4、8×8和16×16三种，预测值源于与方块区域相邻的上方和左方的边缘像素，预测的方式有若干种（参见示意图）。

帧内预测的使用使宏块的解码及宏块内方块的解码具有相互的依赖性，要求宏块解码和宏块内方块的解码遵循严格的自左至右、自上而下的扫描顺序，这也成为解码过程并行化的一大阻碍。

在解码过程中，反变换模块输出的残差值加上预测值即可生成图像数据的重建值，决定预测值的预测模式可以从码流中获取：对于16×16预测，预测模式信息包含于mb_type中，而对于8×8和4×4的预测，预测模式要专门给出，为了节省比特，预测模式本身也是预测编码的，即如果两个相邻块的预测模式相等的话，置一个标志为1而无需给出后一个块的预测模式。

多参考帧

早期的混合型视频压缩方案在编码一个P帧时仅使用该帧在时间轴上向前最近的一个I帧或P帧作为预测帧，编码一个B帧则分别使用时间轴上往前和往后最近的一个I帧或P帧作为前向预测帧和后向预测帧。而多参考帧技术则突破了这一限制，扩大了参考帧的选择范围：P帧可以使用当前帧前面一定范围内任意一个I帧或B帧作为参考帧，而B帧可以从当前帧之前和之后一定范围内的I帧或P帧中分别选择两帧作为前向预测帧和后向预测帧。最早使用这一技术的是H.263+，但参考帧的选择依旧停留在图像级别，这在一定程度上有助于抗误，但就改善压缩性能方面意义不大，H.264则将选择参考帧的层次降低到预测块级别，从而扩大了各个块进行帧间预测的范围，通过采用合适的率失真优化技术能够使压缩效率得到一定程度的提高。

运动预测单元的多样化

早期的视频压缩技术统一使用16×16的矩形作为运动预测的基本单元，这是一个折衷之后的经验值，因为预测单元越大，预测效果越差，但预测单元太小，承载运动信息需要的比特数又太多。在H.263+的Annex F中，提供了8×8运动预测的选项，使用8×8预测模式的宏块需要4个运动矢量。

H.264提供了更多的选择，包括16×8，8×16模式，以及8×8模式，对于采用8×8模式的子块，还有四种子模式可供选择，它们分别以8×8、4×8、8×4、4×4矩形作为运动预测的基本单元。

四分之一像素精度的运动补偿

运动补偿像素精度的提高是通过对参考图像进行内插实现的，非整数点位置的像素值可以由某种滤波器生成，而从理论上讲整个过程可以被认为是某种线性预测滤波，譬如对于四分之一像素精度来说，生成预测值的是一个四抽头线性滤波器。

基于宏块的帧场自适应（MBAFF）

为了保持向前兼容，H.264没有放弃隔行扫描的支持，一个slice可以是场图像，也可以是帧图像，对于帧图像，H.264允许在slice之内自有切换帧场编码模式，也就是帧场自适应技术。在使用帧场自适应的情况下，一个slice被划分为若干宏块对，每个宏块对包括上下相邻的两个宏块，这两个宏块可以是帧编码，也可以是场编码。

基于上下文的熵编码

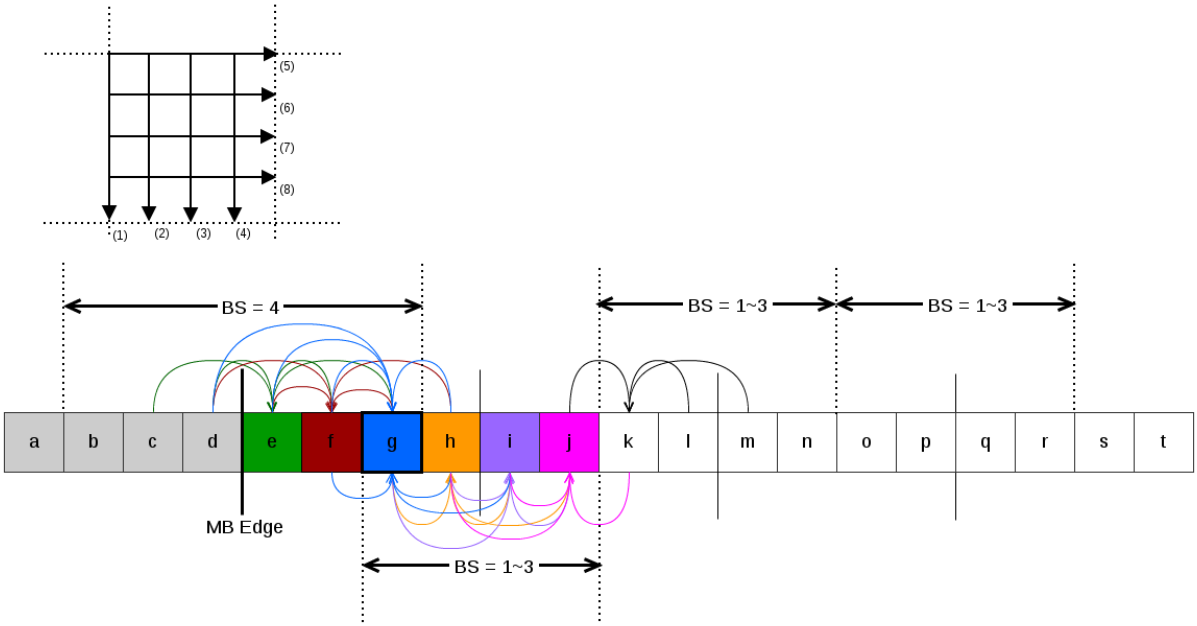
环内滤波

H.264要求对重建图像进行滤波，运动补偿及预测必须使用滤波后的图像作为参考图像，由于这一过程是嵌入在编码循环之中的，因此称为环内滤波，以区别于简单的后处理平滑滤波。引入环内滤波有两个初衷：

1. 消除方块变换引起的块效应，从而改善输出图像的主观质量。
2. 通过使用平滑后的图像作为运动补偿和预测的参考图像，来提升预测编码的率失真性能。

需要注意的是，为了防止滤波过程消除图像中本来就有的边缘信息，必须设定合适的滤波阈值，只有块边界两侧像素点的差值小于设定的阈值时才进行滤波，而这个阈值又往往与量化级相关，量化级越大，阈值也就越大。

滤波过程以宏块为单位按照扫描顺序进行，如果使用了8×8变换，需要对两个垂直边缘和两个水平边缘进行滤波，如果使用了4×4变换，需要对四个垂直边缘和四个水平边缘进行滤波，滤波的顺序为先垂直、后水平。滤波强度由小到大分为四级，一到三级使用四个抽头，最多修改四个像素；第四级使用五个抽头，最多修改六个像素。强度的选择取决于边沿两侧数据的编码方式，譬如对于宏块边缘是边缘两侧有帧内编码的情况滤波强度为最强的四级。



由于影响滤波的因素很多，每修改一个像素点都需要做复杂的判断，这导致环内滤波成为H.264压缩技术中运算量最大的过程之一。而为了节省内存空间，要求环内滤波可以就地执行，也即滤波的结果同时也会作为滤波的输入，因此，环内滤波的执行具有强烈的顺序性，首先上面一行的宏块滤波结束后才能开始下面一行宏块的滤波，一行宏块的滤波必须自左至右执行；其次宏块内各边缘的滤波必须按照先垂直后水平、自左至右、自上而下执行，这样为大规模的并行处理造成了一定困难。从上图可以看出，e，f，g的滤波结果依赖于另一个宏块，h和i的滤波依赖于g的滤波结果，则也依赖于另一个宏块，其中g的最终结果需要经过两次滤波才能得到。

与以往的MPEG标准类似，H.264也定义了不同的档和级别，对编解码器需要支持的编解码技术和运算能力作出了规定。

在ISO标准的2011年修订版中，定义了多达11个档及17个级别。其中，最常见的档有基础档（base profile），主档（main profile）和高级档（high profile），级别则从1.0一直到5.2。在基础档中，除了不能使用B slice和基于上下文的算术编码、不支持场之外，编解码器可以使用上文提到的主要先进技术，如帧内预测、4×4小变换、多参考帧、变尺寸的运动补偿、环内滤波等，并且支持slice组；主档则增加了对B slice、场、以及基于上下文的算术编码技术的支持，高级档还支持选择使用8×8 DCT变换、以单幅图像为单位修正伸缩系数等特性。

4.3.6 Theora和VP3

4.3.6 VP8 & VP9

4.3.7 RealVideo

真正意义上的RealVideo技术应当始自RV30（RealVideo 8），包括之后发布的RV40（RealVideo 9/10）。可以说，2003年H.264草案发布之前，RealVideo的压缩技术一直处于领先地位，RV30和RV40的关键算法均参考自尚处于繁琐的标准化过程中的H.26L，但由于没有及时占领市场(标准化组织在标准制定过程中也一直担心这一点[7])，等到H.264正式发布，RealVideo基本踏上了死亡之路。在数字媒体产业的发展过程中，私有技术和标准之争一直没有停息，相对于标准来说，私有技术的成熟周期短，对市场的反应速度更加快捷，技术授权方面也更加简单，标准则具备更强的兼容性，有利于产业链中各方厂商的协作，然而，由于标准往往是众多企业利益妥协的结果，其中的专利权属问题异常复杂，且从技术角度讲，很容易变得臃肿不堪，增加实现难度。

RealVideo 9主要采用了以下类H.264的技术：

1. 4×4矩阵整数变换
2. 1/4像素运动预测
3. 帧内预测
4. 基于16×8、8×16以及8×8块尺寸的运动预测
5. 双向预测
6. 环内平滑滤波

其没有采用的技术包括：

1. 多参考帧技术
2. 小于8×8块尺寸的运动预测技术
3. 小于16×16块尺寸的双向预测技术
4. 算术编码

这种折衷在某种程度上减轻了编解码器的运算负担，在H.264标准制定的初期是非常有意义的——因为届时尚没有足够强大的硬件来支持JVT庞大的运算模型，然而随着时间的推移，硬件能力很快达到足以应付更复杂的运算需求的程度，此时RealVideo的优势基本丧失。不过，RealVideo技术还有一个亮点，就是环内滤波。RealVideo的环内滤波较H.264更为复杂，因此可以提供更好的主观质量，但这一优势更多体现在低比特率或低分辨率的场景，而随着网络带宽的增加，高比特虑、高解晰度的视频内容成为主流，这个优势也难以具备足够的竞争力。

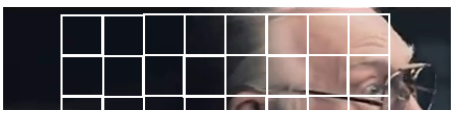
在之后的几年中，Real一直致力于研究下一代视频编码技术（NGV），但其着眼点与JVT的HEVC并不同，NGV强调通过编码与后处理的结合改善视频的主观质量，而HEVC充其量只能作为H.264的一个增强版本。只是，由于计算复杂度远远超出了目前硬件的能力，NGV一直无法进入商用阶段，最终以被转让给Intel而告终。

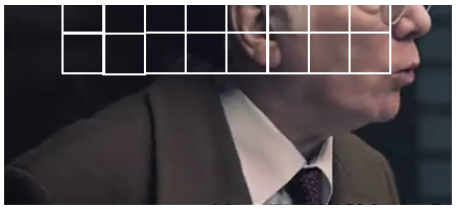
4.3.8 HEVC

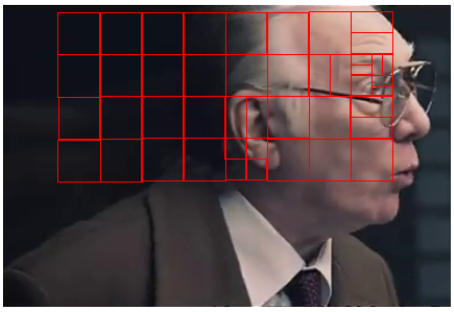
2005年，MPEG组织开始着手探索进一步提高视频压缩效率的可行性，最终，多名专家达成一致，认为通过采用已有的先进技术完全有可能令视频压缩效率继续上升一个台阶。随后，一个负责开发下一代视频压缩标准的联合组织JCT-VC于2010年正式成立，2013年初，冠名为HEVC的新标准发布第一版草案，按照先前的习惯，这个标准也被称为H.265。HEVC的目标是在保持同等质量的前提下，将视频数据的比特率降低为H.264的二分之一，JCT-VC声称这一目标已经达到。

HEVC最大的亮点是利用四叉树结构来组织编码单位，从而使基于变尺寸块的运动补偿预测算法在性能上几近达到其理论上的极致。

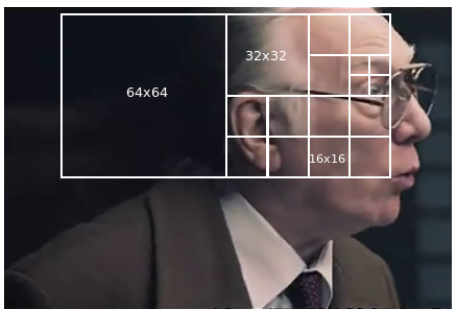
传统以宏块为基本编码组织单位可以达到的运动补偿精度（以AVC为例）：

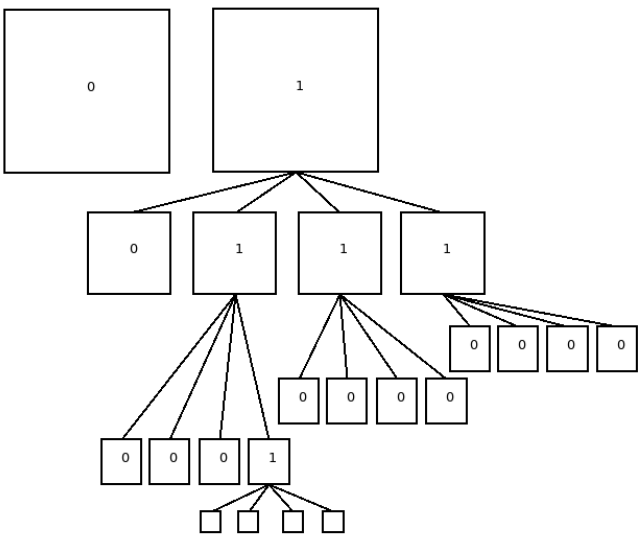


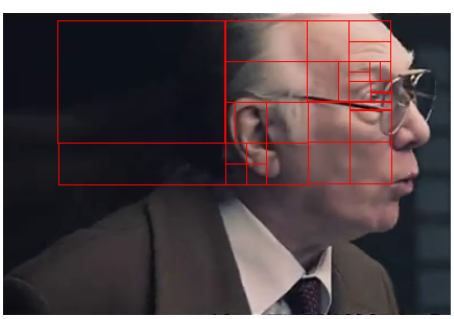


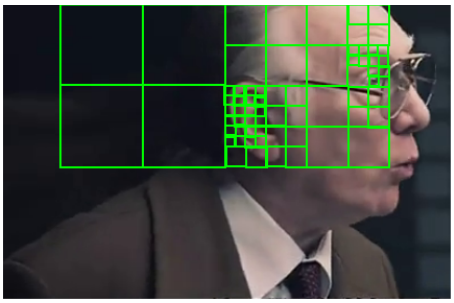


以最大可达64×64的四叉树结构为基本编码组织单位可以达到的运动补偿精度和变换精度：









编码树单元（CTU）和编码树块（CTB）

CTU即上文提到的基本编码组织单位，典型的场景中由一个亮度数据和两个色差数据的四叉树组成，即CTB。亮度CTB的大小可以在编码前预先设定为16、32或64，CTB越大，编码性能越佳。

编码块（CB）和编码单元（CU）

CB是CTB中的四叉树的叶子节点，亮度CB和色差CB共同构成CU。亮度CB的最小尺寸可以在编码前预先设定，不能小于8.

预测单元（PU）和预测块（PB）

PB是运动预测时使用的块，其划分由CB给出，一个CB可以作为一个PB，也可以划分为两个或四个PB，第三种情况仅在CB达到允许的最小尺寸是出现。

变换单元（TU）和变换块（TB）

TB在CB的基础上树状分裂形成，允许的分裂深度可以在编码前预先设定。HEVC定义了4×4、8×8、16×16以及32×32的整数变换矩阵，满足不同TB的变换需求，允许使用的最大TB和最小TB也可以在编码前预先设定。

4.4 语音

4.5 声音

基于其起源于振动的事实，声音信号的特性更易于捉摸，因此声音处理的算法也相对更为成熟。

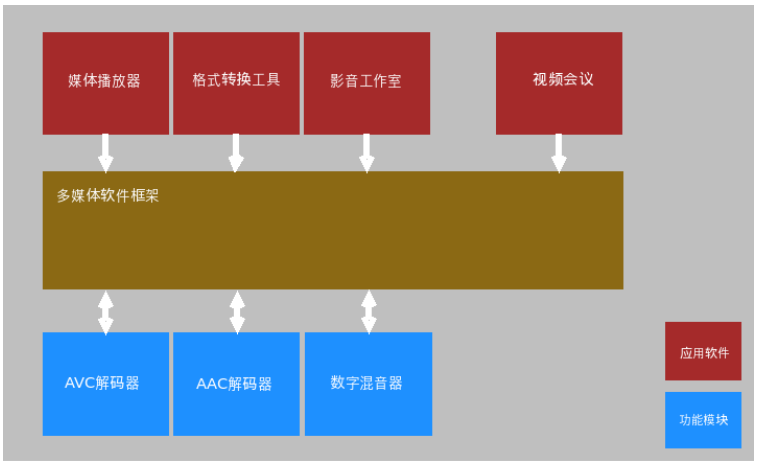
五、质量评估与改善

六、数字版权保护

七、软件构架及解决方案

一个产品级的多媒体应用软件——如媒体播放器、影视剪辑工作室、格式转换工具以及视频会议等——背后必然隐藏着一个高性能且具备极强扩展性软件架构。这一架构需要针对各种不同用例提供便捷的应用编程接口，同时负责组织协调各个功能模块的工作，这些功能模块往往需要实现相当复杂的算法如媒体的编解码和后处理等，因此自耦合性强、运算强度大，通常以库或者插件的形式接入到架构中，由于通信协议和压缩算法演进迅速，架构必须保证功能的可扩展性，以适应由新技术演进引起的应用软件升级。

一个典型的多媒体软件架构常常是这样子的：



一个被称为框架的软件中间层负责为各种不同的应用提供统一的编程接口，从而降低应用软件开发的难度，这个框架还负责以统一的接口接入不同形式的媒体处理功能模块，并规范一个统一的运作策略使这些功能模块能够有效地协同工作。

7.1 DirectShow

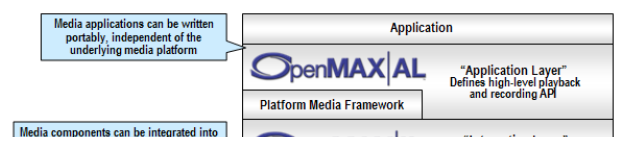
7.2 QuickTime SDK

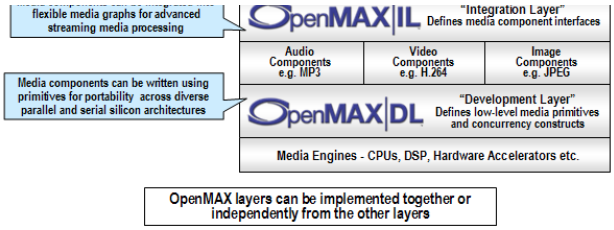
7.3 Helix DNA

7.4 OpenMAX

OpenMAX不是一个软件架构或实现，它是Khronos [http://www.khronos.org]组织制定的一组针对多媒体软件系统的API标准。制定API标准在软件产业界是比较常见的行为，其目的一般是为了增强软件模块的可移植性，但由于整个过程糅合了软件商、芯片公司、OEM厂商等各方的利益，常常是无疾而终的下场，然而某些标准也会得到广泛的应用，比如同样是Khronos制定的OpenGL。OpenMAX的普及程度显然远不及OpenGL，但由于Android系统中采用了OpenMAX IL的接口去调用音视频解码器，各芯片厂商均会提供OpenMAX IL接口的支持，只是接口的行为仍旧会因芯片的不同而不同，似乎也没有完全达到当初制定该标准的初衷。

下图来自Khronos官网，描绘了OpenMAX在制定者心目中的愿景：





但目前应用比较多的只有OpenMAX IL接口，主要在Android中，而且是以独立组件的形式被使用。

7.4.1 OpenMAX IL

OpenMAX IL接口实现了对媒体组件的封装，编解码器、打包模块、解包模块、后处理等等都属于此类组件。定义的API分为核心和组件两部分，核心API负责系统的初始化、组件的管理、组件之间的通信等，组件API负责各具体组件的配置、运行等。在Khronos规范中，调用者、核心和组件的关系如下：

核心API主要包括：

```
  OMX_API OMX_ERRORTYPE OMX_APIENTRY OMX_Init(void);
  OMX_API OMX_ERRORTYPE OMX_APIENTRY OMX_Deinit(void);

  OMX_API OMX_ERRORTYPE OMX_APIENTRY OMX_ComponentNameEnum(
    OMX_OUT OMX_STRING cComponentName,
    OMX_IN  OMX_U32 nNameLength,
    OMX_IN  OMX_U32 nIndex);
  OMX_API OMX_ERRORTYPE OMX_GetComponentsOfRole(OMX_IN OMX_STRING role, OMX_INOUT OMX_U32 *pNumComps, OMX_INOUT OMX_U8  **compNames);
  OMX_API OMX_ERRORTYPE OMX_GetRolesOfComponent(OMX_IN OMX_STRING compName, OMX_INOUT OMX_U32 *pNumRoles, OMX_OUT OMX_U8 **roles);

  OMX_API OMX_ERRORTYPE OMX_APIENTRY OMX_GetHandle(
    OMX_OUT OMX_HANDLETYPE* pHandle,
    OMX_IN  OMX_STRING cComponentName,
    OMX_IN  OMX_PTR pAppData,
    OMX_IN  OMX_CALLBACKTYPE* pCallBacks);
  OMX_API OMX_ERRORTYPE OMX_APIENTRY OMX_FreeHandle(
    OMX_IN  OMX_HANDLETYPE hComponent);

  OMX_API OMX_ERRORTYPE OMX_APIENTRY OMX_SetupTunnel(
    OMX_IN  OMX_HANDLETYPE hOutput,
    OMX_IN  OMX_U32 nPortOutput,
    OMX_IN  OMX_HANDLETYPE hInput,
    OMX_IN  OMX_U32 nPortInput);
```

其中，OMX_Init()和OMX_Deinit()分别用于OpenMAX系统的初始化和退出。

OMX_GetHandle和OMX_FreeHandle分别用于OpenMAX组件的装载和卸载。OpenMAX组件在系统中以名字为标识，如："OMX.Nvidia.h264.decode"，其中"Nvidia"表示芯片厂家，"h264.decode"表示Role。利用OMX_ComponentNameEnum遍历可以获取系统中所有可用的组件的名称：

```
    OMX_ERRORTYPE Error = OMX_ErrorNone;
    OMX_STRING CompName = (OMX_STRING) malloc(OMX_MAX_STRINGNAME_SIZE);
    int i = 0;
    do
    {
        Error = OMX_ComponentNameEnum(CompName, OMX_MAX_STRINGNAME_SIZE, i);
        if (Error == OMX_ErrorNone)
        {
            printf("%s\n", CompName);
            i++;
        }
        else
            break;
    } while (1);
```

此外，通过OMX_GetComponentsOfRole和OMX_GetRolesOfComponent两个函数还可以枚举给定Role的所有组件的名称以及给定名城的某个组件的所有的Role（有时一个组件不仅仅扮演一个Role）。

返回的OMX_HANDLETYPE标识装载的组件，以此可调用组件API。实际上OMX_HANDLETYPE是一个OMX_COMPONENTTYPE结构，内部包含了实现组建接口的一系列函数指针。在装载组件时，需要一个OMX_CALLBACKTYPE结构作为输入参数，用于响应该组件的所有事件，包括：普通事件、EmptyBufferDone事件和FillBufferDone事件。

```
typedef struct OMX_CALLBACKTYPE
{
   OMX_ERRORTYPE (*EventHandler)(
        OMX_IN OMX_HANDLETYPE hComponent,
        OMX_IN OMX_PTR pAppData,
        OMX_IN OMX_EVENTTYPE eEvent,
        OMX_IN OMX_U32 nData1,
        OMX_IN OMX_U32 nData2,
        OMX_IN OMX_PTR pEventData);
    OMX_ERRORTYPE (*EmptyBufferDone)(OMX_IN OMX_HANDLETYPE hComponent, OMX_IN OMX_PTR pAppData, OMX_IN OMX_BUFFERHEADERTYPE* pBuffer);
    OMX_ERRORTYPE (*FillBufferDone)(OMX_OUT OMX_HANDLETYPE hComponent, OMX_OUT OMX_PTR pAppData, OMX_OUT OMX_BUFFERHEADERTYPE* pBuffer);
} OMX_CALLBACKTYPE;
```

组件API包括：

```
OMX_ERRORTYPE (*GetComponentVersion)(
            OMX_IN  OMX_HANDLETYPE hComponent,
            OMX_OUT OMX_STRING pComponentName,
            OMX_OUT OMX_VERSIONTYPE* pComponentVersion,
            OMX_OUT OMX_VERSIONTYPE* pSpecVersion,
            OMX_OUT OMX_UUIDTYPE* pComponentUUID);

OMX_ERRORTYPE (*SendCommand)(
            OMX_IN  OMX_HANDLETYPE hComponent,
            OMX_IN  OMX_COMMANDTYPE Cmd,
            OMX_IN  OMX_U32 nParam1,
            OMX_IN  OMX_PTR pCmdData);

OMX_ERRORTYPE (*GetParameter)(OMX_IN OMX_HANDLETYPE hComponent, OMX_IN OMX_INDEXTYPE nIndex, OMX_INOUT OMX_PTR pComponentParameterStruct);
OMX_ERRORTYPE (*SetParameter)(OMX_IN OMX_HANDLETYPE hComponent, OMX_IN OMX_INDEXTYPE nIndex, OMX_IN OMX_PTR pComponentParameterStruct);
OMX_ERRORTYPE (*GetConfig)(OMX_IN OMX_HANDLETYPE hComponent, OMX_IN OMX_INDEXTYPE nIndex, OMX_INOUT OMX_PTR pComponentConfigStruct);
OMX_ERRORTYPE (*SetConfig)(OMX_IN OMX_HANDLETYPE hComponent, OMX_IN OMX_INDEXTYPE nIndex, OMX_IN OMX_PTR pComponentConfigStruct);
OMX_ERRORTYPE (*GetExtensionIndex)(OMX_IN OMX_HANDLETYPE hComponent, OMX_IN OMX_STRING cParameterName, OMX_OUT OMX_INDEXTYPE* pIndexType);

OMX_ERRORTYPE (*GetState)( OMX_IN  OMX_HANDLETYPE hComponent, OMX_OUT OMX_STATETYPE* pState);
OMX_ERRORTYPE (*UseBuffer)(
            OMX_IN OMX_HANDLETYPE hComponent,
            OMX_INOUT OMX_BUFFERHEADERTYPE** ppBufferHdr,
            OMX_IN OMX_U32 nPortIndex,
            OMX_IN OMX_PTR pAppPrivate,
            OMX_IN OMX_U32 nSizeBytes,
            OMX_IN OMX_U8* pBuffer);
OMX_ERRORTYPE (*AllocateBuffer)(
            OMX_IN OMX_HANDLETYPE hComponent,
            OMX_INOUT OMX_BUFFERHEADERTYPE** ppBuffer,
            OMX_IN OMX_U32 nPortIndex,
            OMX_IN OMX_PTR pAppPrivate,
            OMX_IN OMX_U32 nSizeBytes);
OMX_ERRORTYPE (*FreeBuffer)(OMX_IN  OMX_HANDLETYPE hComponent, OMX_IN OMX_U32 nPortIndex, OMX_IN OMX_BUFFERHEADERTYPE* pBuffer);

OMX_ERRORTYPE (*EmptyThisBuffer)(OMX_IN  OMX_HANDLETYPE hComponent, OMX_IN OMX_BUFFERHEADERTYPE* pBuffer);
OMX_ERRORTYPE (*FillThisBuffer)(OMX_IN  OMX_HANDLETYPE hComponent, OMX_IN OMX_BUFFERHEADERTYPE* pBuffer);

OMX_ERRORTYPE (*SetCallbacks)(OMX_IN  OMX_HANDLETYPE hComponent, OMX_IN OMX_CALLBACKTYPE* pCallbacks, OMX_IN OMX_PTR pAppData);
```

定义OpenMAX IL API的过程中，其设计者尽可能保证普适性，以使其满足不同的芯片商、软件提供商以及产品制造商的需求。最明显的特征包括：

1. 通过回调函数的机制来完成输入输出，以保持对同步机制和异步机制的统一支持。
2. 同时提供分配缓存和复用缓存的API，既支持调用者管理对缓存的管理，也支持组件对缓存的管理。

7.4.1.1 Port参数的查询和设置

OpenMAX IL组件基于Port进行数据处理，承载数据的缓冲必须附着在给定的Port上。一个Port应具备如下几个基本参数：

1. Index：该Port在组件中的唯一索引号；
2. Domain：该Port的作用域（音频、视频、图像或其他）。
3. Dir：该Port的数据传输方向（输入、还是输出）。

一个组件必须实现索引号为0x1000002~0x1000005的参数接口，即OMX_IndexParamAudioInit/OMX_IndexParamImageInit/OMX_IndexParamVideoInit/OMX_IndexParamOtherInit，不同作用域的组件使用不同的索引号。通过调用GetParameter，OpenMAX IL的使用者可以获取组件的Port数目和编号，这些信息存放在OMX_PORT_PARAM_TYPE结构中：

```
typedef struct OMX_PORT_PARAM_TYPE {
    OMX_U32 nSize;              /**< size of the structure in bytes */
    OMX_VERSIONTYPE nVersion;   /**< OMX specification version information */
    OMX_U32 nPorts;             /**< The number of ports for this component */
    OMX_U32 nStartPortNumber;   /** first port number for this type of port */
} OMX_PORT_PARAM_TYPE;
```

Port参数信息的查询和配置则要通过索引号为0x2000001（OMX_IndexParamPortDefinition）的参数接口实现，这个参数接口对应OMX_PARAM_PORTDEFINITIONTYPE结构：

```
typedef struct OMX_PARAM_PORTDEFINITIONTYPE {
    OMX_U32 nSize;                  /**< Size of the structure in bytes */
    OMX_VERSIONTYPE nVersion;       /**< OMX specification version information */
    OMX_U32 nPortIndex;             /**< Port number the structure applies to */
    OMX_DIRTYPE eDir;               /**< Direction (input or output) of this port */
    OMX_U32 nBufferCountActual;     /**< The actual number of buffers allocated on this port */
    OMX_U32 nBufferCountMin;        /**< The minimum number of buffers this port requires */
    OMX_U32 nBufferSize;            /**< Size, in bytes, for buffers to be used for this channel */
    OMX_BOOL bEnabled;              /**< Ports default to enabled and are enabled/disabled by
                                         OMX_CommandPortEnable/OMX_CommandPortDisable.
                                         When disabled a port is unpopulated. A disabled port
                                         is not populated with buffers on a transition to IDLE. */
    OMX_BOOL bPopulated;            /**< Port is populated with all of its buffers as indicated by
                                         nBufferCountActual. A disabled port is always unpopulated.
                                         An enabled port is populated on a transition to OMX_StateIdle
                                         and unpopulated on a transition to loaded. */
    OMX_PORTDOMAINTYPE eDomain;     /**< Domain of the port. Determines the contents of metadata below. */
    union {
        OMX_AUDIO_PORTDEFINITIONTYPE audio;
        OMX_VIDEO_PORTDEFINITIONTYPE video;
        OMX_IMAGE_PORTDEFINITIONTYPE image;
        OMX_OTHER_PORTDEFINITIONTYPE other;
    } format;
    OMX_BOOL bBuffersContiguous;
    OMX_U32 nBufferAlignment;
} OMX_PARAM_PORTDEFINITIONTYPE;
```

前面提到的三个基本参数由nPortIndex、eDir和eDomain给出，与Port相关的缓冲参数由nBufferCountMin，nBufferSize及nBufferCountActual给出，分别表示Port需要的最少的缓冲个数、最小的缓存尺寸以及实际的缓冲个数，其中，只有第三个参数可写，此外，还有两个只读的参数bBuffersContiguous和nBufferAlignment，给出缓冲区是否需要连续内存以及其对齐方式。bEnabled和bPopulated是两个只读的状态标记，分别表示Port是否使能以及Port指定的附属缓冲区是否全部到位。

format则是与作用域相关的具体的参数设定。对于音频，相应的结构为OMX_AUDIO_PORTDEFINITIONTYPE：

```
typedef struct OMX_AUDIO_PORTDEFINITIONTYPE {
    OMX_STRING cMIMEType;
    OMX_NATIVE_DEVICETYPE pNativeRender;
    OMX_BOOL bFlagErrorConcealment;
    OMX_AUDIO_CODINGTYPE eEncoding;
} OMX_AUDIO_PORTDEFINITIONTYPE;
```

对于单一Role的音频组件，其输入Port和输出Port的cMIMEType、eEncoding通常是缺省的。

视频对应的结构为OMX_VIDEO_PORTDEFINITIONTYPE：

```
typedef struct OMX_VIDEO_PORTDEFINITIONTYPE {
    OMX_STRING cMIMEType;
    OMX_NATIVE_DEVICETYPE pNativeRender;
    OMX_U32 nFrameWidth;
    OMX_U32 nFrameHeight;
    OMX_S32 nStride;
    OMX_U32 nSliceHeight;
    OMX_U32 nBitrate;
    OMX_U32 xFramerate;
    OMX_BOOL bFlagErrorConcealment;
    OMX_VIDEO_CODINGTYPE eCompressionFormat;
    OMX_COLOR_FORMATTYPE eColorFormat;
    OMX_NATIVE_WINDOWTYPE pNativeWindow;
} OMX_VIDEO_PORTDEFINITIONTYPE;
```

视频组件具备一些特定的参数。其中，nFrameWidth和nFrameHeight表示以像素为单位的图像的宽度和高度，对于输入Port，如果这两个值设置为0，组件会对图像的尺寸进行自动检测；nStride和nSliceHeight则是描述非压缩图像的缓冲区的参数，nStride表示图像缓冲区的横跨字节数，可读可写；nSliceHeight则表示图像缓冲区的纵向高度，只读。nBitrate表示压缩数据的比特率，0为变比特率或未知比特率；xFramerate表示非压缩数据的帧率，0为变帧率或未知帧率。eCompressionFormat表示压缩格式，如果eCompressionFormat设置为OMX_VIDEO_CodingUnused，则eColorFormat表示非压缩数据的格式。对于视频编码器组件，输入Port的eCompressionFormat可设为OMX_VIDEO_CodingUnused，同时指定eColorFormat，而输出Port的eCompressionFormat需要设置为目的格式，同时还要设置编解码器参数，对于视频解码器组件，输入Port的eCompressionFormat可设可不设，组件会对输入的码流进行自动检测，输出Port的eCompressionFormat可设为OMX_VIDEO_CodingUnused，如果不打算使用缺省的输出格式，还需要同时指定eColorFormat。

图像组件对应的结构为OMX_IMAGE_PORTDEFINITIONTYPE，其成员的含义大致与OMX_VIDEO_PORTDEFINITIONTYPE相同：

```
typedef struct OMX_IMAGE_PORTDEFINITIONTYPE {
    OMX_STRING cMIMEType;
    OMX_NATIVE_DEVICETYPE pNativeRender;
    OMX_U32 nFrameWidth;
    OMX_U32 nFrameHeight;
    OMX_S32 nStride;
    OMX_U32 nSliceHeight;
    OMX_BOOL bFlagErrorConcealment;
    OMX_IMAGE_CODINGTYPE eCompressionFormat;
    OMX_COLOR_FORMATTYPE eColorFormat;
    OMX_NATIVE_WINDOWTYPE pNativeWindow;
} OMX_IMAGE_PORTDEFINITIONTYPE;
```

对于非单一Role的组件，可以使用索引号为0x4000001/0x5000001/0x6000001（OMX_IndexParamAudioPortFormat/OMX_IndexParamImagePortFormat/OMX_IndexParamVideoPortFormat）的参数来查询某个Port支持的数据格式。以视频为例，参数索引号为0x4000001（OMX_IndexParamAudioPortFormat），对应的结构为OMX_AUDIO_PARAM_PORTFORMATTYPE：

```
typedef struct OMX_AUDIO_PARAM_PORTFORMATTYPE {
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U32 nPortIndex;
    OMX_U32 nIndex;
    OMX_AUDIO_CODINGTYPE eEncoding;
} OMX_AUDIO_PARAM_PORTFORMATTYPE;
```

查询时，需要设置除eEncoding之外的其他域，组件返回不同nIndex对应的eEncoding。代码示例如下：

```
    OMX_AUDIO_PARAM_PORTFORMATTYPE AudioFormat;
    CONFIG_SIZE_AND_VERSION(AudioFormat);
    AudioFormat.nPortIndex = in;
    for (i = 0;;i++)
    {
        AudioFormat.nIndex = i;
```

```
        AudioFormat.nIndex = i;

        error = OMX_GetParameter(m_Handle, OMX_IndexParamAudioPortFormat, &AudioFormat);
        if (error != OMX_ErrorNoMore)
        {
            continue;
        }

        if (AudioFormatType == AudioFormat.eEncoding)
        {
            LOGI("Audio Format Found on Input Port, with Index=%d\n", i);
            break;
        }
    }
```

7.4.1.2 组件状态的转换及缓冲管理

组件通过OMX_SendCommand发送OMX_CommandStateSet的命令发起状态转换，转换的结果由OMX_EventCmdComplete事件进行通知，正常的状态转换序列为，：

```
LOADED <-> IDLE <-> EXECUTING <-> PAUSED
```

其中，LOADED为组件的初始状态。 在向IDLE状态转换的命令发起之后，组件各Port所需的缓冲需要得到分配，成功的话才会进入IDLE状态。此时数据处理过程尚不能启动，需要进一步发起向EXECUTING状态的转换，转换成功后，才可以调用OMX_EmptyThisBuffer和OMX_FillThisBuffer请求数据处理。 如果各Port所需的缓冲资源分配失败，组件会进入一个暂时的WAIT_FOR_RESOURCE状态。

若组件遭遇异常，则进入INVALID状态，此时需要重新装载组件。

关联于某Port的缓冲的管理借助数据结构OMX_BUFFERHEADERTYPE实现：

```
typedef struct OMX_BUFFERHEADERTYPE
{
    OMX_U32 nSize;
    OMX_VERSIONTYPE nVersion;
    OMX_U8* pBuffer;
    OMX_U32 nAllocLen;
    OMX_U32 nFilledLen;
    OMX_U32 nOffset;
    OMX_PTR pAppPrivate;
    OMX_PTR pPlatformPrivate;
    OMX_PTR pInputPortPrivate;
    OMX_PTR pOutputPortPrivate;
    OMX_HANDLETYPE hMarkTargetComponent;
    OMX_PTR pMarkData;
    OMX_U32 nTickCount;
    OMX_TICKS nTimeStamp;
    OMX_U32 nFlags;
    OMX_U32 nOutputPortIndex;
    OMX_U32 nInputPortIndex;
}   OMX_BUFFERHEADERTYPE;
```

相关的函数包括：

```
    OMX_ERRORTYPE (*AllocateBuffer)(
            OMX_IN OMX_HANDLETYPE hComponent,
            OMX_INOUT OMX_BUFFERHEADERTYPE** ppBufferHdr,
            OMX_IN OMX_U32 nPortIndex,
            OMX_IN OMX_PTR pAppPrivate,
            OMX_IN OMX_U32 nSizeBytes);
    OMX_ERRORTYPE (*UseBuffer)(
            OMX_IN OMX_HANDLETYPE hComponent,
            OMX_INOUT OMX_BUFFERHEADERTYPE** ppBufferHdr,
            OMX_IN OMX_U32 nPortIndex,
            OMX_IN OMX_PTR pAppPrivate,
            OMX_IN OMX_U32 nSizeBytes,
            OMX_IN OMX_U8* pBuffer);
    OMX_ERRORTYPE (*FreeBuffer)(
            OMX_IN  OMX_HANDLETYPE hComponent,
            OMX_IN  OMX_U32 nPortIndex,
            OMX_IN  OMX_BUFFERHEADERTYPE* pBuffer);
    OMX_ERRORTYPE (*EmptyThisBuffer)(
            OMX_IN  OMX_HANDLETYPE hComponent,
            OMX_IN  OMX_BUFFERHEADERTYPE* pBuffer);
    OMX_ERRORTYPE (*FillThisBuffer)(
            OMX_IN  OMX_HANDLETYPE hComponent,
            OMX_IN  OMX_BUFFERHEADERTYPE* pBuffer);
```

AllocateBuffer分配缓冲及管理缓冲的OMX_BUFFERHEADERTYPE结构，返回给ppBufferHdr，其中的数据域nOutputPortIndex、nInputPortIndex根据输入参数nPortIndex初始化，pAppPrivate、pInputPortPrivate以及pOutputPortPrivate由输入参数pAppPrivate初始化。该函数的调用时机为：组件处于LOADED状态且向IDLE状态转换的命令以及下达；或因资源分配失败致使组件进入WAIT_FOR_RESOURCE状态。如果组件其它状态，需要将该组件禁止之后才能调用此函数。

典型的调用方式如下：

```
/* IL client asks component to allocate buffers */
for (i=0;i<pClient->nBufferCount;i++)
{
    OMX_AllocateBuffer(hComp, &pClient->pBufferHdr[i], pClient->nPortIndex, pClient, pClient->nBufferSize);
}
```

UseBuffer允许组件使用已经由输入参数pBuffer指定的内存，该函数将创建OMX_BUFFERHEADERTYPE结构返回给ppBufferHdr，并采用和AllocateBuffer相同的方式初始化其中的某些数据域，UseBuffer也与AllocateBuffer相同。

典型的调用方式如下：

```
/* supplier port allocates buffers and pass them to non-supplier */
for (i=0;i<pPort->nBufferCount;i++)
{
    pPort->pBuffer[i] = malloc(pPort->nBufferSize);
    OMX_UseBuffer(pPort->hTunnelComponent, &pPort->pBufferHdr[i], pPort->nTunnelPort, pPort, pPort->nBufferSize, pPort->pBuffer[j]);
}
```

无论是使用UseBuffer还是AllocateBuffer，得到的OMX_BUFFERHEADERTYPE结构均需要由FreeBuffer释放；对于后者，FreeBuffer还将同时释放实际的缓冲区。

当组件进入EXECUTING状态之后，其运作由EmptyThisBuffer和FillThisBuffer来驱动。这是两个异步函数，输入参数均为OMX_BUFFERHEADERTYPE结构，EmptyThisBuffer作用于输入Port，请求组件读数据；FillThisBuffer作用于输出Port，请求组件写数据。调用EmptyThisBuffer时，需要在OMX_BUFFERHEADERTYPE结构的nFilledLen字段中给出输入数据的长度。

这两个函数均为非阻塞函数，操作的完成由回调函数来通知：

```
    OMX_ERRORTYPE (*EmptyBufferDone)(OMX_IN OMX_HANDLETYPE hComponent, OMX_IN OMX_PTR pAppData, OMX_IN OMX_BUFFERHEADERTYPE* pBuffer);
    OMX_ERRORTYPE (*FillBufferDone)(OMX_OUT OMX_HANDLETYPE hComponent, OMX_OUT OMX_PTR pAppData, OMX_OUT OMX_BUFFERHEADERTYPE* pBuffer);
```

pAppData指向的正是调用OMX_GetHandle时嵌入的cookie，pBuffer则是处理中的OMX_BUFFERHEADERTYPE结构。

EmptyBufferDone事件产生于组件从输入Port附着的缓冲区中成功读取数据之后；FillBufferDone事件则产生于组件向输出Port附着的缓冲区成功写入数据之后。对于后者，OMX_BUFFERHEADERTYPE结构的字段nOffset和nFilledLen给出数据在缓冲中的偏移和长度。

如果缓冲区内存储的是音视频压缩数据（如解码器的输入缓冲），需要支持三种存储方式：

1. 允许音视频帧的分割和分组
2. 允许音视频帧的分组，但不允许分割
3. 每个缓冲区仅允许存储一个压缩音视频帧

对于第一种方式，要求解码组件进行分帧和组帧，因此对组件的能力要求高，同时需要组件提供额外的帧缓冲，组件外部的控制逻辑也比较复杂；对于第二种方式，要求组件具备分帧能力。而实际上，流行的媒体容器如MOV、MKV等具备很好的组帧和分帧能力，在这种情况下采用第三种方式是可行的，而且能够避免额外的运算量和存储空间，提高性能。

通过nFlags字段可以设置缓冲的附加信息：

- OMX_BUFFERFLAG_EOS：流结束附．当缓冲中包含的数据是媒体流的最后一部分时设置

- OMX_BUFFERFLAG_STARTTIME：
- OMX_BUFFERFLAG_DECODEONLY：
- OMX_BUFFERFLAG_DATACORRUPT：
- OMX_BUFFERFLAG_ENDOFFRAME：暗示缓冲中包含一个完整媒体帧的结束，之后无其余媒体数据。
- OMX_BUFFERFLAG_SYNCFRAME：关键帧标记
- OMX_BUFFERFLAG_EXTRADATA：extra data标记，表示缓冲中的媒体数据后含有以OMX_OTHER_EXTRADATATYPE结构存放的附加数据。
- OMX_BUFFERFLAG_CODECCONFIG：表示缓冲中的数据是配置数据，如H.264压缩数据的SPS, PPS等。注意，OpenMAX IL不允许配置数据和媒体数据混在同一个缓冲中传送。

nTimeStamp字段则标识了缓冲中的媒体数据的播放时间，单位微秒。

无论是nFlag，还是nTimeStamp，都是作用于缓冲区中的第一个起始于该缓冲区的媒体帧。

OMX_CommandMarkBuffer命令实现缓冲标记的功能。如果某个缓冲被标记，即使该缓冲中的数据经多个组件处理之后，仍然可以被发现。OMX_CommandMarkBuffer命令发送给某个Port，则该Port在接下来拿到第一个缓冲时对该缓冲进行标记，标记的信息来源于SendCommand的第二个输入参数，这是一个结构体：

```
typedef struct OMX_MARKTYPE
{
    OMX_HANDLETYPE hMarkTargetComponent;
    OMX_PTR pMarkData;
} OMX_MARKTYPE;
```

hMarkTargetComponent表示需要对标记检测的组件的句柄，pMarkData则是设置的标记信息，这两个参数会被设入对应OMX_BUFFERHEADERTYPE结构的hMarkTargetComponent字段和pMarkData字段。如果某个组件发现自己的句柄正是OMX_BUFFERHEADERTYPE结构中的hMarkTargetComponent，则产生OMX_EventMark事件，注册的EventHandler被调用。

7.4.1.3 OMX_EventPortSettingsChanged事件

组件Port的参数可能会在运行过程中发生变化，这种变化由OMX_EventPortSettingsChanged事件来通知。在收到Port参数变化的事件之后，需要检查新的Port参数，并决定是否重新分配缓冲区。通常的处理流程为：

获取OMX_IndexParamPortDefinition参数 → 发送OMX_CommandFlush命令清空该Port → 发送OMX_CommandPortDisable命令禁止该端口 → 释放已有的缓冲区并重新分配 → 发送OMX_CommandPortEnable命令使能该端口

7.4.1.4 回调函数设计

7.5 FFMPEG

FFMPEG是一个货真价实的多媒体软件解决方案，源码开放，遵循LPGL，提供基于通用CPU的最优化的协议解析，格式解析及媒体编解码能力，并兼容某些硬件API，支持常见的音视频设备，为目前绝大多数多媒体播放器合法或者不合法地使用着。其详情请参见深入浅出FFMPEG。

7.6 Android系统的多媒体结构

作为一个专门针对移动设备的智能系统，其数字媒体的处理能力较之前的传统手机操作系统要强大很多，在如智能电视和平板电脑等产品中甚至扮演着核心角色。

7.6.1 Android系统的多媒体能力

Android系统完整的应用开发接口是以Java的形式给出的，其媒体能力封装到一个名为android.media的包中，包含了大量实现数字影音功能的类以及相关的接口定义。其中，两个最基础的类MediaPlayer和MediaRecorder分别实现了数字媒体内容播放和录制的功能。

7.7 GStreamer

八、系统部署

九、未来展望

1) http://www.olympic.org/berlin-1936-summer-olympics [http://www.olympic.org/berlin-1936-summer-olympics]
2) ISO/IEC, Information technology — Generic coding of moving pictures and associated audio information: Systems, 2nd Edition, 2000
3) ATSC Audio/Video Standards and Solutions A Status Report : http://atsc.org/wp-content/uploads/pdf/audio_seminar/12%20-%20JONES%20-%20Audio%20and%20Video%20synchronization-Status.pdf [http://atsc.org/wp-content/uploads/pdf/audio_seminar/12%20-%20JONES%20-%20Audio%20and%20Video%20synchronization-Status.pdf]
4) C. E. Shannon: A mathematical theory of communication. Bell System Technical Journal, vol. 27, pp. 379-423 and 623-656, July and October, 1948
5) Robert M.Gray and David L.Neuhoff, "Quantization", IEEE Trans Information Theory, Vol.IT-44, Oct.1998: pp2325-2383
6) JAIN, J.R., and JAIN, A.K.: 'Displacement measurement and its application in interframe coding', I E E E Trans., COM-29, (12), 1981, pp. 179%1808
7) Iain E. Richardson, "The H.264 Advanced Video Compression Standard, 2nd Edition", Wiley, July 27 2010,