



H.264 PC 解码库软件

## 开发指南

文档版本	05
发布日期	2008-11-30
BOM编码	N/A

秘密

版权所有 © 深圳市海思半导体有限公司

深圳市海思半导体有限公司为客户提供全方位的技术支持，用户可与就近的海思办事处联系，也可直接与公司总部联系。

## 深圳市海思半导体有限公司

地址：深圳市龙岗区坂田华为基地华为电气生产中心 邮编：518129

网址：<http://www.hisilicon.com>

客户服务电话：+86-755-28788858

客户服务传真：+86-755-28357515

客户服务邮箱：[support@hisilicon.com](mailto:support@hisilicon.com)

**版权所有 © 深圳市海思半导体有限公司 2007-2008。保留一切权利。**

非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

### 商标声明



**HISILICON**、海思和其他海思商标均为深圳市海思半导体有限公司的商标。

本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

### 注意

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

秘密

版权所有 © 深圳市海思半导体有限公司



## 目 录

前 言.....	1
1 概述.....	1-1
2 码流识别与传输.....	2-1
2.1 H.264 基本语法结构简介 .....	2-2
2.2 nalu简介 .....	2-2
2.3 IDR帧的识别方法 .....	2-3
2.4 帧边界识别方法.....	2-3
2.4.1 帧边界识别简介.....	2-3
2.4.2 编码器和解码器配合实现帧边界识别 .....	2-4
2.4.3 高效的帧边界识别方法 .....	2-4
2.5 解码视频通道切换 .....	2-4
2.6 流媒体网络传输的方式 .....	2-5
2.7 解码前的丢包策略 .....	2-6
3 解码示例.....	3-1
3.1 流式解码 .....	3-2
3.1.1 Hi264DecFrame函数 .....	3-2
3.1.2 解码流程.....	3-2
3.1.3 参考代码.....	3-3
3.2 nalu方式解码 .....	3-7
3.3 AU方式解码 .....	3-8
3.3.1 概述.....	3-8
3.3.2 Hi264DecLoadAU函数.....	3-8
3.3.3 Hi264DecLoadAU函数源码.....	3-9
3.3.4 解码流程.....	3-12
3.3.5 参考代码.....	3-13
3.4 数据类型与数据结构 .....	3-17
3.4.1 通用数据类型.....	3-17
3.4.2 ParseContext .....	3-18
A 缩略语 .....	A-1



## 插图目录

图 3-1 流式解码流程图.....	3-3
图 3-2 AU方式解码流程图 .....	3-13



## 表格目录

表 2-1 nalu_type和H.264 语法结构之间的对应关系 .....	2-3
---	-----



## 前 言

### 概述

本节介绍本文档的内容、对应的产品版本、适用的读者对象、行文表达约定、历史修订记录等。

### 产品版本

与本文档相对应的产品版本如下所示。

产品名称	产品版本
Hi3510 通信媒体处理器	V100
Hi3511 H.264 编解码处理器	V100
Hi3512 H.264 编解码处理器	V100

### 读者对象

本文档主要适用于以下工程师：

- 软件开发工程师
- 技术支持工程师

### 内容简介

本文档内容组织如下。

章节	内容
1 概述	简单介绍 H.264 视频编解码标准的基本特性。
2 码流识别与传输	介绍 H.264 码流结构以及 IDR 帧识别、AU 识别方法。








章节	内容
3 解码示例	提供流式解码、nalu 方式解码和 AU 方式解码的参考示例。
A 缩略语	列举了本文档出现的缩略语。

## 约定

### 符号约定

在本文中可能出现下列标志，它们所代表的含义如下。

符号	说明
 危险	以本标志开始的文本表示有高度潜在危险，如果不能避免，会导致人员死亡或严重伤害。
 警告	以本标志开始的文本表示有中度或低度潜在危险，如果不能避免，可能导致人员轻微或中等伤害。
 注意	以本标志开始的文本表示有潜在风险，如果忽视这些文本，可能导致设备损坏、数据丢失、设备性能降低或不可预知的结果。
 窍门	以本标志开始的文本能帮助您解决某个问题或节省您的时间。
 说明	以本标志开始的文本是正文的附加信息，是对正文的强调和补充。

### 通用格式约定

格式	说明
宋体	正文采用宋体表示。
黑体	一级、二级、三级标题采用黑体。
楷体	警告、提示等内容一律用楷体，并且在内容前后增加线条与正文隔离。
“Terminal Display” 格式	“Terminal Display” 格式表示屏幕输出信息。此外，屏幕输出信息中夹杂的用户从终端输入的信息采用加粗字体表示。



## 修订记录

修订记录累积了每次文档更新的说明。最新版本的文档包含以前所有文档版本的更新内容。

修改日期	版本	修改说明
2008-08-30	05	<ul style="list-style-type: none"><li>• 2.6 流媒体网络传输的方式：删除了不恰当的描述。</li><li>• 2.7 解码前的丢包策略：修改描述。</li><li>• 增加 Hi3512 芯片信息。</li></ul>
2008-04-03	04	<ul style="list-style-type: none"><li>• 2.5 解码视频通道切换中优化了描述。</li><li>• 3.1.3 参考代码增加了启动 Deinterlace 的说明。</li><li>• 3.5 参考代码中增加了启动 Deinterlace 的说明。</li></ul>
2008-01-15	03	<ul style="list-style-type: none"><li>• 3.3.3 Hi264DecLoadAU 函数源码：修改了 Hi264DecLoadAU 函数的源码。</li><li>• 3.3.5 参考代码：修改了 AU 方式解码参考代码。</li></ul>
2007-11-20	02	<ul style="list-style-type: none"><li>• 3.3.2 Hi264DecLoadAU 函数：在“参数”中增加“PrevFirstMBAAddr”成员，并修改了“描述”中的内容。</li><li>• 3.3.3 Hi264DecLoadAU 函数源码：修改了 Hi264DecLoadAU 函数的源码。</li><li>• 3.3.5 参考代码：修改了 AU 方式解码参考代码。</li><li>• 3.4.2 ParseContext：在“参数”中增加“PrevFirstMBAAddr”成员。</li></ul>
2007-09-30	01	第 1 次版本发布。





# 1 概述

与以往的视频压缩标准相比，H.264 视频压缩标准（简称 H.264）具有更出色的性能，因此 H.264 被称为新一代视频压缩标准。

H.264 与 H.263 或 MPEG-4 相比，主要新增特性如下：

- 采用更为精细和丰富的帧内编码及帧间预测方式，有效地减少残差数据。
- 引入新的算术编码方式，使得数据压缩比更高。
- 视频数据分层更为合理，引入 NAL 更利于网络传输。
- 取消传统的帧结构，引入 slice 结构和参数集，提高码流的抗误码能力。
- 引入灵活的参考帧管理机制，参考帧数目最多可以达到 16 个。

上述特性使得 H.264 在视频信噪比、图像质量以及应用的灵活性上有了质的飞跃，但带来的问题是 H.264 在实现上复杂度较高。

海思的 H.264 PC 解码库软件具有较高的性能和可靠性，同时提供简单易用的 API 接口，可以极大地提高基于 H.264 产品的开发速度。



## 2 码流识别与传输

### 关于本章

本章描述内容如下表所示。

标题	内容
<a href="#">2.1 H.264 基本语法结构简介</a>	介绍 H.264 的基本语法结构。
<a href="#">2.2 nalu 简介</a>	介绍 nalu 的相关信息。
<a href="#">2.3 IDR 帧的识别方法</a>	介绍 IDR 帧的识别方法。
<a href="#">2.4 帧边界识别方法</a>	介绍帧边界的识别方法。
<a href="#">2.5 解码视频通道切换</a>	介绍解码视频通道切换的相关信息。
<a href="#">2.6 流媒体网络传输的方式</a>	介绍流媒体网络传输的方式。
<a href="#">2.7 解码前的丢包策略</a>	介绍解码前的丢包策略。



## 2.1 H.264 基本语法结构简介

H.264 相对以往的视频压缩编码标准来说，在语法结构上有很大的改变，其中最大的改变体现在以下两个方面：

- 取消帧级语法单元

H.264 语法中没有 `frame_header` 之类的语法单元，帧信息全部放在 `slice_header`、SPS 和 PPS 中，这样可增强 `slice` 单元解码的独立性，提高码流的抗丢包、抗误码能力。但由于一帧图像可以对应多个 `slice`，因此解码器无法通过解析类似 `frame_header` 的语法来识别码流中的一帧数据。

- 引入 SPS、PPS 等参数集概念

- 将一个视频序列(从 IDR 帧开始到下一个 IDR 帧之前的数据称为一个视频序列)全部图像的共同特征抽取出来，放在 SPS 语法单元中。
- 将各个图像的典型特征抽取出来，放在 PPS 语法单元中。
- 只有视频序列之间才能切换 SPS，即只有 IDR 帧的第一个 `slice` 可以切换 SPS。
- 只有图像之间才能切换 PPS，即只有每帧图像的第一个 `slice` 才能切换 PPS。

从宏观上来说，SPS、PPS、IDR 帧（包含一个或多个 I-Slice）、P 帧（包含一个或多个 P-Slice）、B 帧（包含一个或多个 B-Slice）共同构成典型的 H.264 码流结构。除上述典型语法结构外，为了方便传输私有信息，H.264 还定义了 SEI 语法结构。除非编码器和解码器进行特定的语法协商，解码器一般不对 SEI 包进行解析。

## 2.2 nalu 简介

nalu 是 H.264 的最高抽象层，H.264 的所有语法结构最终都被封装成 nalu。码流中的 nalu 单元必须定义合适的分隔符，否则无法区分。H.264 视频压缩标准的附录 B 采用前缀码“00 00 01”作为 nalu 的分隔符，可以通过搜索前缀码“00 00 01”来识别一个 nalu。

nalu 有自己的语法元素，但是仅占用一个字节，即 nalu 单元除了紧跟前缀码“00 00 01”之后的第一个字节外，其他载荷都是 H.264 某种语法结构的有效载荷。

`nalu_type` 是 nalu 最重要的语法元素，它表征 nalu 内封装的 H.264 语法结构的类型。

`nalu_type` 可以按照以下方式解析：

```
nalu_type = first_byte_in_nal & 0x1F
```

这里用 `first_byte_in_nal` 表示 nalu 第一个字节。

`nalu_type` 和 H.264 语法结构之间的对应关系如表 2-1 所示。



表2-1 nalu\_type 和 H.264 语法结构之间的对应关系

nalu_type	H.264 语法结构类型	说明
01	P-Slice B-Slice I-Slice	一个 P 帧（或 B 帧）可能包含 I-Slice，但 I-Slice 对应的 nalu_type 和 IDR 帧中 I-Slice 的 nalu_type 不同。
05	I-Slice	一个 IDR 帧可以划分为若干个 I-Slice，每个 I-Slice 对应一个 nalu。
06	SEI	一个 SEI 对应一个 nalu。
07	SPS	一个 SPS 对应一个 nalu，最多可有 32 个不同的 SPS。
08	PPS	一个 PPS 对应一个 nalu，最多可有 256 个不同的 PPS。
09	AU delimiter	接入单元分界符（帧边界）。

## 2.3 IDR 帧的识别方法

解码器只能从 IDR 帧开始才能正常解码，所以播放器为了完成快进、快退、解码通道切换功能时需要识别 IDR 帧。H.264 协议虽然没有定义帧语法结构，但可以通过 nalu\_type 来识别 IDR 帧。例如，可以从码流中搜索并提取连续存放的若干个 nalu\_type 等于 05 的 nalu，即可获得一个完整的 IDR 帧，详细信息请参考“2.2 nalu 简介”。



### 注意

一个 IDR 帧可以划分为若干个 I-Slice，每个 I-Slice 对应一个 nalu，也就是说一个 IDR 帧可以对应多个 nalu，提取时需要保证 IDR 帧的完整性。

SPS 和 PPS 的搜索方法和 IDR 帧相同，但每个 SPS 或者 PPS 仅对应一个 nalu。

## 2.4 帧边界识别方法

### 2.4.1 帧边界识别简介

H.264 将构成一帧图像所有 nalu 的集合称为一个 AU，帧边界识别实际上就是识别 AU。因为 H.264 取消帧级语法，所以无法简单地从码流中获取 AU。解码器只有在解码的过程中，通过某些语法元素的组合才能判断一帧图像是否结束。一般来说，解码器必须在



完成一帧新图像的第一个 slice\_header 语法解码之后, 才能知道前一帧图像已经结束。因此, 最严谨的 AU 识别步骤如下:

- 步骤 1 对码流实施“去 03 处理”。
- 步骤 2 解析 nalu 语法。
- 步骤 3 解析 slice\_header 语法。
- 步骤 4 综合判断前后两个 nalu 以及对应的 slice\_header 中的若干个语法元素, 看是否发生变化。如果发生变化, 则说明这两个 nalu 属于不同的帧, 否则说明这两个 nalu 属于同一帧。

----结束

显然, 在解码前完成上述的 AU 识别消耗许多 CPU 资源, 因此不推荐使用 AU 方式解码。

## 2.4.2 编码器和解码器配合实现帧边界识别

为了提供一种简单的 AU 识别方案, H.264 规定一种类型为 09 的 nalu (详细信息请参见“表 2-1 nalu\_type 和 H.264 语法结构之间的对应关系”), 即编码器在每次完成一个 AU 编码后, 在码流中插入一个类型为 09 的 nalu, 在这个前提下, 解码器只需要从码流中搜索类型为 09 的 nalu 即可获得一个 AU。



说明

H.264 并不强制要求编码器插入类型为 09 的 nalu, 因此并非所有的码流都具备这种特征。对于编码器和解码器协同工作的应用场景, 建议让编码器插入类型为 09 的 nalu, 这样可以降低解码器识别 AU 的代价。

## 2.4.3 高效的帧边界识别方法

为了降低解码器在解码前识别 AU 的代价, 本文提出一种高效的 AU 识别方法, 其主要思路是利用一帧图像的第一个 slice\_header 中的语法元素 first\_mb\_in\_slice 一般等于 0 这个特征 (对于包含 ASO 或者 FMO 特性的码流, 这个条件不一定成立)。

这种高效的帧边界识别方法主要是对配置的一段线性连续的缓冲区中的码流进行搜索, 获得帧边界后就返回帧边界的位置信息, 不进行任何拷贝操作, 详细信息请参见“3.3 AU 方式解码”。

## 2.5 解码视频通道切换

一般来说, 在视频点播或者视频通道切换时, 仅搜索 IDR 帧送给解码器是不够的, 还必须将对应的 SPS 和 PPS 送给解码器。H.264 并没有要求 SPS、PPS 以及 IDR 帧捆绑传输, 只是规定一定要在 IDR 帧之前传输相应的 SPS 和 PPS。因此, 最严谨的做法是从视频码流的起始位置开始搜索, 将获取的全部 SPS 和 PPS 依次送给解码器, 直到搜索到一个 IDR 帧。

为了降低应用的复杂性, 在视频点播或者实时编解码应用场景中, 编码器一般连续传递 SPS、PPS 和 IDR 帧, 即严格在 IDR 帧之前传递当前视频序列所需要的 SPS 参数和若干个 PPS 参数。Hi351x 编码器遵循这一准则, 因此可以在每个 IDR 帧的第一个 nalu 前面找到 4 个尺寸较小的 nalu, 它们即为当前视频序列解码所需要的全部参数, 其中第一个



nal 为 SPS。简单地说，对于 Hi351x 码流，只要找到 SPS，并从此位置开始送给解码器解码，那么一定可以得到完整的视频图像。

在视频输入通道不断切换的应用场景中（如单通道解码器用时间片的方式点播多个编码通道码流的场景），需要不断地切换解码器的输入视频源通道。基于上述描述的解码完整性原则，切换不是从任意位置开始的。也就是说，如果解码器需要切换到某个视频通道，那么必须从获取这个视频通道的 SPS 的位置开始切换，否则在切换点，视频图像会出现异常。

## 2.6 流媒体网络传输的方式

基于 MPEG-4 的方案在网络传输时通常采用以下两种方案：

- 用尺寸小于网络层 MTU 的定长数据段来切分线性的原始码流。
- 直接传输增加私有格式头的一整帧数据，网络层对整帧数据进行切分，而应用层则忽略由于网络拥塞而丢弃的包。

H.264 提出比 MPEG-4 更优秀的网络传输抗误码解决方案。H.264 包含 VCL 和 NAL，VCL 和 NAL 的特点如下：

- VCL 包括核心压缩引擎和块 / 宏块 / slice 的语法级别定义，应该尽量将 VCL 独立于网络之外。
- NAL 将 VCL 产生的比特字符串适配到各种各样的网络环境中。NAL 覆盖所有 slice 级以上的语法级别，并包含以下机制：
  - 提供每一个 slice 解码时所需要的参数数据。
  - 预防起始码冲突。
  - 支持 SEI。
  - 提供将编码 slice 的比特字符串在基于比特流的网络上进行传送。

H.264 将 NAL 与 VCL 分离的主要原因如下：

- 定义一个 VCL 信号处理与 NAL 传输的接口，这样允许 VCL 和 NAL 工作在完全不同的处理器平台上。
- VCL 和 NAL 都被设计成适应于异质传输环境，当网络环境不同时，网关不需要对 VCL 比特流进行重构和重编码。

IP 网络的类型可分为以下三种：

- 不可控 IP 网络（如 Internet）。
- 可控 IP 网络（如广域网）。
- 无线 IP 网络（如 3G 网络）。

这三种 IP 网络有不同的 MTU、比特出错概率和 TCP 使用标记。两个 IP 节点之间的 MTU 是动态变化的，通常假定有线 IP 网络的 MTU 为 1.5KB，无线 IP 网络 MTU 的范围从 100byte 到 500byte。

在以 UDP 发包时建议以 nal 为基本单元，这样便于在应用层进行封装。推荐使用 RTP 进行组包传输，一个 RTP 分组里放一个 nal，将 nal（包括同步头）放入 RTP 的载荷





中，并设置 RTP 的头信息。由于包传送的路径不同，接收端需要重新对 slice 分组进行排序，RTP 包含的次序信息可以用来解决这一问题。

不要随意丢弃尺寸大于 MTU 的 UDP 分组，可直接发送，由网络驱动层进行拆包和组包，对应用层不会产生影响。即使网络出现 UDP 丢包，丢弃也是一个完整的 nalu。在 H.264 中，nalu 是基于 VCL slice 的网络层封装，每个 slice 对应于多个宏块（不同于 MPEG-4）。H.264 在编码时不同 slice 的宏块之间语法是非相关的，丢掉少量的 slice 不会对其他 slice 的解码产生影响。

H.264 的应用越来越广泛，在实际的应用方案中将存在 MPEG-4 与 H.264 并存的情形。为了解码端的系统稳定，避免软件崩溃，建议以 nalu 为基本单元放入分组包里传送，这样可以做到 H.264 与 MPEG-4 互不干扰，稳定并存于一套系统中。



### 注意

- 尽量避免将不完整的 nalu 送给解码器，否则可能会出现解码器崩溃等异常现象。
- 有 MPEG-4 应用经验的用户在处理 H.264 码流时，一定要避免这样的思维定式：在发包侧，将一帧线性的码流直接等分成小于 MTU 的块加上私有头进行发包。

## 2.7 解码前的丢包策略

播放器在下列情况下需要进行丢包处理：

- 计算机 CPU 负荷过重或者网络抖动等原因引起播放器前端码流缓冲区即将溢出。
- 应用层检测到当前网络丢包现象严重。
- 播放器调度需要进行丢包处理。

目前通用的主动丢包处理方法如下：

- 尽可能将收到的所有码流信息送至解码器，不要轻易丢弃已经收到的码流信息。
- 如果播放器在上述几种情况下需要主动丢包，则播放器应从当前帧开始丢弃，一直丢弃到下一个 IDR 帧。



### 注意

当播放器需要主动丢包时，应保证丢包结束时重新送给解码器的码流是从 IDR 帧的起始位置开始，避免将参差不齐的数据送给解码器。



# 3 解码示例

## 关于本章

本章描述内容如下表所示。

标题	内容
3.1 流式解码	提供流式解码参考示例。
3.2 nalu 方式解码	提供 nalu 方式解码参考示例。
3.3 AU 方式解码	提供 AU 方式解码参考示例。
3.4 数据类型与数据结构	介绍 API 用到的通用数据类型和 ParseContext 数据结构。





## 3.1 流式解码

### 3.1.1 Hi264DecFrame 函数

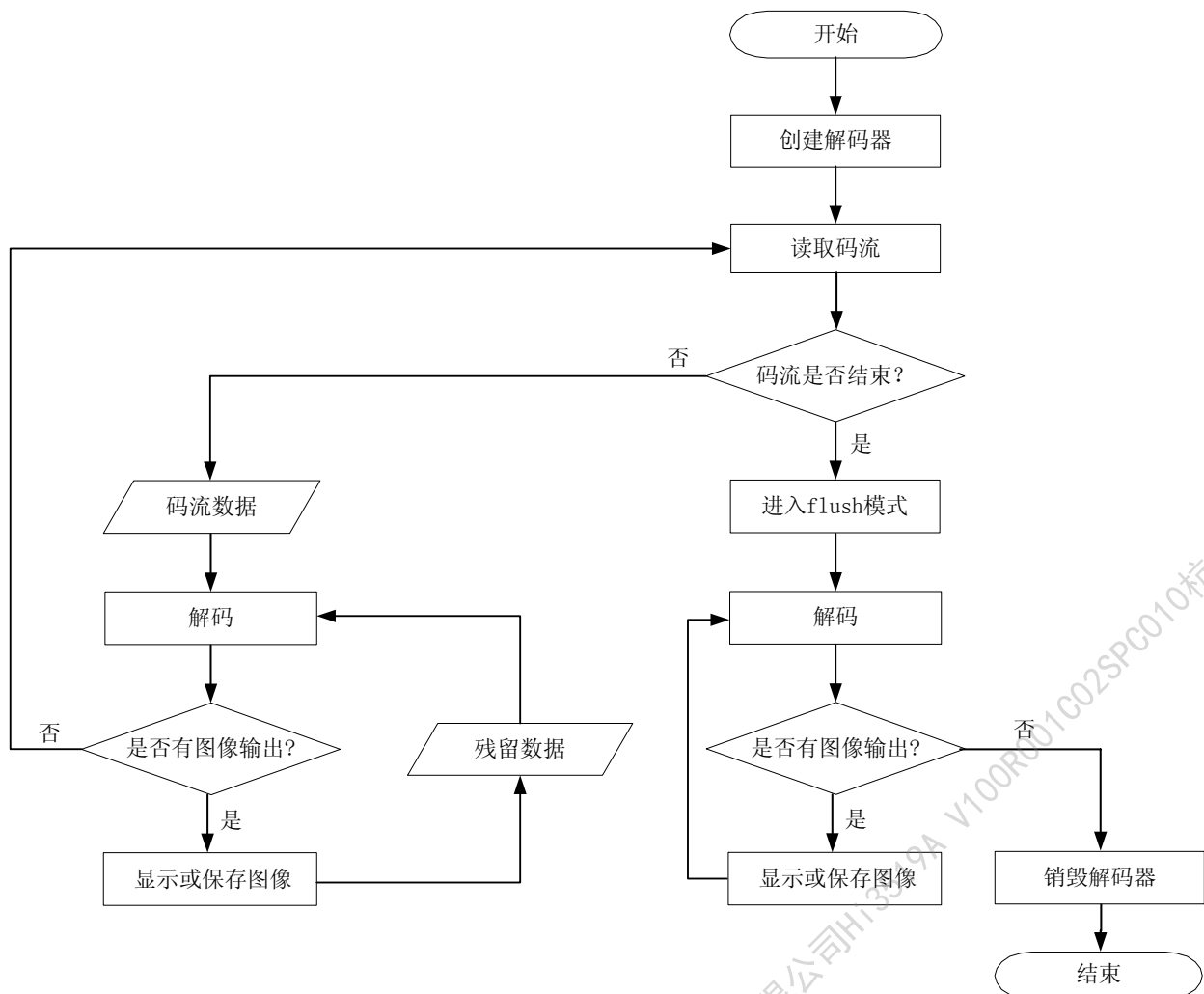
海思的解码库提供统一的流式解码 Hi264DecFrame 函数（详细信息请参见《H.264 PC 解码库软件 API 参考》），该函数具有以下功能：

- 解码一段任意长度的码流。  
输入参数有码流缓冲区的地址 pStream 和输入码流字节数 iStreamLen。
  - 如果这段码流不足一帧，需要继续输入码流进行解码。
  - 如果这段码流包括若干帧，需要在解码出一帧图像后反复调用 Hi264DecFrame 函数解码剩余码流。在码流结束后，必须进入 flush 模式清空解码器中的残留图像，直到解码器中没有残留图像为止。
- 解码库提供时间戳透传功能，输入时间戳 ullPTS 将会透传到输出图像结构体中，方便应用层控制播放。
- 输出参数 pDecFrame 包括图像类型标志 uPicFlag。如果解码输出的是场图像，需要在解码后调用 de\_interlace 函数将两场图像合成一帧图像。
- 输出参数 pDecFrame 包括当前帧出错信息 bError，可以根据 bError 判断当前图像是否有错。
- 如果码流中含有用户数据，输出参数 pDecFrame 将包含用户数据，否则用户数据为空指针。

### 3.1.2 解码流程

流式解码流程如图 3-1 所示。

图3-1 流式解码流程图



### 3.1.3 参考代码

流式解码参考代码如下：

```
#include "hi_config.h"
#include "hi_h264api.h"

#define STREAM_BUFFER_LEN 0x8000

int main(int argc, char** argv)
{
    HI_S32 end = 0;

    /* 码流缓冲区 */
    HI_U8 buf[STREAM_BUFFER_LEN];
    H264_DEC_ATTR_S dec attribute;
```



```
H264_DEC_FRAME_S dec_frame;
HI_HDL handle = NULL;

/* 解码时间和帧数 */
LARGE_INTEGER lpFrequency;
LARGE_INTEGER t1;
LARGE_INTEGER t2;
HI_U32 time;
HI_U32 pic_cnt = 0;

FILE *h264 = NULL;          /* 输入码流文件 */
FILE *yuv = NULL;           /* 输出yuv文件 */

if (argc < 2)
{
    fprintf(stderr, "error comand format or no H.264 stream!\n");
    fprintf(stderr, "the Example: hi_264sample stream_file.264\n");
    goto exitmain;
}
else
{
    /* 打开输入码流文件 */
    h264 = fopen(argv[1], "rb");
    if (NULL == h264)
    {
        fprintf(stderr, "Unable to open a h264 stream file %s \n", argv[1]);
        goto exitmain;
    }
    printf("decode file: %s...\n", argv[1]);

    if (argc > 2)
    {
        /* 打开输出yuv文件 */
        yuv = fopen(argv[2], "wb");
        if (NULL == yuv)
        {
            fprintf(stderr, "Unable to open the file to save yuv %s.\n",
                argv[2]);
            goto exitmain;
        }
        printf("save yuv file: %s...\n", argv[2]);
    }
}
```



```
}

/* 初始化解码器属性参数 */
/* 解码器最大参考帧数: 16 */
dec_attrbute.uBufNum = 16;

/* 解码器最大图像宽高, D1图像(720x576) */
dec_attrbute.uPicHeightInMB = 36;
dec_attrbute.uPicWidthInMB = 45;

/* 没有用户数据 */
dec_attrbute.pUserData = NULL;

/* 以 "00 00 01" 或 "00 00 00 01" 开始的码流 */
dec_attrbute.uStreamInType = 0x00;

/* bit0 = 1: 标准输出模式; bit0 = 0: 快速输出模式 */
/* bit4 = 1: 启动内部Deinterlace; bit4 = 0: 不启动内部Deinterlace */
dec_attrbute.uWorkMode = 0x10;

/* 创建解码器 */
handle = Hi264DecCreate(&dec_attrbute);
if (NULL == handle)
{
    goto exitmain;
}

/* 统计解码时间: 开始计时 */
QueryPerformanceFrequency(&lpFrequency);
QueryPerformanceCounter(&t1);

/* 解码流程 */
while (!end)
{
    /* 从文件 "h264" 读取一段码流 */
    HI_U32 len = fread(buf, 1, sizeof(buf), h264);
    /* 如果码流文件结束进入flush模式 */
    HI_U32 flags = (len > 0)?0:1;
    HI_S32 result = 0;

    result = Hi264DecFrame(handle, buf, len, 0, &dec_frame, flags);

    while (HI_H264DEC_NEED_MORE_BITS != result)
```



```

{
    if (HI_H264DEC_NO_PICTURE == result)    /* 解码器中已经没有残留图像 */
    {
        end = 1;
        break;
    }

    if (HI_H264DEC_OK == result)            /* 输出一帧图像 */
    {
        if (NULL != yuv)                   /* 保存YUV文件 */
        {
            const HI_U8 *pY = dec_frame.pY;
            const HI_U8 *pU = dec_frame.pU;
            const HI_U8 *pV = dec_frame.pV;
            HI_U32 width      = dec_frame.uWidth;
            HI_U32 height     = dec_frame.uHeight;
            HI_U32 yStride    = dec_frame.uYStride;
            HI_U32 uvStride   = dec_frame.uUVStride;

            fwrite(pY, 1, height* yStride, yuv);
            fwrite(pU, 1, height* uvStride/2, yuv);
            fwrite(pV, 1, height* uvStride/2, yuv);
        }

        /* 读取和打印用户数据 */
        if (NULL != dec_frame.pUserData)
        {
            HI_U32 i;
            printf("frame:%d, user data type: %d; user data length: %d\n",
                pic_cnt,
                dec_frame.pUserData->uUserDataSize);
            for(i=0; i<dec_frame.pUserData->uUserDataSize; i++)
                printf(" %d ", dec_frame.pUserData->pData[i]);

            printf("\n");
        }
        pic_cnt++;
    }

    /* 继续解码剩余H.264码流 */
    result = Hi264DecFrame(handle, NULL, 0, 0, &dec_frame, flags);
}

```



```
    }  
}  
  
/* 统计解码时间：结束计时 */  
QueryPerformanceCounter(&t2);  
time=(HI_U32)((t2.QuadPart-t1.QuadPart)*1000000/lpFrequency.QuadPart);  
  
/* 打印统计信息：解码时间、解码帧数、帧率，时间单位为微秒 */  
printf("time= %d us\n", time);  
printf("%d frames\n",pic_cnt);  
printf("fps: %d\n", pic_cnt*1000000/(time+1));  
  
/* 销毁解码器 */  
Hi264DecDestroy(handle);  
  
exitmain:  
if (NULL != h264)  
{  
    fclose(h264);  
}  
  
if (NULL != yuv)  
{  
    fclose(yuv);  
}  
  
return 0;  
}
```

## 3.2 nalu 方式解码

nalv 方式解码流程与流式解码流程基本相同，区别在于每次输入的码流对应一个 nalv，详细信息请参见“3.1 流式解码”。

nalv 方式解码需要注意以下几点：

- 输入的 nalv 必须符合 H.264 协议附录 B 规范，即 nalv 前面必须有前缀码“00 00 01”或“00 00 00 01”，nalv 不能进行“去 03 处理”。
- 输入参数 iStreamLen 为一个 nalv 的字节数，iStreamLen 必须包括前缀码“00 00 01”或“00 00 00 01”的长度。
- 输入参数 pStream 为输入码流缓冲区指针，必须保证该缓冲区大于 iStreamLen，不能出现越界。



- 由于 H.264 的特点，只有在下一帧图像解码之前才能判断本帧图像结束。因此解码器中至少会残留一帧图像，在结束解码之前需要将 `uflag` 置为 1，并且反复调用 `Hi264DecFrame` 函数解码剩余码流，直到解码器中没有残留图像为止。

## 3.3 AU 方式解码

### 3.3.1 概述

AU 方式解码主要应用于在解码前进行音视频同步的场合。因为传统视频协议的语法中都有帧结构，所以从码流中识别一帧图像比较容易。但 H.264 取消帧语法结构，导致直接从码流中识别一帧图像比较困难。因此，为了降低帧识别带来的解码性能损失，不推荐使用 AU 方式解码。

本章介绍一种基于高效帧识别方案的 AU 方式解码，主要原理是在解码之前引入识别一帧码流的过程，此后的解码过程与流式解码完全相同，详细信息请参见“3.1 流式解码”。



#### 注意

这种帧识别技术有一定的局限性，当码流中包含 FMO 或者 ASO 时，不能够准确识别一帧图像。

### 3.3.2 Hi264DecLoadAU 函数

#### 【目的】

确定一幅图像的边界。

#### 【语法】

```
HI_S32 Hi264DecLoadAU(HI_U8* pStream, HI_U32 iStreamLen, ParseContext *pc);
```

#### 【描述】

该函数从输入码流中找到一幅完整图像的边界。

#### 【参数】

参数	成员	取值范围	输入/输出	描述
pc	FrameStartFound	0, 1	输出	表示是否从码流缓冲区中找到 AU 起始地址。 <ul style="list-style-type: none"><li>• 0: 找到 AU 起始地址。</li><li>• 1: 未找到 AU 起始地址。</li></ul>
	iFrameLength	-	输出	一幅完整图像对应的码流长度。



参数	成员	取值范围	输入/输出	描述
	PrevFirstMBAddr	-	输出	记录上一个 Slice 的起始宏块地址。
pStream	-	-	输入	码流缓冲区的起始地址。
iStreamLen	-	-	输入	外部分配的码流缓冲区长度。应该保证 iStreamLen 大于一帧码流的长度。

#### 【返回值】

返回值	宏定义	描述
0	无	找到 AU 的边界。
-1	无	未找到 AU 的边界。

#### 【注意】

- Hi264DecLoadAU 函数不属于海思 H.264 PC 解码库的 API 函数。
- Hi264DecLoadAU 函数可以将 SPS、PPS 或者 SEI 包单独提取出来。
- 在帧模式和帧场自适应模式下，Hi264DecLoadAU 函数搜索的是一帧图像的边界；在场模式下，Hi264DecLoadAU 函数搜索的是一场图像的边界。
- Hi264DecLoadAU 函数默认输入码流缓冲区的首地址指向上一帧图像的结尾，通过已获取的一帧图像的长度来确定下一个 nalu 的起始位置和结束位置。
- 输入码流缓冲区必须保证至少能够存放下一个完整的 nalu。当输入码流缓冲区不能存放下一个完整的 nalu 时，Hi264DecLoadAU 函数将无法正确地获取该 nalu 的起始位置和结束位置。输入码流缓冲区的设置原则如下：
  - 图像格式为 CIF 格式时，输入码流缓冲区建议设置为 150KB。
  - 图像格式为 D1 格式时，输入码流缓冲区建议设置为 620KB。
- 对一段输入码流完成 AU 搜索后，解码器中可能会残留不足一个 AU 的码流，此时应该将残留码流搬运到一个新 Buffer，然后在新 Buffer 的剩余空间填充新的码流。

### 3.3.3 Hi264DecLoadAU 函数源码

Hi264DecLoadAU 函数的源码如下：

```
typedef struct ParseContext{  
    HI_U32  FrameStartFound;          /* 表示是否从码流缓冲区中找到AU起始地址。  
                                       0: 找到AU起始地址; 1:未找到AU起始地址。*/  
    HI_U32  iFrameLength;             /* 一幅完整图像对应的码流长度。*/  
    HI_U32  PrevFirstMBAddr;          /* 记录上一个Slice的起始宏块地址 */  
};
```





```
} ParseContext;

#define MOST_BIT_MASK 0x80000000
#define MAX_AU_SIZE 0x80000
static HI_U32 CountPrefixZeros(HI_U32 CodeNum)
{
    HI_U32 ZeroCount, i;
    ZeroCount = 0;
    for(i = 0; i < 32; i++)
    {
        if((CodeNum & MOSTBITMASK))
        {
            break;
        }
        else
        {
            ZeroCount++;
            CodeNum = CodeNum<<1;
        }
    }

    return ZeroCount;
}

HI_U32 CheckFirstMbAddr(HI_U8* p)
{
    HI_U32 code, zeros;
    code = (*p++)<<24;
    code |= (*p++)<<16;
    code |= (*p++)<<8;
    code |= *p++;
    zeros = CountPrefixZeros(code);
    code = code << zeros;
    return ((code >> (31 - zeros)) - 1);
}

HI_S32 Hi264DecLoadAU(HI_U8* pStream, HI_U32 iStreamLen, ParseContext *pc)
{
    HI_U32 i;
    HI_U32 FirstMbAddr;
    HI_U8* p;
    HI_U32 state = 0xffffffff;
    if( NULL == pStream || iStreamLen <= 4)
    {
```



```
        return -1;
    }

    pc->FrameStartFound = 0;
    pc->PrevFirstMBAAddr = 0;

    for( i = 0; i < iStreamLen; i++)
    {
        /*find a I-slice or a P-slice*/
        if( (state & 0xFFFFFFF1F) == 0x101 ||
            (state & 0xFFFFFFF1F) == 0x105 )
        {
            p = &pStream[i];
            FirstMbAddr = CheckFirstMbAddr(p);
            if( 1 == pc->FrameStartFound )
            {
                if( FirstMbAddr <= pc->PrevFirstMBAAddr )
                {
                    pc->iFrameLength = i - 4;
                    pc->PrevFirstMBAAddr = FirstMbAddr;
                    state = 0xffffffff;
                    return 0;
                }
            }
            else
            {
                pc->PrevFirstMBAAddr = FirstMbAddr;
            }
        }
        else
        {
            pc->PrevFirstMBAAddr = FirstMbAddr;
            pc->FrameStartFound = 1;
        }
    }

    /*find a sps, pps or au_delimiter*/
    if( (state&0xFFFFFFF1F) == 0x107 ||
        (state&0xFFFFFFF1F) == 0x108 ||
        (state&0xFFFFFFF1F) == 0x109 )
    {
        if(1 == pc->FrameStartFound)
        {
            pc->iFrameLength = i - 4;
            pc->PrevFirstMBAAddr = 0;
        }
    }
}
```



```
        state = 0xffffffff;
        return 0;
    }
    else
    {
        pc->FrameStartFound = 1;
        pc->PrevFirstMBAddr = 0;
    }
}

state = (state << 8) | pStream[i];

if( MAX_AU_SIZE - 1 <= i )
{
    pc->iFrameLength = i - 3;
    return 0;
}

}

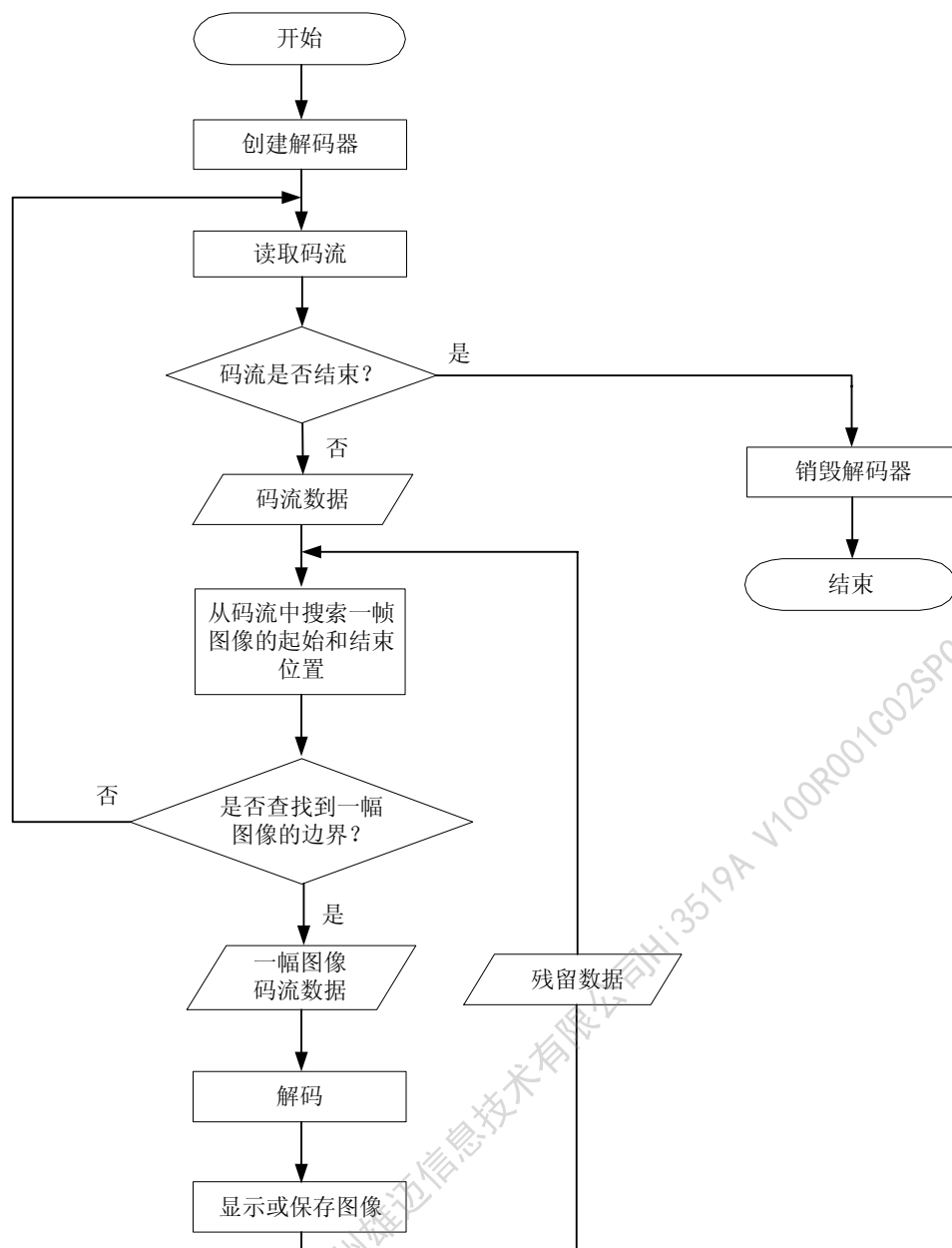
pc->iFrameLength = i;
return -1;
}
```

### 3.3.4 解码流程

AU 方式解码流程如图 3-2 所示。



图3-2 AU 方式解码流程图



### 3.3.5 参考代码



#### 说明

AU 方式解码的解码器初始化以及打开码流文件的过程与流式解码完全相同。AU 方式解码流程加入识别 AU 的过程。

AU 方式解码参考代码如下：

```
#define BYTE_LEN 0x98000 /*解码码流缓冲区，至少能缓冲一帧码流*/
int main(int argc, char** argv)
{
```



```

HI_U8  buf[BYTE_LEN];                                /* 码流缓冲区 */
H264_DEC_ATTR_S dec_attribute;
H264_DEC_FRAME_S dec_frame;
HI_HDL handle = NULL;
FILE * h264= NULL;                                    /* 输入码流文件 */
FILE * yuv = NULL;                                    /* 输出yuv文件 */
HI_U8 * pBuf;
HI_S32 result = 0;
HI_S32 ReturnValue, count;

/* 解码时间和帧数 */
LARGE_INTEGER lpFrequency;
LARGE_INTEGER t1;
LARGE_INTEGER t2;
HI_U32 time;
HI_U32 pic_cnt = 0;
HI_U32 iStreamLen = 0;
HI_U32 end;

/* 分析码流上下文结构体 */
ParseContext PC;
PC.FrameStartFound          = 0;
PC.iFrameLength             = 0;
PC.PrevFirstMBAddr          = 0;

if (argc < 2)
{
    fprintf(stderr, "error comand format or no H.264 stream!\n");
    fprintf(stderr, "the Example: hi_264sample stream_file.264
[yuvfile]\n");
    goto exitmain;
}
else
{
    /* 打开输入码流文件 */
    h264 = fopen(argv[1], "rb");
    if (NULL == h264)
    {
        fprintf(stderr, "Unable to open a h264 stream file %s \n", argv[1]);
        goto exitmain;
    }
    printf("decode file: %s...\n", argv[1]);

```



```
if (argc > 2)
{
    /* 打开输出yuv文件 */
    yuv = fopen(argv[2], "wb");
    if (NULL == yuv)
    {
        fprintf(stderr, "Unable to open the file to save yuv %s.\n",
            argv[2]);
        goto exitmain;
    }
}

printf("save yuv file: %s...\n", argv[2]);
}

/* 初始化解码器属性参数 */
dec_attribute.uBufNum = 16; /* 解码器最大参考帧数: 16 */

/* 解码器最大图像宽高, D1图像(720x576) */
dec_attribute.uPicHeightInMB = 36;
dec_attribute.uPicWidthInMB = 45;

/* 没有用户数据 */
dec_attribute.pUserData = NULL;

/* 以 "00 00 01" 或 "00 00 00 01" 开始的码流 */
dec_attribute.uStreamInType = 0x00;
/* bit4 = 1: 启动内部Deinterlace; bit4 = 0: 不启动内部Deinterlace */
dec_attribute.uWorkMode = 0x10;

/* 创建解码器 */
handle = Hi264DecCreate(&dec_attribute);
if (NULL == handle)
{
    goto exitmain;
}

/* 统计解码时间: 开始计时 */
QueryPerformanceFrequency(&lpFrequency);
QueryPerformanceCounter(&t1);
count = 0;
end = 0;
```



```

/* 解码流程 */
while ( !end )
{
    /* 将码流从文件读入到缓冲区 */
    count = fread( buf + iStreamLen, sizeof(char), (BYTE_LEN - iStreamLen),
        h264);
    pBuf = buf;
    end = (count == 0);          /* end == 1: end of file */
    iStreamLen += count;
    do
    {
        ReturnValue = Hi264DecLoadAU( pBuf, iStreamLen, &PC );
        if ( ReturnValue == 0 || end )          /*获取一帧码流*/
        {
            result = Hi264DecAU( handle, pBuf, PC.iFrameLength, 0 ,
                &dec_frame, 0 );

            if ( HI_H264DEC_OK == result )          /* 获得一帧图像 */
            {
                const HI_U8 *pY = dec_frame.pY;
                const HI_U8 *pU = dec_frame.pU;
                const HI_U8 *pV = dec_frame.pV;
                HI_U32 width  = dec_frame.uWidth;
                HI_U32 height = dec_frame.uHeight;
                HI_U32 yStride = dec_frame.uYStride;
                HI_U32 uvStride = dec_frame.uUVStride;

                fwrite(pY, 1, height* yStride, yuv);
                fwrite(pU, 1, height* uvStride/2, yuv);
                fwrite(pV, 1, height* uvStride/2, yuv);
            }
            pBuf += PC.iFrameLength;          /* 码流指针指向下一帧*/
            iStreamLen -= PC.iFrameLength;    /*计算剩余码流长度*/
        }
    }while ( ReturnValue == 0 );

    if ( ReturnValue == -1 && !end )          /* 无法从缓冲区获取一帧码流*/
    {
        /* 将剩余码流拷贝到缓冲区起始位置 */
        memmove(buf, pBuf, iStreamLen );
    }
}

```



```
/* 解码结束, 统计解码时间和速度 */
QueryPerformanceCounter(&t2);
time=(HI_U32)((t2.QuadPart-t1.QuadPart)*1000000/lpFrequency.QuadPart);

printf("time= %d us\n", time);
printf("%d frames\n",pic_cnt);
printf("fps: %d\n", pic_cnt*1000000/(time+1));

/* 销毁解码器 */
Hi264DecDestroy(handle);

exitmain:
if (NULL != h264)
{
    fclose(h264);
}

if (NULL != yuv)
{
    fclose(yuv);
}

return 0;
}
```

## 3.4 数据类型与数据结构

### 3.4.1 通用数据类型

在 win32 环境下, API 用到的主要数据类型定义如下:

```
/* 通用数据类型定义 */
typedef unsigned char    HI_U8;
typedef unsigned char    HI_UCHAR;
typedef unsigned short    HI_U16;
typedef unsigned int      HI_U32;
typedef signed char       HI_S8;
typedef signed short      HI_S16;
typedef signed int        HI_S32;
typedef __int64           HI_S64;
typedef unsigned __int64  HI_U64;
typedef char              HI_CHAR;
typedef char*             HI_PCHAR;
```





```
typedef void* HI_HDL;
```

### 3.4.2 ParseContext

#### 【说明】

扫描码流过程中保存的上下文信息。

#### 【定义】

```
typedef struct ParseContext{  
    HI_U32  FrameStartFound;    /* 表示是否从码流缓冲区中找到AU起始地址。  
                                0: 找到AU起始地址; 1:未找到AU起始地址。*/  
  
    HI_U32  iFrameLength;      /* 一幅完整图像对应的码流长度。*/  
  
    HI_U32  PrevFirstMBAAddr;  /* 记录上一个Slice的起始宏块地址 */  
} ParseContext;
```

#### 【注意事项】

无。



# A 缩略语

## A

**ASO** Arbitrary Slice Order 任意片顺序

**AU** Access Unit 接入单元

## F

**FMO** Flexible Macroblock Order 灵活宏块顺序

## I

**IDR** Instantaneous Decoding Refresh 即时解码刷新

## M

**MTU** Maximum Transmission Unit 最大传输单元

## N

**NAL** Network Abstraction Layer 网络抽象层

**nalu** Network Abstraction Layer Unit NAL 单元

## R

**RTP** Real-Time Transport Protocol 实时传输协议

## P

**PPS** Picture Parameter Set 图像参数集



A 缩略语

---

S

<b>SEI</b>	Supplemental Enhancement Information	补充增强信息
<b>SPS</b>	Sequence Parameter Set	序列参数集

V

<b>VCL</b>	Video Coding Layer	视频编码层
------------	--------------------	-------