

H.264 PC Decoding Library Software

Development Guide

Issue 05

Date 2008-11-30

Part Number N/A

HiSilicon Technologies CO., LIMITED. provides customers with comprehensive technical support and service. Please feel free to contact our local office or company headquarters.

HiSilicon Technologies CO., LIMITED.

Address: Manufacture Center of Huawei Electric, Huawei Base,

Bantian, Longgang District, Shenzhen, 518129, People's Republic of China

Website: http://www.hisilicon.com

Tel: +86-755-28788858 Fax: +86-755-28357515

Email: support@hisilicon.com

Copyright © HiSilicon Technologies CO., LIMITED. 2008. All rights reserved.

No part of this document may be reproduced or transmitted in any form or by any means without prior written consent of HiSilicon Technologies CO., LIMITED.

Trademarks and Permissions

(IMITED.), and other Hisilicon icons are trademarks of HiSilicon Technologies CO.,

All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute the warranty of any kind, express or implied.

i

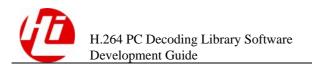
Contents

About This Document	1
1 Overview	1-1
2 Stream Identification and Transmission	2-1
2.1 Overview of the H.264 Basic Syntax Structures	2-2
2.2 Overview of NALU	2-2
2.3 Identification of IDR Frames	2-3
2.4 Identification of Frame Borders	2-4
2.4.1 Overview	2-4
2.4.2 Frame Border Identification Implemented Through the Negotiation of Decoder and Encoder .	2-4
2.4.3 Highly Efficient Method of Frame Border Identification	2-4
2.5 Switch of Decoding Video Channels	2-5
2.6 Network Transmission Mode of the Stream Media	2-5
2.7 Packet Discard Strategy Before Decoding.	2-6
3 Decoding Examples	3-1
3.1 Stream Decoding	3-2
3.1.1 Hi264DecFrame	3-2
3.1.2 Decoding Flow	3-2
3.1.3 Reference Codes	3-3
3.2 NALU Decoding	3-7
3.3 AU Decoding	3-8
3.3.1 Overview	3-8
3.3.2 Hi264DecLoadAU	3-8
3.3.3 Source Codes of Hi264DecLoadAU	3-10
3.3.4 Decoding Flow	3-13
3.3.5 Reference Codes	3-13
3.4 Data Types and Data Structures	3-17
3.4.1 Common Data Types	3-17
3.4.2 ParseContext	3-18
A Acronyms and Abbreviations	A_1

Figures

Figure 3-1 Stream decoding flow	3-3
Figure 3-2 AU decoding flow	3-13

Tables



About This Document

Purpose

This document describes the document contents, related product versions, intended audience, conventions and update history.

Related Versions

The following table lists the product versions related to this document.

Product Name	Version
Hi3510 Communications Media Processor	V100
Hi3511 H.264 Encoding and Decoding Processor	V100
Hi3512 H.264 Encoding and Decoding Processor	V100

Intended Audience

This document is intended for:

- Software development engineers
- Technical support engineers

Organization

This document is organized as follows:

Chapter	Description	
1 Overview	Describes the basic attributes of the H.264 video encoding and decoding standard.	
2 Stream Identification and Transmission	Describes the H.264 stream structures, IDR frame identification, and AU identification.	

Chapter	Description
3 Decoding Examples	Provides examples of the stream decoding, NALU decoding, and AU decoding.
Appendix A Abbreviations	Lists the acronyms and abbreviations used in this document.

Conventions

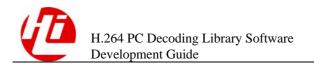
Symbol Conventions

The following symbols may be found in this document. They are defined as follows.

Symbol	Description
DANGER	Indicates a hazard with a high level of risk which, if not avoided, will result in death or serious injury.
WARNING Indicates a hazard with a medium or low level of risk not avoided, could result in minor or moderate injury.	
CAUTION Indicates a potentially hazardous situation that, if no avoided, could cause equipment or component dam loss, and performance degradation, or unexpected relative to the country of the country	
©—¹ TIP	Indicates a tip that may help you solve a problem or save time.
NOTE	Provides additional information to emphasize or supplement important points of the main text.

General Conventions

Convention	Description	
Times New Roman	Normal paragraphs are in Times New Roman.	
Boldface	Names of files, directories, folders, and users are in boldface . For example, log in as user root .	
Italic	Book titles are in <i>italics</i> .	
Courier New	Terminal display is in Courier New.	



Update History

Updates between document versions are cumulative. Therefore, the latest document version contains all updates made to previous versions.

Updates in Issue 05 (2008-11-10)

The fourth commercial release has the following updates:

Chapter 2 Stream Identification and Transmission

- Information about the Hi3512 is added.
- Descriptions of section 2.6 "Network Transmission Mode of the Stream Media" are changed.
- Descriptions of section 2.7 "Packet Discard Strategy Before Decoding" are changed.

Chapter 3 Decoding Examples

Descriptions of section 3.1.1 "Hi264DecFrame" are changed.

Updates in Issue 04 (2008-04-03)

The third commercial release has the following updates:

Chapter 2 Stream Identification and Transmission

Descriptions in section 2.5 " Switch of Decoding Video Channels" are changed.

Chapter 3 Decoding Examples

- Descriptions of enabling the Deinterlace function are added in section 3.1.3 "Reference Codes."
- Descriptions of enabling the Deinterlace function are added in section 3.3.5 "Reference Codes."

Updates in Issue 03 (2008-01-15)

The second commercial release has the following updates:

Chapter 3 Decoding Examples

- Source codes of the Hi264DecLoadAU function are changed in section 3.3.3 "Source Codes of Hi264DecLoadAU."
- Reference codes for AU decoding are changed in section 3.3.5 "Reference Codes."

Updates in Issue 02 (2007-11-20)

The initial commercial release has the following updates:

Chapter 3 Decoding Examples

Paramter "PrevFirstMBAddr" is added and decriptions of section 3.3.2 "Hi264DecLoadAU" are changed.

Source codes of the Hi264DecLoadAU function are changed in section 3.3.3 "Source Codes of Hi264DecLoadAU."

Reference codes for AU decoding are changed in section 3.3.5 "Reference Codes."

Parameter "PrevFirstMBAddr" is added in section 3.4.2 "ParseContext."

Updates in Issue 01 (2007-09-30)

Initial release.

1 Overview

Compared with the previous video compression standards, the H.264 video compression standard (hereinafter referred to as the H.264) has better performance. Therefore, the H.264 is regarded as a new generation of video compression standard.

Compared with the H.263 or the MPEG-4, the H.264 has the following features:

- The intra coding and the inter prediction are more precise so that the residual data is reduced effectively.
- The new arithmetic encoding is introduced so that the data compression ratio becomes higher.
- The video data delamination is more reasonable and the newly introduced network abstraction layer (NAL) is more suitable for the network transmission.
- The traditional frame structure is abolished, and the slice structure as well as parameter SEIs are introduced. As a result, the anti error code capacity is improved.
- The flexible management mechanism of reference frames is introduced. The maximum count of reference frames is 16.

Because of the features, the H.264 performance improves a lot in terms of video signal-to-noise ratio, picture quality and application flexibility. However, the implementation of the H.264 becomes more complicated.

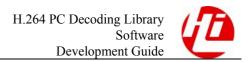
The HiSilicon H.264 PC decoding library software has better performance and reliability and provides easier APIs, which can greatly speed up the H.264 application development.

2 Stream Identification and Transmission

About This Chapter

The following table lists the contents of this chapter.

Section	Describes	
2.1 Overview of the H.264 Basic Syntax Structures	The basic syntax structures of the H.264.	
2.2 Overview of NALU	The information related to NALU.	
2.3 Identification of IDR Frames	The method of identifying IDR frames.	
2.4 Identification of Frame Borders	The method of identifying frame borders.	
2.5 Switch of Decoding Video Channels	The information related to the decoding video channel switch.	
2.6 Network Transmission Mode of the Stream Media	The network transmission modes of the stream media.	
2.7 Packet Discard Strategy Before Decoding	The -packet discard strategy before decoding.	



2.1 Overview of the H.264 Basic Syntax Structures

Compared with the previous video compression and encoding standards, the H.264 is modified a lot in terms of syntax. The most significant modifications are:

- Abolishing the syntax unit of the frame level.
 - There is no syntax unit such as frame_header in the H.264 syntax. All the frame information is saved in slice_header, sequence parameter set (SPS), and picture parameter set (PPS). As a result, the slice unit decoding becomes more independent, and the anti packet loss capacity as well as the anti error code capacity are improved. A frame of picture can correspond to multiple slices, so the decoder cannot identify a frame of data in the stream by parsing the syntax unit such as frame_header.
- Introducing parameter set definitions such as SPS and PPS.
 - Common features of all the pictures in a video sequence (data between an instantaneous decoding refresh (IDR) frame and the next IDR frame) are abstracted and saved in the SPS syntax unit.
 - Typical features of all the pictures are abstracted and saved in the PPS syntax unit.
 - SPS can be switched only between video sequences, that is, only the first slice of the IDR frame can have the SPS switch.
 - PPS can be switched only between video sequences, that is, only the first slice of the IDR frame can have the PPS switch.

The typical H.264 stream structure consists of SPS, PPS, IDR frames (including one or more I-Slices), P frames (including one or more P-Slices), and B frames (including one or more B-Slices). To make the private information transmission easier, the H.264 defines the SEI syntax structure besides the above typical syntax structures. The decoder usually does not parse SEI packets unless the encoder and the decoder have a specific syntax agreement.

2.2 Overview of NALU

Network abstraction layer unit (NALU) is the topmost abstraction layer of the H.264. All the H.264 syntax structures are ultimately encapsulated in a NALU. NALUs must be separated with separators in the data stream; otherwise, the NALUs cannot be identified. The prefix code 00 00 01 is the NALU separator defined in the H.264 video compression standard in appendix B. An NALU can be identified by searching for the prefix code 00 00 01.

NALU has its own syntax unit, which occupies only one byte. Except the first byte behind the prefix code 00 00 01, all other loads in a NALU are the payload of the H.264 syntax structure.

nalu_type is the most important NALU syntax element. It indicates the type of the H.264 syntax structure encapsulated in a NALU. nalu_type can be parsed as follows:

```
nalu type = first byte in nal & 0x1F
```

The parameter first_byte_in_nal represents the first byte in a NALU.

Table 2-1 shows the mapping relationship between nalu_type and H.264 syntax structures.

Accessing unit delimiter

(frame border).

nalu_type	Type of the H.264 Syntax Structure	Description
01	P-Slice B-Slice I-Slice	A P frame (B frame) may include an I slice. But nalu_type of the I slice is different from that of the I slice in the IDR frame.
05	I-Slice	An IDR frame can be divided into several I slices, and each I slice corresponds to a NALU.
06	SEI	Every SEI has a corresponding NALU.
07	SPS	Every SPS has a corresponding NALU. Up to 32 different SPSs exist.
08	PPS	Every PPS has a corresponding NALU. Up to 256 different PPSs exist.

Table 2-1 Mapping relationship between nalu_type and H.264 syntax structures

2.3 Identification of IDR Frames

09

The decoder starts decoding only from the IDR frame. In order to implement functions such as fast forward, fast backward, and decoding channel switch, the decoder needs to identify IDR frames. H.264 protocol dose not define frame syntax structures, but the IDR frame can be identified through nalu_type. If you search for and abstract several continuous NALUs (nalu_type is 05), you can obtain a complete IDR frame. For details, see section 2.2 "Overview of NALU."



CAUTION

AU delimiter

An IDR frame can be divided into several I slices. Every I slice has a corresponding NALU. In other words, an IDR frame can correspond to multiple NALUs. When abstracting an IDR frame, ensure that it is integrated.

The method of searching for SPS and PPS is the same to that of the IDR frame. But every SPS or PPS corresponds to only one NALU.



2.4 Identification of Frame Borders

2.4.1 Overview

The H.264 regards all the NALUs constituting a frame of picture as an AU. The frame border identification is implemented by identifying Access Units (AUs). The frame level syntax is abolished in the H.264 protocol, so AUs cannot be obtained from the stream. During decoding, the decoder judges the end of a frame through the combination of some syntax elements. Generally, after decoding the first slice_header of a new frame, the decoder knows the previous frame has ended. Therefore, the most precise AU identification procedure is as follows:

- **Step 1** Implement the processing of removing 03 to the stream.
- **Step 2** Parse the NALU syntax.
- **Step 3** Parse the syntax of slice header.
- **Step 4** Check two continuous NALUs and the syntax elements in the corresponding slice_header and see whether there are changes. If there are changes, it indicates that the two NALUs belong to different frames; otherwise, the two NALUs belong to the same frame.

----End

Obviously, AU identification before decoding cost much CPU resource. Therefore, the AU decoding is not recommended.

2.4.2 Frame Border Identification Implemented Through the Negotiation of Decoder and Encoder

To provide a simpler AU identification solution, H.264 defines a NALU of the 09 type. For details, see Table 2-1. In other words, the decoder inserts a NALU of the 09 type in the stream each time after the AU encoding. To obtain an AU, the decoder needs only to search for a NALU of the 09 type.

MOTE

The H.264 does not require the encoder to insert NALUs of the 09 type, so not all segments of stream have the feature. If the encoder and the decoder work together, it is recommended to insert NALUs of the 09 type. As a result, the cost of AU identification can be reduced.

2.4.3 Highly Efficient Method of Frame Border Identification

To reduce the cost of AU identification, the document puts forward a highly efficient AU identification method. The method is based on the feature that the syntax element first_mb_in_slice of the first slice_header in a frame is 0. The method does not work if the stream has the arbitrary slice order (ASO) or the flexible macroblock order (FMO) feature.

The highly efficient method of the frame border identification refers to searching for a segment of linear, continuous stream in the buffer and returning the position of the frame border after getting the frame border. No copying operation is performed. For details, see section 3.3 "AU Decoding."

2.5 Switch of Decoding Video Channels

Generally, searching for IDR frames is not enough during the video request service or the video channel switch. The corresponding SPS and PPS must be sent to the decoder. It is unnecessary that the SPS, PPS, and IDR frames are transmitted together. However, the H.264 requires that the corresponding SPS and PPS must be transmitted before IDR frames are transmitted. Therefore, the best way is to search from the start position of the video stream. All the obtained SPSs and PPSs are sent to the decoder in order till a target IDR frame is found.

To reduce the application complexity, the encoder generally transmits SPS or PPS and IDR frame continually, that is, the SPS and PPS parameters used in the current video sequence are transmitted directly before the IDR frame. The Hi351X encoder works in that way, so four smaller NALUs can be found before the first NALU of an IDR frame. They are the parameters required in the current video sequence decoding. The first NALU is SPS. For the Hi351X stream, the SPS must be found. After receiving a segment of stream starting from SPS, the decoder can decode a complete video picture.

If the video input channel switch occurs repeatedly, the channels of the input video sources should be changed constantly. According to the preceding integrity principles of decoding, the switch operation does not occur from any point. In other words, if a decoder should be switched to a video channel, the switch operation must be started from the SPS where the video channel is obtained. Otherwise, the video images may be abnormal at the switch point.



Hi351X refers to Hi3510, Hi3511, or Hi3512.

2.6 Network Transmission Mode of the Stream Media

There are the following two network transmission solutions based on the MPEG-4 solution:

- The data segment with fixed length whose size is smaller than the network layer maximum transmission unit (MTU) is used to divide the raw linear stream.
- During the direct transmission, a complete frame of data is added in the private format header. The network layer divides the whole frame of data, and the application layer ignores the packets lost during the network congestion.

Compared with the MPEG-4, the H.264 puts forward better anti error code solution used in the network transmission. The H.264 consists of video coding layer (VCL) and NAL. Features of VCL and NAL are as follows:

- VCL contains the core compression engine and the block/macro block/slice level syntax definition. VCL should be independent from the network.
- NAL adapts the bit strings generated by VCL to multiple different network environments.
 NAL covers all the syntax levels above the slice level, including the following mechanisms:
 - Providing parameters required in the slice decoding.
 - Preventing the start code confliction.
 - Supporting supplemental enhancement information (SEI).
 - Transmitting the encoding slice bit strings on the network based on the bit streams.

The reasons why the H.264 divides NAL and VCL are as follows:



- An interface for VCL signal processing and NAL transmission is defined, so VCL and NAL can work on different processing platforms.
- VCL and NAL are designed to adapt the heterogeneous transmission environment. When
 the network environment changes, the gateway need not restructure and recode the VCL
 bit stream.

IP networks are classified into the following three types:

- Uncontrollable IP networks, such as the Internet.
- Controllable IP networks, such as the wide area network.
- Wireless IP networks, such as the 3G network.

The three types of IP networks have different MTUs, bit error ratios, and TCP usage flags. The MTU between two IP nodes are changed dynamically. Generally, the MTU of the cable IP network is presumed to be 1.5 KB, and the MTU of the wireless IP network ranges from 100 bytes to 500 bytes.

It is recommended to use the basic unit NALU during the UDP transmission. As a result, the encapsulation in the application layer is easier. It is recommended to use real-time transport protocol (RTP) for encapsulation and transmission. An RTP contains a NALU. Put the NALU (including the synchronization header) into the RTP loader, and then set the RTP header information. Because packet transmission paths are different, the receiver needs to reorder slice groups. The order information contained in RTP can be used to solve the problem.

Do not cast the UDP groups whose sizes are bigger than that of the MTU. These UDP groups can be sent directly. Disassembly and assembly can be implemented by the network driver layer. The application layer is not affected. If a UDP packet is cast by the network, the packet contains a complete NALU. In the H.264, NALU is the network layer encapsulation based on the VCL slice. Each slice corresponds to multiple macro blocks (different from the MPEG-4). During the H.264 encoding, the macro block syntaxes of different slices are not relevant. Casting a few of slices does not affect the decoding of other slices.

The H.264 application becomes wider. Actually, the MPEG-4 and the H.264 usually exist in the same application solution. To have a stable decoding system and avoid the software crash, it is recommended to transmit packets in the unit of NALU. In that case, the H.264 and the MPEG-4 do not interfere each other in the same system.



CAUTION

- Do not send incomplete NALUs to the decoder; otherwise, the decoder may go abnormal, such as the decoder crash.
- When processing the H.264 stream, users who are familiar with the MPEG-4 application should avoid the operation: divide a frame of linear stream into blocks which are smaller than MTU and transmit packets after adding private headers.

2.7 Packet Discard Strategy Before Decoding

The player casts packets in the following cases:

• Network jitter leads to the PC CPU overload or the overflow of the player front stream buffer.

- The application layer detects that the network packet discard occurs frequently.
- The player scheduling needs the packet discard processing.

The current principles for discarding packets actively are as follows:

- Try to send all the received streams to the decoder unless the streams have to be discarded.
- If the player must actively discard packets in the preceding cases, it should start to discard packets from the current frame till the next IDR frame occurs.



CAUTION

If the player must discard packets actively, ensure that the streams from the starting address of an IDR frame are send to the decoder after the discarding operation is complete. Do not send incomplete data to the decoder.

3 Decoding Examples

About This Chapter

The following table lists the contents of this chapter.

Section	Describes	
3.1 Stream Decoding	The examples of the stream decoding.	
3.2 NALU Decoding	The examples of the NALU decoding.	
3.3 AU Decoding	The examples of the AU decoding.	
3.4 Data Types and Data Structures	The common data types and the ParseContext data structures used in APIs.	

3.1 Stream Decoding

3.1.1 Hi264DecFrame

The decoding library provided by HiSilicon has a uniform Hi264DecFrame function for the stream decoding. For details, see the *H.264 PC Decoding Library Software API Reference*. Hi264DecFrame has the following functions:

- Decoding a stream of any length.
 - Input parameters are pStream (the address of the stream buffer) and iStreamLen (the byte count of the input stream).
 - If the stream is not enough for a frame, more stream data is needed for decoding.
 - If the stream contains several frames, call Hi264DecFrame repeatedly to decode the remaining stream after decoding a frame of picture.

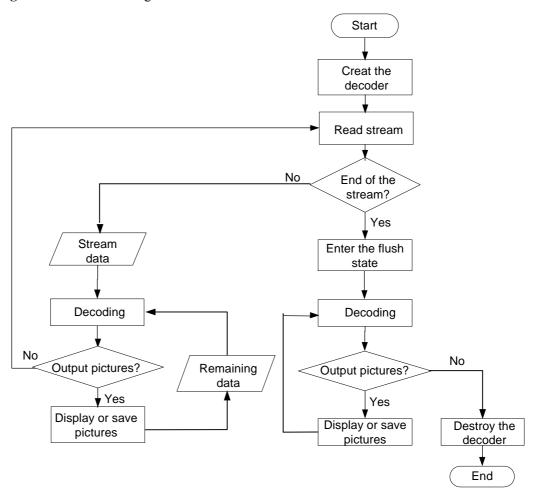
When the stream ends, enter the flush mode to clear the remaining picture in the decoder till no remaining picture exists.

- The decoder provides the time stamp transparent transmission function. The input time stamp ullPTS is transmitted to the output picture structure transparently so that the application layer can control the playing easily.
- The output parameter pDecFrame contains the picture type flag uPicFlag. If field pictures are exported, call de_interlace to combine two fields of pictures into a frame of picture.
- The output parameter pDecFrame contains the current frame error information bError. bError can be used to judge whether the current picture has errors.
- If the stream contains user data, the output parameter pDecFrame contains user data; otherwise, user data is a null pointer.

3.1.2 Decoding Flow

Figure 3-1 shows the stream decoding flow.

Figure 3-1 Stream decoding flow



3.1.3 Reference Codes

The reference codes of stream decoding are as follows:

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

#define STREAM_BUFFER_LEN 0x8000

int main(int argc,char** argv)
{
    HI_S32 end = 0;
    /*Stream buffer.*/
    HI_U8 buf[STREAM_BUFFER_LEN];
    H264_DEC_ATTR_S dec_attrbute;
    H264_DEC_FRAME_S dec_frame;
    HI_HDL handle = NULL;
```

```
/*Decoding time and frame count.*/
LARGE INTEGER lpFrequency;
LARGE INTEGER t1;
LARGE_INTEGER t2;
HI U32 time;
HI_U32 pic_cnt = 0;
FILE *h264 = NULL;
                         /* Input stream file.*/
FILE *h264 = NULL;
                          /* Output YUV file.*/
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
   /*Open the input stream file.*/
   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)
   /*Open the output YUV file.*/
   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]);
}
/*Initialize the attribute parameters of the decoder.*/
/*The maximum reference frame count of the decoder: 16.*/
dec_attrbute.uBufNum = 16;
```

}

```
/*The maximum picture width and height supported by the decoder, D1 picture
    (720 x 576).*/
   dec attrbute.uPicHeightInMB = 36;
   dec attrbute.uPicWidthInMB = 45;
   /*No user data.*/
   dec attrbute.pUserData = NULL;
   /*The stream started with 00 00 01 or 00 00 00 01.*/
   dec attrbute.uStreamInType = 0x00;
   /*bit0 = 1: standard output mode; bit0 = 0: quick output mode.*/
   /* bit4 = 1: enable the internal Deinterlace function; bit4 = 0: do not
enable the internal Deinterlace function */
   dec attrbute.uWorkMode
                              = 0x10:
   /*Create a decoder.*/
   handle = Hi264DecCreate(&dec_attrbute);
   if (NULL == handle)
      goto exitmain;
   /*Query the decoding time: start timing.*/
   QueryPerformanceFrequency(&lpFrequency);
   QueryPerformanceCounter(&t1);
   /*Decoding process.*/
   while (!end)
      /*Read a segment of stream from the file "h264".*/
      HI U32 len = fread(buf, 1, sizeof(buf), h264);
      /*Enter the flush mode if the end of stream file is reached.*/
      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)
                                                /*No remaining picture
                                                   exists in the decoder.*/
             end = 1;
```

```
break;
      }
      if (HI_H264DEC_OK == result)
                                           /*Output a frame.*/
          if(NULL != yuv)
                                                /*Save the YUV file.*/
             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);
          }
          /*Read and print the user data.*/
          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->uUserDataType,
             dec frame.pUserData->uUserDataSize);
             for(i=0; i<dec frame.pUserData->uUserDataSize; i++)
             printf(" %d ",dec_frame.pUserData->pData[i]);
             printf("\n");
          pic cnt++;
      }
      /*Continue to decode the remaining H.264 stream.*/
      result = Hi264DecFrame(handle, NULL, 0, 0, &dec_frame, flags);
   }
/*Query the decoding time: end timing.*/
QueryPerformanceCounter(&t2);
```

}

```
time=(HI_U32)((t2.QuadPart-t1.QuadPart)*1000000/lpFrequency.QuadPart);

/*Print the calculation information: decoding time, count of the decoded
frames, frame rate in the unit of millisecond.*/
  printf("time= %d us\n", time);
  printf("%d frames\n",pic_cnt);
  printf("fps: %d\n", pic_cnt*1000000/(time+1));

/*Destroy the decoder.*/
  Hi264DecDestroy(handle);

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

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

  return 0;
}
```

3.2 NALU Decoding

The process of the NALU decoding is similar to that of the stream decoding. The difference is that the stream input each time corresponds to a NALU. For details, see section 3.3 "AU Decoding."

During the NALU decoding, pay attention to the following points:

- The input NALU must comply with the specifications in the Appendix B of the H.264 protocol. In other words, the prefix code 00 00 01 or 00 00 00 01 must be put before the NALU. Do not implement the processing of removing 03 to the NALU.
- The input parameter iStreamLen is the byte count of a NALU. iStreamLen must contain the length of the prefix code 00 00 01 or 00 00 00 01.
- The input parameter pStream is the pointer to the input stream buffer. The buffer must be greater than iStreamLen, and no infringement is permitted.
- For the H.264, the end of picture can be identified only before the decoding of the next picture. Therefore, at least a frame of picture exists in the decoder. Before ending a decoding process, configure uflag to 1 and call Hi264DecFrame repeatedly to decode the remaining stream till no picture exists in the decoder.

3.3 AU Decoding

3.3.1 Overview

The AU decoding is mainly used for the video and audio synchronization before decoding. The syntax of traditional video protocols has the frame structure, so it is easy to identify a frame of picture in the stream. But the H.264 abolishes the frame syntax structure. As a result, it is difficult to identify a frame of picture in the stream directly. To reduce the decoding performance loss caused by the frame identification, it is not recommended to use the AU decoding.

This chapter describes the AU decoding based on the highly efficient frame identification solution. The main principle is to introduce the process of identifying a frame before decoding. Except for this point, the decoding process is the same as the stream decoding. For details, see section 3.1 "Stream Decoding."



CAUTION

The frame identification technology has a limitation. When the stream contains FMO or ASO, a frame of picture cannot be identified.

3.3.2 Hi264DecLoadAU

[Purpose]

Find the border of a picture.

[Syntax]

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

[Description]

The function is used to find the border of a complete picture from the input stream.

[Parameter]

Parameter	Member	Value Range	Input/ Output	Description
pc	FrameStartFound	0, 1	Output	The flag indicating whether the AU start address is found from the stream buffer.
				0: The AU start address is found.
				1: The AU start address is not found.
	iFrameLength	-	Output	Length of a segment of stream corresponding to a complete picture.
	PrevFirstMBAddr	-	Output	Start address of the macro block of the previous Slice.
				First mb address of the previous Slice.
pStream	-	-	Input	Start address of the stream buffer.
iStreamLen	-	-	Input	Length of the buffer allocated externally. iStreamLen must be greater than the length of a frame.

[Return Value]

Return Value	Macro Definition	Description
0	None	The AU border is found.
-1	None	The AU border is not found.

[Note]

- The Hi264DecLoadAU function does not belong to the HiSilicon H.264 PC decoding library.
- The Hi264DecLoadAU function can extract the SPS, PPS, or SEI packet separately.
- In the frame mode and the frame and field self-adaptive mode, Hi264DecLoadAU searches for the border of a frame. In the field mode, Hi264DecLoadAU searches for the border of a field.
- For Hi264DecLoadAU, the start address of the input stream buffer points to the end of the previous frame by default. The start position and the end position of the next NALU are determined by the length of the obtained frame.
- The input stream buffer must be able to accommodate as least a complete NALU. If the
 input stream buffer cannot accommodate a complete NALU, Hi264DecLoadAU cannot
 obtain the start position and the end position of the NALU. The configuration rules of the
 input stream buffer are as follows:

- When the picture format is CIF, the recommended capacity of the input stream buffer is 150 KB.
- When the picture format is D1, the recommended capacity of the input stream buffer is 620 KB.
- When the AU of a segment of input stream is found, an incomplete AU may be left in the decoder. The remaining stream should be transferred into a new buffer, and then the new buffer is filled with new stream.

3.3.3 Source Codes of Hi264DecLoadAU

The source codes of Hi264DecLoadAU are as follows:

```
typedef struct ParseContext{
   HI U32 FrameStartFound;
                                   /*Whether the AU start address is found
                                     from the stream buffer.
                                     0: The AU start address is found.
                                     1: The AU start address is not found.*/
                                   /*Length of the stream corresponding to a
   HI_U32 iFrameLength;
                                     complete picture.*/
   HI_U32 PrevFirstMBAddr;
                                   /*First mb address of the previous Slice.*/
} 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;</pre>
       }
   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;</pre>
   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)</pre>
      return -1;
   pc->FrameStartFound = 0;
   pc->PrevFirstMBAddr = 0;
   for(i = 0; i < iStreamLen; i++)
      /*find a I-slice or a P-slice*/
      if( (state & 0xFFFFFF1F) == 0x101 ||
          (state & 0xFFFFFF1F) == 0x105)
          p = &pStream[i];
          FirstMbAddr = CheckFirstMbAddr(p);
          if( 1 == pc->FrameStartFound )
             if( FirstMbAddr <= pc->PrevFirstMBAddr)
                 pc->iFrameLength = i - 4;
                 pc->PrevFirstMBAddr = FirstMbAddr;
                 state = 0xffffffff;
                 return 0;
             else
                 pc->PrevFirstMBAddr = FirstMbAddr;
```

```
else
          pc->PrevFirstMBAddr = FirstMbAddr;
          pc->FrameStartFound = 1;
   }
   /*find a sps, pps or au_delimiter*/
   if( (state&0xFFFFFF1F) == 0x107 \mid \mid
       (state&0xFFFFFF1F) == 0x108 | |
       {
      if(1 == pc->FrameStartFound)
         pc->iFrameLength = i - 4;
          pc->PrevFirstMBAddr = 0;
          state = 0xffffffff;
          return 0;
      else
          pc->FrameStartFound = 1;
          pc->PrevFirstMBAddr = 0;
   }
   state = (state << 8) | pStream[i];</pre>
   if( MAX_AU_SIZE - 1 <= i )</pre>
     pc->iFrameLength = i - 3;
      return 0;
   }
pc->iFrameLength = i;
return -1;
```

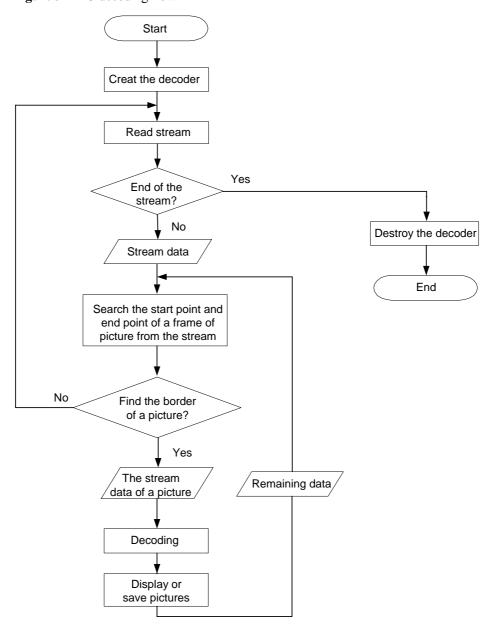
}

}

3.3.4 Decoding Flow

Figure 3-2 shows the AU decoding flow.

Figure 3-2 AU decoding flow



3.3.5 Reference Codes

■ NOTE

The decoder initialization and the process of opening the stream file of the AU decoding are the same to those of the stream decoding. The AU identification process is added into the AU decoding.

The reference codes of AU decoding are as follows:

```
#define BYTE LEN 0x98000
                                    /*The decoding stream buffer contains a
                                    frame of stream at least*/
int main(int argc,char** argv)
   HI_U8 buf[BYTE_LEN];
                                                 /*Stream buffer.*/
   H264_DEC_ATTR_S dec_attrbute;
   H264 DEC FRAME S dec frame;
   HI HDL handle = NULL;
   FILE * h264= NULL;
                                                 /*Input stream file.*/
   FILE * yuv = NULL;
                                                 /*Output YUV file.*/
   HI U8 * pBuf;
   HI_S32 result = 0;
   HI S32 ReturnValue, count;
   /*Decoding time and frame count.*/
   LARGE_INTEGER lpFrequency;
   LARGE INTEGER t1;
   LARGE INTEGER t2;
   HI_U32 time;
   HI_U32 pic_cnt = 0;
   HI U32 iStreamLen = 0;
   HI U32 end;
   /*Parse the stream context structure.*/
   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
      /*Open the input stream file.*/
      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)
          /*Open the output YUV file.*/
          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]);
   }
   /*Initialize the attribute parameters of the decoder*/
   dec attrbute.uBufNum = 16;
                                             /*Maximum reference frame
                                             count of the decoder: 16.*/
   /*Maximum picture width and length supported by the decoder, D1 picture
   (720 \times 576).*/
   dec_attrbute.uPicHeightInMB = 36;
   dec attrbute.uPicWidthInMB = 45;
   /*No user data.*/
   dec_attrbute.pUserData = NULL;
   /*Stream started with "00 00 01" or "00 00 00 01".*/
   dec_attrbute.uStreamInType = 0x00;
   /* bit4 = 1: enable the internal Deinterlace function; bit4 = 0: do not
enable the internal Deinterlace function*/
   dec attrbute.uWorkMode
                               = 0x10;
   /*Create a decoder.*/
   handle = Hi264DecCreate(&dec attrbute);
   if (NULL == handle)
      goto exitmain;
   /*Calculate the decoding time: start timing.*/
   QueryPerformanceFrequency(&lpFrequency);
```

```
QueryPerformanceCounter(&t1);
count = 0;
end = 0;
/*Decoding process.*/
while ( !end )
   /*Read streams from the file to the buffer*/
   count = fread( buf + iStreamLen, sizeof(char), (BYTE_LEN - iStreamLen),
   h264);
   pBuf = buf;
                                      /* end == 1: end of file */
   end = (count == 0);
   iStreamLen += count;
   do
      ReturnValue = Hi264DecLoadAU( pBuf, iStreamLen, &PC );
      if (ReturnValue == 0 | end)
                                            /*Obtain a frame of stream*/
          result = Hi264DecAU( handle, pBuf, PC.iFrameLength, 0 ,
          &dec frame, 0 );
          if (HI H264DEC OK == result)
                                            /*Obtain a frame of stream*/
             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;
                                            /*The stream pointer points
                                             to the next frame/
                                             /*Calculate the lengths of
          iStreamLen -= PC.iFrameLength;
                                             the remaining streams/
      }
   }while ( ReturnValue == 0 );
   if ( ReturnValue == -1 && !end )
                                            /*Fail to obtain a frame of
                                             stream from the buffer*/
```

```
/*Copy the remaining streams to the start position of the buffer*/
      memmove(buf, pBuf, iStreamLen );
   }
 }
/*Calculate the decoding time and rate after the decoding is completed*/
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));
/*Destroy the decoder.*/
Hi264DecDestroy(handle);
exitmain:
if (NULL != h264)
   fclose(h264);
}
if (NULL != yuv)
   fclose(yuv);
return 0;
```

3.4 Data Types and Data Structures

3.4.1 Common Data Types

The major data structures used in the APIs of the win32 environment are as follows:

```
/*Definitions of the common data types.*/
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;
                           HI_HDL;
typedef void*
```

3.4.2 ParseContext

[Description]

The context information that is saved when the stream is scanned.

[Definition]

```
typedef struct ParseContext{
   HI_U32 FrameStartFound;
                                 /*Whether the AU start address is found from
                                  the stream buffer. 0: AU start address is
                                  found; 1: AU start address is not found.*/
                                /*Length of a segment of stream corresponding
   HI_U32 iFrameLength;
                                  to a complete picture.*/
   HI U32 PrevFirstMBAddr;
                                /*Start address of the macro block of the
                                 previous Slice.*/
} ParseContext;
[Note]
```

None.

A

Acronyms and Abbreviations

A

ASO Arbitrary Slice Order

AU Access Unit

F

FMO Flexible Macroblock Order

I

IDR Instantaneous Decoding Refresh

M

MTU Maximum Transmission Unit

N

NAL Network Abstraction Layer

NALU Network Abstraction Layer Unit

 \mathbf{R}

RTP Real-Time Transport Protocol

P

PPS Picture Parameter Set

 \mathbf{S}

SEI Supplemental Enhancement Information

SPS Sequence Parameter Set

V

VCL Video Coding Layer