SDK6 AN: USB Host UVC

Version 1.0

May 18, 2015

Confidentiality Notice:

US
3101 Jay Street
Ste.110
Santa Clara, CA 95054, USA
Phone: +1.408.734.8888
Fax: +1.408.734.0788

Taiwan
Suite C1, No. 1, Li-Hsin Road 1
Science-Based Industrial Park
Hsinchu 30078, Taiwan
Phone: +886.3.666.8828
Fax: +886.3.666.1282

Hong Kong
Unit A&B, 18/F, Spectrum Tower
53 Hung To Road
Kwun Tong, Kowloon
Phone: +85.2.2806.8711
Fax: +85.2.2806.8722

Japan - Yokohama
Shin-Yokohama Business Center Bldg. 5th Floor
3-2-6 Shin-Yokohama, Kohoku-ku,
Yokohama, Kanagawa, 222-0033, Japan
Phone: +81.45.548.6150
Fax: +81.45.548.6151

Korea
6 Floor, Hanwon-Bldg.
Sunae-Dong, 6-1, Bundang-Gu
SeongNam-City, Kyunggi-Do
Republic of Korea 463-825
Phone: +031.717.2780
Fax: +031.717.2782

China - Shenzhen
Unit E, 5th Floor
No. 2 Finance Base
8 Ke Fa Road
Shenzhen, 518057, China
Phone: +86.755.3301.0366
Fax: +86.755.3301.0966

China - Shanghai
9th Floor, Park Center
1088 Fangdian Road, Pudong New District
Shanghai 201204, China
Phone: +86.21.6088.0608
Fax: +86.21.6088.0366

I Contents

II Preface

This document provides technical details using a set of consistent typographical conventions to help the user differentiate key concepts at a glance.

Conventions include:

Example	Description
AmbaGuiGen, **Chameleon** **Save, File > Save** **Power, Reset, Home**	Software names GUI commands and command sequences Computer keys (+ simultaneous) Hardware buttons
GPIO81, **CLK_AU**	Hardware external pins
VIL, VIH, VOL, VOH	Hardware pin parameters
amb_performance_t ***amb_operating_mode_t*** **amb_set_operating_mode()**	API details (e.g., functions, structures, and type definitions)

Table II-1. Typographical Conventions for Technical Documents.

Command Prompts:

The following is a list of the typographical conventions used in this document. Commands are preceded by prompts that indicate the environment where the command is to be typed.

Indicate the command to be typed in the Linux workstation that is used to compile A5s Flexible Linux SDK.

```
build $
```

Indicate the command to be typed in the Linux machine that is connected to A5s Flexible Linux platform.

```
host $
```

Indicate the command to be typed in the console window connected to the A5s Flexible Linux platform.

```
a5s #
```

1 Overview

1.1 Overview: Introduction

This document describes how to implement the USB Video Class (UVC) in the application layer by using APIs provided by the USB kernel stack.

1.2 Introduction: Scope of Document

This document includes UVC flow, behavior and issue in the USB unit test. The cooperation of the UVC with other functions (eg. video muxer) and CPU/ memory resource distribution are beyond the scope of this document.

1.3 Introduction: Required Tools

In the UVC unit test, data payloads are received and stored in the SD card. Data format depends on the support list in web camera's USB descriptor. To verify received data payloads, the user uses the free software tools introducedbelow:

- (Section 1.3.1) Tools: YUV Player
- (Section 1.3.2) Tools: SM Player

1.3.1 Tools: YUV Player

When the format of data payload is YUV, the user can parse it with the YUV player. With the frame size and color space given, thess tools can display the image frame by frame.

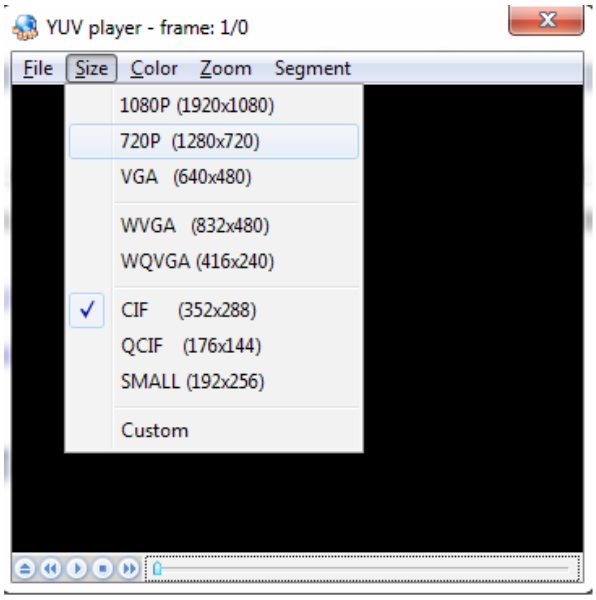

Figure 1-1. YUV Player.

1.3.2 Tools: SM Player

When the format of data payload is MPEG, the user can parse it with the SM player. Note that the file extension of data payloads should be changed to '.mpeg'.

Figure 1-2. SM Player.

1.4 Introduction: Web Camera

In this document, the user uses Logitech HD Pro C920 as the USB device web camera.

Figure 1-3. Logitech HD Pro C920 Web Camera.

2 Streaming Start

2.1 Streaming Start: Overview

This chapter provides the USB video class definition and standardizes video streaming functionality on the USB. This document explains how the USB video class works in the unit test of SDK6. People who design the application software shall reference this document and change the flow and resource distribution according to their requirements.
In the current SDK6, USB Host stack supports isochronous transfer, which is used in most web cameras (UVC with bulk transfer is beyond the scope of this document). Data transferring in isochronous endpoint has guaranteed bandwidth (maximum payload is 3072 bytes/125us) and bounded latency. However, there is no re-try mechanism for a delivery failure in the isochronous transfer.

In this chapter, the flow starting from the enumeration of the web camera to the start of streaming is explained as follows.

- (Section 2.2) Streaming Start: Enumeration
- (Section 2.3) Streaming Start: Probe and Commit
- (Section 2.4) Streaming Start: AmbaUSBH_Uvc_StreamingStart
- (Section 2.5) Streaming Start: AmbaUSBH_Uvc_StreamingStop
- (Section 2.6) Streaming Start: Change User Control
- (Section 2.7) Streaming Start: Summary

2.2 Streaming Start: Enumeration

After the USB stack is initialized and UVC class starts, USB begins enumeration between host and device once web camera is connected. All media information is obtained during the enumeration, including the supported data format, frame size, packet bandwidth, frame rate and user control properties.

2.2.1 Eumeration: AppUsbh_ClassInit

The application can call “AppUsbh_ClassInit(AMBA_USB_HOST_CLASS_VIDEO)” to start the USB stack and UVC class with or without the camera device being connected. When the camera is connected, the device information is given through the enumeration. The application can use APIs in the sections below to get specified information.

```
a:\> t usb init_host uvc
[00014033][CA9] [USBX] USBX Host version : 2015 Ambarella USBX Host stack
[00014033][CA9] [USBX] OHCI: Disabled
a:\> [00014234][CA9]
[00014077][USB] [EHCI] Connect Status Change. 0x1803

[00014277][CA9] [USBH] port 0 insertion detected.
[00014277][CA9] [USBH] ENABLE PORT [0] RESULT 0x0.
[00014531][CA9] [USBH] The 0 times RESET PORT 0, Result: 0x0.
[00014849][CA9] [USBH] VID 0x46D PID 0x821
[00014849][CA9] [USBH] Class 0xEF SubClass 0x2 Protocol 0x1
[00015302][CA9] _uvch_control_thread_entry(): insance = 0xB2CFB790
[00015380][CA9] Unsupported descriptor subtype = 0xd
[00015380][CA9] Unsupported descriptor subtype = 0xd
[00015396][CA9] uvch_device_change_func(): state = live, instance = 0xB2CFB790
[00015396][CA9] [UVCH] Activated
```

Figure 2-1. Console Log when USB Stack Initializes with a Connected Camera Device.

2.2.2 Eumeration: AmbaUSBH_Uvc_RegisterCallback

Application can register the callback for the camera insertion/removal events by the API **AmbaUSBH_Uvc_RegisterCallback**.

2.2.3 Eumeration: AmbaUSBH_Uvc_GetDeviceInfo

Application can get supported format and frame information by API **AmbaUSBH_Uvc_GetDeviceInfo**.

Example:

```
UVCH_DEVICE_INFO dev_info;
        dev_info.size = sizeof(UVCH_DEVICE_INFO);
        nRet = AmbaUSBH_Uvc_GetDeviceInfo(&dev_info);
   ….
```

By using **AmbaUSBH_Uvc_GetDeviceInfo**, the application can compose a support list for the connected camera. The frame rate (fps) can be calculated by given FrameInterval. For example, if the FrameInterval is 0x51615, which mean 33333300 (ns). The frame rate should be 1000000000/33333300 = 30 (fps).

2.2.4 Eumeration: AmbaUSBH_Uvc_GetPuControlInfo

Application can get the information of user control property by the API **AmbaUSBH_Uvc_GetPuControlInfo**.

Example:

```
UVCH_CONTROL_ITEM item;
                    UINT8 max_buffer[20];
                    UINT8 min_buffer[20];
                    UINT8 def_buffer[20];
                    UINT8 cur_buffer[20];
                    item.max = max_buffer;
                    item.min = min_buffer;
                    item.def = def_buffer;
                    item.cur = cur_buffer;
                    item.buffer_size = 20;
        AmbaUSBH_Uvc_GetPuControlInfo(&item, UVC_PU_BRIGHTNESS_CONTROL);
```

The “size” field decides the size of data buffers. For example, if the “item.size” is 2, data in xxx_buffer is a word in little endian order (xxx could be max/min/def/cur). If “item.size” is 4, data in xxx_buffer is a double-word in little endian order.

2.3 Streaming Start: Probe and Commit

Before starting streaming, the USB host needs to do a series “probe and commit” to confirm the data format and frame information. The application should issue API **AmbaUSBH_Uvc_ProbeAndCommit**”\ for this purpose. The “fomatIndex” and “frameIndex” should be in the support list created by the API **AmbaUSBH_Uvc_GetDeviceInfo**. The “fps” is frame rate in frame per second, which should also be in the support list (Section 2.2).

2.4 Streaming Start: AmbaUSBH_Uvc_StreamingStart

The application can request the camera to start streaming through the API **AmbaUSBH_Uvc_StreamingStart**. The application could take note that “Probe and Commit (see Section 2.3)” should be done before starting streaming.

2.5 Streaming Start: AmbaUSBH_Uvc_StreamingStop

The application can request the camera to stop streaming using the API **AmbaUSBH_Uvc_StreamingStop**.

2.6 Streaming Start: Change User Control

The application can change the property of the user control (see Section 2.2.4) through the API **AmbaUSBH_Uvc_SetCurValues**.

2.7 Streaming Start: Summary

- The application should register the callback functions of the insertion/removal event before starting the UVC class.
- The application should start the USB stack and the UVC class with or without the camera being connected.
- The application should compose the support list by the APIs in Sections 2.2.3 and 2.2.4, which includes the supported streaming format, frame information in each format and the user control property.
- The application can decide the streaming properties which are in the supported list by “probe and commit” in Section 2.3.
- After “probe and commit”, the application can start streaming by the API in Section 2.4.
- After streaming starts, the application can stop the streaming by the API in Section 2.5.
- After streaming starts, the application can change the user control properties by the API in Section 2.6.

3 Stream Handling

3.1 Stream Handling: Overview

After the stream starts, the application can make a transfer request to ask the USB stack to receive the data at any time with appointed size. Before the user starts, the application designer should know the characteristic of isochronous transfer.

Data in isochronous transfer is transferred periodically. Normally, the interval between each packet is 125 us.

The bandwidth of isochronous transfer is guaranteed. The maximum bandwidth of high-speed isochronous transfer is 3072 bytes in 125us. However, the actual bandwidth depends on the connected camera.

The USB stack supports up to 2000 packets in each request of data reception, which takes 250ms (2000x125us=250ms).

Corrupted data in isochronous transfer cannot be recovered since there's no re-try mechanism.
In this chapter, we will introduce how the transfer request works. Then, the user will explain the flow of stream handling in the unit test. The application designer can modify it according to different requirements.

3.2 Stream Handling: Transfer Request

The USB stack provides some APIs for the application to make the transfer request according to different conditions. There are also APIs to help application decide the correct parameters of the transfer request.

3.2.1 Transfer Request: AmbaUSBH_Uvc_ReadBlock

The application can make a transfer request by the API **AmbaUSBH_Uvc_ReadBlock**.
Before the user calls the API **AmbaUSBH_Uvc_ReadBlock**, some jobs need to be done. First, the request array with length of packet number should be allocated and filled with responded values. For example,

```
#define UVCH_MAX_PACKET_NUM  2000
#define UVCH_MAX_PACKET_SIZE  3072
#define UVCH_REQUEST_ARRAY_SIZE  UVCH_MAX_PACKET_NUM*sizeof(UX_EHCI_ISO_REQUEST)
#define UVCH_READ_BUFF_SIZE  UVCH_MAX_PACKET_SIZE* UVCH_MAX_PACKET_NUM

// Allocate the data buffers and request buffers.
AmbaKAL_MemAllocate(&AmbaBytePool_Cached, &uvch_read_buffer, UVCH_READ_BUFF_SIZE, 32);
AmbaKAL_MemAllocate(&AmbaBytePool_Cached, &uvch_request, UVCH_REQUEST_ARRAY_SIZE, 32);

// Fill the request array with responded data.
UX_EHCI_ISO_REQUEST        *TmpRequest = uvch_request. pMemAlignedBase;
UINT8  * ReadBuf = uvch_read_buffer. pMemAlignedBase;
for (i=0; i< UVCH_MAX_PACKET_NUM; i++) {
   TmpRequest->buf = (UCHAR*)(ReadBuf + UVCH_MAX_PACKET_SIZE *i);
   TmpRequest->request_length = UVCH_MAX_PACKET_SIZE;
   TmpRequest++;
}
```

In this example, maximum packet size is 3072 bytes, which could vary in different stream setup. The application designer can refer to Section 3.1.3 for details.

Second, the complete function should be designed carefully since it's called by the USB stack task. The application should take care of the occupied time and task stack for the complete function. For example,

```
static void uvch_rx_complete(UX_EHCI_ISO_REQUEST *request)
{
        AmbaPrint("receive complete 0x%x", request);
    // Only send the queue message to other task, which handles the parsing job.
        uvch_send_parse_queue_msg(request);
}
```

After above preparations are done, application can call the API **AmbaUSBH_Uvc_ReadBlock** to make a transfer request.

Application should also understand that, although **AmbaUSBH_Uvc_ReadBlock** is not blocked in USB stack (which means the user can call it again immediately), USB stack would return 0x41 (UX_NO_BANDWIDTH_AVAILABLE) if application issues another transfer request during the current transfer period. For example, if the application makes a request with 2000 packets, another request can only be issued after 250ms. Otherwise, it would return a 0x41 error.

There could be a time gap between every transfer request due to the above characteristic, which would cause non-continuous data to be received. To solve this issue, the application can use the API function in Section 3.1.2.

3.2.2 Transfer Request: AmbaUSBH_Uvc_ReadAppend

The API Syntax of **AmbaUSBH_Uvc_ReadAppend** is the same as **AmbaUSBH_Uvc_ReadBlock**.

The only difference is **AmbaUSBH_Uvc_ReadAppend** can be called successively and USB stack would make sure the received data between every transfer request are continuous. USB stack also makes the necessary delay which may block the task.

3.2.3 Transfer Request: AmbaUSBH_Uvc_GetMaxPacketSize

The maximum packet size is changed according to different USB interface. The application can get the current maximum packet size by the API **AmbaUSBH_Uvc_GetMaxPacketSize**.

The application should use the return value as the request length in the request array.

3.3 Stream Handling: Flow in Unit Test

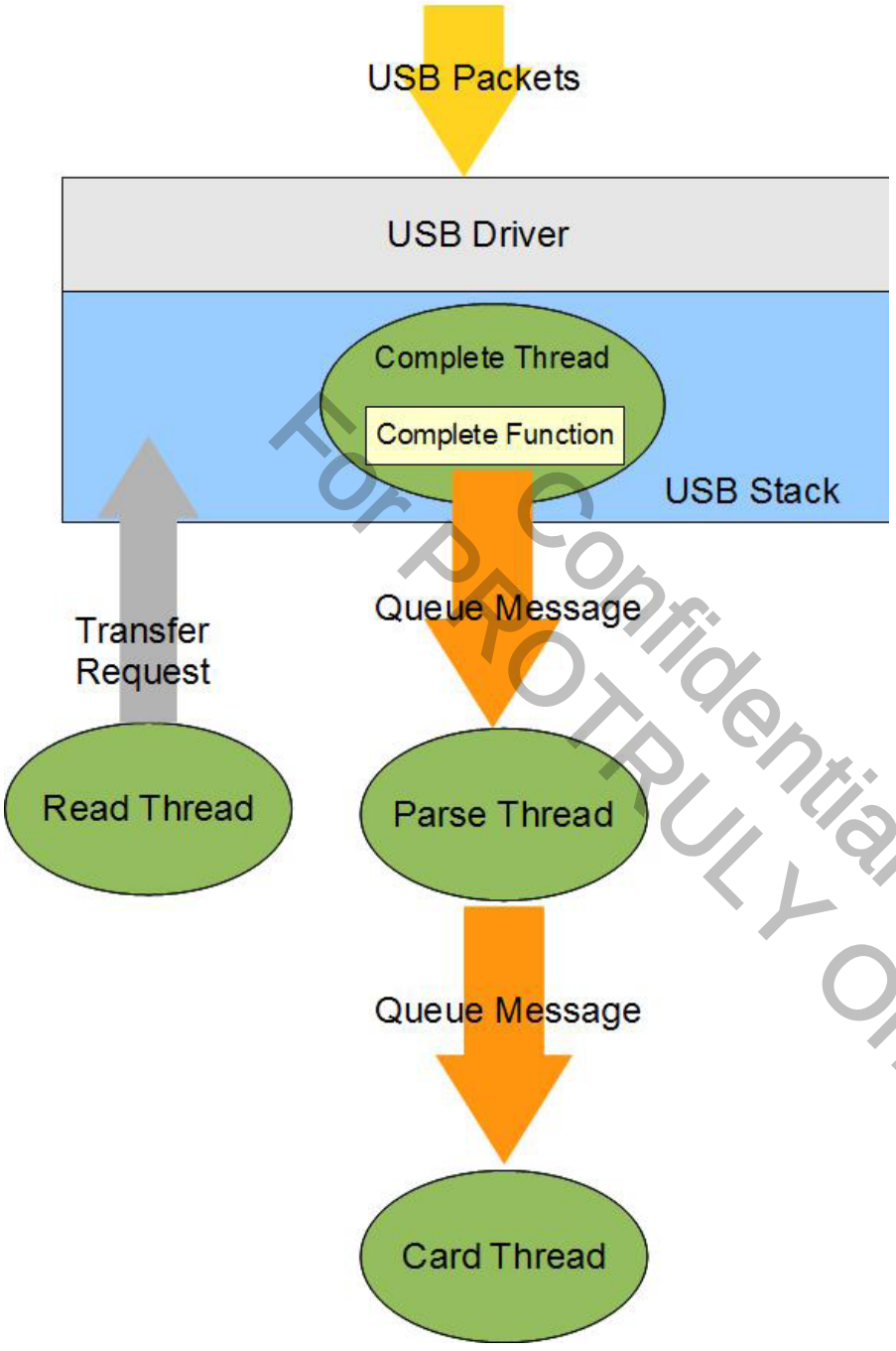

Figure 3-1. The Flow in the Unit Test.

There are four major threads handling the received packets – **Read Thread**, **Complete Thread**, **Parse Thread** and **Card Thread** in the unit test code. Application should create all threads except for **Complete Thread**, which is in the USB stack. **Read thread** is responsible for preparing data buffers, composing request arrays and complete function, then sending transfer request to the USB stack. When the specified transfer is complete, **Complete Thread** in the USB stack would call the complete function, which activates the **Parse Thread** via the queue message.
Parse Thread would search all packet headers and find the video content. The video content would be stored in write buffer temporarily and move the SD card by Card Thread once the write buffer is full.

In the unit test, after the streaming starts, the application calls **AppUvch_ReadStart** to allocate the necessary system resource, including all stack memory, threads, queue message and mutex. Similarly, the application calls **AppUvch_ReadStop** to release them.

3.4 Stream Handling: Read Thread

Read Thread which initiates the transfer request is created after the application calls **AppUvch_ReadStart** after the streaming starts. The main purpose of this thread is to allocate data buffers and compose the request arrays. These memory resources shall be released either after video content is processed in the **Parse Thread** or the transfer request returns an error condition.

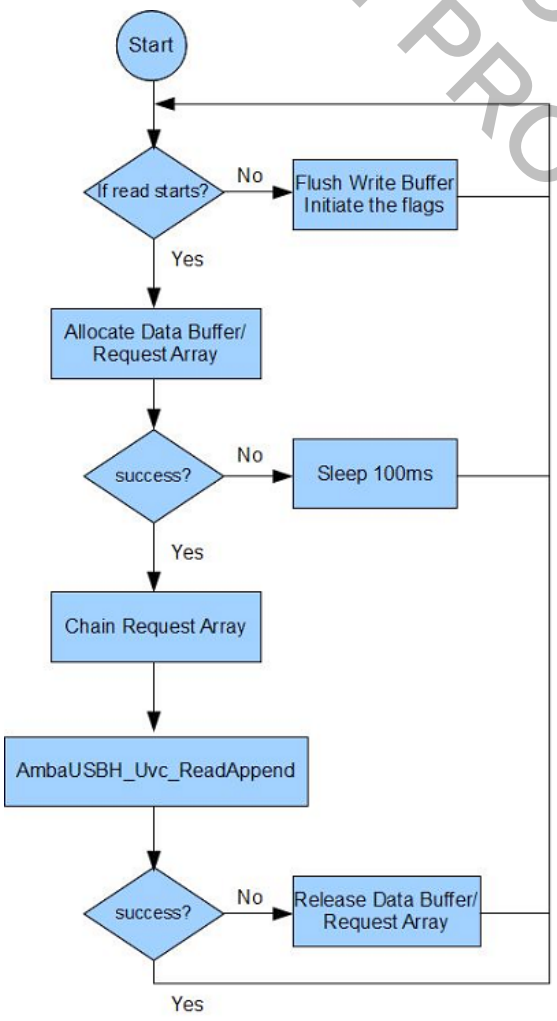

Figure 3-2. The Flow Block of READ Thread.

There are some flags of flow control in the READ Thread. These flags decide if the user needs to issue another transfer request. If not, the user flushes all data stored in the write buffers (if any) and initialize these flags.

A transfer request is complete when its responded complete function is called by **Complete Thread** or when an error is returned. It is possible that more than one transfer request are queued in the USB stack waiting for the request to be accomplished. **Read Thread** can allocate at most 5 data buffers and request arrays at the same time. The size of each data buffers should be PACKET_NUMBER x MAX_PACKET_SIZE. PACKET_NUMBER is the packet number of one transfer request, which is up to 2000 and must be multiples of 8. MAX_PACKET_SIZE can be obtained by the API **AmbaUSBH_Uvc_GetMaxPacketSize** (See Section 3.1.3). The size of each request array is PACKET_NUMBER x size of (UX_EHCI_ISO_REQUEST). The application can have the critical region protection of the resource allocate/release code (ie. mutex) since they could be called by different threads.

If the process of allocating responded resource fails, we shall wait for 100ms and try again.
The request arrays should be filled with correct buffer pointer and data length of each packet. Normally, it should be mapped to a continuous memory.

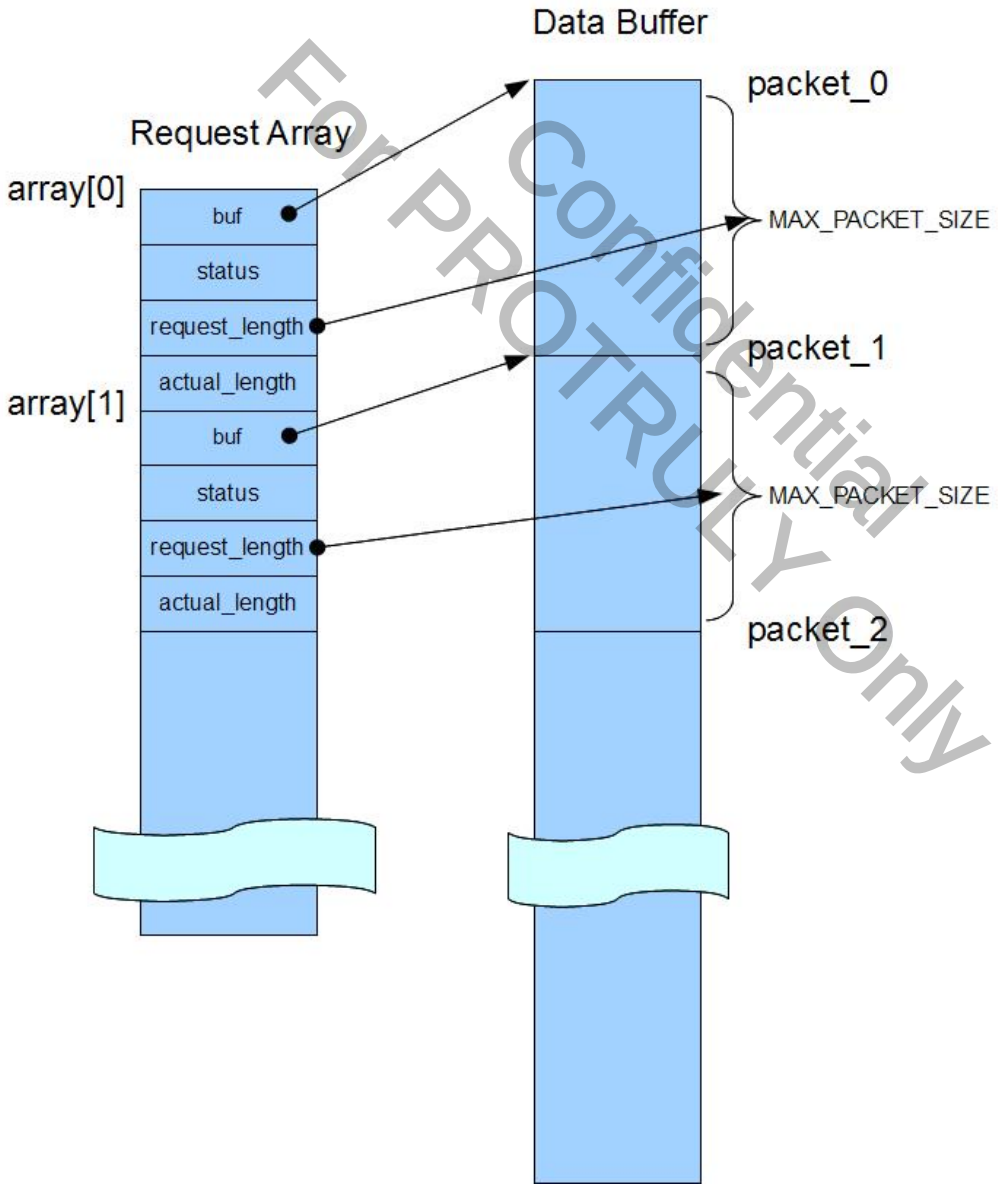

Figure 3-3. Chain the Request Array.

After the request array is chained, **Read Thread** should issue **AmbaUSBH_Uvc_ReadAppend** to start a transfer request. See Section 3.1.2 for more details.

If a transfer request fails, it should release all allocated resources and wait for 1ms before starting another transfer request.

3.5 Stream Handling: Complete Thread

Application does not need to create **Complete Thread** since it is created in the USB stack. When the transfer is complete, the received data should be stored in the responded buffer by the USB hardware. Then, **Complete Thread** would fill the transfer status and actual data length in the request array. Finally, it would call the complete function which has been registered by **AmbaUSBH_Uvc_ReadAppend** or **AmbaUSBH_Uvc_ReadBlock**. To avoid time latency in complete function, the user just sends a queue message to inform the **Parse Thread** which data has been received.

```
static void uvch_rx_complete(UX_EHCI_ISO_REQUEST *request)
{
        AmbaPrint("receive complete 0x%x", request);
        uvch_send_parse_queue_msg(request);
}
```


3.6 Stream Handling: Parse Thread

The received data packets are composed of the stream headers and the stream payload.

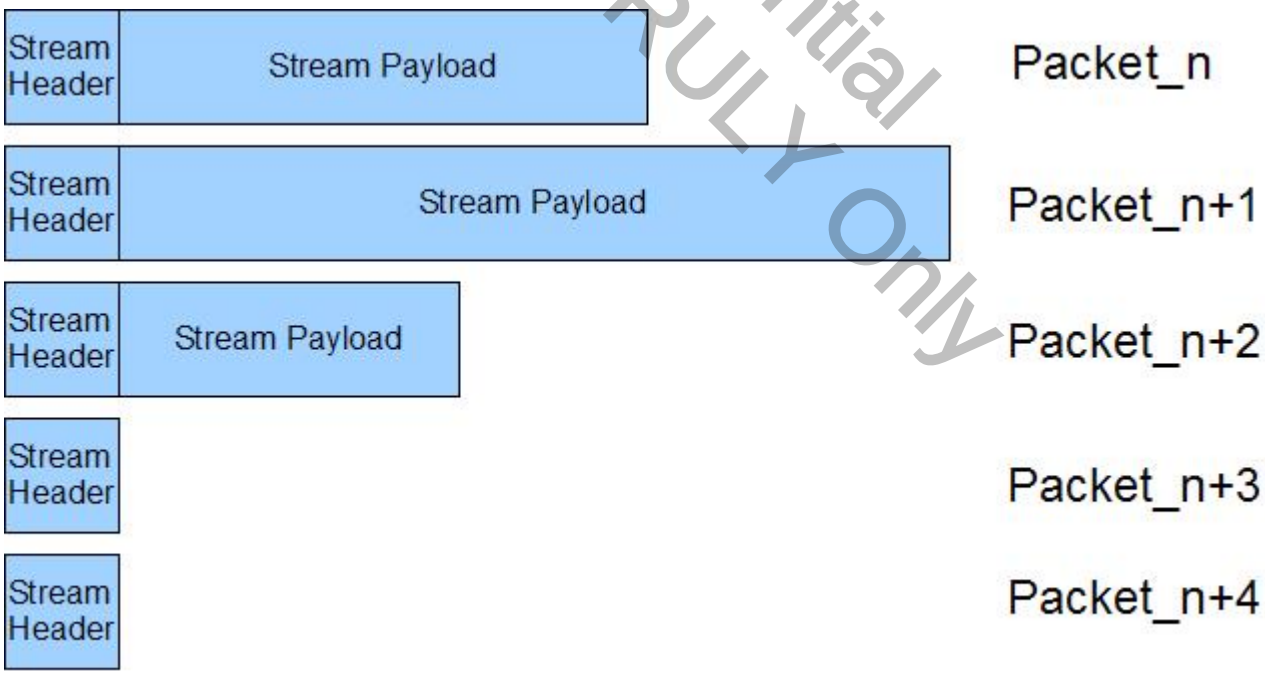

Figure 3-4. Data Structure of the Stream Payload.

Each packet starts with a stream header with/without a stream payload following it. **Parse Thread** would parse the stream header of received packets and extract the specific frame data. Unit test supports two kinds of video format: Uncompressed (YUV) and MJPEG. The 12-bytes stream header format in YUV and MJPEG is as follows:

Header Length							
EOH	ERR	STI	RES	SCR	PTS	EOF	FID
PTS [7:0]							
PTS [15:8]							
PTS [23:16]							
PTS [31:24]							
SCR [7:0]							
SCR [15:8]							
SCR [23:16]							
SCR [31:24]							
SCR [39:32]							
SCR [47:40]							

Figure 3-5. Stream Header of Video Payload.

About the specification of stream header, application designer can check [Universal Serial Bus Device Class Definition for Video Devices: Uncompressed Payload] and [Universal Serial Bus Device Class Definition for Video Devices: Motion-JPEG Payload] for detail. In unit test, Parse Thread uses only FID and EOF to locate the start-of-frame (sof) and end-of-frame (eof) then store the stream payload between sof and eof to write buffers. Below are the functions of FID and EOF bits:

- FID: Frame Identifier. This bit toggles at each frame start boundary and stays constant for the rest of the frame.

- EOF: End of Frame. This bit indicates the end of a video frame and is set in the last video sample that belongs to a frame.

According to the characteristic of stream header, the user could locate the *sof* and *eof* of stream payload.

- *sof*: Start of frame. The first stream payload after FID toggles.
- *eof*: End of frame. The stream payload with EOF bit set

Parse Thread would store the extracted stream payload between sof and eof to write the buffer. When the write buffer is full, **Parse Thread** would issue a queue message for **Card Thread**, which would write the stored data to the SD card. To optimize the performance of card write, the buffer size is the multiples of 128K.

It is possible that the (n)th transfer request ends without finding *eof*, which means the video frame could cross the boundary of transfer request. In this case, **Parse Thread** should reserve the top-half part of the video payload of the (n)th transfer request and combine it with the bottom-half part of the (n+1)th transfer request.

3.7 Stream Handling: Card Thread

To avoid the latency, an independent thread handling card writing is necessary. **Card Thread** activates when receiving the queue message from other threads.

4 Deprecated APIs

4.1 Deprecated APIs: Overview

This chapter provides the deprecated APIs.

4.2 Deprecated APIs: AmbaUSBH_Uvc_Read

This API is replaced by **AmbaUSBH_Uvc_ReadBlock** (Section 3.1.1) and **AmbaUSBH_Uvc_ReadAppend** (Section 3.1.2).

5 Trouble Shooting

5.1 Trouble Shooting: Overview

The chapter provides the solutions to issues.

5.2 Trouble Shooting: Actual Frame Rate is not Expected (Low)

The frame rate would vary depending on the captured scene. For example, the fps is set as 30. Normally the actual frame rate is lower than 30fps. When the captured scene is brighter, the actual frame rate is closer to 30 fps. Ambarella suggests the customers to ask their camera vendors for help.

5.3 Trouble Shooting: Glitch when Playing the Video

Since there is no re-try and error detection mechanism in isochronous transfer, the user needs to make sure the video stream provided by camera is correct. Try to verify the video by connecting the camera to PC with the same configuration. Then, try to check the parser process and see if the corner case occurs.

5.4 Trouble Shooting: Can the user use Multiple Cameras?

The current USB stack does not support multiple cameras.

Appendix 1 Additional Resources

Please contact an Ambarella representative for digital copies.

Appendix 2 Important Notice

All Ambarella design specifications, datasheets, drawings, files, and other documents (together and separately, “materials”) are provided on an “*as is*” basis, and Ambarella makes no warranties, expressed, implied, statutory, or otherwise with respect to the materials, and expressly disclaims all implied warranties of noninfringement, merchantability, and fitness for a particular purpose. The information contained herein is believed to be accurate and reliable. However, Ambarella assumes no responsibility for the consequences of use of such information.

Ambarella Incorporated reserves the right to correct, modify, enhance, improve, and otherwise change its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

All products are sold subject to Ambarella’s terms and conditions of sale supplied at the time of order acknowledgment. Ambarella warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with its standard warranty. Testing and other quality control techniques are used to the extent Ambarella deems necessary to support this warranty.

Ambarella assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using Ambarella components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

Ambarella does not warrant or represent that any license, either expressed or implied, is granted under any Ambarella patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which Ambarella products or services are used. Information published by Ambarella regarding third-party products or services does not constitute a license from Ambarella to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from Ambarella under the patents or other intellectual property of Ambarella.

Reproduction of information from Ambarella documents is not permissible without prior approval from Ambarella.

Ambarella products are not authorized for use in safety-critical applications (such as life support) where a failure of the product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Customers acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of Ambarella products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by Ambarella. Further, Customers must fully indemnify Ambarella and its representatives against any damages arising out of the use of Ambarella products in such safety-critical applications.

Ambarella products are neither designed nor intended for use in automotive and military/aerospace applications or environments. Customers acknowledge and agree that any such use of Ambarella products is solely at the Customer’s risk, and they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

Appendix 3 Revision History

NOTE: Page numbers for previous drafts may differ from page numbers in the current version.

Version	Date	Comments
1.0	18 May 2015	Preliminary Release

Table A3-1. Revision History.