Universal Serial Bus Device Class Definition for Video Devices: Frequently Asked Questions (FAQ)

Revision 1.5

August 9, 2012

Contributors

Abdul R. Ismail	Intel Corp.
Allison Hicks	Texas Instruments
Anand Ganesh	Microsoft Corp.
Anshuman Saxena	Texas Instruments
Bertrand Lee	Microsoft Corp.
David Goll	Microsoft Corp.
Eric Luttmann	Cypress Semiconductor Corp.
Geraud Mudry	Logitech Inc.
Hiro Kobayashi	Microsoft Corp.
Jean-Michel Chardon	Logitech Inc.
Jeff Zhu	Microsoft Corp.
Olivier Lechenne	Logitech Inc.
Remy Zimmermann	Logitech Inc.

Please send comments via electronic mail to <video-chair>@usb.org

Revision History

Revision	Date	Description
1.0		Initial released version
1.0a		Added: 2.21 Interlaced Video
1.0b		Added: 2.22 Maximum Values of the PTS and STC fields within the Payload Header 2.23 Protocol STALL behavior 2.24 Current and Future Payload Header Format and Extensibility

1.0c	June 5, 2004	Added: 2.25 Motion JPEG Characteristics
1.1	June 1st, 2005	Added MPEG2 APT information. Added Terminal association information (RR0052). Added Forward and Backward Compatibility Guidelines (RR0055). Added support for Stream Based Payload (MPEG4-SL, VC1, H264) relying on MPEG2 systems specification (RR0061). Added 1.30 Host Behavior of the Still Image Capture Method 2. (RR0068). Applied RR0064. Change VDC to UVC in **Terms and Abbreviation**. Update section 2.1 with updated document set.

Table of Contents

List of Tables

List of Figures

1 Introduction

1.1 Purpose

This document provides guidelines and answers to frequently asked questions regarding implementation of USB video class-compliant devices. It also describes possible implementation-specific support for compliant devices. This document is provided as an aid to implementers of the USB Video Device Class specification and, as such, is informative only. Should a conflict arise between this document and a specification, the specification shall take precedence.

1.2 Scope

Implementation and host-specific issues not part of the base document *USB Device Class Definition for Video Devices* are addressed in this document.

1.3 Related Documents

USB Specification Revision 2.0, April 27, 2000, www.usb.org
USB Device Class Definition for Video Devices, www.usb.org

1.4 Terms and Abbreviations

The following table defines terms and abbreviations used throughout this document.

Term	Description
CSM	Content security method
FAQ	Frequently asked questions
HID	Human interface device
UVC	USB Video Class

2 Frequently Asked Questions (FAQ)

Frequently asked questions are addressed in the following sections.

2.1 Set of USB Video Class Documents

Question: What are the documents specifying the USB Video Device Class standard?

Answer: The following table lists the USB Video Device Class documents that are included in this documentation. In addition to the base specification, other sections describe specific areas.

Table 2-1 Set of USB Video Class Documents

Category	Document topic / Filename	Description
Base specification	Universal Serial Bus Device Class Definition for Video Devices (USB_Video_Class)	Defines the overall USB Video Class framework.
Terminal specification	Media Transport Terminal (USB_Video_Transport)	Defines the Media Transport Terminal for Digital Video Cameras/Decks.

Payload specification	MJPEG (USB_Video_Payload_MJPEG)	Defines the payload and the header usage (if necessary), based on both bulk and isochronous video streams for the MJPEG video format.
Payload specification	DV (USB_Video_Payload_DV)	Defines the payload and the header usage (if necessary), based on both bulk and isochronous video streams for the DV video format.
Payload specification	MPEG2TS (USB_Video_Payload_MPEG2-TS)	Defines the payload and the header usage (if necessary), based on both bulk and isochronous video streams for the MPEG2TS video format.
Payload specification	MPEG1-SS, MPEG2-PS (USB_Video_Payload_MPEG1-SS_MPEG2-PS)	Revision 1.0. Obsolete. See Payload Stream Based Specification. Defines the payload and the header usage (if necessary), based on both bulk and isochronous video streams for the MPEG1-SS and MPEG2-PS video formats.
Payload specification	Uncompressed (USB_Video_Payload_Uncompressed)	Defines the payload and the header usage (if necessary), based on both bulk and isochronous video streams for the Uncompressed video formats.
Payload specification	Vendor (USB_Video_Payload_Vendor)	Revision 1.0. Obsolete. See Payload Stream Based Specification or Payload Frame Based specification. Defines the payload and the header usage (if necessary), based on both bulk and isochronous video streams for a Vendor specific video format.
Payload specification	Stream Based (USB_Video_Payload_Stream_Based)	Defines the payload and the header usage (if necessary),

		based on both bulk and isochronous video streams for a Stream Based video format.
Payload specification	Frame Based (USB_Video_Payload_Frame_Based)	Defines the payload and the header usage (if necessary), based on both bulk and isochronous video streams for a Frame Based video format.
Example	Video Camera Example (USB_Video_Example)	Provides the descriptors and requests based on sample video device topologies.
FAQ	Frequently Asked Questions (USB_Video_FAQ)	Answers questions about the USB video device specification, implementation of compliant devices as well as host/device interactions.

2.2 Host USB Bandwidth Management Support

Question: How do hosts manage the limited amount of USB bandwidth?

Answer: The following paragraphs describe the bandwidth-management policies and mechanisms that are available in various operating systems.

2.2.1 Microsoft Windows

Bandwidth management is at the discretion of the Video Class driver (usbvideo.sys). The only mechanism for adjusting isochronous bandwidth usage in Microsoft Windows operating systems is via Alternate Interface selection. All current versions of Microsoft Windows require that bandwidth be released and reallocated during the process of changing from one non-zero Alternate Interface to another non-zero Alternate Interface. This means that the isochronous stream must be stopped and restarted.

Bandwidth reallocation is atomic, but *only* if the new bandwidth requirement can be met. This is possible because all alternate interface selection requests are serialized through the port driver (usbport.sys). However, if the new bandwidth requirements cannot be met, the old bandwidth reservation is lost and must be reacquired via a separate Set Alternate Interface request (thus becoming non-atomic).

When selecting alternate interfaces (also known as configuring endpoints), the class driver is able to override the **wMaxPacketSize** of the isochronous endpoint that is part of that interface. In combination with the bandwidth negotiation between the Video Class driver and the device, this would allow the device to specify just one Alternate Interface (other than Alternate Interface 0), and the class driver would specify the agreed-upon **wMaxPacketSize** when configuring the endpoints.

2.2.2 Apple Mac OS X

Bandwidth management for isochronous endpoints on Mac OS X is at the discretion of the Video Class Driver. The primary mechanism for adjusting isochronous bandwidth usage in Mac OS X is via the Alternate Interface selection. Mac OS X requires that bandwidth be released and reallocated during the process of changing from one non-zero Alternate Interface to another non-zero Alternate Interface. This means that the isochronous stream will be stopped and restarted.

Bandwidth reallocation is atomic, but *only* if the new bandwidth requirement can be met. This is possible because all alternate interface selection requests are serialized through the USB Controller driver. However, if the new bandwidth requirements cannot be met, the old bandwidth reservation is lost and must be reacquired via a separate Alternate Interface request (thus becoming non-atomic).

Mac OS X provides a mechanism to adjust the allocated bandwidth for an isochronous endpoint: If not enough bandwidth is available to successfully create the isochronous endpoint (using the wMaxPacketSize specified in the isochronous endpoint descriptor for the selected interface), the

endpoint will get an allocation of 0 bytes. The Video Class driver can then negotiate a bandwidth requirement with the Video Class device and use the SetPipePolicy() API to specify the actual bandwidth required by the endpoint. This would allow the device to specify just one Alternate Interface (other than Alternate Interface 0) and lets the Video Class driver and the Video Class device negotiates an appropriate bandwidth selection.

Bandwidth allocation for full speed devices attached to a high speed hub is done in the same manner as described above. However, the amount of bandwidth available to a device depends on the type of high speed hub. High speed hubs can have one or more transaction translators attached to some or all of the ports. Each transaction translator will provide 12 Mb/s of bandwidth. So, a high speed hub with only one transaction translator will need to allocate the 12Mb/s bandwidth *among* all its ports. A 4-port USB Hub with four transaction translators can allocate 12 Mb/s to *each* of the ports.

2.3 Device Alternate Interface Set

Question: How would a device implementer determine the number of alternate interface settings that a particular **VideoStreaming** interface should support?

Answer: This would depend on the bandwidth usage of the various video parameter combinations supported by that particular **VideoStreaming** interface.

For example, suppose that a **VideoStreaming** interface supports the following parameter set:
Video Format: NV12 (12 bits per pixel)
Video Frame Size: 320 x 240 or 640 x 480
Frame Rate: 5, 10, or 15 fps

This parameter set would have the bandwidth usage characteristics described in the following table (assuming high-speed transfers with one packet per microframe):

Table 2-2 Device Alternate Interface Set

Packet size required (in bytes)	5 fps	10 fps	15 fps
320 x 240	72	144	216
640 x 480	288	576	864

The device implementer may choose to implement the following alternate settings:
0: MaxPacketSize = 0
1: MaxPacketSize = 290
2: MaxPacketSize = 580
3: MaxPacketSize = 870

With this combination of alternate settings, no more than 218 bytes per frame is wasted (that is, reserved but left unused while streaming), which amounts to less than 3% of total bus bandwidth. Of course, finer granularity and less waste of bandwidth can be achieved by implementing more alternate settings. However, this may result in increased device cost.

The final choice of alternate settings and MaxPacketSize ranges is left to the device implementer. The general guideline is that they should be chosen to minimize bandwidth waste for any supported video parameter combination.

2.4 USB Video Class and HID Interfaces

Question: Is HID required for USB Video Class devices? If not, when is it appropriate to use HID?

Answer: USB Video Class devices are not required to implement HID interfaces in order to communicate device state to the host. The key distinction of HID interfaces is the communication of user intent through an arbitrary Human Interface on a device.

For example, a device with a tape transport will probably have buttons that allow the user to play, pause, or stop the tape. If the device designer wishes to model the device such that the host software is aware of the state of these buttons, then a HID interface should be used. However, the state of the tape transport, although controlled by the buttons on the device, can be independently represented through the USB Video Class interface. That is to say, the class driver is only concerned with the state of the transport; it is not concerned with how the transport was made to change its state.

The USB Video Class interface will be the sole interface through which a host controls the device state. This is true even if the device implements a HID interface to represent the buttons on the device. With HID, the user controls on the device would be detached from the internal device mechanism, and all button events would go through the host. The host would respond to these button events by translating them into function-specific requests and relaying those requests to the device through the currently active, video device driver instance.

The following illustration shows a recommended HID implementation on a Microsoft Windows platform.

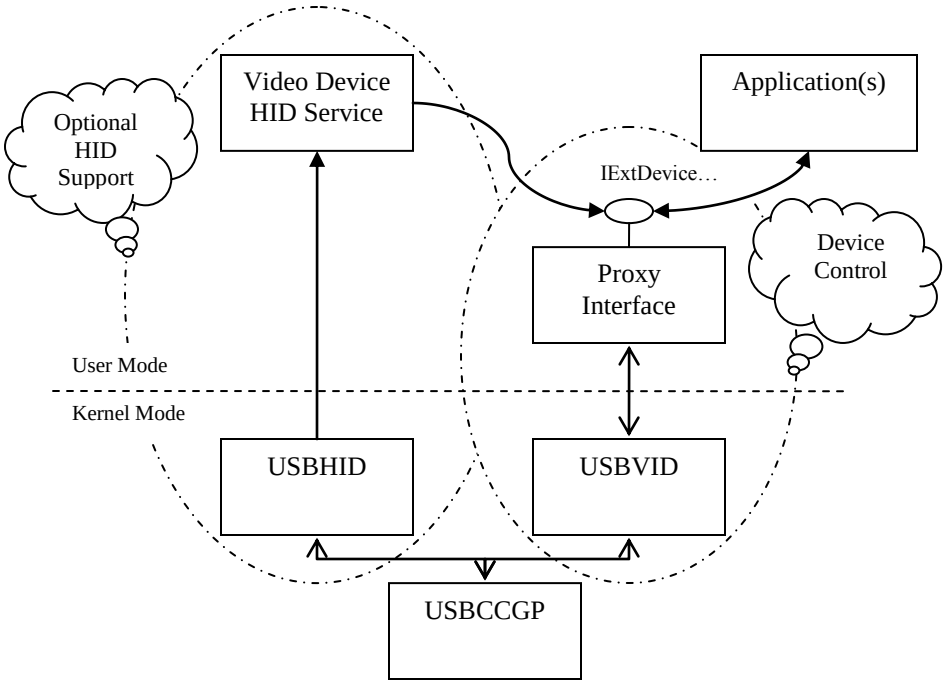


Figure 2-1 Recommended HID Implementation

2.5 USB Video Class Scope

Question: What is the Video Device Class scope?

Answer: The Video Device Class definition applies to all devices or functions within composite devices that are used to manipulate video and video-related functionality. This includes devices such as desktop video cameras (or "webcams"), analog video converters, analog and digital television tuners, and still image cameras and digital camcorders with video-streaming functionality.

Device types covered in these specifications:

- Desktop video cameras (Web cameras)
- Digital video cameras/decks (digital camcorders)
- Digital still-image cameras with video-streaming functionality
- Analog video converters

Device types NOT covered in these specifications:

- Analog television tuners
- Digital television tuners
- Other video devices

Note: Each device category could be added as a terminal or a unit in the future.

2.6 Bulk versus Isochronous Transfers

2.6.1 Characteristics of bulk and isochronous transfers

Question: What are the characteristics of bulk versus isochronous transfers?

Answer: Bulk transfer ensures error-free transmission at much higher potential bit rates than isochronous transfer. Bulk transfer uses any bandwidth not already being used by isochronous and/or interrupt endpoints; however, bulk-transfer bandwidth cannot be reserved or guaranteed. This means that competition with other devices on the bus may reduce the available bulk-transfer bandwidth below that needed for real-time transfer of video.

For real-time transfer of video, such as with video conferencing, bulk transfer cannot guarantee that all data will be delivered, or if delivered, it is not guaranteed to be delivered on time. Ironically, the reason has to do with the error-free nature of bulk transfer. Data can be dropped when buffering at both the source and the destination is kept to a minimum (a requirement for low latency). For instance, data could be dropped at the source if the previous data transfer is still in progress due to retries. Or, data might have to be discarded when it finally reaches the destination, because the presentation time for the sample has already passed.

Therefore, while certainly possible, bulk transfer is not the optimal approach for real-time transfer (for example, video conferencing) due to the lack of bulk-bandwidth reservation and the potentially unbounded latency of data transfer. However, if the latencies in producing and transferring an entire video frame fall well enough within the frame interval, and contention with other devices is managed by the user, bulk transfer can ensure error-free and timely video data transmission.

Bulk transfer for USB Video Class devices is best used for transfer from sequential storage (for example, capture for Non-Linear Editing), where data accuracy is the primary concern, rather than the timeliness of delivery.

Bulk transfer may impose extra buffering requirements at the source and destination. The buffering in the device allows data to accumulate, so that in the event of transmission retries, additional data can be collected instead of discarded. This would be especially important for transfer to and from sequential storage, where the transport mechanism does not have fine-grained throttling capabilities. Buffering introduces latency, which if too great, is unsuitable for video conferencing.

2.6.2 Bulk pipe transfers

Question: Why do we need a bulk pipe for video streams?

Answer: In general, a bulk pipe has the following three major features:

1) Access to the USB on a bandwidth-available basis.
2) Retry of transfers, in the case of occasional delivery failure due to errors on the bus.
3) Guaranteed delivery of data but not guaranteed bandwidth or latency.

For the following scenarios, a bulk pipe with the previous features (high speed, reliability) is more suitable than an isochronous pipe for transferring video stream data.

Video Editing Scenario

In video editing, it takes one hour to transfer one hour of video data from a device to a host over an isochronous pipe. However, it may take less time using a bulk pipe instead. Depending on the USB traffic conditions, the bulk pipe may manage data faster than the actual length of time. Furthermore, the reliability of video data, which is very important for video editing, can be guaranteed with the bulk transfer due to the error-correction functionality. The bulk video transfer could be possible with mass storage class devices; however, it is not applicable for sequential media storage devices that do not have a file system like FAT, such as a DV camcorder.

In conclusion, bulk video transfer is the preferable method for video editing, because video data stored in the device could be transferred to the host more quickly and reliably. Therefore, video editing could be done without data loss, which would be of significant value to video editors.

Multiple Camera Control Scenario

In the case of a surveillance-camera system, many cameras could be connected to a host, although every single camera would not need to send the video data at a high frame rate. If an isochronous pipe was used, each camera would have to reserve the bandwidth; in this case, the host application would have to control the frame rate and calculate the bandwidth of all of the cameras to avoid a bandwidth shortage. For example, if twenty cameras were connected to the host, the bandwidth control in the host application would be very complicated. However, if a bulk pipe is used, bandwidth control would not be necessary, and it would be easier for the host application to control many cameras on the bus at the same time.

Direct Recording Video Capturing from USB Video Device to USB Mass Storage Device Scenario

This scenario involves recording data directly from a USB video device to a USB DVD-RW device (mass storage class).

If a USB video device uses an isochronous video transfer, the isochronous pipe could occupy most of the USB bandwidth. In that case, a bulk-pipe bandwidth of a USB DVD-RWMSC device that is on the same bus would be limited, because the bulk pipe has a lower priority than an isochronous pipe. As a result, the recorded video picture might not be of an acceptable quality.

However, if the USB video device uses a bulk video transfer, it would not reserve the bus bandwidth. Therefore, you can capture the video in the best way possible, since the bulk pipes for both devices have equal priority.

2.6.3 DATA PID Ordering of Bulk Transfers

Question: Is my bulk endpoint required to start every payload transfer with a packet having a DATA0 PID?

Answer: No. The bulk transfer examples in section 2 of the *USB Device Class Definition for Video Devices* specification show DATA0 PIDs at the start of each transfer, but this is only meant to simplify the diagrams. The *USB Specification Revision 2.0* requires alternating DATA0 and DATA1 PIDs for bulk transfer packets.

2.7 Audio and Video Stream Synchronization

Question: What are the synchronization mechanisms provided by the USB Video Device Class?

Answer: To properly synchronize multiple audio and video streams from a media source, the media source must provide certain information to the media sink. This information includes its local stream latency, periodic clock-reference information, and a way for the media sink to determine the proper presentation time for samples from each stream (relative to the other streams).

Latency

The media source is required to report its internal latency (delay from data acquisition to data delivery on the bus). This latency reflects the lag introduced by any buffering, compression, decompression, or processing done by the stream source. Without latency information for each stream, a media sink (or rendering device) cannot properly correlate the presentation times of each stream.

In the case of a video source, this means that the source must guarantee that the portion of a sample fully acquired as of SOF_n (start of frame *n*) will have been completely sent to the bus as of $SOF_{n+\delta}$. Latency δ is the source's internal delay expressed in frames. For high-speed endpoints, the resolution increases to 125µs, and the delay will be specified in micro-frames.
Every video streaming interface must report this latency value. For more information, see the description of the *wDelay* parameter in the "Video Probe and Commit Controls" section of the *USB Device Class Definition for Video Devices* specification. By following these latency rules, phase jitter is limited to ±1 ms (or ±125µs for high-speed endpoints). It is up to the video sink to synchronize streams by scheduling the rendering of samples at the correct moment, taking into account the internal delays of all media streams being rendered.

Per Frame Latency

If the media source expects variable delay per frame, potentially due to encoding or other video processing on the source, it should implement PTS and SCR as described below. Note that this type of implementation is requires for H.264 and VP8 payloads.

Video delay between sensor capture and host driver timestamp is calculated in two parts. The delay on the camera due to pipeline processing and encoding, and the delay caused by USB transport and host processing.
The webcam generates two pieces of data that aid in calculating these two delays, Presentation Time Stamp (PTS) and Source Clock Reference (SCR). PTS and SCR are attached to the payload header as described in the associated payload specification. PTS should be attached to every frame and SCR at the frequency required to address clock drift. An abbreviated definition is as follows:

- Presentation Time Stamp (PTS) is the value of the Source Time Clock (STC) in native device clock units when the raw frame capture begins. The PTS is in the same units as specified in the **dwClockFrequency** field of the Video Probe Control response.
- The Source Clock Reference (SCR) contains two fields that enable the host to correlate between the device clock and the USB clock.
  - STC: device's STC value in units of the **dwClockFrequency** field of the Probe and Commit response of the device
  - SOFTC: Start-of-Frame (SOF) token counter for USB expressed in units of the 1 KHz USB host controller clock.

In generating the SCR, both the STC clock and SOF clocks are sampled at the SOF boundary when the video frame is sent over USB. While the UVC 1.1 specification states that the SOF is not required to match the 'current' frame number, for this solution, the SOF must be the same frame number as that of the USB packet to which the SCR is attached.
The delay of the video frame on the camera is calculated as:

DeviceDelay = (SCR_STC) - PTS Equation 1

This delay is expressed in units of dwClockFrequency, where dwClockFrequency is provided by the device as part of Probe & Commit. The delay caused by USB transport and processing is calculated as the difference between the SOF marker when the driver receives the video payload and the SOF in the SCR from the device:

TransportDelay = SOF_Driver – SOF_SCR Equation 2

TransportDelay is expressed in units of the 1 KHz USB host controller clock.
The total delay for each video frame between capture and the video class driver is calculated as the sum of the two delays calculated in Equation 1 and Equation 2 above, and is expressed in units of the host QPC clock.

Total Video Delay = DeviceDelay + TransportDelay Equation 3

The next task is to correlate between Device and PC clocks. Since the capture time of the video frame (PTS) is indicated by the device using the STC, and A/V sync will rely on PC clock values, we need to correlate the two clocks. On Windows, the PC clock is exposed via the Query Performance Counter (QPC), and this clock will be used as the PC clock in the following

discussion. The correlation 'constant' between PTS and QPC can be calculated as the most recent Total Video Delay.

Clock Correlation Constant (CCC) = Total Video Delay Equation 4

The timestamp applied by the video driver to the current video frame is calculated as the timestamp for the current frame – CCC.

Timestamp for current frame = QPC - CCC Equation 5

For video samples that are split into multiple slices per frame, timestamp calculated above should be applied to all slices belonging to the same picture. New pictures can be detected by inspecting the payload header to see if the FID bit has flipped.

Clock Reference
Clock reference information is used by a media sink to perform *clock-rate matching*. Rate matching refers to the synchronization of the media sink's rendering clock with the media source's sampling clock. Without clock-rate matching, a stream will encounter buffer over- or under-run errors. This has not been a problem with audio streams due to the relative ease of performing audio sample-rate conversion. But with video, sample-rate conversion is significantly more difficult, thus a method for rate matching is required.

To understand the problem of clocks running at slightly different rates, consider the following example. For simplicity, assume that video buffers can be filled instantaneously, and that there is one buffer available to be filled at any given time within the video frame interval. Also assume that the two crystals governing the source and rendering clocks operate with 100 ppm (parts per million) accuracy. The accuracy value is a ratio that can be applied such that for every frame, the clock will drift by a fraction of the frame that is equal to the ratio. In other words, two clocks with accuracy of 100 ppm could have a worst-case drift relative to each other of $1/5{,}000^{th}$ of a frame (two clocks at opposite extremes of their valid operating range for a cumulative error ratio of 2 * 100/1,000,000). Therefore, a frame glitch will occur once every 5,000 frames. At a frame rate of 30 fps, this would equate to a glitch every 166.67 seconds. At a frame rate of 60 fps, the glitch rate is worse, with one glitch every 83.3 seconds.

Frame glitches can be postponed, *but not avoided*, by adding additional buffers to hold video frames before they are rendered. If the source clock is running slower than the rendering clock, the buffer underrun could only be postponed by letting the extra buffers fill to a certain threshold before rendering, resulting in unacceptable latency. Once the first glitch occurs, the extra buffers are effectively useless, since the behavior will degrade to the single-buffer case from that point onward.

This specification assumes that in all cases, the media sink has no control over the media source clock, and that the source and sink do not "slave" to a common clock (the bus clock lacking sufficient resolution). Also, due to cost constraints, additional isochronous endpoints to

communicate clock-rate information will not be used. Therefore, this specification requires that a video stream include clock-reference information that can be used to adjust the rendering clock rate. The clock-reference information may be encapsulated in a transport stream, or it may be provided via an optional field in each sample header. This field becomes required in the latter case.

Presentation Time
For fixed-rate streams, the presentation time can be derived from the data stream. For a fixed-rate audio stream (for example, PCM), the media sink can derive the presentation time from the stream offset (typically the count of bytes since start of capture). For variable-rate streams, each sample must be accompanied by a presentation time stamp. The media sink is responsible for converting the time stamp to native units and adjusting the time stamp to account for the local clock offset when a stream starts, as well as accounting for source-stream latency. Even though video streams might arrive at the media sink at a fixed frame rate, if they are subject to variable-rate compression/encoding, they are not considered fixed-rate streams and will require time stamps on the samples.

2.8 Support for Streaming Encrypted Video Content over the USB

Question: What support would be available for streaming encrypted video content over the USB?

Answer: The USB Content Security (CS) class defines a mechanism for audio and video devices to support the streaming of encrypted data.

The base CS class specification standardizes on the endpoints, descriptors and requests needed for a content-security interface in a composite device. This CS interface is intended to supplement existing audio/video interfaces, and allows for the enumeration and application of content security methods (CSM) to particular audio/video/other data-transport endpoints or interfaces on a device.

The various CSMs that are supported are outlined in separate companion specs, with two currently available for v 1.0:

- CSM 1: Basic Authentication Protocol - This spec simply defines a host request to retrieve a unique channel ID from the device.

- CSM 2: Digital Transmission Content Protection (DTCP) (aka 5C) - This spec defines the requests, interrupts, descriptors and packet formats that are needed to support DTCP over USB. The DTCP spec itself defines the AKE (Authentication and Key Exchange) protocols, cipher negotiation and transmission management needed to send encrypted data between the device and the host.

In order to leverage the CS class to support encrypted video streams, the following would have to occur (for Microsoft Windows platforms):

- Due to legal and/or licensing issues, DTCP is not supported on Microsoft Windows platforms. Therefore, a new CSM would have to be defined that provides similar functionality (Authentication/Key Exchange/Cipher negotiation/Transmission management).
- A Content Security (CS) class driver would have to be developed to control the CS interface on the device, and to provide support for the new CSM that is defined, including encryption/decryption capability.
- A new Secure Video Path would have to be defined for Windows platforms that would secure the driver and application components that form the data transport channel. This is not currently available on Microsoft Windows platforms, but is being explored for future versions.

2.9 Layout of a GUID Data Structure

Question: What is the layout of the GUID data structure and how should a device transmit and parse this data structure?

Answer:

Table 2-3 Layout of a GUID Data Structure

Offset	Field	Size (bytes)	Example
0	Data1	4	47504a4d
4	Data2	2	0000
6	Data3	2	0010
8	Data4a	1	80
9	Data4b	1	00
10	Data4c	1	00
11	Data4d	1	AA
12	Data4e	1	00
13	Data4f	1	38
14	Data4g	1	9B
15	Data4h	1	71

Therefore, for a GUID defined as {47504A4D-0000-0010- 8000-00AA00389B71}, as described in the table above, the little-endian byte-stream representation would be:
0x4D, 0x4A, 0x50, 0x47, 0x00, 0x00, 0x10, 0x00, 0x80, 0x00, 0x00, 0xAA, 0x00, 0x38, 0x9B, 0x71

2.10 Host Processing of Incoming and Outgoing Data

The following paragraphs describe the host data processing state machines.

2.10.1 Diagram Terminology

These are descriptions of the terms used in the state diagrams that follow:

- **Data** – This refers to an incoming (or outgoing) video data unit arriving from (or going to) the device through a UVC VideoStreaming pipe. For isochronous pipes, a data unit is defined as a USB packet. For bulk pipes, a data unit is defined as the smallest format-specific block to which a UVC header is attached. It is assumed that the size of a single data unit is always less than or equal the size of a single buffer.
- **New Frame** – This indicates the condition where the host driver detects a new frame/sample through a transition in the state of the FID bit from the previous data packet received (if available).
- **Buffer** – This refers to the data buffer currently being processed. Data buffers are submitted by the host application to the host driver as part of sequential read or write IRPs (I/O request packets). The driver will queue these IRPs, and the buffer belonging to the IRP at the head of the queue is the current buffer. For incoming data, the driver will

fill this buffer with video data received from the device, and complete the read request (that is, return the filled buffer back to the application) when a frame boundary is encountered (as indicated by the EOF and/or FID bits), or when the buffer is full. For outgoing data, the driver will transmit the data in the current buffer to the device, setting the EOF and/or FID bits as necessary, and complete the write request when the all the buffer data has been transmitted.

- **Buffer Ready** – This indicates whether there is a current buffer available. This condition is true only when the "Fetch Buffer" state was passed through, and a pending buffer was available in the driver queue on the last buffer fetch attempt, and the current buffer was not dequeued and completed.
- **Empty Buffer** – This condition is true only if a current buffer is available and it contains no data.
- **Buffer Full** – This indicates that the current buffer is full, or that the remaining buffer space available to be filled is less than the size of a single data unit.
- **Fast Complete** – This flag is set at the Fill Buffer stage if the host detects a frame boundary (EOF=1) or a full buffer after filling the current buffer. It is reset to zero at the Fetch Data stage.

2.10.2 Incoming Data State Diagram for Frame-based Video Formats

The following state diagram depicts the host behavior in handling data packets arriving from the device for frame-based video formats. Support of the FID and EOF bits is required.

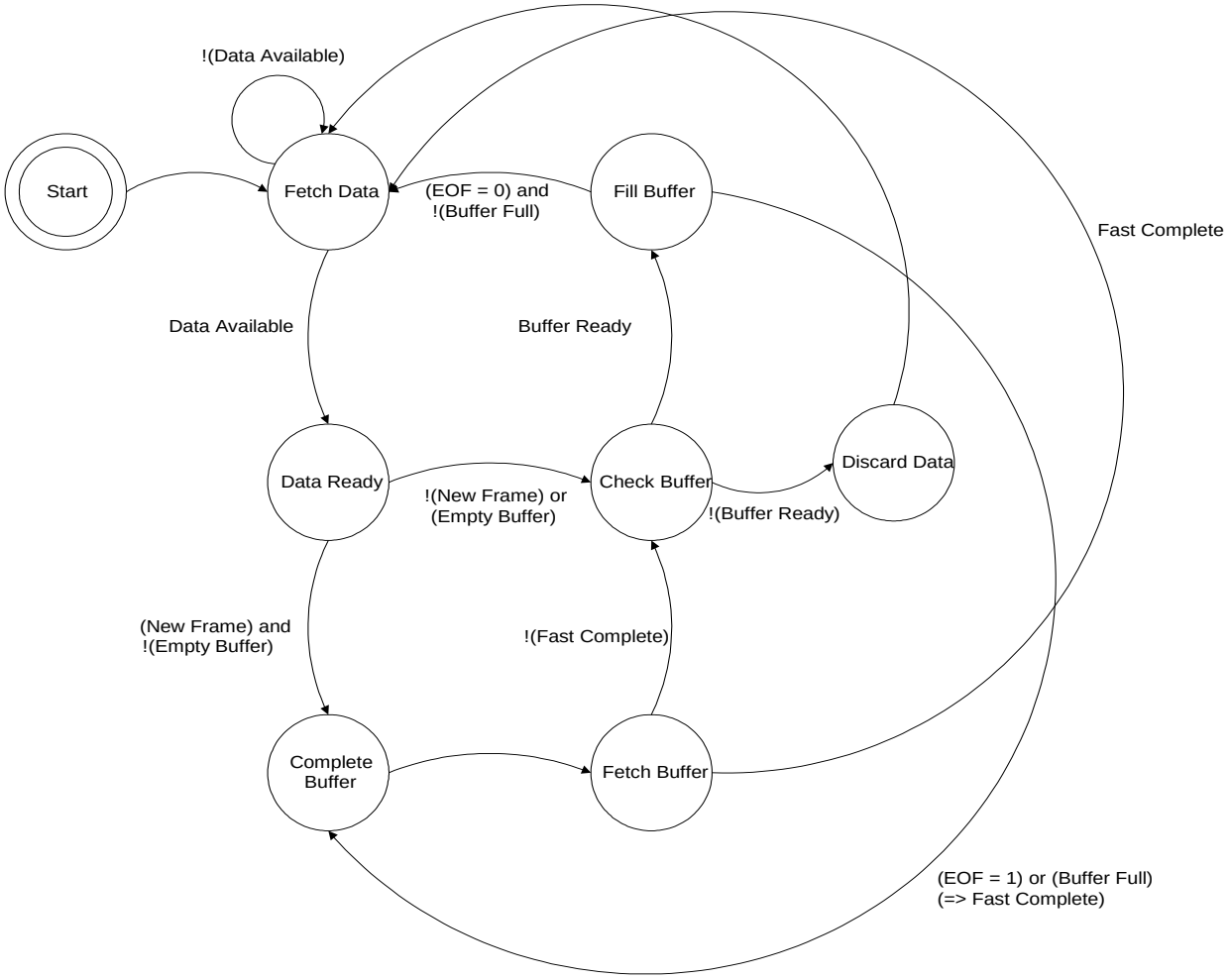


Figure 2-2 Incoming Data State Diagram for Frame-based Video Formats

2.10.3 Incoming Data State Diagram for Stream-based Video Formats

The following state diagram depicts the host behavior in handling data packets arriving from the device for stream-based video formats. Support of the FID and EOF bits is not required.

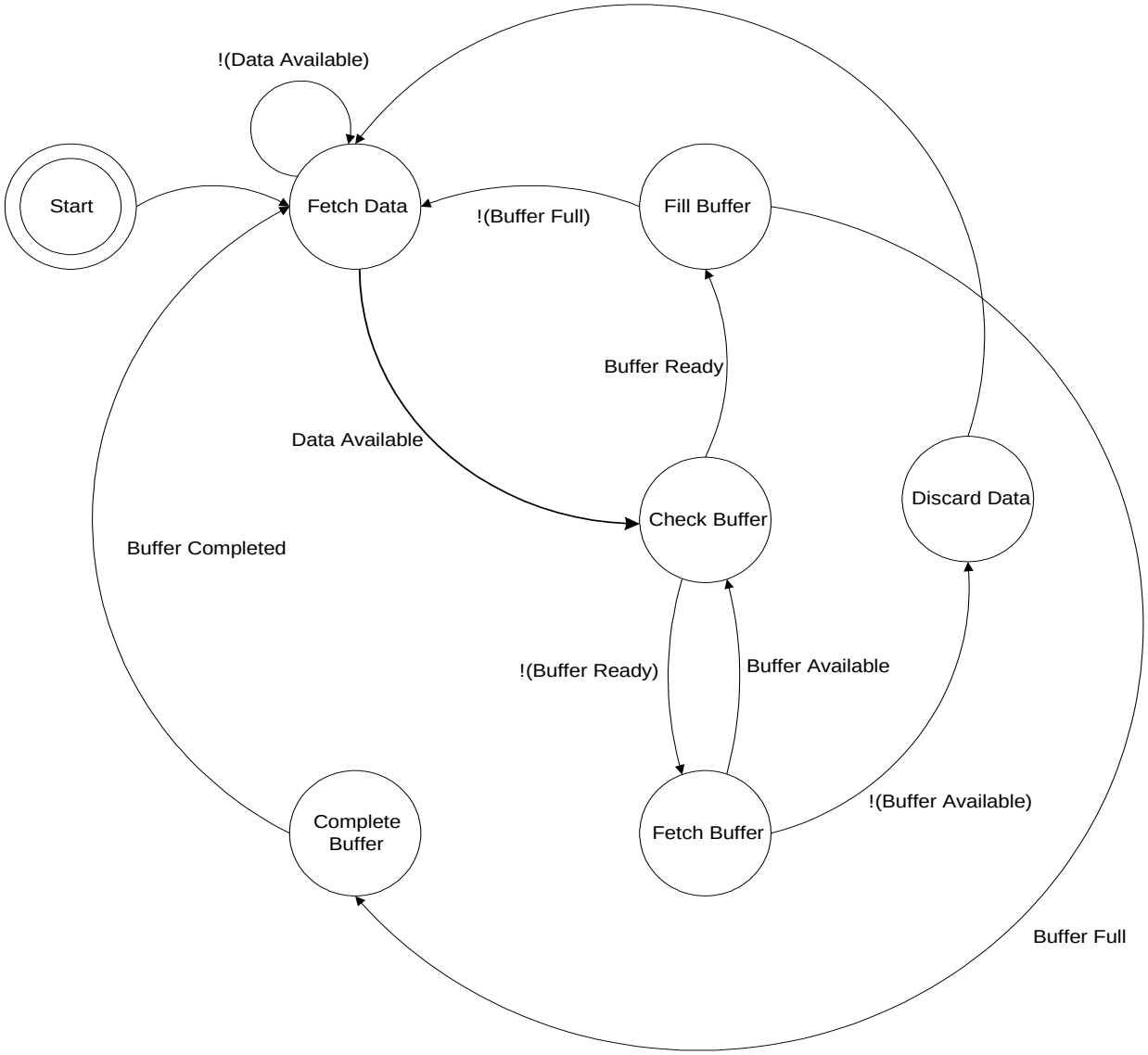


Figure 2-3 Incoming Data State Diagram for Stream-based Video Formats

2.10.4 Outgoing Data State Diagram for Frame-based and Stream-based Video Formats

The following state diagram depicts the host behavior in handling data being transmitted to the device for frame-based and stream-based video formats. The FID and EOF bits are supported by the host for frame-based video formats.

For data formats that support embedded bus timing information, the host could possibly parse this information and selectively throttle transmission, to prevent data overrun in the device buffer. This is indicated by the optional states depicted in dashed lines in the following diagram. Note that this is distinct from the timing information present in the UVC packet header BFH[0] (SCR and PTS fields), which the device should parse in all cases where they are made available, to support rate matching and stream synchronization.

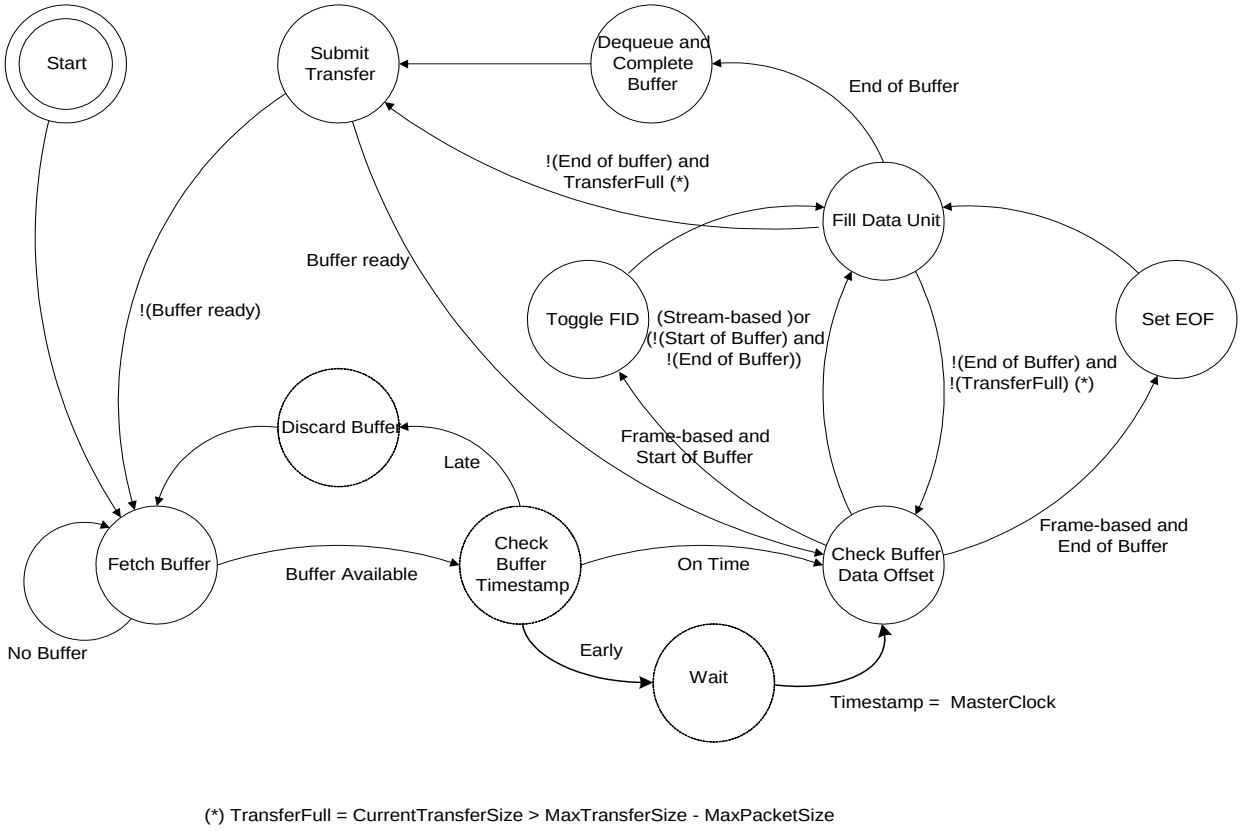


Figure 2-4 Outgoing Data State Diagram for Frame-based & Stream-based Video Formats

2.11 Relationship between Frame Interval and Frame Rate

Question: How is the video frame interval (used in various payload frame descriptors and VS interface controls) derived from the frame rate?

Answer: The video frame interval is specified in 100 ns units and is derived from the frame rate as follows: For a frame rate x, the video frame interval is (10,000,000/x) truncated to an integer value.

For example:
15 fps: Frame interval = (10000000/15) = 666666
30 fps: Frame interval = (10000000/30) = 333333
25 fps (PAL): Frame interval = (10000000/25) = 400000
29.97 fps (NTSC): Frame interval = (10000000/29.97) = 333667

2.12 Clock Recovery Mechanism

Question: What is the clock recovery mechanism used by the USB Video Class?

Answer: The USB Video Class has adopted much of the same principles and terminology used in Annex D of reference [1], which covers this subject in detail. Although applying the same principles, the USB Video Class modifies certain aspects, as described here.

Reference [1] differentiates between SCRs (System Clock References as defined for MPEG2 Program Stream) and PCRs (Program Clock References as defined for MPEG2 Transport Stream). The USB Video Class has settled on a single term, SCR (Source Clock Reference). MPEG2 SCRs and PCRs have differing interval requirements due to the intended method of transmission of the respective types of streams. The USB Video Class standardizes on a single maximum interval of 100ms for both bulk and isochronous streams.

The MPEG2 SCR/PCR values are used to reconstruct the source's System Time Clock (STC) at the destination. In a typical implementation, the destination will implement a PLL to slave a rendering clock to the STC. See [1], Annex D for a description of such a PLL.

In order to support the production of useful *e* values by a Low Pass Filter (LPF) within a PLL, the SCR must have sufficient resolution to allow the clock to be adjusted within the constraints of the maximum slew rate.

The MPEG2 approach is to use a 42 bit value for the SCR/PCR, where the most-significant 33 bits contain 90 KHz clock values, and the least significant 9 bits contain 27 MHz clock values. A 27 MHz clock provides more than enough resolution to remain within the bounds of a 0.1Hz/sec maximum slew rate (which requires at least 10 MHz resolution). The 27 MHz clock portion of the SCR will roll over many times during an SCR/PCR interval, but the 90 KHz clock portion can be used to reconstruct the full 27 MHz clock. The 27 MHz clock values are used as the input to the LPF.

The USB Video Class standardizes on a 48 bit value for the SCR. The least-significant 32 bits (D31..D0) contain clock values sampled from the System Time Clock (STC) at the source. This clock may be of any resolution. If the source is the USB device, the resolution is at the discretion of the device implementer; if the source is the host, the resolution is based on the highest resolution clock available to the host.

The times at which the STC is sampled must be correlated with the USB Bus Clock. To that end, the next most-significant 11 bits of the SCR (D42..D32) contain a 1 KHz SOF counter, representing the frame number at the time the STC was sampled. The STC is sampled at SOF boundaries. It is the same size and frequency as the frame number associated with USB SOF tokens; and is required to match the current frame number.

The most-significant 5 bits (D47..D43) are reserved, and must be set to zero.

The maximum interval between STC samples is 100ms, or the video frame interval, whichever is greater. Shorter intervals are permitted.

Host-Side Clock Recovery Algorithm

The host software exposes a clock that is slaved to the SCR stream. The same device clock is used by the device to generate the PTS values. The host converts the PTS values into native host timestamps by converting from device clock units to host clock units. Aside from rebasing the time stamps such that they are zero at the start of the stream, no other scaling or adjustment is made. Clock rate matching is achieved by applying a scaling value to the clock exposed by the host software. The scaling value is obtained by calculating the slope between the device and host clock ticks counted over a fixed-length sliding time window.

The host software samples a host-based high-resolution counter every time an SCR is received. This high resolution counter is referred to as the Performance Counter (or QPC for short). Each QPC is associated with the USB Frame Counter value at the time the QPC was sampled. The result is a time stamp that varies over a 1ms window around each USB Frame Counter. Over time, the timestamps average out and are effectively free of jitter. The slope SCR(delta) / QPC(delta) would be 1.0 if the clocks are advancing at identical rates. If they are not, the slope provides a direct scaling value to skew the QPC clock values to the SCR clock.

The algorithm given below is shown in two major parts. The first part shows the processing that occurs on the receipt of an SCR from the device. The second part shows the processing that occurs when the host software is queried for the current master clock value. It is this value that is scaled to keep pace with the device clock. The algorithms are shown from the perspective of the host in a capture scenario; however the same principles can be applied to a device implementation in a render scenario. For simplicity, no care is taken to detect system errors, discontinuities, nor arithmetic overflow. Floating point arithmetic is used except where noted. The algorithm can be modified to use integer arithmetic.

The following values and functions are defined (temporary variables are not described in this table):

Table 2-4 Host-Side Clock Recovery Algorithm

Accumulate_QPC()	Samples the host clock and USB Frame number, incorporating the new pair with the previous clock and frame number pairs.
Accumulate_SCR()	Incorporates the device clock and frame counter from the current SCR with the previous clock and frame number pairs.
SOF_delta	The number of frames observed between SCR(*n*) through SCR(*m*), where SCR(*n*) represents the earliest SCR available, and SCR(*m*) represents the most recent. Note that the SCR history may be trimmed to permit a clock slaving mechanism that is more responsive to short-term fluctuations in the host and device clocks.
QPC_delta(*n*)	The number ticks observed in the host clock over the most recent *n* SOF packets (returns 0 if insufficient history available)
SCR_delta(*n*)	The number of ticks observed in the device clock over the most recent *n* SOF packets (returns 0 if insufficient history available)
QPC_freq	The frequency of the host clock (initialized via a host-dependent mechanism)
SCR_freq	The frequency of the device clock (provided by the device through a USB Video descriptor)
ClockSlaveRatio	The clock slave ratio calculated on each receipt of an SCR (initial value 1.0)
MinSlavable	The minimum ratio considered valid for clock slaving (constant value 0.8)
MaxSlavable	The maximum ratio considered valid for clock slaving (constant value 1.2)
MinSOF_delta	The minimum number of milliseconds before clock slaving can begin (constant value 2000)
QPC()	Current host clock value
BaseQPCForLastReturnedTime	Record of last unscaled host clock value
CarriedError	Record of error resulting from cast to integer from floating point
SlavedClock	The slaved clock value

Part 1: Receipt of an SCR

```
Accumulate_QPC();
Accumulate_SCR();

if (SOF_delta > MinSOF_delta) {

    dHostDelta = QPC_delta(SOF_delta) / QPC_Freq;
    dMasterDelta = SCR_delta(SOF_delta) / SCR_Freq;

    if (dHostDelta != 0) {

        dNewRatio = dMasterDelta / dHostDelta;

        if (dNewRatio >= MinSlavable &&
            dNewRatio <= MaxSlavable) {

            ClockSlaveRatio = dNewRatio;
        }
    }
}
```

Part 2: Master Clock Query

```
llNow = QPC();

llDelta = llNow - BaseQPCForLastReturnedTime;

dDelta = (double)llDelta;

// scale according to our slope
dDeltaScaled = dDelta * ClockSlaveRatio;

// accumulate the error
dDeltaScaled += CarriedError;

// compute the non FP value that we are going to use
llDelta = (LONGLONG)dDeltaScaled;

//  carry the error to the next call
CarriedError = dDeltaScaled - (double)llDelta;

//  save the basis for the last returned time
BaseQPCForLastReturnedTime = llNow;

SlavedClock += llDelta;
```

References:

[1] *ISO/IEC 13818-1:* Information technology -- Generic coding of moving pictures and associated audio information: Systems

2.13 MaxPayloadTransferSize selection

Question: How should the MaxPayloadTransferSize value be selected?

Answer: As defined, the MaxPayloadTransferSize parameter depends on the type of USB transfer supported by a video streaming interface.
For an isochronous streaming pipe, this value shall be lower or equal to the USB wMaxPacketSize of the streaming endpoint in the alternate interface selected. The host software is responsible for selecting the appropriate isochronous transfer size based on buffering, latency and processing overhead.
For bulk transfers, this value is implementation specific and is subject to the following tradeoffs:

- a large value potentially increases decoding latency as the USB host controller will only complete an I/O request packet after a completed bulk transfer. As this value is used for memory buffer allocation on the host, this potentially increases the host buffering requirements.
- a small value increases the amount of I/O request packet traffic and increases the number of stream headers to be transferred.

Depending on the target application and tradeoffs mentioned above, the MaxPayloadTransferSize should be set to a value comprised between 10ms worth of data and dwMaxVideoFrameSize.

2.14 Payload Format Forward Compatibility

Question: How can future payload formats be supported?

Answer: Future payload formats can be supported by defining the payload according to the Vendor Payload specification document and require no change to the host video class driver.

2.15 Device/Host Processing Partitioning

Question: Are devices with device/host processing partitioning supported?

Answer: Devices relying on the host for some processing are supported by the device class specification. These devices rely on a vendor specific payload format and rely on host software class driver functionality providing a control and event path from the associated codec to the device's unit(s) and terminal(s).

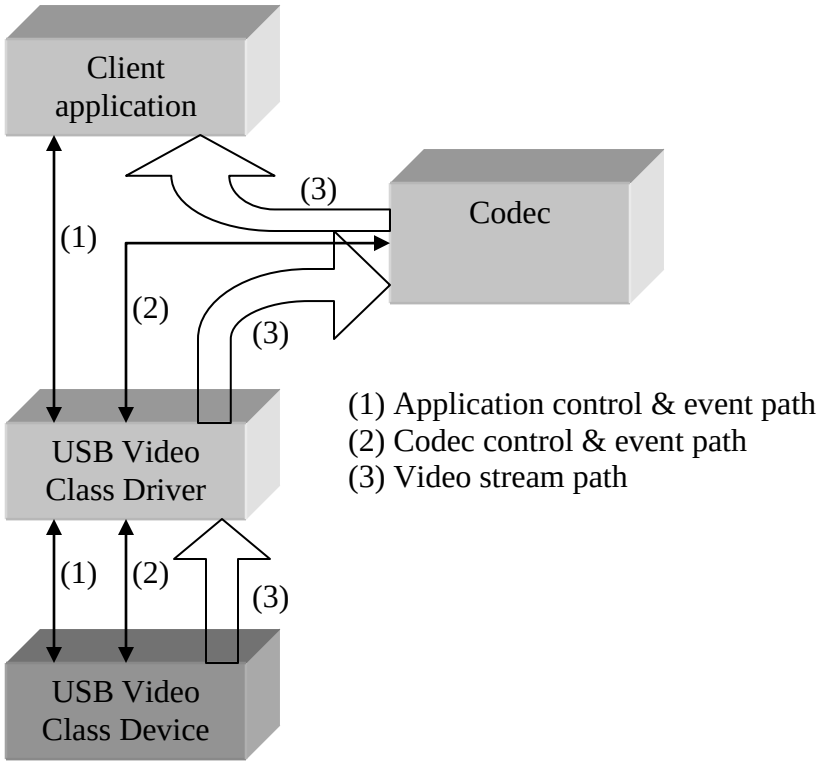


Figure 2-5 Device/Host Processing Partitioning

2.16 Power Mode Control

Question: What does the power mode state mean?

Answer: POWER MODE CONTROL describes the controlling device power mode. There are three power sources available to operate a device:
1) A.C. power
2) USB
3) Battery

In the case of the first two power sources, the user doesn't need to be concerned with the power consumption. However, if the host can not supply enough power to make the device work when another device is also connected to it, the host can use POWER MODE CONTROL to allow the device to work with low power.

If the power source is a battery, the user needs to be concerned with power consumption, and should use POWER MODE CONTROL to conserve battery power consumption. For example, if

the device has the capability to reduce the power consumption by not using some of the device functionality, use POWER MODE CONTROL to do so.

Table 2-5 Video Device Power Source

Power source	Description
A.C. power	Device driven by A.C. power supply
USB	Device driven by USB power supply
Battery	Device driven by battery power supply

A configuration descriptor can indicate to the host which power source(s) a device can use (bus powered and/or self powered). Further a host can use the GetStatus command to determine whether the device is currently self powered. However, the host has no way to know whether a self powered device is using A.C. power or battery power. Specifying the power source would enable the host to know the condition of the power supply on the device side.

2.17 Probe and Commit operations

Question: My device supports only one streaming parameters set (format, resolution, frame rate, etc), what should be implemented for handling the probe/commit controls?

Answer: All the requests and attributes need to be supported but all the GET_XXX request will always return the same data structure content. The commit SET_CUR requests must still validate the required parameters.

Question: How do I cycle through supported stream attributes?

Answer: Issue a request with the GET_XXX attribute. In the returned control data structure, change the stream field (dwFrameInterval, wKeyFrameRate, wPFrameRate, wCompQuality, wCompWindowSize) you want to cycle through by subtracting 1 and issue a SET_CUR request: this will force the device to use the next lower supported value for this field according to the negotiation loop avoidance when a GET_CUR request is issued. The host detects the end of the enumeration when the device state doesn't change anymore (or the enumerated field reached its minimum as returned by the GET_MIN request).

Question: In what cases can a probe/commit request fail?

Answer: A probe/commit request will fail if:

- bFormatIndex and/or bFrameIndex have invalid values
- A given stream specified by bFormatIndex and bFrameIndex cannot be supported with a specified dwMaxPayloadTransferSize.

All the other fields will be negotiated

Question: How does the host software use the Video Probe and Commit Controls to negotiate streaming parameters and video compression properties such as key frame rate, etc.?

Answer: The Video Probe and Commit controls are extremely versatile and can be used in many different ways depending on host software design and requirements.

The following is just one specific example of how the host software could use the Probe and Commit controls.

(Note: The video compression fields/parameters referenced in this example refer to the **KeyFrameRate**, **PFrameRate**, **CompQuality** and **CompWindowSize** fields)

Retrieving range information for the video compression fields (key frame rate, etc.)
1. Host sets format index (and frame index/interval, if supported), sets all other fields to zero, issues SET_CUR to probe control
2. GET_MIN to probe control to get min values for video compression fields
3. GET_MAX to probe control to get max values for video compression fields
4. GET_DEF to probe control to get default values for video compression fields
5. GET_RES to probe control to get resolution values for video compression fields
6. Host repeats from step 1 with a different set of format (and frame index/interval) indices

Setting Video Compression parameters before streaming commences (user-selected format/frame index and frame interval already known)
1. Set all **bmHint** fields to fixed/variable according to user preferences
2. Set **MaxPayloadTransferSize** fields to zero (**Delay** and **MaxVideoFrameSize** fields are always set by device)
3. Prepare probe/commit data structure:
   - Set **FormatIndex**, **FrameIndex** and **FrameInterval** fields to values chosen by user/application
   - Set Video Compression Parameters (**KeyFrameRate**, **PFrameRate**, **CompQuality**, **CompWindowSize**) to values chosen by user/application
4. SET_CUR to probe control
5. GET_CUR to probe control
6. Save **KeyFrameRate**, **PFrameRate**, **CompQuality**, **CompWindowSize**, **MaxVideoFrameSize**, **Delay** and **MaxPayloadTransferSize** fields returned from device.

Getting Video Compression parameters before streaming commences (user-selected format/frame index and frame interval already known)
1. If required parameter is cached, return cached value to client application
2. If required parameter is not cached, prepare ProbeCommit data structure:
   - Set **FormatIndex**, **FrameIndex** and **FrameInterval** fields to values chosen by user/application
   - Set Video Compression Parameters (**KeyFrameRate**, **PFrameRate**, **CompQuality**, **CompWindowSize**) to values chosen by user/application.

- Set bmHint fields to fixed/variable based on user preference and previously initialized parameters.

3. SET_CUR to probe control
4. GET_CUR to probe control
5. Return requested parameter to client application

Stream Format Negotiation (setup for streaming)

1. Set all bmHint fields to fixed/variable according to user preferences
2. Set **MaxPayloadTransferSize** fields to zero (**Delay** and **MaxVideoFrameSize** fields are always set by device)
3. Prepare probe/commit data structure:
   - Set **FormatIndex**, **FrameIndex** and **FrameInterval** fields to values chosen by user/application
   - Set Video Compression Parameters (**KeyFrameRate**, **PFrameRate**, **CompQuality**, **CompWindowSize**) to values chosen by user/application
4. SET_CUR to probe control
5. GET_CUR to probe control
6. Save **KeyFrameRate**, **PFrameRate**, **CompQuality**, **CompWindowSize**, **MaxVideoFrameSize**, **Delay** and **MaxPayloadTransferSize** fields returned from device.
7. When streaming is about to begin, SET_CUR to commit control using values from Step 6 (this should always succeed since it was based on values returned from the probe control)
8. Attempt to select alt setting based on **MaxPayloadTransferSize** saved in step 6 (isoch pipes only)
9. If failed, return failure code or, for isoch pipes, update **MaxPayloadTransferSize** and attempt to select smaller alternate interface setting by going to step 3.

Getting Video Compression Parameters during streaming

1. GET_CUR to commit control

Setting Video Compression Parameters during streaming

1. GET_CUR to commit (to retrieve active device state)
2. Update parameter that is being set
3. Set **MaxPayloadTransferSize** field to current alternate interface max packet size
4. SET_CUR to probe control
5. GET_CUR to probe control
6. If returned parameters are acceptable, issue SET_CUR to commit control
7. If SET_CUR was issued, save **KeyFrameRate**, **PFrameRate**, **CompQuality**, **CompWindowSize**, **Delay** and **MaxPayloadTransferSize** fields
8. If SET_CUR was not issued, return failure code

Notes:
Probe control refers to VS_PROBE_CONTROL
Commit control refers to VS_COMMIT_CONTROL

2.18 Input and Output Terminals association

Question: For bAssocTerminal field in Input or Output terminal descriptors, what is the meaning of an association between two Terminals?

Answer: The implied meaning of Associated Terminals in the Specification is that if we associate two Terminals, we cannot use them at the same time.

Example 1: If we use a Media Transport Terminal such as a Tape.

Since a Tape Terminal cannot be used at the same time for both reading and writing Data, the Input and Output Terminals corresponding to this Tape shall be associated in the corresponding Descriptors.

Example 2: If we use a Media Transport Terminal such as a Random Access Media.

Since a Random Access Media can be used for both reading and writing Data at the same Time, the Input and Output Terminals corresponding to this Random Access Media shall not be associated in the corresponding Descriptors.

2.19 Multiple Color Matching Descriptors

Question: How to deal with multiple color matching descriptors?

Answer: Changes in color matching descriptors are handled by the same mechanism used for dynamic format changes.

Question: Is there justification for the three color matching fields in the descriptor?

Answer: Yes there is. There are numerous devices and standards related to color and very often the standards do not reflect the current usage scenarios. To resolve these ambiguities the color format needs to be explicitly stated. The problem is partly illustrated by the table that follows. It shows different scenarios and the standards used in defining the color.

Table 2-6 Matrix of Color Standards and Scenarios

Color Primary	Transfer Function	Luma Matrix	Example Scenario
709	709	709	DVB HDTV Standard Rec.
709	709	601	Typical webcam
240	709	601	NTSC tuner/video capture device
709	709	601	PAL tuner/video capture device
709	240	240	HDTV(1035/60)
170	170	170	DVB 30Hz SDTV
470	*	*	DVD’s
240	470(M)	601	NTSC SDTV

709	470(B,G)	601	PAL SDTV
709	709	709	ATSC (1920/60)

Question: Why was NV12 chosen as one of the recommended 4:2:0 YUV planar format?

Answer: NV12 was chosen by the USB Video Class because it is the preferred format for graphics cards. This is because NV12 has significant advantages in terms of cache-coherence and overall graphics card performance. Current graphics cards have begun to support this format, and in the future all graphics cards are expected to do so.

2.20 Request Error Code Control

Question: When is the Request Error Code Control value updated?

Answer: The Request Error Code Control is updated after any control request (including the asynchronous control requests and requests to the Request Error Code Control itself). On success, the Request Error Code Control value is reset to 0. On failure (protocol stall on the control pipe), the Request Error Code Control value is set to the failure reason. Subsequent completions of asynchronous control operations (via the Status Interrupt Endpoint) do not affect the Request Error Code Control.

2.21 Isochronous pipes and data availability

Question: What is the expected behavior of a transmitter on an isochronous endpoint when no data is available to be transmitted?

Answer: Isochronous endpoints must accept (OUT endpoints) or return (IN endpoints) a zero-length packet for each bus interval in which data is not available. Note that a payload transfer consisting of a payload header alone (with no payload data) is permitted if the maximum interval for delivery of a Source Clock Reference (SCR) would otherwise expire during a period without data.

Devices that use high bandwidth isochronous endpoints, the device must be ready to accept (OUT endpoints) or return (IN) zero length packets with DATA pids of 0, 1 or 2 depending on the number of additional transaction opportunities per microframe and the amount of buffer remaining. Please to refer to section 5.9.2 and 8.5.5 of the *USB Specification Revision 2.0* for further details. In addition, Appendix D of the EHCI specification for USB Revision 1.0 clearly states the rules that USB 2.0 host controller (and from that derive what a complaint device) should follow to correctly handle high bandwidth isochronous endpoints.

2.22 Interlaced Video

Question: What do the following mean in the context of interlaced video fields: top, bottom, field 1, field 2, odd and even?

Answer: Top and bottom are spatial terms referring to the actual picture lines in a video frame. The first picture line is associated with the top field, and the second picture line is associated

with the bottom field. Field 1 and field 2 are temporal terms. The first field to be captured within a frame is called Field 1, and the other field is called Field 2. Odd and even are overloaded terms since they could refer to odd/even picture lines, or odd/even lines of the analog video field, and for this reason we will refrain from using these terms in our explanations.

Question: What are the kinds of interlaced video formats supported by the USB Video Class?

Answer: The USB Video Class requires interlaced video content to be sent in one of two forms:
1) Line interleaved: Both fields are sent in the same video sample, but the lines of the top and bottom fields are interleaved. i.e. the lines from the two fields keep alternating.
2) One field per sample: Each sample contains only one field and the FID bit is used to indicate whether a given video sample contains a top field or a bottom field (see item below).

Question: How is the FID bit used to indicate top and bottom fields for interlaced video?

Answer: In the case of one-field-per-sample interlaced video, an FID value of 1 indicates a top field, and a value of 0 indicates a bottom field. The FID bit, when used with the D2 bit of the **dwInterlaceFlags** (Field1First) field of the uncompressed format descriptor, lets the host PC combine the fields in an unambiguous manner. Note that temporally, field 1 is always assumed to be sent before field 2 in one field per sample video data.

Question: What does the "Field1First" bit of the interlace flags mean?

Answer: "Field1First" must be read as "Field1First - Spatially". This means the spatially first field, or the top field, is captured first, and is the first field to be streamed in one field per sample video data.

Question: How should the frame descriptor fields be interpreted for one-field-per-sample video data? In particular, what should the value of **wHeight** be?

Answer: The values of the frame descriptor should refer to the field only, and not to the entire video frame. Thus the **dwMaxVideoFrameBufferSize** refers to the number of bytes in one sample (field), frame interval refers to the time interval between fields, and **wHeight** refers to the number of lines in a field. Thus for a 29.97 fps, 720*576 video, the frame descriptor will report 59.94 (fields per second) and **wHeight** = 288 lines.

Question: What is the recommended format on Microsoft Windows operating systems?

Answer: On Microsoft operating systems, two fields per sample, line-interleaved format is the best supported interlaced video format.

2.23 Maximum Values of the PTS and STC fields within the Payload Header

Question: What are the maximum values of the PTS and STC (a portion of the SCR) fields of the Payload Header?

Answer: The maximum value of these fields can be either 0xFFFFFFFF (maximum value of a 32-bit unsigned value), or the clock resolution of the stream minus one. For example, if the stream is delivered with a clock resolution of 13.5MHz, the PTS and STC values are permitted to wrap to zero after reaching either 0xFFFFFFFF or 13,499,999. It is the responsibility of the video sink (the host software when capturing from the device, the device firmware when capturing from the host) to determine which maximum is being used while processing these fields at runtime. The fields may independently wrap in either of the two ways within the same implementation.

2.24 Protocol STALL behavior

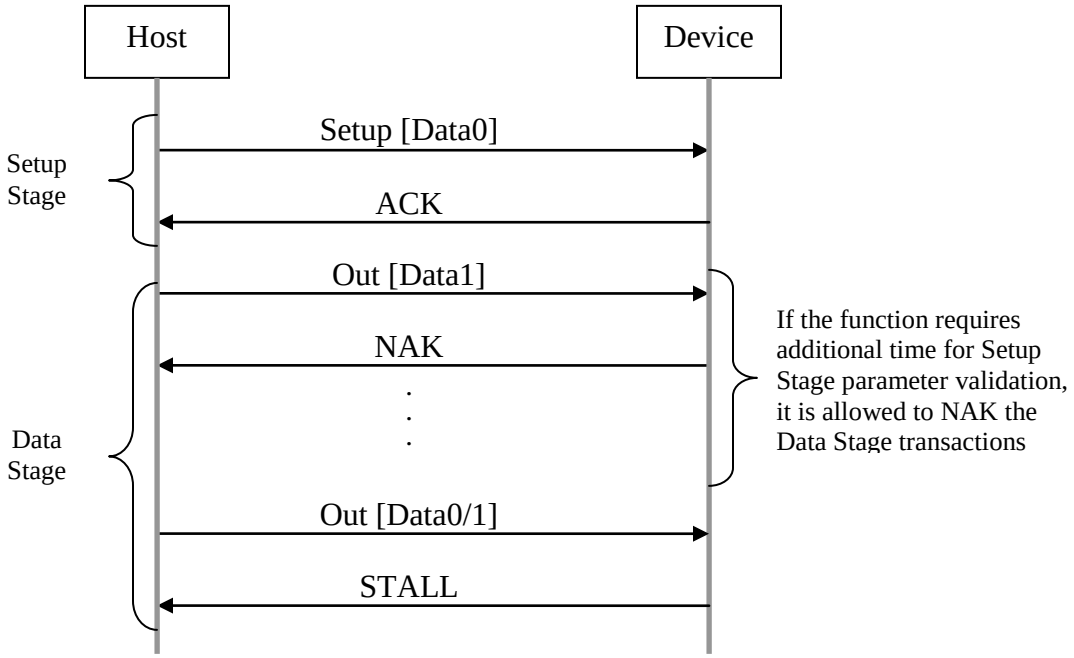
Question: How does a video function handle request parameters validation and error reporting?

Answer: The USB Video Device Class specification defines a protocol STALL to allow the video function to report errors; the function is required to validate control transfer parameters and report errors when invalid.
Additionally, a Request Error Code Control is defined to allow precise reporting of the error's cause.

The following paragraphs describe potential handling of control transfer errors.

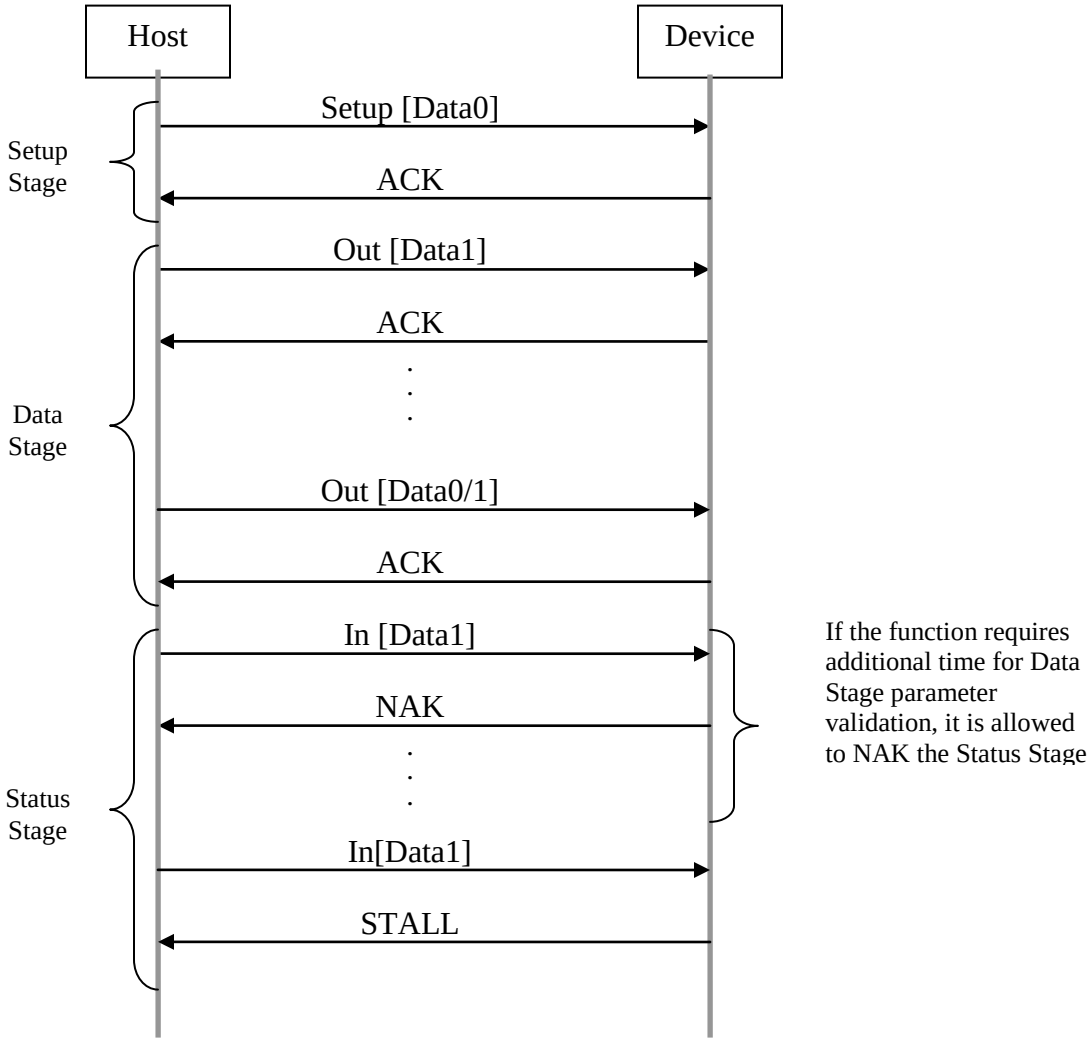
2.24.1 SET_XXX request with Setup Stage error

This example shows a failing control write transfer due to invalid Setup Stage parameters. Note that the STALL condition could also be reported during the Status Stage.



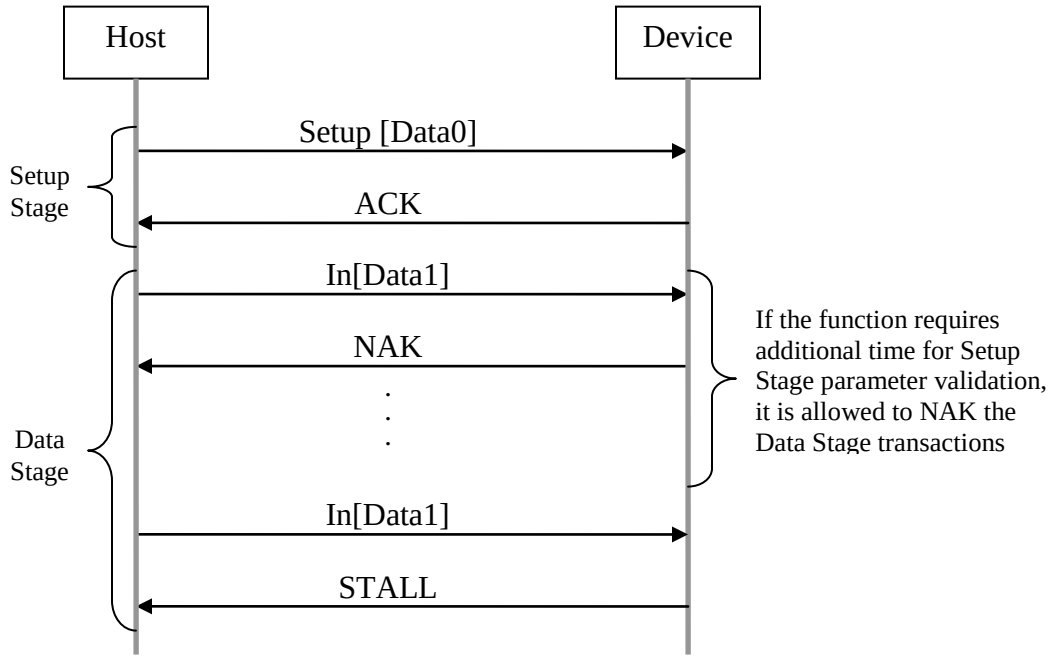
2.24.2 SET_XXX request with Data Stage error

This example shows a failing control write transfer due to invalid Data Stage parameters.



2.24.3 GET_XXX request with Setup Stage error

This example shows a failing control read transfer due to invalid Setup Stage parameters.



2.25 Current and Future Payload Header Format and Extensibility

Question: Can the Payload Header be fixed size?

Answer: Yes. The bHeaderLength value of the Payload Header can specify any length as long as it is sufficient to allow the Payload Header to hold the information indicated by the bmHeaderInfo bitmap field(s). If the bHeaderLength value is larger than necessary, the extra bytes are simply ignored.

Question: Can the bmHeaderInfo byte be extended while maintaining backward compatibility with versions of class drivers released prior to the addition of new bmHeaderInfo bytes?

Answer: Yes. As long as the previous class driver versions use the "End of Header" (D7) bit of the bmHeaderInfo byte(s) to determine when the remaining Payload Header values (if any) start, the bmHeaderInfo portion of the Payload Header can be any number of bytes up to the maximum allowed by the bHeaderLength value and any additional Payload Header data.

Question: Are there any restrictions placed on the bit assignments in bmHeaderInfo extensions?

Answer: Yes. The "End of Header" (D7) bit must be present in each extended bmHeaderInfo byte. The value of the D7 bit will be 0 for all but the last bmHeaderInfo byte.

2.26 Motion JPEG Characteristics

Question: Why do the colors output from my MJPEG device appear to be artificially boosted?

Answer: Newer video cards assume video is in the YCbCr color space (i.e. using BT-601 video ranges). If the images are encoded with the luma and chroma units in the 0-255 range that is used for typical JPEG still images, then the saturation and contrast will look artificially boosted when the video is rendered under the assumption that the levels were in the YCbCr color space. BT-601 specifies eight-bit coding where Y is in the range of 16 (black) to 235 (white) inclusive. Similarly, Cb and Cr are in the range of 16 through 240 inclusive.

2.27 MPEG2-TS APT

Question: For the MPEG TS APT method, how are the implementation-dependent constant value and the required buffer size determined?

Answer: The following two examples show how to determine the implementation-dependent constant value and the required buffer size.

Example 1: If the sink's local counter (*microframe_count* : *microframe_offset*) remains unmodified:

In this case, the sink will calculate a constant value to be added to each APT value when it is received. The value is constant for a particular stream. The constant value will directly determine when each MPEG packet is released to the application and indirectly determine the sink's buffer size. The constant value is equal to the difference between the sink and source's counters plus some number of microframes. The extra number of microframes will compensate for any USB jitter and delivery latency. For instance, if the first APT value in a stream is exactly 10 microframes behind the local (*microframe_count* : *microframe_offset*) value when it is received, the constant value can be set to 13 microframes so that this particular packet will stay in the buffer for exactly 3 microframes. Due to the inherent packet delivery jitter over USB, the successive packets will stay in the buffer between 1 and 5 microframes, provided that the same constant value is used. Therefore, the buffer size should be big enough to store at least 5 microframes worth of data in this example.

Note: While a value of "3 microframes" is mentioned in this example for buffering, the value for each system is implementation dependent.

Example 2: If the sink's local counter (*microframe_count* : *microframe_offset*) value is initialized at the beginning of each stream:

If the sink's local (*microframe_count* : *microframe_offset*) value is properly initialized at the beginning of a stream, each APT value can be compared against the local (*microframe_count* : *microframe_offset*) value to determine the time of delivery to the application. No implementation-dependent constant is required. To cope with the inherent packet delivery jitter over USB, the sink should alter its local timer to be several microframes behind the source's counters so each packet stays in the buffer for some time before it is released to the application. For instance, if the first APT value in a stream is exactly 6 microframes behind the local (*microframe_count* : *microframe_offset*) value, a value of 9 microframes can be subtracted from the local *microframe_count* counter so that this particular packet will stay in the buffer for exactly 3 microframes. Due to the inherent packet delivery jitter over USB, the successive packets will stay in the buffer between 1 and 5 microframes. Therefore, the buffer size should be big enough to store at least 5 microframes worth of data in this example.

2.28 Host and Device interoperability

Question: How is backward and forward compatibility among devices and drivers handled?

Answer: The UVC specification defines the protocol by which devices and drivers must abide; as the specification evolves, devices will integrate those changes and drivers must be resilient to them. Devices are expected to properly respond to host driver requests based on earlier specification version(s) and assume meaningful defaults for unspecified fields. Below are guidelines addressing such interoperability issues.

Table 2-7 Interoperability guidelines

Specification Modification	Expected Host Behavior	How to address issue/comments
Redefinition of a field		Keep old field and define new field.
Additional descriptor	Skip it based on type and subtype (if unknown, skip)	
Addition of descriptor field(s)	The descriptor length field can be used to skip over unknown fields.	
New terminal/unit	The host class driver should either expose it generically or ignore it.	If the host class driver cannot expose the new terminal/unit generically, new descriptors will be ignored by older drivers. The device should have reasonable default behavior when used in conjunction with an older driver.
New control on existing unit/terminal	Expose it generically or ignore it	Can be exposed generically by the host class driver.

Additional probe/commit field	Ignore it (set to default)	The GET_LEN request allows the host class driver to size the probe/commit structure properly. The host is required to always submit requests of the length specified by the device, setting all unknown fields to zero. It must subsequently preserve the values returned by the device when it submits the final set to the Commit control.
Existing control change	Host ignores unknown functionality.	The following guidelines allow forward/backward compatibility to be maintained: • New functionality is in the form of additional fields • The original fields remain semantically and syntactically the same as previously defined. • The GET_LEN request shall become mandatory if it wasn't previously defined for the control. • If the GET_LEN request is supported, the host must submit requests with the length specified by the device (even if GET_LEN was not previously defined for that control). • The GET_DEF request shall become mandatory if it wasn't previously defined for the control. The host will only attempt a GET_DEF request if the GET_LEN

		indicates that there are fields that the host does not understand. If the GET_DEF request is not supported for earlier spec revisions, the device will only return defaults for the fields within the control that were added on or after the revision in which the GET_DEF request became mandatory.
Additional stream header field	Ignore it	Can be supported with current header definition
Payload content change	Cannot currently be supported	The UVC specification version 1.1 provides the necessary versioning by extending the Probe and Commit requests.

2.29 Stream based payload support

Question: Are stream based payloads such as MPEG-4 SL, VC1, H.264 supported by the USB Video Device Class specification?

Answer: Such payloads rely on the MPEG-2 systems specification (ISO/IEC 13818: 2000 / ITU-T Rec. H.222.0).

2.29.1 MPEG-4 SL

The MPEG-2 systems specification (ISO/IEC 13818: 2000 / ITU-T Rec. H.222.0) section 2.11 describes encapsulation of MPEG-4 SL.

2.29.2 VC1

The SMPTE VC1 specification describes encapsulation of VC1 payload in MPEG-2 systems.

2.29.3 H.264

ITU-T Rec. H.222.0 amendment 3 describes encapsulation of H.264 in MPEG-2 systems.

2.30 Host Behavior for Still Image Capture Method 2

Question: My device supports Still Image Capture Method 2 (i.e. video streaming and still image capture via a shared isochronous endpoint). What is the expected host behavior during format negotiation and still image acquisition?

Answer: The host will use the Still Probe and Commit controls to inform the device of the desired still image capture parameters (format, frame size and compression index), and to obtain

the capture buffer and payload transfer sizes that the device will use when transmitting a still image matching those parameters. The Still Probe and Commit negotiation will be performed at least once, but possibly multiple times, prior to still image capture. The number of times is application-dependent. The host is required to perform at least one VS_STILL_COMMIT_CONTROL (SET_CUR) operation prior to the first still image transfer. Also prior to still image capture, the host will determine the *optimal* alternate interface setting for that isochronous endpoint by comparing the **dwMaxPayloadTransferSize** from the Still Probe and Commit control with the **wMaxPacketSize** of each isochronous endpoint descriptor. Although it seems logical that the maximum available isochronous bandwidth would be used during still image transfer, this would not be appropriate if the device is unable to provide still image data at that rate. Therefore, the host will select an alternate interface that most closely matches the **dwMaxPayloadTransferSize** value provided by the device.

When a still image is to be captured, the host will:

1. halt video streaming over the endpoint (if active)
2. select the optimal alternate interface for still image capture
3. trigger the transfer
4. collect the still image bytes
5. restore the previous alternate interface selection
6. restart video streaming (only if previously active)

The host will not change the alternate interface setting if the device is actively streaming *and* the bandwidth reservation of the optimal alternate interface for still image capture would be equal to (or lower than) that of the current alternate setting.

If the bandwidth required for still image capture is not available, the host should gracefully degrade, selecting an alternate interface representing the maximum bandwidth available at that time. Under normal circumstances, this should never be less than the bandwidth already reserved for any active video streaming. The device will only be aware of available bandwidth as a passive recipient of alternate interface selections, so host behavior in low-bandwidth conditions is not addressed here.