MCCI Corporation
3520 Krums Corners Road
Ithaca, New York 14850 USA
Phone +1-607-277-1029
Fax +1-607-277-6844
www.mcci.com

MCCI USB DataPump Video Protocol User's Guide

Engineering Report 950000736
Rev. C
Date: 2011/09/30

PROPRIETARY NOTICE AND DISCLAIMER

Document Release History

Rev. A	2008/05/29	Original release
Rev. B	2011/06/13	Changed document numbers to nine digit versions. DataPump 3.0 Updates.
Rev. C	2011/09/30	Added source code disclaimer.

TABLE OF CONTENTS

LIST OF TABLES

1 Introduction

This document describes architecture of the MCCI Video Class Protocol for the MCCI USB DataPump and specifies APIs to access it.

1.1 References

[DPREF] *MCCI USB DataPump User's Guide*, MCCI Engineering Report 950000066

[USBCORE] *Universal Serial Bus Specificatio*n, version 2.0/3.0 (also referred to as the *USB Specificatio*n). This specification is available on the World Wide Web site **http://www.usb.org**.

[USBRC] *USBRC User's Guide*, MCCI Engineering Report 950000061

[USBVIDEO] *Universal Serial Bus Device Class Definition for Video Devices, Revision 1.1.* This specification is available at https://www.usb.org/developers/devclass.

[VIDEOMJPEG] *Universal Serial Bus Device Class Definition for Video Devices: Motion-JPEG Payload, Revision 1.1.* This specification is available at https://www.usb.org/developers/devclass.

1.2 Goals

This design has the following goals:

1. Must be simple, easy-to-understand and suitable for use by MCCI-supplied clients, by platform-supplied clients, or by OEM-supplied clients.

2. The Video Class API is structured as a pure service, and must be defined in terms of a functional interface, as opposed to explicit messages.

3. The API must support zero-copy transfers (at least for the isoch data transfer paths). Command and response will be zero-copy to the maximum extent convenient (without overly complicating the code)

4. As is normal, outward calls from the service to a client (in this case, from Video Class Protocol to the clients using it) must be done as callbacks using function pointers.

5. Must support high-speed operation, according to the conventions of the high-speed DataPump.

2 Overall Architecture

The intention of this design is to make Video Class Protocol as general as possible and suitable for high-speed operation and easy to use. The configuration and functionality that a USB Video device can have are different from device to device. MCCI Video Class Protocol must be able to support all kinds of USB Video device.

Each instance of Video Class Protocol API needs to have a certain amount of memory for tracking requests, delivering callbacks, etc. Therefore, we will have a DataPump object that represents the API to the Video Class Protocol implementation and to clients. For purposes of discussion, we'll call this object `UPROTO_VIDEO`, and we'll use `pVideo` as a generic pointer of this type.

A `UPROTO_VIDEO` instance creates and maintains Video stream objects when it is created, each Video stream object representing a Video stream interface which has an isochronous/bulk endpoint. All Video stream related APIs require a Video stream object handle, which client gets when it opens Video stream. As there's one-to-one mapping between Video stream interface represented by Video stream object and linked isochronous/bulk endpoint, endpoint related stuff is encapsulated into Video stream object.

3 Error Codes

Video Class Protocol API defines the following error codes.

Table 1. Error codes returned by Video Class Protocol

Name	Code	Meaning
`UPROTO_VIDEO_ERROR_OK`	0	No error occurred, success
`UPROTO_VIDEO_ERROR_INVALID_PARAMETER`	1	An invalid parameter was passed in API call
`UPROTO_VIDEO_ERROR_ALREADY_OPEN`	2	Already opened by another client
`UPROTO_VIDEO_ERROR_NOT_OPEN`	3	Not opened by client
`UPROTO_VIDEO_ERROR_INVALID_REQUEST`	4	An invalid API call in current context
`UPROTO_VIDEO_ERROR_NO_BUFQE`	5	Too many I/O requests currently pending – client can configure number of I/O requests that Video Class Protocol could accept
`UPROTO_VIDEO_ERROR_INTERNAL_ERROR`	6	Internal error happened in Video Class Protocol

Name	Code	Meaning
UPROTO_VIDEO_ERROR_NO_SUCH_STREAM	7	No such Video stream that client wants to open by specifying parameters to Video stream open API
UPROTO_VIDEO_ERROR_IO_BUSY	8	Busy in handling I/O request
UPROTO_VIDEO_ERROR_IO_ERROR	9	Error happened in handling I/O request
UPROTO_VIDEO_ERROR_IO_KILL	10	I/O request was cancelled
UPROTO_VIDEO_ERROR_IO_STALL	11	I/O request failed because endpoint was stalled
UPROTO_VIDEO_ERROR_IO_NOTCFG	12	I/O request failed because endpoint is not in configured state

4 Initialization

4.1 Start-Up (Configuring the DataPump to provide the Video Class Protocol)

The DataPump binds protocols to USB interfaces at startup using the "application initialization vector". This vector specifies the interface class, subclass and protocol, and provides a pointer to the entry point that will create a protocol instance. The DataPump startup code looks at the descriptors and creates instances based on descriptors matches. Generally speaking, a UPROTO_VIDEO instance is provisioned by placing the appropriate entry in the application initialization vector. For more details on how to initialize the DataPump, please refer *[DPREF]*.

The code snippet below is part of Videodemo, an MCCI Video test application, initialization vector. It creates a UPROTO_VIDEO instance if it finds matching interface descriptors (Video class code 0x0E and Video control subclass code 0x01).

```
static CONST USBPUMP_PROTOCOL_INIT_NODE InitNodes[] =
        {
        USBPUMP_PROTOCOL_INIT_NODE_INIT_V2(                             \
            /* dev class, subclass, proto */ -1, -1, -1,                \
            /* ifc class */ USB_bInterfaceClass_Video,                  \
            /* subclass */ USB_bInterfaceSubClass_VideoControl,         \
            /* proto */ -1,                                             \
            /* cfg, ifc, altset */ -1, -1, -1,                          \
            /* speed */ -1,                                             \
            /* uProbeFlags */ USBPUMP_PROTOCOL_INIT_FLAG_AUTO_ADD,      \
            /* probe */ UsbPumpVideo_ProtocolProbe,                     \
```

```
            /* create */ UsbPumpVideo_ProtocolCreate,                  \
            /* pQualifyAddInterface */ NULL,                           \
            /* pAddInterface */ NULL,                                  \
            /* optional info */ (VOID *) &gk_VideoDemo_ProtoConfig     \
            ),
        };
```

4.2 Using the Video Class Protocol API

The platform specific API will need to find the Video Class Protocol API using the standard DataPump APIs. It will look for objects named USBPUMP_OBJECT_NAME_ENUM_VIDEO ("`video.*.fn.mcci.com`").

```
while ((pFunctionObject = UsbPumpObject_EnumerateMatchingNames(
                          &pPumpRoot->Header,
                          pFunctionObject,
                          USBPUMP_OBJECT_NAME_ENUM_VIDEO
                          )) != NULL)
            {
            g_pVidemoDemoContext = VideoDemo_MjpegDemoInit (
                pPlatform,
                pFunctionObject
                );
            }
```

Once located, the platform-specific API uses the object pointer to find the entry points. (For convenience the platform can also use Video Class Protocol APIs direct function calls. The Video Class Protocol library exports Video client library functions named `VideoClientLib_XXXX`)

5 Down-Call APIs

This section describes Video Class Protocol Down-Call APIs.

Please note that the notation of `IN` and `OUT` in parameter list is from the perspective of the client. So, all parameters of callback should be `OUT`.

5.1 VIDEO_CONTROL_OPEN

This API opens the `UPROTO_VIDEO` instance, and sets a number of callback functions to be used for processing asynchronous host operations.

```
typedef VOID
UPROTO_VIDEO_CONTROL_OPEN_FN(
        UPROTO_VIDEO *                      /* pVideo */,
        UPROTO_VIDEO_CONTROL_OPEN_CB_FN *   /* pCallback */,
        VOID *                              /* pCallbackCtx */,
```

```
        VOID *                              /* pClientContext */,
        CONST UPROTO_VIDEO_OUTSWITCH *      /* pOutSwitch */
        );
```

This is an IN function, used by applications to open video protocol instance. The callback function pointer pCallback and data pointer pCallbackCtx are used for combining with variant callback functions. The data pointer pClientContext is used for the context for each of the asynchronous callback functions. The pointer pOutSwitch is used for outswitching of video, which will be called by lower layer to inform application of certain events. Currently available outswitch functions are listed below. This function doesn't return anything.

```
struct UPROTO_VIDEO_OUTSWITCH
        {
        UPROTO_VIDEO_CONTROL_STATUS_EVENT_CB_FN *pControlStatusEventFn;
        UPROTO_VIDEO_CONTROL_REQUEST_CB_FN *     pControlRequestFn;
        UPROTO_VIDEO_STREAM_STATUS_EVENT_CB_FN * pStreamStatusEventFn;
        UPROTO_VIDEO_STREAM_REQUEST_CB_FN *      pStreamRequestFn;
        };
```

On completion of this API, `UPROTO_VIDEO` instance invokes callback, whose function pointer is provided as pCallback parameter. The data pointer pCallbackCtx is used for this callback.

```
typedef VOID
UPROTO_VIDEO_CONTROL_OPEN_CB_FN(
        VOID *                  /* pCallbackCtx */,
        UINT32                  /* ErrorCode */
        );
```

5.2 VIDEO_CONTROL_CLOSE

This API closes the `UPROTO_VIDEO` instance.

```
typedef VOID
UPROTO_VIDEO_CONTROL_CLOSE_FN(
        UPROTO_VIDEO *                      /* pVideo */,
        UPROTO_VIDEO_CONTROL_CLOSE_CB_FN *  /* pCallback */,
        VOID *                              /* pCallbackCtx */
        );
```

The callback function pointer pCallback and data pointer pCallbackCtx are used for combining with variant callback functions. This module is an IN function called by the application to close video protocol instance. This function doesn't return anything.

On completion of this API, `UPROTO_VIDEO` instance invokes the callback function pointed by pCallback parameter, and it passes the pCallbackCtx pointer to it.

```
typedef VOID
UPROTO_VIDEO_CONTROL_CLOSE_CB_FN(
        VOID *                  /* pCallbackCtx */,
        UINT32                  /* ErrorCode */
        );
```

5.3 VIDEO_REQUEST_REPLY

These API functions send a reply to the host's Video control interface, Video stream interface and isochronous/bulk endpoint, as a response to a Video request. Client will receive the reply via its supplied buffer pointed out by pReply (reply buffer pointer). The reply message shall be USB byte-ordered (little endian).

```
typedef VOID
UPROTO_VIDEO_CONTROL_REQUEST_REPLY_FN(
        UPROTO_VIDEO *                                  /* pVideo */,
        UPROTO_VIDEO_CONTROL_REQUEST_REPLY_CB_FN *      /* pCallback */,
        VOID *                                          /* pCallbackCtx */,
        UPROTO_VIDEO_CONTROL_REQUEST *                  /* pReply */
        );

typedef VOID
UPROTO_VIDEO_STREAM_REQUEST_REPLY_FN(
        UPROTO_VIDEO *                                  /* pVideo */,
        UPROTO_VIDEO_STREAM_REQUEST_REPLY_CB_FN *       /* pCallback */,
        VOID *                                          /* pCallbackCtx */,
        UPROTO_VIDEO_STREAM_REQUEST *                   /* pReply */
        );
```

These APIs send the reply data on the control pipe. The callback function will be called after sending data. If there is any error, the error code will be returned by the callback. pCallback is used for the callback function and pCallbackCtx is passed as a parameter for the callback.

UPROTO_VIDEO_ERROR_OK means success. The possible error return values include:

- `UPROTO_VIDEO_ERROR_OK`
- `UPROTO_VIDEO_ERROR_INVALID_PARAMETER`
- `UPROTO_VIDEO_ERROR_NOT_OPEN`
- `UPROTO_VIDEO_ERROR_INVALID_REQUEST`
- `UPROTO_VIDEO_ERROR_NO_BUFQE`

On the completion of this API, `UPROTO_VIDEO` instance invokes callback, whose prototype is provided below.

```
typedef VOID
UPROTO_VIDEO_CONTROL_REQUEST_REPLY_CB_FN(
        VOID *                  /* pCallbackCtx */,
```

```
        UINT32                  /* ErrorCode */
        );

typedef VOID
UPROTO_VIDEO_STREAM_REQUEST_REPLY_CB_FN(
        VOID *                  /* pCallbackCtx */,
        UINT32                  /* ErrorCode */
        );
```

Below are the definitions for control request and stream request:

```
union UPROTO_VIDEO_CONTROL_REQUEST
        {
        UPROTO_VIDEO_CONTROL_REQUEST_HDR                Hdr;
        UPROTO_VIDEO_CONTROL_GET_INFO_REQUEST           GetInfo;
        UPROTO_VIDEO_VC_REQUEST_POWER_MODE_CONTROL      PowerMode;
        UPROTO_VIDEO_VC_REQUEST_ERROR_CODE_CONTROL      ErrorCode;
        UPROTO_VIDEO_CT_SCANNING_MODE_CONTROL           Scanning;
        UPROTO_VIDEO_CT_AE_MODE_CONTROL                 AeMode;
        UPROTO_VIDEO_CT_AE_PRIORITY_CONTROL             AePriority;
        UPROTO_VIDEO_CT_EXPOSURE_TIME_ABSOLUTE_CONTROL  ExpTimeAbs;
        UPROTO_VIDEO_CT_EXPOSURE_TIME_RELATIVE_CONTROL  ExpTimeRel;
        UPROTO_VIDEO_CT_FOCUS_ABSOLUTE_CONTROL          FocusAbs;
        UPROTO_VIDEO_CT_FOCUS_RELATIVE_CONTROL          FocusRel;
        UPROTO_VIDEO_CT_FOCUS_AUTO_CONTROL              FocusAuto;
        UPROTO_VIDEO_CT_IRIS_ABSOLUTE_CONTROL           IrisAbs;
        UPROTO_VIDEO_CT_IRIS_RELATIVE_CONTROL           IrisRel;
        UPROTO_VIDEO_CT_ZOOM_ABSOLUTE_CONTROL           ZoomAbs;
        UPROTO_VIDEO_CT_ZOOM_RELATIVE_CONTROL           ZoomRel;
        UPROTO_VIDEO_CT_PANTILT_ABSOLUTE_CONTROL        PantiltAbs;
        UPROTO_VIDEO_CT_PANTILT_RELATIVE_CONTROL        PantiltRel;
        UPROTO_VIDEO_CT_ROLL_ABSOLUTE_CONTROL           RollAbs;
        UPROTO_VIDEO_CT_ROLL_RELATIVE_CONTROL           RollRel;
        UPROTO_VIDEO_CT_PRIVACY_CONTROL                 Privacy;
        UPROTO_VIDEO_SU_INPUT_SELECT_CONTROL            InputSelect;
        UPROTO_VIDEO_PU_BACKLIGHT_COMPENSATION_CONTROL  Backlight;
        UPROTO_VIDEO_PU_BRIGHTNESS_CONTROL              Brightness;
        UPROTO_VIDEO_PU_CONTRAST_CONTROL                Contrast;
        UPROTO_VIDEO_PU_GAIN_CONTROL                    Gain;
        UPROTO_VIDEO_PU_POWER_LINE_FREQUENCY_CONTROL    PowerLine;
        UPROTO_VIDEO_PU_HUE_CONTROL                     Hue;
        UPROTO_VIDEO_PU_HUE_AUTO_CONTROL                HueAuto;
        UPROTO_VIDEO_PU_SATURATION_CONTROL              Saturation;
        UPROTO_VIDEO_PU_SHARPNESS_CONTROL               Sharpness;
        UPROTO_VIDEO_PU_GAMMA_CONTROL                   Gamma;
        UPROTO_VIDEO_PU_WHITE_BALANCE_TEMPERATURE_CONTROL WhiteTemp;
        UPROTO_VIDEO_PU_WHITE_BALANCE_TEMPERATURE_AUTO_CONTROL  WhiteTempAuto;
        UPROTO_VIDEO_PU_WHITE_BALANCE_COMPONENT_CONTROL  WhiteComp;
        UPROTO_VIDEO_PU_WHITE_BALANCE_COMPONENT_AUTO_CONTROL  WhiteCompAuto;
        UPROTO_VIDEO_PU_DIGITAL_MULTIPLIER_CONTROL      Digital;
```

```
        UPROTO_VIDEO_PU_DIGITAL_MULTIPLIER_LIMIT_CONTROL DigitalLimit;
        UPROTO_VIDEO_PU_ANALOG_VIDEO_STANDARD_CONTROL    AnalogVideo;
        UPROTO_VIDEO_PU_ANALOG_LOCK_STATUS_CONTROL       AnalogLock;
        UPROTO_VIDEO_XU_GENERIC                          Extension;
        };

union UPROTO_VIDEO_STREAM_REQUEST
        {
        UPROTO_VIDEO_STREAM_REQUEST_HDR              Hdr;
        UPROTO_VIDEO_STREAM_GET_INFO_REQUEST         GetInfo;
        UPROTO_VIDEO_VS_PROBE_COMMIT_CONTROL         Video;
        UPROTO_VIDEO_VS_STILL_PROBE_COMMIT_CONTROL   Still;
        UPROTO_VIDEO_VS_SYNCH_DELAY_CONTROL          SyncDelay;
        UPROTO_VIDEO_VS_STILL_IMAGE_TRIGGER_CONTROL  StillImage;
        UPROTO_VIDEO_VS_GENERATE_KEY_FRAME_CONTROL   GenerateKey;
        UPROTO_VIDEO_VS_UPDATE_FRAME_SEGMENT_CONTROL UpdateFrame;
        UPROTO_VIDEO_VS_STREAM_ERROR_CODE_CONTROL    StreamError;
        };
```

For success, ErrorCode parameter is set to UPROTO_VIDEO_ERROR_OK. The possible ErrorCode value includes:

- UPROTO_VIDEO_ERROR_OK
- UPROTO_VIDEO_ERROR_IO_NOTCFG
- UPROTO_VIDEO_ERROR_IO_STALL
- UPROTO_VIDEO_ERROR_IO_KILL
- UPROTO_VIDEO_ERROR_IO_ERROR
- UPROTO_VIDEO_ERROR_INTERNAL_ERROR

5.4 VIDEO_STREAM_OPEN

This API opens the Video stream object, which represents the Video stream interface in the Video interface collection belonging to a UPROTO_VIDEO instance.

```
typedef VOID
UPROTO_VIDEO_STREAM_OPEN_FN(
        UPROTO_VIDEO *                      /* pVideo */,
        UPROTO_VIDEO_STREAM_OPEN_CB_FN *    /* pCallback */,
        VOID *                              /* pCallbackCtx */,
        INT8                                /* bInterfaceNumber */,
        INT8                                /* bAlternateSetting */,
        INT8                                /* bTerminalLink */,
        BOOL                                /* fTrueIfVideoInput */
        );
```

There can be many Video stream objects representing Video stream interfaces belonging to a single UPROTO_VIDEO instance. The client can designate a Video stream object that it wants to open by providing the appropriate parameters.

- `bInterfaceNumber : InterfaceNumber of Video stream interface descriptor`
- `bAlternateSetting : AlternateSetting of Video stream interface descriptor`
- `bTerminalLink : TerminalLink of class-specific Video stream interface descriptor`
- `FTrueIfVideoInput : TRUE if want to open Video Input, (OUT in USB Terminology)`

Client can input –1 for OPTIONAL parameters, in this case, this API ignores those parameters and tries to find matching Video stream object, using only other parameters. This API sets phVideoStream out-parameter to a valid object handle for a Video stream object, only when it returns UPROTO_VIDEO_ERROR_OK.

UPROTO_VIDEO_ERROR_OK is returned for success. The possible return values include:

- `UPROTO_VIDEO_ERROR_OK`
- `UPROTO_VIDEO_ERROR_ALREADY_OPEN`
- `UPROTO_VIDEO_ERROR_NO_SUCH_STREAM`

On completion of this API, UPROTO_VIDEO instance invokes the pCallback function, and it passes pCallbackCtx to the callback.

```
typedef VOID
UPROTO_VIDEO_STREAM_OPEN_CB_FN(
          VOID *                          /* pCallbackCtx */,
          UINT32                          /* ErrorCode */,
          UPROTO_VIDEO_STREAM_HANDLE      /* hVideoStream */
          );
```

5.5 VIDEO_STREAM_CLOSE

This API closes the Video stream object.

```
typedef VOID
UPROTO_VIDEO_STREAM_CLOSE_FN(
          UPROTO_VIDEO *                        /* pVideo */,
          UPROTO_VIDEO_STREAM_CLOSE_CB_FN *     /* pCallback */,
          VOID *                                /* pCallbackCtx */,
          UPROTO_VIDEO_STREAM_HANDLE            /* hVideoStream */
          );
```

pCallback is used as the callback function while UPROTO_VIDEO_STREAM_CLOSE completes. The parameters used by pCallback include the return value of UPROTO_VIDEO_STREAM_CLOSE. UPROTO_VIDEO_ERROR_OK is returned for success. The possible return values are:

- UPROTO_VIDEO_ERROR_OK
- UPROTO_VIDEO_ERROR_INVALID_PARAMETER
- UPROTO_VIDEO_ERROR_NOT_OPEN

On completion of this API, UPROTO_VIDEO instance invokes pCallback function and it passes pCallbackCtx to the callback function.

```
typedef VOID
UPROTO_VIDEO_STREAM_CLOSE_CB_FN(
         VOID *                  /* pCallbackCtx */,
         UINT32                  /* ErrorCode */
         );
```

5.6 VIDEO_STREAM_GET_INFO

This API retrieves information about Video stream object.

```
typedef VOID
UPROTO_VIDEO_STREAM_GET_INFO_FN(
         UPROTO_VIDEO *                        /* pVideo */,
         UPROTO_VIDEO_STREAM_GET_INFO_CB_FN * /* pCallback */,
         VOID *                                /* pCallbackCtx */,
         UPROTO_VIDEO_STREAM_HANDLE            /* hVideoStream */
         );
```

Client can input NULL for OPTIONAL parameters if it doesn't want to get that information. The Error Code will be UPROTO_VIDEO_ERROR_INVALID_PARAMETER if the input parameters are NULL. The parameters and the status of the client's object forces API to set all out-parameters to a valid value only when the Error code is UPROTO_VIDEO_ERROR_OK. The callback function will be called at the completion of this API and it will pass the error code to the client application.

- bInterfaceNumber : InterfaceNumber of Video stream interface descriptor
- bAlternateSetting : AlternateSetting of Video stream interface descriptor
- bTerminalLink : TerminalLink of class-specific Video stream interface descriptor
- wMaxPacketSize : MaxPacketSize of isochronous/bulk endpoint descriptor
- wTransportHeaderSize : TransportHeaderSize of Video stream bulk endpoint descriptor

UPROTO_VIDEO_ERROR_OK is returned for success. The possible return values include:

- UPROTO_VIDEO_ERROR_OK

- UPROTO_VIDEO_ERROR_INVALID_PARAMETER
- UPROTO_VIDEO_ERROR_NOT_OPEN

On completion of this API, UPROTO_VIDEO instance invokes the pCallback function and it passes the pCallbackCtx to the callback.

```
typedef VOID
UPROTO_VIDEO_STREAM_GET_INFO_CB_FN(
          VOID *               /* pCallbackCtx */,
          UINT32               /* ErrorCode */,
          UINT8                /* bInterfaceNumber */,
          UINT8                /* bAlternateSetting */,
          UINT8                /* bTerminalLink */,
          UINT16               /* wMaxPacketSize */,
          UINT16               /* wTransportHeaderSize */
          );
```

5.7 VIDEO_STREAM_READ

This API reads data from the host using the isochronous/bulk endpoint associated with the video stream object, which is specified by hVideoStream parameter. Data is read into client provided buffer (pointed by pBuffer). The size is also specified by the client in nBuffer (buffer size in bytes).

Video stream read/write APIs use scattered buffer mechanism because of the property of USB isochronous/bulk transfer. The major difference between read and write API mechanism is the buffer writer. In case of read, DataPump fills this buffer; in the case of a write the client will fill the buffer.

```
typedef VOID
UPROTO_VIDEO_STREAM_READ_FN(
          UPROTO_VIDEO *                        /* pVideo */,
          UPROTO_VIDEO_STREAM_READ_CB_FN *      /* pCallback */,
          VOID *                                /* pCallbackCtx */,
          UPROTO_VIDEO_STREAM_HANDLE            /* hVideoStream */,
          UINT8 *                               /* pBuffer */,
          BYTES                                 /* nBuffer */
          );
```

UPROTO_VIDEO_ERROR_OK means success. The possible values of error code include:

- UPROTO_VIDEO_ERROR_OK
- UPROTO_VIDEO_ERROR_INVALID_PARAMETER
- UPROTO_VIDEO_ERROR_NOT_OPEN
- UPROTO_VIDEO_ERROR_INVALID_REQUEST
- UPROTO_VIDEO_ERROR_NO_BUFQE

If the input parameters are valid, the stream data will be manipulated and sent to the queue unit. If an error occured, UsbPumpVideoI_StreamRead_Done will be called to indicate the error.

On completion of this API, UPROTO_VIDEO instance invokes callback, whose function pointer is provided by the pStreamReadCallback parameter. The data pointer pCallbackCtx is used for this callback.

```
typedef VOID
UPROTO_VIDEO_STREAM_READ_CB_FN(
         VOID *                  /* pCallbackCtx */,
         UINT32                  /* ErrorCode */,
         UINT8 *                 /* pBuffer */,
         BYTES                   /* nBuffer */,
         BYTES                   /* nActualRead */
         );
```

For success, ErrorCode parameter is set to UPROTO_VIDEO_ERROR_OK and nActualRead is set to the length of the filled buffer. The possible ErrorCode values include:

- UPROTO_VIDEO_ERROR_OK
- UPROTO_VIDEO_ERROR_IO_NOTCFG
- UPROTO_VIDEO_ERROR_IO_STALL
- UPROTO_VIDEO_ERROR_IO_KILL
- UPROTO_VIDEO_ERROR_IO_ERROR
- UPROTO_VIDEO_ERROR_INTERNAL_ERROR

5.8 VIDEO_STREAM_WRITE

This API writes data to the host via the isochronous/bulk endpoint associated with the Video stream object, specified by hVideoStream parameter. The client will provide the data to be written in the buffer pointed by pBuffer (data pointer) and will specify the size using nBuffer (data length in bytes).

```
typedef VOID
UPROTO_VIDEO_STREAM_WRITE_FN(
         UPROTO_VIDEO *                      /* pVideo */,
         UPROTO_VIDEO_STREAM_WRITE_CB_FN *   /* pCallback */,
         VOID *                              /* pCallbackCtx */,
         UPROTO_VIDEO_STREAM_HANDLE          /* hVideoStream */,
         UINT8 *                             /* pBuffer */,
         BYTES                               /* nBuffer */,
         BOOL                                /* fEndOfData */
         );
```

UPROTO_VIDEO_ERROR_OK means success. The possible error return values include:

- UPROTO_VIDEO_ERROR_OK
- UPROTO_VIDEO_ERROR_INVALID_PARAMETER
- UPROTO_VIDEO_ERROR_NOT_OPEN
- UPROTO_VIDEO_ERROR_INVALID_REQUEST
- UPROTO_VIDEO_ERROR_NO_BUFQE

If the input parameters are valid, the stream data will be manipulated and sent to the queue unit.

On the completion of this API, UPROTO_VIDEO instance invokes callback, whose function pointer is provided by pCallback parameter. The data pointer pCallbackCtx is used for this callback. The error code of UPROTO_VIDEO_STREAM_WRITE is one of the parameters of the callback function.

```
typedef VOID
UPROTO_VIDEO_STREAM_WRITE_CB_FN(
         VOID *                  /* pCallbackCtx */,
         UINT32                  /* ErrorCode */,
         UINT8 *                 /* pBuffer */,
         BYTES                   /* nBuffer */
         );
```

For success, ErrorCode parameter is set to UPROTO_VIDEO_ERROR_OK and nActualWrite is set to the length of data buffer which is processed. The possible ErrorCode values include:

- UPROTO_VIDEO_ERROR_OK
- UPROTO_VIDEO_ERROR_IO_NOTCFG
- UPROTO_VIDEO_ERROR_IO_STALL
- UPROTO_VIDEO_ERROR_IO_KILL
- UPROTO_VIDEO_ERROR_IO_ERROR
- UPROTO_VIDEO_ERROR_INTERNAL_ERROR

6 Up-call APIs

This section describes the Video Class Protocol Up-Call APIs. The up-call functions should be provided to UPROTO_VIDEO instance using the pOutSwitch parameter (function pointer table) when the client calls VIDEO_CONTROL_OPEN API. The APIs are later called when asynchronous events occur. If there is no function pointer provided in a given function pointer entry, the UPROTO_VIDEO instance does not perform the up-call (it cannot). In case an API is for informing one of Video class specific request received, UPROTO_VIDEO instance replies with STALL by itself.

6.1 VIDEO_CONTROL_STATUS_EVENT

This API is called when a USB bus status change or Video control interface status change has occured.

```
typedef VOID
UPROTO_VIDEO_CONTROL_STATUS_EVENT_CB_FN(
        VOID *              /* pClientContext */,
        UPROTO_VIDEO_STATUS  /* VideoStatusCode */
        );
```

The possible VideoStatusCode values include:

- `UPROTO_VIDEO_STATUS_START : USB bus start`
- `UPROTO_VIDEO_STATUS_SUSPEND_WITH_WAKEUP : USB bus suspend with wakeup`
- `UPROTO_VIDEO_STATUS_SUSPEND_WITHOUT_WAKEUP : USB bus suspend without wakeup`
- `UPROTO_VIDEO_STATUS_RESUME : USB bus resume`
- `UPROTO_VIDEO_STATUS_RESET : USB bus reset`
- `UPROTO_VIDEO_STATUS_ATTACH : USB cable attach (not always possible)`
- `UPROTO_VIDEO_STATUS_DETACH : USB cable detach (not always possible)`
- `UPROTO_VIDEO_STATUS_ACTIVATE : Video Control Interface configured`
- `UPROTO_VIDEO_STATUS_DEACTIVATE : Video Control Interface not-configured`

6.2 VIDEO_CONTROL_REQUEST

This API is called when the DataPump receives Video class specific control requests. Please refer to *[USBVIDEO]* for the details of possible bRequest, bControlSelector values. For the format of UPROTO_VIDEO_CONTROL_REQUEST refer to section 5.3 of the *[USBVIDEO]* document. The parameters are in USB byte-order (little endian).

```
typedef BOOL
UPROTO_VIDEO_CONTROL_REQUEST(
        UINT8                           /* bRequestType */,
        UINT8                           /* bControlSelector */,
        UINT8                           /* bRequest */,
        UINT16                          /* wLength */,
        UINT8                           /* bAcceptRequest */,
        );
```

When client is handling the requests, it has to reply by calling the VIDEO_REQUEST_REPLY API and then returning TRUE. The client should return FALSE if it does not support the request or the parameter(s) of the request are not valid for the client. This client behavior will cause the UPROTO_VIDEO instance to send an error handshake (STALL PID) to the host.

```
typedef VOID
UPROTO_VIDEO_CONTROL_REQUEST_CB_FN(
        VOID *                          /* pClientContext */,
        UPROTO_VIDEO_CONTROL_REQUEST *  /* pRequest */
        );
```

6.3 VIDEO_STREAM_STATUS_EVENT

This API is called when there's a status change in Video stream interface. The Video stream interface is abstracted by the Video stream object and it has an associated hVideoStream handle. The only possible status changes are "activated" or "deactivate", which will be reflected in the fActivate parameter.

```
typedef VOID
UPROTO_VIDEO_STREAM_STATUS_EVENT_CB_FN(
          VOID *                        /* pClientContext */,
          UPROTO_VIDEO_STREAM_HANDLE    /* hVideoStream */,
          BOOL                          /* fActivate */
          );
```

6.4 VIDEO_STREAM_REQUEST

This API is called when DataPump receives a Video class specific request targeted to Video stream interface. Video stream interface is abstracted by the Video stream object and its associated hVideoStream handle. Please note that the parameters are in USB byte-order (little endian). Refer to *[USBVIDEO]* for the details of bRequest, bControlSelector values.

```
typedef BOOL
UPROTO_VIDEO_STREAM_REQUEST(
          UINT8                             /* bControlSelector */,
          UINT8                             /* bRequest */,
          UINT16                            /* wLength */,
          UINT8                             /* bAcceptRequest */,
          );
```

When client is handling the requests, it has to reply by calling the VIDEO_REQUEST_REPLY API and then returning `TRUE`. The client should return `FALSE` if it does not support the request or the parameter(s) of the request are not valid for the client. This client behavior will cause the UPROTO_VIDEO instance to send an error handshake (STALL PID) to the host.

```
typedef BOOL
UPROTO_VIDEO_STREAM_REQUEST_CB_FN(
          VOID *                            /* pClientContext */,
          UPROTO_VIDEO_STREAM_HANDLE        /* hVideoStream */,
          UPROTO_VIDEO_STREAM_REQUEST *     /* pRequest */
          );
```

7 Demo applications

The DataPump installations contain a Motion-JPEG demo in usbkern/app/videodemo.

7.1 videodemo_device.urc

This URC file is used to generate the device/configuration descriptors using the usbrc.exe tool provided by MCCI. The content of this file should be modified for each application. The proper values should be provided in the VC_INPUT_TERMINAL, VC_PROCESSING_UNIT, VS_FORMAT_MJPEGs and VS_STILL_IMAGE_FRAME descriptors. For reference please see *[USBVIDEO]* for the details of the descriptors and *[USBRC]* for .urc file details.

7.2 VideoDemo_MjpegDemoInit

VideoDemo_ClientCreate() will be called at the initialization stage to search for video protocol instances by UsbPumpObject_EnumerateMatchingNames() and then create client-application related instance by VideoDemo_MjpegDemoInit() if it finds a suitable video protocol instance. VideoDemo_MjpegDemoInit() will set the initial value of application context used to store settings like sharpness, saturation... and so on, and these settings can be manipulated by Video Request from host afterwards. The porting engineer should modify these initial settings for the specific application by changing the constant in videodemo.h.

7.3 VideoDemo_ProcessingUnitEvent

VideoDemo_ProcessingUnitEvent() will be called upon the Video class requests of GET_INFO/ GET_MIN/GET_MAX/GET_RES/GET_DEF/GET_CUR/SET_CUR if the destination is the Processing Unit. The initial setting of the manipulated information is defined in videodemo.h, which must be changed according to different application uses. A concrete function should be added following each SET_CUR request to effectively change the Process Unit's characteristics.

7.4 VideoDemo_StreamInterfaceEvent

VideoDemo_StreamInterfaceEvent() will be called upon special Video class requests if the destination is the Streaming interface. The host and the device can negotiate through a mechanism called PROBE/COMMIT to agree on video streaming or still image settings, as explained in *[USBVIDEO]*. For example, the host can choose among the VS_FORMAT_MJPEGs listed in configuration descriptor, through this mechanism. Once the host confirms settings by issuing SET_CUR on the COMMIT selector, a specific routine to change this characteristic should be called to satisfy the request from the host.

7.5 VideoDemo_WriteOneFrame

Users can use VideoDemo_WriteOneFrame() to send 1 frame data to host. The frequency of sending the frame data and the size of the frame data vary for different applications. These parameters can be negotiated through the PROBE/COMMIT

mechanism explained above. Users should create a task to call VideoDemo_WriteOneFrame() periodically based on the constant frequency setting.

```
VOID
VideoDemo_WriteOneFrame(
          VIDEODEMO_CONTEXT *  pVideoDemoCtx,
          CONST UINT8 *    pData,
          UINT16  wNumberOfByte,
          UINT32  dwPTS,
          UINT16  wSofCounter,
          BOOL    bStillImage,
          BOOL    bContinue
          );
```

pData is the pointer to the starting position of the video streaming/still image data. The length is specified by wNumberOfByte. dwPTS and wSofCounter are the time-stamps mentioned in *[USBVIDEO]* and *[VIDEOMJPEG]*. bStillImage is used to indicate if this frame data is still image data or video stream data, and bContinue should always be set to zero. bContinue is used by VideoDemo_WriteOneFrame_Done() if there is still remaining data and a second try is needed.

The combination of dwPTS, wSofCounter and bStillImage will form a header, which occupies the first 12 bytes for each payload. VideoDemo_WriteOneFrame() will automatically generate these fields, but users should supply the correct dwPTS/wSofCounter/bStillImage values.

Currently, VideoDemo_WriteOneFrame_Done() is called in VideoDemo_StreamStatusEvent() in order to trigger sending the demo frame data to host when the device is attached to host. Upon the completion of the frame data VideoDemo_WriteOneFrame_Done() will call VideoDemo_WriteOneFrame() to send another frame. This sequence is just for demo purpose because there is no another task sending frame data periodically, so users should remove all these function calls and have another task for this purpose.