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 VSC V2 Protocol User's Guide

Engineering Report 950001297
Rev. A
Date: 2013/05/23

Document Release History

Rev. A	2013/05/23	Initial Publication

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

This document describes architecture of the MCCI VSC (Vendor Specific Class) V2 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 Specification*, version 2.0/3.0 (also referred to as the *USB Specification*). This specification is available on the World Wide Web site **http://www.usb.org**.
[USBRC]	*USBRC User's Guide*, MCCI Engineering Report 950000061

1.2 Goals

This design has the following goals:

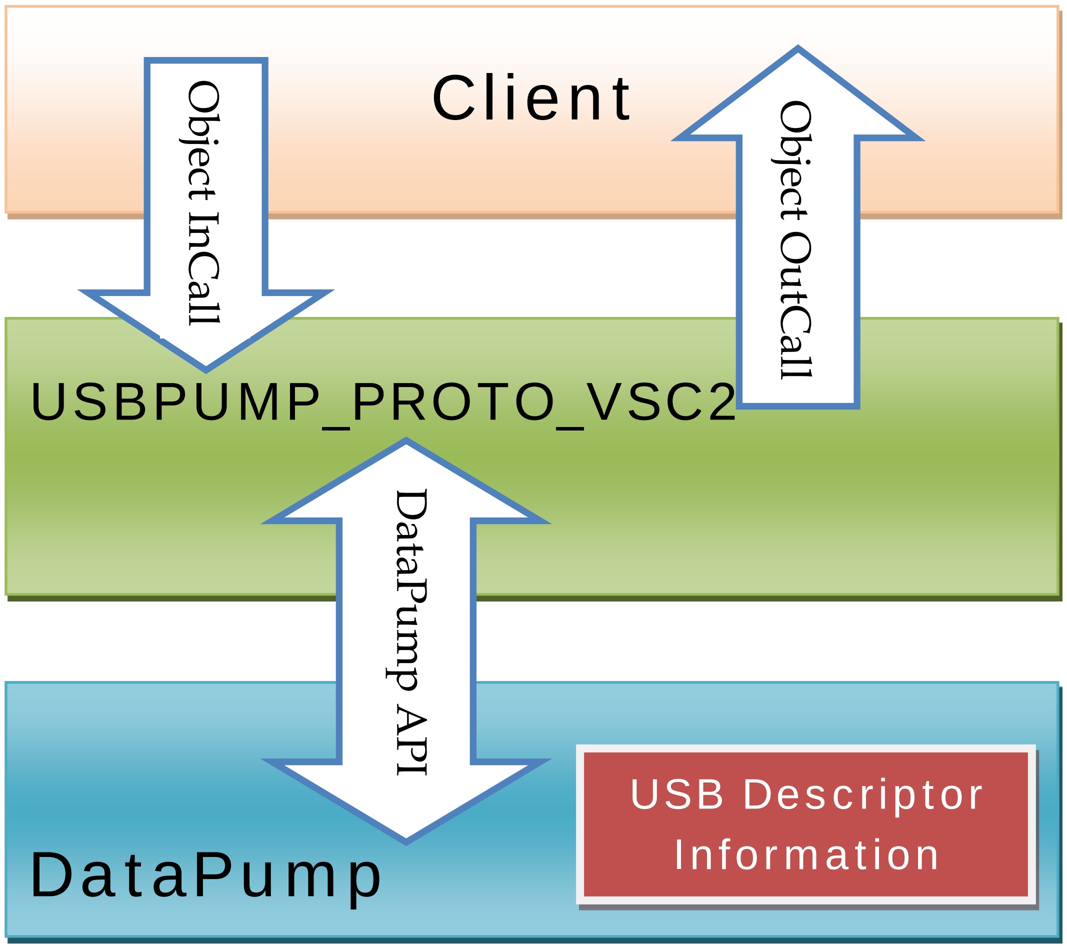
1. Must be simple, easy-to-understand and suitable for use by MCCI-supplied clients, by platform-supplied clients, or by OEM-supplied clients.
2. The Vendor Specific Class API is structured as a pure service, and must be defined in terms of a functional interface, as opposed to explicit messages.
3. 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 Vendor Specific Class Protocol to the clients using it) must be done as outcalls using function pointers.
5. Must support high-speed and super-speed operation, according to the conventions of the high-speed and super-speed DataPump.

2 Overall Architecture

The Vendor Specific Class Protocol V2 is a general-purpose easy-to-use design suitable for high-speed and super-speed operation. The configuration and functionality that a support USB vendor specific interface can have are different from device to device, and variations are supported here. The MCCI Vendor Specific Class Protocol V2 is able to support all kinds of USB devices.

Each instance of Vendor Specific Class Protocol V2 API needs to have a certain amount of memory for tracking requests, delivering callbacks, etc. Therefore, we have a DataPump object that represents the API to the Vendor Specific Class Protocol V2 implementation and to clients. For the purposes of discussion, we call this object `USBPUMP_PROTO_VSC2`, and we use `pProtoVsc` as a generic pointer of this type.

Figure 1. Objects and APIs



3 Initialization

3.1 Start-Up (Configuring the DataPump to provide the VSC protocol V2)

The DataPump binds protocols to USB interfaces at startup using the "application initialization vector". This vector specifies interface class, subclass and protocol, and gives 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 matches. Generally speaking, a `USBPUMP_PROTO_VSC2` instance is provisioned by placing the appropriate entry in the application initialization vector. The code snippet below is part of the initialization vector for the MCCI VSC test application. It creates `USBPUMP_PROTO_VSC2` instance if it finds the matched USB interface descriptor (vendor specific code `0xFF`) from the USB configuration descriptors exported by the device.

```
/*
|| This table provides the initialization information for the protocols
|| that we might load for this device.
*/
static CONST USBPUMP_PROTOCOL_INIT_NODE sk_InitNodes[] =
       {
       USBPUMP_PROTOCOL_INIT_NODE_INIT_V2(                         \
              /* dev class, subclass, proto */ -1, -1, -1,         \
              /* ifc class */ 0xFF,                                \
              /* subclass */ 0,                                    \
              /* proto */ 0,                                       \
              /* cfg, ifc, altset */ -1, -1, -1,                   \
              /* speed */ -1,                                      \
              /* uProbeFlags */ 0,                                 \
              /* probe */ UsbPumpProtoVsc2_Probe,                  \
              /* create */ UsbPumpProtoVsc2_Create,                \
              /* pQualifyAddInterface */ NULL,                     \
              /* pAddInterface */ NULL,                            \
              /* optional info */ &sk_ProtoVsc_Config              \
              )
       };
```

`UsbPumpProtoVsc2_Probe` is a probe function which checks if the matched USB interface is applicable to `USBPUMP_PROTO_VSC2` implementation. If the probe fails, the DataPump will not call `UsbPumpProtoVsc2_Create` to create a `USBPUMP_PROTO_VSC2` instance.

`sk_ProtoVsc_Config` in the above code snippet is a VSC protocol V2 configuration structure which is used while initializing `USBPUMP_PROTO_VSC2` instance. Below is the sample configuration for VSC test application.

```
/* Object name for USBPUMP_PROTO_VSC2 */
CONST TEXT gk_ProtoVsc_ObjectName[] = USBPUMP_PROTO_VSC2_NAME("myObject");

/* Configuration of USBPUMP_PROTO_VSC2 */
static CONST USBPUMP_PROTO_VSC2_CONFIG sk_ProtoVsc_Config =
       USBPUMP_PROTO_VSC2_CONFIG_INIT_V1(
              /* SizeControlBuffer */    512,
              /* pObjectName */          gk_ProtoVsc_ObjectName,
              /* DebugFlags */           0
              );
```

`SizeControlBuffer` specifies the buffer size that will be allocated by the VSC protocol V2. This buffer will be used to send / receive control transfer data.

`pObjectName` specifies name of the VSC protocol V2 instance object. It should not be NULL. Client can use `USBPUMP_PROTO_VSC2_NAME` macro to generate object name.

`DebugFlags` specifies the VSC protocol V2 object debug flags. If `DebugFlags` is zero, it will inherit debug flag of parent object which is UDEVICE object.

3.2 Using the Vendor Specific Class Protocol API

The platform specific API will be needed to find the Vendor Specific Protocol instance object. It can be done using the standard DataPump object APIs. It will look for objects named `USBPUMP_PROTO_VSC2_NAME("myObject")` (or you can use `gk_ProtoVsc_ObjectName`).

```
MY_VSC_CONTEXT *
VscDemo_Create(
          UPLATFORM *     pPlatform
          )
          {
          USBPUMP_OBJECT_ROOT * CONST  pRootObject =
              UsbPumpObject_GetRoot(&pPlatform->upf_Header);
          USBPUMP_OBJECT_HEADER *      pObjectHeader;
          MY_VSC_CONTEXT *             pSelf;
          RECSIZE                      SizeOpenMemory;
          VOID *                       pOpenMemory;

          /*
          || Find VSC protocol V2 instance object.
          */
          pObjectHeader = UsbPumpObject_EnumerateMatchingNames(
                      &pRootObject->Header,
                      NULL,
                      gk_ProtoVsc_ObjectName
                      );

          If (pObjectHeader == NULL)
              {
              /* No VSC protocol instance found. Nothing to do. */
              TTUSB_OBJPRINTF((
                  &pPlatform->upf_Header,
                  UDMASK_ERRORS,
                  "? VscDemo_Create:"
                  " can’t find VSC protocol instance!\n"
                  ));
              return NULL;
              }

          /*
          || Allocate my VSC context
```

```
*/
pSelf = UsbPumpPlatform_MallocZero(pPlatform, sizeof(*pSelf));
if (pSelf == NULL)
    {
    TTUSB_OBJPRINTF((
        pObjectHeader,
        UDMASK_ERRORS,
        "?VscDemo_Create:"
        " pSelf allocation fail\n"
        ));
    return NULL;
    }

SizeOpenMemory = UsbPumpObject_SizeOpenSessionRequestMemory(
            pObjectHeader
            );
pOpenMemory = UsbPumpPlatform_Malloc(pPlatform, SizeOpenMemory);
if (pOpenMemory == NULL)
    {
    TTUSB_OBJPRINTF((
        pObjectHeader,
        UDMASK_ERRORS,
        "?VscDemo_Create:"
        " pOpenMemory allocation fail\n"
        ));

    UsbPumpPlatform_Free(
        pPlatform,
        pSelf,
        sizeof(*pSelf)
        );
    return NULL;
    }

pSelf->pVscObject = pObjectHeader;
pSelf->pPlatform = pPlatform;

TTUSB_OBJPRINTF((
    pObjectHeader,
    UDMASK_FLOW,
    " VscDemo_Create:"
    " pSelf(%p) binding with pObjectHeader(%p)\n",
    pSelf,
    pObjectHeader
    ));

UsbPumpObject_OpenSession(
    pObjectHeader,
    pOpenMemory,
    SizeOpenMemory,
    VscDemoI_OpenSession_Callback,
    pSelf,  /* pCallBackContext */
    &gk_UsbPumpProtoVsc2_Guid,
    NULL,   /* pClientObject -- OPTIONAL */
```

```
              &pSelf->VscInCall.GenericCast,
              sizeof(pSelf->VscInCall),
              pSelf,  /* pClientHandle */
              &sk_VscDemo_OutCall.GenericCast,
              sizeof(sk_VscDemo_OutCall)
              );

          return pSelf;
          }

VOID
VscDemoI_OpenSession_Callback(
          VOID *          pClientContext,
          USBPUMP_SESSION_HANDLE   SessionHandle,
          UINT32          Status,
          VOID *          pOpenRequestMemory,
          RECSIZE         sizeOpenRequestMemory
          )
          {
          MY_VSC_CONTEXT * CONST   pSelf = pClientContext;

          TTUSB_OBJPRINTF((
              pSelf->pVscObject,
              UDMASK_ENTRY,
              "+VscDemoI_OpenSession_Callback: Status=%d\n",
              Status
              ));

          if (pOpenRequestMemory)
              {
              UsbPumpPlatform_Free(
                  pSelf->pPlatform,
                  pOpenRequestMemory,
                  sizeOpenRequestMemory
                  );
              }

          if (USBPUMP_PROTO_AUDIO_STATUS_FAILED(Status))
              {
              TTUSB_OBJPRINTF((
                  pSelf->pVscObject,
                  UDMASK_ERRORS,
                  "?VscDemoI_OpenSession_Callback:"
                  " failed to open a VSC session (%d)\n",
                  Status
                  ));
              return;
              }

          /* Save SessionHandle */
          pSelf->SessionHandle = SessionHandle;

          /* Do more work here... For example, Open Streams */
          }
```

The client finds Vendor Specific Class protocol V2 instance objects using the DataPump object API `UsbPumpObject_EnumerateMatchingNames()`. If it finds a VSC protocol V2 instance object, the client should call `UsbPumpObject_OpenSession()` API to open a session of the `USBPUMP_PROTO_VSC2` instance. With the open, the client receives VSC InCall functions and sets OutCall functions to be used for processing asynchronous host operations. The data pointer `pSelf` is used for the context of each of the OutCall functions.

The client needs to save the returned `SessionHandle` in the `VscDemoI_OpenSession_Callback`. This `SessionHandle` will be used when calling InCall functions.

4 Error Codes

The Vendor Specific Class protocol V2 defines the following error codes.

Table 1. Error codes returned by VSC protocol V2

Name	Code	Meaning
USBPUMP_PROTO_VSC2_STATUS_OK	0x0	No error occurred, success
USBPUMP_PROTO_VSC2_STATUS_INVALID_PARAMETER	0x1	If parameter passed is invalid
USBPUMP_PROTO_VSC2_STATUS_BUFFER_TOO_SMALL	0x3	If sizeInCallApiBuffer is too small
USBPUMP_PROTO_VSC2_STATUS_INVALID_SESSION_HANDLE	0x6	If the session handle is invalid
USBPUMP_PROTO_VSC2_STATUS_INVALID_STREAM_HANDLE	0x100	If an invalid stream handle is passed
USBPUMP_PROTO_VSC2_STATUS_ALREADY_OPENED	0x101	If a session is already opended by another client
USBPUMP_PROTO_VSC2_STATUS_NO_STREAM_TO_OPEN	0x102	If there is no stream to open
USBPUMP_PROTO_VSC2_STATUS_INVALID_REQUEST	0x103	If a control reply request is invalid
USBPUMP_PROTO_VSC2_STATUS_NOT_CONFIGURED	0x104	If an I/O request failed because interface is not configured state

5 InCall APIs

This section describes the Vendor Specific Class protocol InCall APIs. The client should get InCall functions using `UsbPumpObject_OpenSession()` API. Below is the VSC protocol InCall structure definition.

```
typedef struct __TMS_USBPUMP_PROTO_VSC2_INCALL_CONTENTS
       {
       USBPUMP_API_INCALL_HEADER;

       /* VSC protocol in-calls */
       USBPUMP_PROTO_VSC2_OPEN_STREAM_FN *     pOpenStreamFn;
       USBPUMP_PROTO_VSC2_CLOSE_STREAM_FN *    pCloseStreamFn;
       USBPUMP_PROTO_VSC2_CONTROL_REPLY_FN *   pControlReplyFn;
       USBPUMP_PROTO_VSC2_SUBMIT_REQUEST_FN *  pSubmitRequestFn;

       USBPUMP_API_INCALL_TRAILER;
       } USBPUMP_PROTO_VSC2_INCALL_CONTENTS;

typedef union __TMS_USBPUMP_PROTO_VSC2_INCALL
       {
       USBPUMP_PROTO_VSC2_INCALL_CONTENTS      Vsc;
       USBPUMP_API_INCALL_CONTENTS__UNION;
       } USBPUMP_PROTO_VSC2_INCALL;
```

5.1 USBPUMP_PROTO_VSC2_OPEN_STREAM_FN

This API opens data stream associated with a single pipe. The client should call this API in the DataPump context after open session complete. This API allows callers to discover the pipes that are from a specified UINTERFACE. `BindingFlags`, `EpAddrMask`, and `PipeOrdinal` are used to select a subset of the pipes, based on caller's preferences.

```
       typedef USBPUMP_PROTO_VSC2_STATUS
       (*USBPUMP_PROTO_VSC2_OPEN_STREAM_FN)(
              USBPUMP_SESSION_HANDLE                 SessionHandle,
              UINT                                   BindingFlags,       // IN
              UINT                                   EpAddrMask,         // IN
              UINT                                   PipeOrdinal,        // IN
              USBPUMP_PROTO_VSC2_STREAM_HANDLE *     pStreamHandle       // OUT
              );
```

`BindingFlags` specifies the type (or types) of pipes that are acceptable. The available types are:

```
       UPIPE_SETTING_MASK_ISO_OUT
       UPIPE_SETTING_MASK_ISO_IN
       UPIPE_SETTING_MASK_BULK_OUT
       UPIPE_SETTING_MASK_BULK_IN
       UPIPE_SETTING_MASK_INT_OUT
       UPIPE_SETTING_MASK_INT_IN
```

specifying the obvious pipe types. These bit values may be OR'ed together. The API also allows:

```
UPIPE_SETTING_MASK_CONTROL_OUT
UPIPE_SETTING_MASK_CONTROL_IN
```

but these are not fully supported because it is default pipe. However, CONTROL_IN means the to-host half of a control endpoint; CONTROL_OUT means the from-host half.

Because flags may be or'ed together, it's possible to accept multiple kinds of pipes with a single binding. The following abbreviations are defined:

```
UPIPE_SETTING_MASK_ANY     matches any possible pipe.
UPIPE_SETTING_MASK_DATA    matches any data pipe (iso, bulk, int) x (in, out)
UPIPE_SETTING_MASK_DATA_IN matches for any IN data pipe (iso, bulk, int) x in
UPIPE_SETTING_MASK_DATA_OUT matches any OUT data pipe (iso, bulk, int) x out
UPIPE_SETTING_MASK_BULKINT_IN matches any bulk or int IN pipe (bulk, int) x in
UPIPE_SETTING_MASK_BULKINT_OUT stand for bulk or int OUT pipe (bulk, int) x out
```

After considering the endpoint binding flags, the endpoint address is checked. If `EpAddrMask` is zero, then any endpoint can match. Otherwise, `EpAddrMask` is a bit mask, where bit[0] corresponds to endpoint address 0, bit[1] corresponds to endpoint address 1, and so forth; endpoints are matched by setting the bit corresponding to the **desired** endpoint(s).

`PipeOrdinal` is used to select sequential matching pipes. This is a zero-based index.

This API returns `USBPUMP_PROTO_VSC2_STATUS_OK` if stream opened successfully. Possible error codes are

- `USBPUMP_PROTO_VSC2_STATUS_INVALID_SESSION_HANDLE`
- `USBPUMP_PROTO_VSC2_STATUS_INVALID_PARAMETER`
- `USBPUMP_PROTO_VSC2_STATUS_NO_STREAM_TO_OPEN`

5.2 USBPUMP_PROTO_VSC2_CLOSE_STREAM_FN

This API closes the opened data stream. The client should call this API in the DataPump context.

```
typedef USBPUMP_PROTO_VSC2_STATUS
(*USBPUMP_PROTO_VSC2_CLOSE_STREAM_FN)(
      USBPUMP_SESSION_HANDLE           SessionHandle,
      USBPUMP_PROTO_VSC2_STREAM_HANDLE  StreamHandle
      );
```

This API returns `USBPUMP_PROTO_VSC2_STATUS_OK` if stream closed successfully. Possible error codes are

- `USBPUMP_PROTO_VSC2_STATUS_INVALID_SESSION_HANDLE`
- `USBPUMP_PROTO_VSC2_STATUS_INVALID_STREAM_HANDLE`

5.3 USBPUMP_PROTO_VSC2_CONTROL_REPLY_FN

This API handles the reply of a vendor specific class control request to the VSC protocol V2 driver. This API will be used to send control request data or status to the host.

```
typedef USBPUMP_PROTO_VSC2_STATUS
(*USBPUMP_PROTO_VSC2_CONTROL_REPLY_FN)(
      USBPUMP_SESSION_HANDLE     SessionHandle,
      VOID *                     pReplyBuffer,
      UINT16                     nReplyBuffer
      );
```

To send control request status OK (or to send a zero length packet / ZLP), the client should set `pReplyBuffer` to not NULL and set `nReplyBuffer` to zero. To send STALL, the client should set `pReplyBuffer` to NULL and `nReplyBuffer` to zero. The caller should ensure that `pReplyBuffer` is the same data buffer as the `USBPUMP_PROTO_VSC2_SETUP_PROCESS_FN()` callback.

This API returns `USBPUMP_PROTO_VSC2_STATUS_OK` if sent the reply of control request successfully. Possible error codes are

- `USBPUMP_PROTO_VSC2_STATUS_INVALID_SESSION_HANDLE`
- `USBPUMP_PROTO_VSC2_STATUS_INVALID_ REQUEST`

USBPUMP_PROTO_VSC2_SUBMIT_REQUEST_FN

This API submits data transfer request to the VSC protocol V2 driver. This API may be called from the arbitration context.

The client should provide `pRequest->Qe.uqe_donefn`. This API always calls `pRequest->Qe.uqe_donefn` function when either the request complete or an error happens. If this API returns `USBPUMP_PROTO_VSC2_STATUS_OK`, then the client should wait `pRequest->Qe.uqe_donefn` to be called. If this API returns not `USBPUMP_PROTO_VSC2_STATUS_OK`, `pRequest->Qe.uqe_donefn` will not be called.

In general, if there is no input parameter error, `pRequest->Qe.uqe_donefn` will be called from the DataPump context.

```
typedef USBPUMP_PROTO_VSC2_STATUS
(*USBPUMP_PROTO_VSC2_SUBMIT_REQUEST_FN)(
      USBPUMP_SESSION_HANDLE            SessionHandle,
      USBPUMP_PROTO_VSC2_REQUEST *      pRequest
      );
```

This API returns `USBPUMP_PROTO_VSC2_STATUS_OK` if submitted the I/O request successfully. Possible error codes are

- `USBPUMP_PROTO_VSC2_STATUS_INVALID_SESSION_HANDLE`
- `USBPUMP_PROTO_VSC2_STATUS_INVALID_STREAM_HANDLE`

- USBPUMP_PROTO_VSC2_STATUS_INVALID_PARAMETER
- USBPUMP_PROTO_VSC2_STATUS_NOT_CONFIGURED

USBPUMP_PROTO_VSC2_REQUEST is a wrapper structre of UBUFQE_GENERIC. The client needs to allocate this request and submit the request to the VSC protocol V2 instance.

```
typedef struct __TMS_USBPUMP_PROTO_VSC2_REQUEST
        {
        USBPUMP_PROTO_VSC2_STREAM_HANDLE hStream;
        UBUFQE_GENERIC                   Qe;
        /* VSC protocol uses below variables */
        USBPUMP_PROTO_VSC2 *             pSelf;
        UCALLBACKCOMPLETION              Callback;
        } USBPUMP_PROTO_VSC2_REQUEST;
```

The UsbPumpProtoVscRequest_PrepareLegacy() macro helps to initialize the legacy mode UBUFQE of the USBPUMP_PROTO_VSC2_REQUEST structure.

```
UsbPumpProtoVscRequest_PrepareLegacy(
        pRequest,
        hStream,
        pBuffer,
        hBuffer,
        nBuffer,
        TransferFlags,
        pDoneFn,
        pDoneCtx
        );
```

The UsbPumpProtoVscRequest_PrepareIsochIn/Out()macros help to initialize the isochronous transfer mode UBUFQE of the USBPUMP_PROTO_VSC2_REQUEST structure.

```
UsbPumpProtoVscRequest_PrepareIsochIn(
        pRequest,
        hStream,
        pBuffer,
        hBuffer,
        nBuffer,
        TransferFlags,
        pIsochDescr,
        hIsochDescr,
        nIsochDescr,
        IsochStartFrame,
        pDoneFn,
        pDoneCtx
        );
```

6 OutCall APIs

This section describes the Vendor Specific Class protocol V2 OutCall APIs. These OutCall functions should be provided to the USBPUMP_PROTO_VSC2 instance through the

`UsbPumpObject_OpenSession()` API. Below is the VSC protocol V2 OutCall structure definition and initialization macro.

```
typedef struct __TMS_USBPUMP_PROTO_VSC2_OUTCALL_CONTENTS
       {
       USBPUMP_API_OUTCALL_HEADER;

       USBPUMP_PROTO_VSC2_EVENT_FN *           pEventFn;
       USBPUMP_PROTO_VSC2_SETUP_VALIDATE_FN *  pSetupValidateFn;
       USBPUMP_PROTO_VSC2_SETUP_PROCESS_FN *   pSetupProcessFn;

       USBPUMP_API_OUTCALL_TRAILER;
       } USBPUMP_PROTO_VSC2_OUTCALL_CONTENTS;

typedef union __TMS_USBPUMP_PROTO_VSC2_OUTCALL
       {
       __TMS_USBPUMP_PROTO_VSC2_OUTCALL_CONTENTS      Vsc;
       __TMS_USBPUMP_API_OUTCALL_CONTENTS__UNION;
       } USBPUMP_PROTO_VSC2_OUTCALL;

/* compile-time initializer */
USBPUMP_PROTO_VSC2_OUTCALL_INIT_V1(
       USBPUMP_PROTO_VSC2_EVENT_FN *pEventFn,
       USBPUMP_PROTO_VSC2_SETUP_VALIDATE_FN *pSetupValidateFn,
       USBPUMP_PROTO_VSC2_SETUP_PROCESS_FN *pSetupProcessFn
       );

/* run-time initializer */
USBPUMP_PROTO_VSC2_OUTCALL_SETUP_V1(
       USBPUMP_PROTO_VSC2_OUTCALL *pOutCall,
       USBPUMP_PROTO_VSC2_EVENT_FN *pEventFn,
       USBPUMP_PROTO_VSC2_SETUP_VALIDATE_FN *pSetupValidateFn,
       USBPUMP_PROTO_VSC2_SETUP_PROCESS_FN *pSetupProcessFn
       );
```

All these OutCall functions will be called from the DataPump context. The client must post event to the original task if it is required.

6.1 USBPUMP_PROTO_VSC2_EVENT_FN

This API will be called by the VSC protocol V2 driver to deliver a VSC protocol event to the registered client.

```
typedef VOID
(*USBPUMP_PROTO_VSC2_EVENT_FN)(
      VOID *                     ClientHandle,
      USBPUMP_PROTO_VSC2_EVENT   Event,
      CONST VOID *               pEventInfo
      );
```

`ClientHandle` is the client's context pointer that was provided by the client through the `UsbPumpObject_OpenSession()` API. The `USBPUMP_PROTO_VSC2_EVENT` codes are defined as below:

```
USBPUMP_PROTO_VSC2_EVENT_INTERFACE_UP   VSC interface is configured
USBPUMP_PROTO_VSC2_EVENT_INTERFACE_DOWN VSC interface is not configured
USBPUMP_PROTO_VSC2_EVENT_SUSPEND        USB bus is suspended
USBPUMP_PROTO_VSC2_EVENT_RESUME         USB bus is resumed
USBPUMP_PROTO_VSC2_EVENT_RESET          USB bus reset
USBPUMP_PROTO_VSC2_EVENT_ATTACH         USB cable attached
USBPUMP_PROTO_VSC2_EVENT_DETACH         USB cable detached
```

The `pEventInfo` contains extra event information. If there is no extra event information, the VSC protocol will pass NULL. The following structure are used for extra event information for the specific event.

```
typedef struct __TMS_USBPUMP_PROTO_VSC2_EVENT_INTERFACE_UP_INFO
        {
        UINT            CurrentAltSetting;
        } USBPUMP_PROTO_VSC2_EVENT_INTERFACE_UP_INFO;

typedef struct __TMS_USBPUMP_PROTO_VSC2_EVENT_SUSPEND_INFO
        {
        BOOL            fRemoteWakeEnabled;
        } USBPUMP_PROTO_VSC2_EVENT_SUSPEND_INFO;
```

6.2 USBPUMP_PROTO_VSC2_SETUP_VALIDATE_FN

This API validates a Vendor Specific Class control request. It will be called by the VSC protocol V2 driver when it receive vendor specific request from host. The client providing this function should validate the vendor specific control request. If the client can accept this control request, the client should return `USBPUMP_PROTO_VSC2_SETUP_STATUS_ACCEPTED`. If this control request is unknown (the client can't accept this control request), the client should return `USBPUMP_PROTO_VSC2_SETUP_STATUS_NOT_CLAIMED`. If the client recognizes this control request but wants to reject this control request for some reason, the client should return `USBPUMP_PROTO_VSC2_SETUP_STATUS_REJECTED`.

If the client accepts this vendor specific control request, then the client will subsequently receive a `USBPUMP_PROTO_VSC2_SETUP_PROCESS_FN()` callback to process the accepted control request.

If the client returns `USBPUMP_PROTO_VSC2_SETUP_STATUS_REJECTED`, the VSC protocol will send STALL.

If the client returns `USBPUMP_PROTO_VSC2_SETUP_STATUS_NOT_CLAIMED`, the protocol will do nothing for this vendor specific command and DataPump core will handle this command.

```
typedef USBPUMP_PROTO_VSC2_SETUP_STATUS
(*USBPUMP_PROTO_VSC2_SETUP_VALIDATE_FN)(
        VOID *          ClientHandle,
        USETUP *        pSetup
```

```
          );
```

Defined `USBPUMP_PROTO_VSC2_SETUP_STATUS` code are

```
USBPUMP_PROTO_VSC2_SETUP_STATUS_ACCEPTED
USBPUMP_PROTO_VSC2_SETUP_STATUS_REJECTED
USBPUMP_PROTO_VSC2_SETUP_STATUS_NOT_CLAIMED
```

6.3 USBPUMP_PROTO_VSC2_SETUP_PROCESS_FN

This API processes Vendor Specific Class control request. It will be called by the VSC protocol V2 driver to let client process vendor specific control request.

If the direction of the vendor request is from host to device, the VSC protocol receives the data from the host, passing the received data in `pBuffer`, with the count of valid data indicated by `nBuffer`. The client should send its reply using `USBPUMP_PROTO_VSC2_CONTROL_REPLY_FN()`.

If the direction of the vendor request is from device to host, the VSC protocol provides a data buffer `pBuffer` of size `nBuffer`. The client processes the control request and copies the resulting data to the `pBuffer`, setting `nBuffer` to the actual count of valid response data bytes. Then the client sends the reply using `USBPUMP_PROTO_VSC2_CONTROL_REPLY_FN()`.

If this process function returns FALSE, the VSC protocol driver will send STALL for this setup packet. If it returns TRUE, this indicates the client has handled this setup packet.

```
typedef BOOL
(*USBPUMP_PROTO_VSC2_SETUP_PROCESS_FN)(
      VOID *             ClientHandle,
      USETUP *           pSetup,
      VOID *             pBuffer,
      UINT16             nBuffer
      );
```

7 Sequence Diagrams

The sequence diagrams depicting the operation of these APIs are shown in Figure 2 through Figure 5. In these diagrams the details of USB bus level have been omitted. Since USB mixes layer 2 & layer 3 at the hardware level, it is difficult to draw the sequence diagrams that are simultaneously comprehensible and fully accurate. However, these diagrams still provide adequate details of the API operations.

The names of the function in the diagram are mere suggestions and are not specified by this document. The client is the agent that issues calls based on activities elsewhere in the system. At client level, the code resides within the DataPump task, but it can still receive messages from anywhere in the system. The message receive mechanisms are not specified in this document and are likely to vary from platform to platform.

Figure 2. Sequence Diagram for open VSC protocol instance

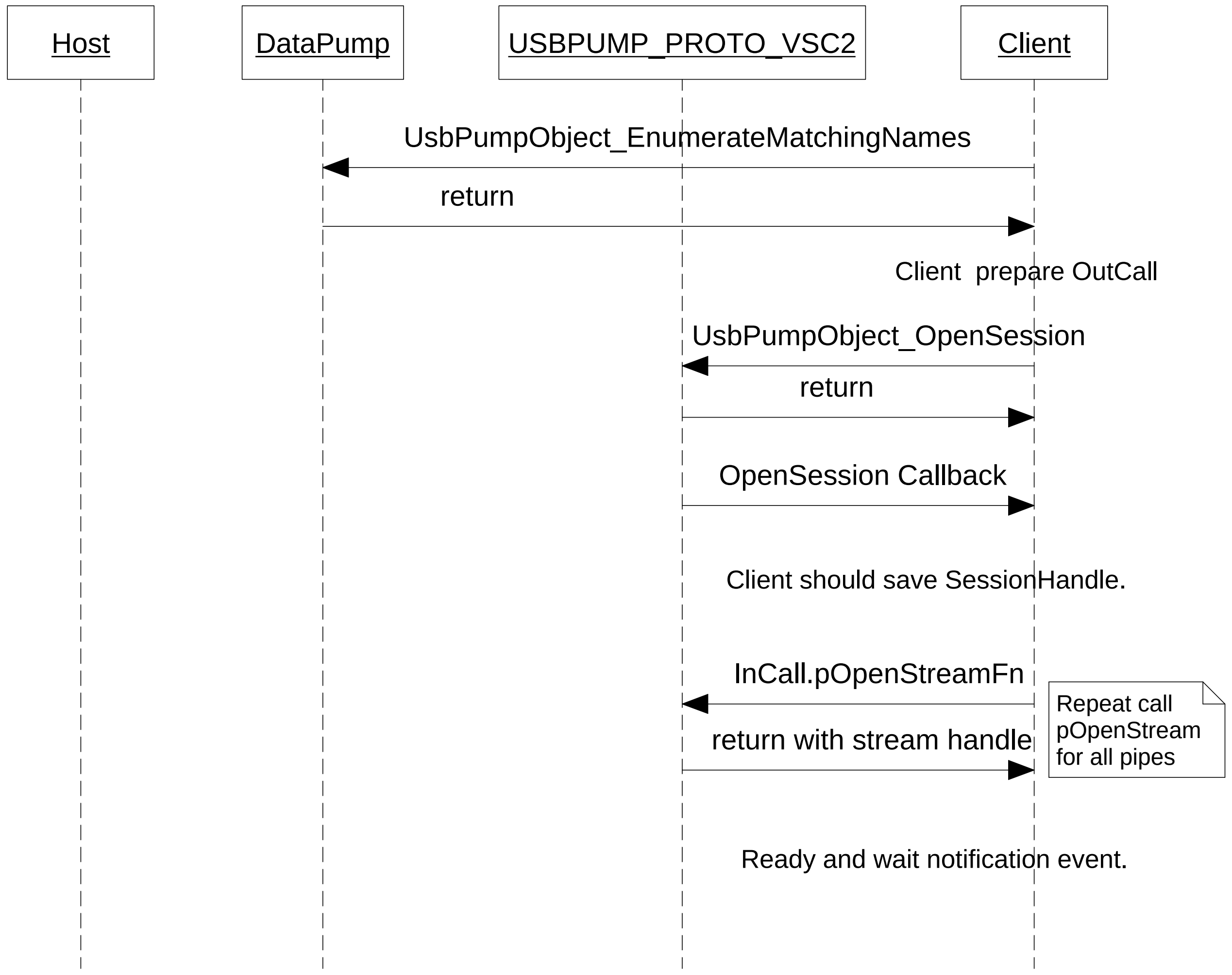

Host
DataPump
USBPUMP_PROTO_VSC2
Client
UsbPumpObject_EnumerateMatchingNames
return
Client prepare OutCall
UsbPumpObject_OpenSession
return
OpenSession Callback
Client should save SessionHandle.
InCall.pOpenStreamFn
return with stream handle
Repeat call pOpenStream for all pipes
Ready and wait notification event.

Figure 3. Sequence Diagram for VSC control request

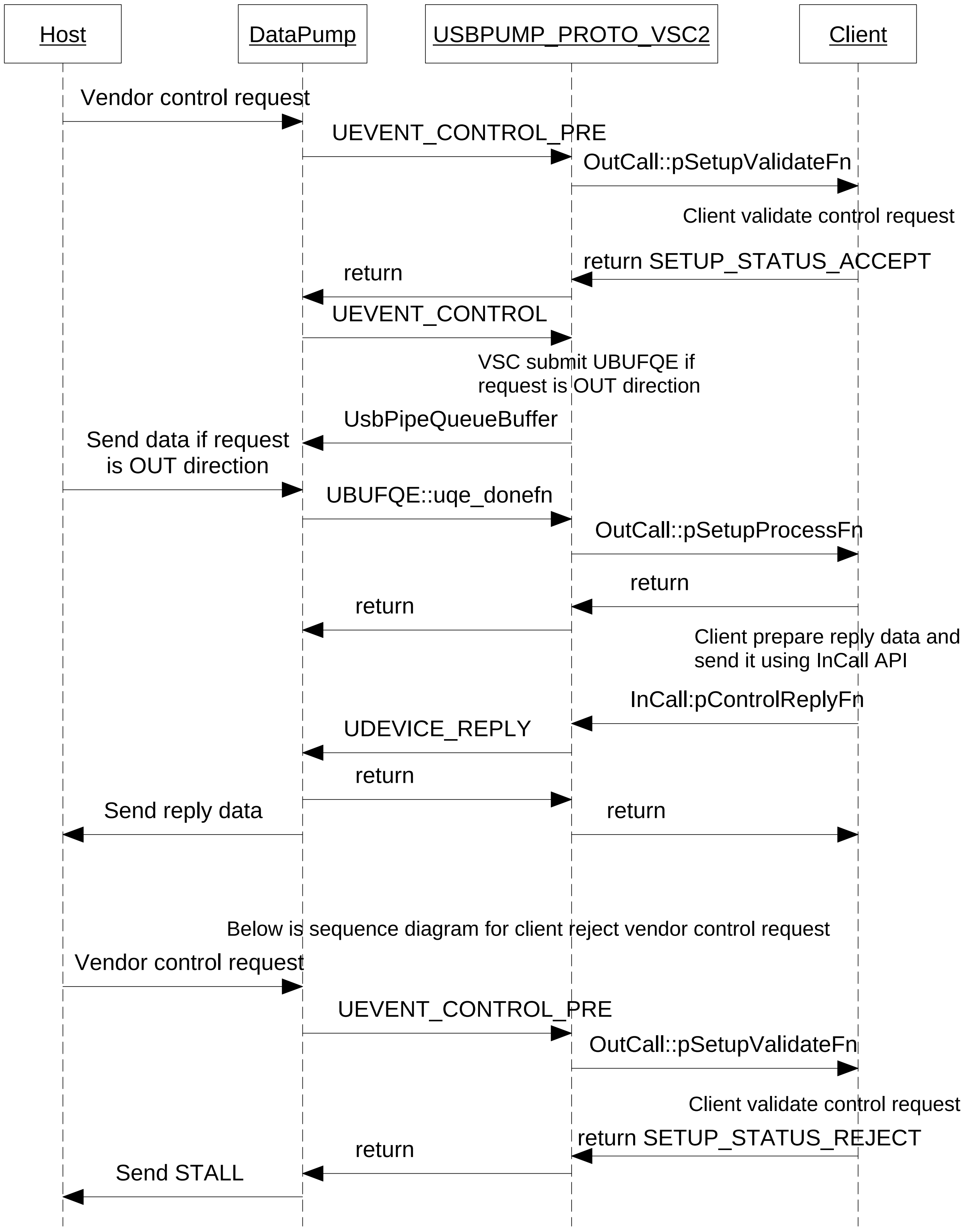

Host
DataPump
USBPUMP_PROTO_VSC2
Client
Vendor control request
UEVENT_CONTROL_PRE
OutCall::pSetupValidateFn
Client validate control request
return SETUP_STATUS_ACCEPT
return
UEVENT_CONTROL
VSC submit UBUFQE if request is OUT direction
UsbPipeQueueBuffer
Send data if request is OUT direction
UBUFQE::uqe_donefn
OutCall::pSetupProcessFn
return
return
Client prepare reply data and send it using InCall API
InCall:pControlReplyFn
UDEVICE_REPLY
return
Send reply data
return
Below is sequence diagram for client reject vendor control request
Vendor control request
UEVENT_CONTROL_PRE
OutCall::pSetupValidateFn
Client validate control request
return SETUP_STATUS_REJECT
return
Send STALL

Figure 4. Sequence Diagram for VSC event notification

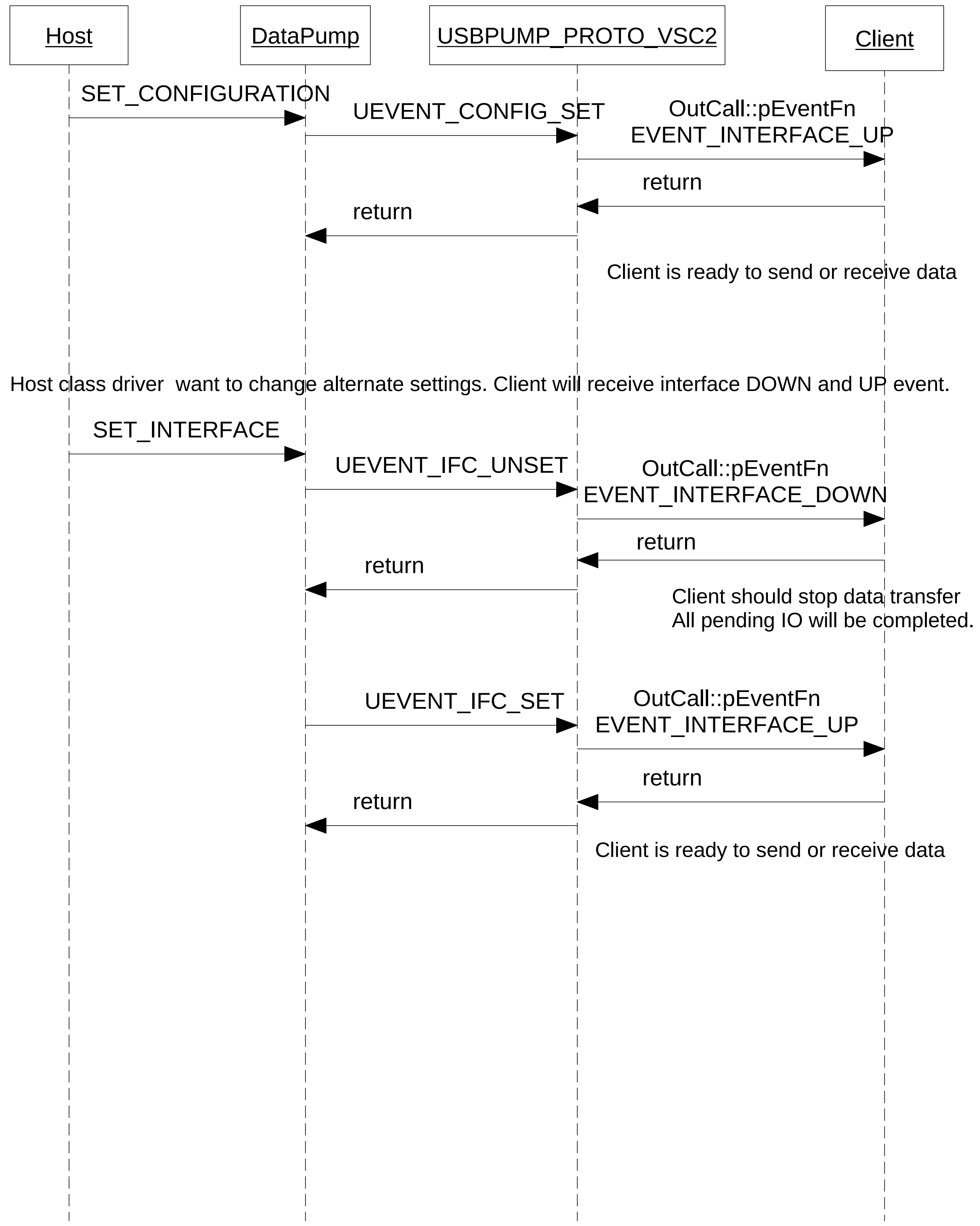
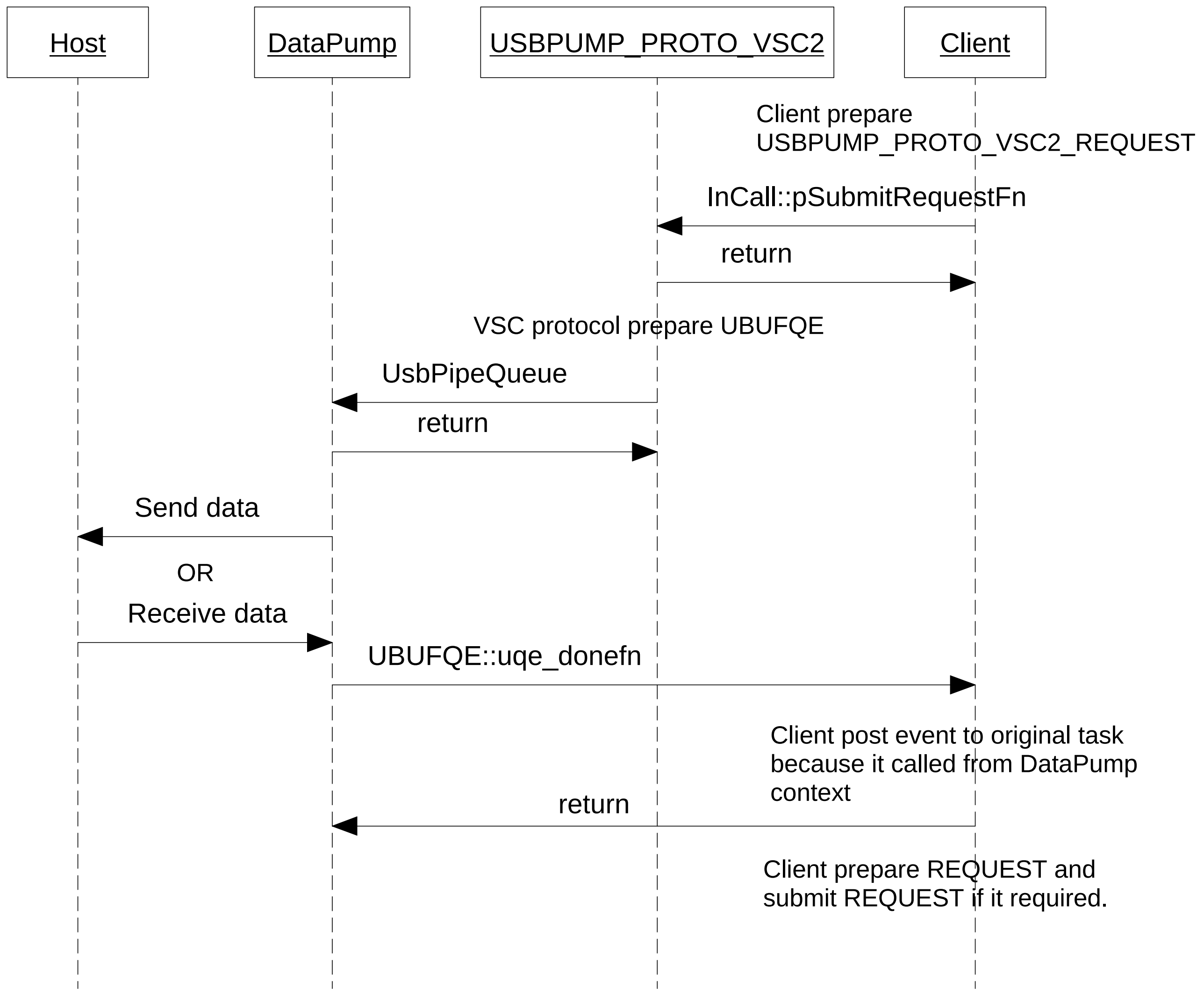


Figure 5. Sequence Diagram for VSC data transfer


Host
DataPump
USBPUMP_PROTO_VSC2
Client
Client prepare USBPUMP_PROTO_VSC2_REQUEST
InCall::pSubmitRequestFn
return
VSC protocol prepare UBUFQE
UsbPipeQueue
return
Send data
OR
Receive data
UBUFQE::uqe_donefn
Client post event to original task because it called from DataPump context
return
Client prepare REQUEST and submit REQUEST if it required.