# USB2 Compliance Device
## A Functional Device Interface Specification for USB2 Compliance Testing

**Date:** January 2, 2002

**Revision:** 1.0

**Scope of this Revision**

The 1.0 revision of the specification is intended for review purposes only.

**Revision History**

| Revision | Issue Date | Comments |
|---|---|---|
| 0.9 | 8/30/2001 | Essentially the same as 0.75. Added some clarifications on bFixedRespCaps use with Isochronous endpoints. Also set pagination. |
| 0.91 | 10/1/2001 | Editorial clarifications on side-effects of VEN_Change_EP and added requirement that high-speed device must support high-bandwidth. |
| 0.92 | 10/9/2001 | Vendor ID of 0xFFFF was causing problems. So, went to a Class code model. |
| 0.93 | 10/15/2001 | Fixed some typos, and added clarification that device does not have to go to the suspend power state when a VEN_Set_RMTWK_TO command is active. |
| 0.94 | 10/22/2001 | Added a clarification about when the device may return a STALL handshake. |
| 1.0 rc1 | 12/20/2001 | Final updates to clarify exporting control pipe capabilities and fix numerous typos. |



**Significant Contributors:**

| | |
|---|---|
| John S. Howard (Author) | Intel Corporation |
| Brad Hosler | Intel Corporation |
| Dan Froelich | Intel Corporation |

***Please send comments via electronic mail to: john.howard@intel.com***

# Table of Contents

THIS PAGE INTENTIONALLY LEFT BLANK

# 1. Introduction

This document provides the definition and requirements for a device that operates as the target device for the compliance programs for Universal Serial Bus 2.0 and Enhanced Host Controller Interface for USB. This document is intended to be useful for two purposes:

- A hardware device vendor or firmware engineer intending to build and program compliance devices which adhere to this specification, and
- A software driver developer programming to use the compliance device.

NOTE: THIS INFORMATION IS PROVIDED AS A COURTESY FOR REFERENCE ONLY WITHOUT ANY WARRANTY AND MAY CONTAIN ERRORS OR INACCURACIES. SEE THE COMPLETE LEGAL NOTICE ON PAGE 2 FOR MORE DETAILS.

This specification is organized as follows:

- Section 2 Overview:
- Section 3 Device Requirements:
- Section 4 Device Framework:

# 2. Overview

The USB2 compliance software test suites for host controller and hub testing, define a complete test environment, from the application and driver software on the host, to the requirements and functionality of 'general-purpose' test devices. The premise of the test environment is to ensure that the only *unknown* is the host controller or hub that is under test evaluation. The 'general purpose' test devices used in the compliance testing (referred to hereafter as compliance devices), regardless of device speed, share the same interface architecture and definition. The common interface and pre-defined requirements yields a set of known and reliable functional characteristics upon which the tests can depend.

The USB2 compliance test methodology depends on being able to re-configure or, in some sense, re-program the compliance devices to meet the functional requirements of the current test. The tests in the test suites are rigorous in their coverage and they exercise as many normal and boundary conditions as reasonably possible. The result is a requirement that the compliance devices be easily and quickly re-programmable, with as few side effects across the device as possible. For example, a runtime instance of a test suite could have more than one test active at the same time; each utilizing different endpoint resources on the same compliance device. The re-programming of the compliance device must allow each test to accomplish its own re-programming without effecting the other running tests.

The standard USB method of multiple interfaces and alternate settings was considered by the compliance development team to be too ROM intensive and inflexible for the compliance programs' purposes. For example, it would take 3072 (1024x3) alternate interfaces in order to fully express all of the valid maximum packet size settings of one high-bandwidth interrupt endpoint. A new device/endpoint configuration model is defined (in this specification) which accommodates the high-degree of re-programming required by the compliance tests.

The compliance device model is similar in concept to the standard USB model in that it is based on a 1:N phase interaction where the device reports its capabilities to the host and then the host repeatedly sets the device's operating characteristics. The compliance device capabilities reported to the host are structured to indicate the capabilities of how each actual hardware endpoint resource can be programmed for operation. The host can program (and use) each endpoint (independently) within the capability parameters provided by the device.

A compliance device's capabilities are reported to the host via a structure called an *EndPoint Table*. The test host will extract the *EndPoint Table*, using the specific request defined in Section 4.2.3. This table is an array of endpoint capabilities, where each entry corresponds to a hardware addressable

endpoint on the device. Entries provide the host with information on how the endpoint may be (re-) programmed. For example, it provides information on which transfer types the endpoint supports and the maximum packet size the endpoint can be programmed to support. The format of the *EndPoint Table* is defined in Section 4.2.3.

The host uses a specific request (see Section 4.2.1) to reprogram a specific endpoint on the compliance device. The host uses the endpoint's index (from the table) in the request to identify which of the endpoints it intends to reprogram. This re-programming effectively sets the current operating characteristics of the endpoint. The re-programming does not change the contents of the *EndPoint Table*. The device must keep separate state to reflect the current operating characteristics. In addition, a required side-effect of programming an endpoint is that the device must re-calculate its standard configuration descriptor set to reflect the current programmed endpoint characteristics. This feature allows the host to program the device endpoints and then use the standard GetDescriptor(Configuration) request to determine the current characteristics.

# 3. Device Requirements

This section provides an overview of the requirements of a compliance device for use in the USB 2.0 compliance programs. These feature requirements may be implemented utilizing any method (firmware, dedicated hardware, or combination).

- Implement the compliance device framework and command set as defined in this specification. Optional commands, if implemented, must be implemented to fulfill the requirements defined in this specification.
- At least one IN and one OUT endpoint that can support data streams of each transfer type defined in the USB core specification, for the device speed (with the exception of Control).

  High Speed – Isochronous, Interrupt, and Bulk.

  Full Speed – Isochronous, Interrupt, and Bulk.

  Low Speed – Interrupt.

  Note1: These can all be the same endpoint. For example a sufficient high-speed device could have one IN and one OUT endpoint each capable of supporting Isochronous, Interrupt, and Bulk.

  Note2: For a high-speed device, at least one (each) of the Isochronous and Interrupt capable IN and OUT endpoints must support the high-bandwidth feature. High-bandwidth capable endpoints must support 1, 2 and 3 packets per micro-frame, per period.
- At least one endpoint resource that supports the control transfer type. The Default Control Endpoint is sufficient. However, a device may provide support for additional control endpoints.
- At least one IN and one OUT endpoint that can source or sink data without responding with a NAK to a token (i.e. perfect sink or source). This means the endpoint (depending on direction) will always take or provide data at the rate presented by the host, and will always respond with either DATA or an ACK.
- Endpoints must support each of the following required *fixed* response modes. This feature allows the host to instruct an endpoint to provide a fixed handshake, until the endpoint is re-programmed.

  ***Required***
  - Always STALL handshake
  - Always NAK handshake
  - Always Timeout

  ***Optional***
  - Always CRCERROR
- The compliance device must provide at least one pair of loop-back endpoints. The preferred solution is to have two pairs of loop-back endpoints as many tests use more than one pair. This reduces the number of compliances devices required for some tests. This feature, when enabled by

system software, will accept data on an OUT endpoint and will provide that data (in received order) on an IN endpoint.

High-speed devices must allow each member of the loop back pair to be programmed with different transfer types. For example, the OUT can be programmed to be a Bulk and the IN can at the same time be programmed as an Isochronous. This is not a requirement for Full- or Low-speed compliance devices.

- High- and Full-speed compliance devices must provide additional data buffering (over and above the endpoint maximum packet size) for at least one IN and one OUT endpoint. These endpoints must additionally be part of the same loop back pair. Tests use this additional buffering to send or receive multiple packets of data before encountering either data flow or losing data. Many "off the shelf" USB device controllers have a shared pool of additional buffering available that is allocated by the device across its available hardware endpoints. For these devices the actual amount of additional buffering available to a loop-back pair depends on prior endpoint allocations. When additional buffering is available, the device is required to allocate it in even multiples of the configured maximum packet size. If the IN and OUT maximum packet sizes for the loop-back pair are different, then the device must allocate the buffer in even multiples of the larger maximum packet size. The host determines how much additional data buffering is available on an endpoint by issuing a VEN_Get_EP_BufferSize request. It is recommended that a high-speed device provide at least 16K bytes of buffering for looping data from an OUT in its IN partner.
- The compliance device may optionally be capable of programmed disconnect/reconnect. This means that the host will send a specific request to tell the device to disconnect and provide a count value (in units of seconds), which directs the device how long to stay disconnected before pulling on the appropriate data line to signal a re-connect. Note, although this feature is optional, it is strongly recommended. If this feature is not supported, then the device will respond with a STALL handshake in response to a VEN_Do_Disconnect request (see Section 4.2.2).
- The compliance devices must be capable of programmed remote wakeup. This means that the host will send a specific request that essentially arms the device for remote wakeup and provides a count value (in units of seconds). When the device detects a suspend event, it will wait the programmed count number of seconds before initiating a resume.

# 4. Device Framework

The compliance device exports a single configuration with a single interface. The number of endpoint descriptors in the configuration depends on the number of entries explicitly marked as valid in the device's *EndPoint Table*. There are no valid endpoints in the default configuration. This means, after initial enumeration (after power up), the device will provide a configuration descriptor set that includes only a configuration descriptor and a single interface descriptor and no endpoint descriptors, (i.e. an empty interface). The host system uses vendor-specific requests (defined below) to explicitly manage the endpoint resources in terms of available, maximum packet size, transfer type, etc.

The compliance device will respond to all vendor-specific commands defined below, regardless of whether the device has received a **SetConfiguration(1)** command.

## 4.1 Compliance Device Descriptors

**Table 4–1. Device Descriptor**

| Offset | Size | Value | Description |
|---|---|---|---|
| 0 | 1 | 12H | **bLength**. Standard length (in bytes) of a device descriptor |
| 1 | 1 | 01H | **bDescriptorType.** DEVICE Descriptor type. |
| 2 | 2 | 200H | **BCD.** USB Specification Release number |
| 4 | 1 | DCH | **bDeviceClass.** This code indicates a Diagnostic Device. |

**Table 4–1. Device Descriptor (cont.)**

| Offset | Size | Value | Description |
|---|---|---|---|
| 5 | 1 | 01H | **bDeviceSubClass.** This code indicates a re-programmable compliance device . |
| 6 | 1 | 01H | **bDeviceProtocol.** This value indicates the device conforms to the programming interface as defined in this specification. |
| 7 | 1 | XXH | **bMaxPacketSize0.** Maximum packet size of Default Endpoint. Actual value depends on current device speed. Valid values are:<br>**Speed** **Value**<br>LS 08H<br>FS/HS 40H |
| 8 | 2 | XXXXH | **idVendor.** Vendor Identifier. This field identifies the Vendor of the compliance device.. |
| 10 | 2 | XXXXH | **idProduct.** This is optionally re-programmable from the framework interface. Its value depends on the last programmed encoding. From a power-up, the value depends on the manufacturer. |
| 12 | 2 | XXXXH | **bcdDevice.** The specification version number of the *Functional Device Interface Specification for USB2 Compliance Testing* to which this device implementation conforms. The value of this field may be the same as that reported in the VEN_Get_Version request. |
| 14 | 1 | XXH | **iManufacturer.** (Optional) |
| 15 | 1 | XXH | **iProduct.** (Optional) |
| 16 | 1 | XXH | **iSerialNumber.** (Optional) |
| 17 | 1 | 01H | **bnumConfigurations.** Device supports one configuration. |

**Table 4–2. Configuration Descriptor**

| Offset | Size | Value | Description |
|---|---|---|---|
| 0 | 1 | 09H | **bLength.** Length of standard Configuration descriptor. |
| 1 | 1 | 02H | **bDescriptorType.** Configuration Descriptor Type. |
| 2 | 2 | VAR | **wTotalLength.** Value depends on the currently programmed active endpoint characteristics. Varies by the number of valid endpoint descriptors. |
| 4 | 1 | 01H | **bNumInterfaces.** The compliance device has one interface. |
| 5 | 1 | 01H | **bConfigurationValue.** Used by host to activate configuration in a SetConfiguation() request. |
| 6 | 1 | XXH | **iConfiguration.** (Optional) |
| 7 | 1 | XXH | **bmAttributes.** Indicates whether device is bus or self-powered. Implementation-dependent. |
| 8 | 1 | XXH | **bMaxPower.** Indicates how much power the device draws from VBus when configured. Implementation-dependent. |

There is one interface descriptor. It is a standard interface descriptor with zeros for most fields (*bInterfaceNumer*, *bAlternateSetting*, *bInterfaceClass*, *bInterfaceSubClass*, and *bInterfaceProtocol*). The *iInterface* field is optional. The value of the *bNumEndpoints* field depends on the number of endpoints enabled in the endpoint characteristics table.

## 4.2 Compliance Device Command Set

The compliance device should implement all required standard commands in the core device framework. In addition, the following commands must be implemented. These are identified on the USB as vendor-specific commands.

**Table 4–3. Compliance Device Interface Command Summary**

| Request Mnemonic | bRequest | Required | Description/Purpose |
|---|---|---|---|
| VEN_Change_EP | 51H | Required | This command allows the host to change the parameters for a single endpoint resource. |
| VEN_Do_Disconnect | 52H | Optional | In response to this command, the device will disconnect from the bus. This command includes two values, one for how long to wait before performing the disconnect, and one for how long to wait (after the disconnect is complete) before doing a connect. Units are in seconds. May be used with VEN_Change_EP commands to get the device to re-enumerate as a different device.<br>Although optional, it is strongly recommended that the compliance device implement this feature. |
| VEN_Get_EP_Table | 53H | Required | In response to this command, the device will return the entire contents of the active, endpoint parameter image. |
| VEN_Set_PID | 54H | Optional | This command sets the device's PID to the value specified in the setup data (wValue). This value will be returned in the next GetDescriptor(DEVICE). |
| VEN_Set_RMTWK_TO | 55H | Required | This command is used to set the time the device should wait after detecting a suspend event before issuing a remote-wakeup. Units are in seconds. |
| Reserved | 56H | N/A | Reserved for future use. |
| Reserved | 57H | N/A | Reserved for future use. |
| VEN_Get_Count | 58H | Required | This command reads from the device an endpoint-specific count value. The count value is reset as a side effect of being read. The count value is 32 bits. |
| VEN_Get_Version | 59H | Required | This command reads the firmware and FPGA version information from the device. |
| VEN_Set_Chirp | 5AH | HS: Optional,<br>FS/LS: N/A. | This command sets the chirp behavior. This command is optional, but recommended for high-speed devices. This command is not to be implemented for a pure full / low-speed device. |
| VEN_Control_Write_Data | 5BH | Required | This command is used to write arbitrary data to the compliance device over a control pipe. |
| VEN_Control_Read_Data | 5CH | Required | This command is used to read data from the control pipe, that was written using VEN_Control_Write_Data. |

**Table 4–3. Compliance Device Interface Command Summary (cont.)**

| Request Mnemonic | bRequest | Required | Description/Purpose |
|---|---|---|---|
| VEN_Get_EP_BufferSize | 5DH | Required | This command is used to query a device endpoint index for the size of the additional buffering provided for the current configured setting of the selected endpoint. |

Note, when the device receives a mal-formed request (i.e. invalid parameters), or an optional request that it does not implement, it must return a STALL PID in response to the next Data stage. As per core-USB specification requirement, this STALL event does NOT set a HALT bit associated with the Default Control Endpoint of the compliance device. The host must set reserved fields in requests and data to zero. Doing otherwise will result in undefined behavior.

### 4.2.1 VEN_Change_EP

This command is used to change the characteristics of a particular endpoint.

| bmRequestType | bRequest | wValue | wIndex | wLength | Data |
|---|---|---|---|---|---|
| 01000000B | VEN_Change_EP | Zero | EP table Index | Length | Endpoint change data |

The *wIndex* field contains the EP table index of the endpoint to modify with this command.

The *wLength* field contains the number of bytes in data stage of this request. The *wLength* value is generically expressed as **`sizeof(EpChangeData_t)`** (see below).

The format of the endpoint change data is specified below.

```
typedef struct _vendor_epChange {
    unsigned char     bFlags;             /*   [0] valid                      */
                                          /*   [1] do loop back               */
                                          /* [3:2] endpoint type              */
                                          /* [5:4] number of additional       */
                                          /*  transaction opportunities per   */
                                          /*  micro-frame                     */
                                          /* [7:6] reserved                   */
    unsigned char     r0;                 /* reserved for future use          */
    unsigned short    wm_MaxPacketSize;   /* the max, modifiable              */
    unsigned long     dm_Pattern;         /* base data pattern by device      */
    unsigned long     dm_Modifer;         /* data pattern modifier            */
    unsigned char     bPeriod             /* for interrupt, the poll interval */
    unsigned char     bFixedRespMode;     /* for setting fixed response mode  */
                                          /*   mode encodings are:            */
                                          /*   00H   normal response mode     */
                                          /*   01H   Always Stall handshake   */
                                          /*   02H   Always Nak handshake     */
                                          /*   04H   Always TimeOut response  */
                                          /*   08H   Always CRCERROR          */
    unsigned short    r1;                 /* reserved for future use          */
} EpChangeData_t, *pEpChangeData_t;
```

The encoding of the endpoint type sub-field in the *bFlags* field are the same as the encoding defined in chapter 8 of the USB Specification:

| Value | Meaning | Value | Meaning |
|---|---|---|---|
| 0b00 | Control | 0b10 | Bulk |
| 0b01 | Isochronous | 0b11 | Interrupt |

**Note:** Only the *Normal* and *Always CRCERROR* encodings in the field *bFixedRespMode* are applicable to the endpoint when *bFlags.endpoint type* indicates an Isochronous endpoint.

A *VEN_Change_EP* command has the following side effects:

- This is essentially a re-configuration event to the endpoint. When request completes, the endpoint has initialized its endpoint-specific resources (data buffer pointers, etc. are at their default, initial values) and the data sequence bit (data toggle) is initialized to zero. Additionally, the device may optionally assign and initialize additional buffering to the endpoint.
- The re-programmed endpoint characteristics (including all local resources such as additional buffering) must persist as static values at least, until the device loses power. For example, a bus-powered device must retain the last programmed values until the device loses power. If a self-powered device, the preferred behavior is that the values be retained until at least, a power-down condition. It is acceptable for a device to store the programmed values in EEPROM so that it will persist across power cycles.
- The device must re-compute its configuration descriptor based on the new endpoint setting. The next time the host requests a configuration descriptor, each endpoint reported must match the last programmed value (since power-up).

If *bflags.valid* is set to a one, then the device must provide the functionality specified in the endpoint change data. The exception is if the host requests functionality explicitly not exported via the Endpoint Capabilities Table (see Section 4.2.3). If this occurs, the device must respond to this request with a STALL PID in the status stage. In addition, the device is allowed to return a STALL PID when the request parameters are valid in the programmable space, but due to previous *VEN_Change_EP* requests, the device does not currently have sufficient local resources (like buffering) to deliver the behavior requested in the current request.

An IN endpoint configured as a loop back must NAK all IN tokens until the host has delivered data to it's OUT loop back partner endpoint. The endpoint table index of the loop back partner is specified in the endpoint capabilities for this endpoint. An OUT endpoint configured as a loop back must ACK OUT tokens until it's internal buffering (between it and its IN loop back partner) is full. It must implement the behavior required by any standard OUT endpoint as specified in the USB core specification.[1]

An IN endpoint configured with *bFlags.do loop back* set to a zero, must behave as an infinite source of data. The fields *dm_Pattern* and *dm_Modifier* are only used by the device for an IN endpoint in *infinite source* mode. On reception of a VEN_Change_EP command where *bFlags.do loop back* bit is a zero, the device will initialize the endpoint's entire dedicated data buffer to a DWORD pattern based on the value of *dm_Pattern*. It is assumed and sufficient that the endpoint data buffer be no larger than *wm_MaxPacketSize* when not in loop-back mode. The field *dm_Modifier* has the following functional effects to an IN endpoint:

- When *dm_Modifier* is non-zero, it is used to change the contents of the endpoint's buffer (initialized to *dm_Pattern* above) for each packet sent to the host. The device is allowed to NAK IN tokens while it adjusts the data buffer. The preferred algorithm for adjusting the

[1] Note, the implication here is that the OUT endpoint must ACK, NYET or NAK an OUT token depending on the availability of internal buffering between itself and its IN loop back partner.

buffer by modifier is to maintain a DWORD aligned offset into the endpoint buffer. Each time an IN token occurs, the endpoint buffer at the current offset is modified by adding the value of *dm_Modifier* to the DWORD value at the offset. Then the offset is incremented by the size of a DWORD (modulo *wm_MaxPacketSize*). The following table illustrates.

| Event | Action (Side-effects) |
|---|---|
| VEN_Change_EP | • Entire endpoint buffer (EP_Buffer) is set to *dm_Pattern*.<br>• Offset is set to 0. |
| $IN^X$ | • After IN transaction is complete, device updates buffer before enabling the endpoint for the next IN.<br>`EP_Buffer[Offset] = EP_Buffer[Offset] + dm_Modifier;`<br>`Offset += 1;`<br>`Enable endpoint` |
| $IN^{X+1}$ | • After IN transaction is complete, update buffer, then re-enable endpoint to send.<br>`EP_Buffer[Offset] = EP_Buffer[Offset] + dm_Modifier;`<br>`Offset += 1;`<br>`Enable endpoint` |
| $IN^{X+N}$ | Repeat |

This algorithm will create a buffer pay load pattern similar to:

| | |
|---|---|
| Packet[X] | EP_Buffer [*dm_Pattern, …dm_Pattern*] |
| Packet[X+1] | EP_Buffer [*dm_Pattern + dm_Modifier, …dm_Pattern*] |
| Packet[X+2] | EP_Buffer [*dm_Pattern + dm_Modifier, dm_Pattern + dm_Modifier …dm_Pattern*] |

- If the endpoint is capable of being a perfect source (as determined by the value of the endpoint capabilities *bf_Attributes*[2] field being a one), and *dm_Modifier* is a zero, then the endpoint must not NAK IN tokens. Otherwise, the endpoint is allowed to NAK IN tokens.

An OUT endpoint configured with *bFlags.do loop back* set to zero, must behave as an infinite sink of data. Additionally, if the OUT endpoint is capable of being a perfect sink, as specified by the endpoint capabilities *bf_Attributes*[2] field being a one, then it must never NAK OUT tokens.

When disabling an endpoint (setting bFlags.valid to zero) all other fields in the endpoint change data structure are undefined and must not be verified by the device. Disabling an endpoint that is already disabled is a valid operation. The device never returns a STALL PID in the status stage when disabling an endpoint, even when the endpoint is already disabled.

### 4.2.2 VEN_Do_Disconnect

This command is used to cause the device to perform a program-controlled disconnect, then subsequent reconnect.

| bmRequestType | bRequest | wValue | wIndex | wLength | Data |
|---|---|---|---|---|---|
| 01000000B | VEN_Do_Disconnect | T_disconnect | Speed Flag | Zero | None |
| | | T_connect | | | |

The most significant byte of the *wValue* field (*T_disconnect*) specifies the number of seconds after completion of this command that the device should wait before initiating disconnection from the bus.

The least significant byte of the *wValue* field (*T_connect*) specifies the number of seconds after the disconnect is complete, that the device will wait before initiating a connect to the bus.

*T_disconnect* and *T_connect* must be in the range: [1,60].

When the device performs a disconnect and reconnect in response to this command it is required to retain its current configuration.

*Speed Flag* is an optional field that is used by the host to request that the device re-connect at a specific speed. For example, the host application may need to test with a full-speed device. This mechanism allows the host to force the device to disconnect and then reconnect and enumerate at the desired speed. In this example case, inhibiting the chirp during the next reset. *Speed Flag* is the least significant bit of the *wIndex* field. The bit value has the following meaning:

| *Speed Flag* | Explanation |
|---|---|
| 0 | Remain at the current speed. This means: if currently connected as a high-speed device, do chirp in response to the bus reset observed after the next connect. If currently connected as a full-speed device, do not do chirp in response to the bus reset observed after the next connect. |
| 1 | Flip to the other speed. This means: if currently connected as a high-speed device, do not do chirp in response to the bus reset observed after the next connect. If currently connected as a full-speed device, do chirp in response to the bus reset observed after the next connect. |

### 4.2.3 VEN_Get_EP_Table

This command extracts from the compliance device, the endpoint capabilities table.

| bmRequestType | bRequest | wValue | wIndex | wLength | Data |
|---|---|---|---|---|---|
| 11000000B | VEN_Get_EP_Table | Zero | Zero | Length | EP Table |

The *wLength* field contains the number of bytes in the data stage of this request. The actual length of the endpoint table is implementation dependent. In order to extract the entire table, the host should first use this request to read the first byte of the table (i.e. *length* field). The value of this byte reports the number of elements in the table. The actual length of the table (in bytes) is computed by:

```
1 + (sizeof(EpChangeData_t) * EP_Table.length)
```

The format of the endpoint capabilities information is organized as a record that contains a length field, followed by an array of endpoint capability records. There is one endpoint capability record for each discretely addressable endpoint on the device. The value of the length field reports the number of discrete endpoints supported by this device.

On power-up, the default configuration is that all endpoints in the endpoint table image have all *valid* fields set to zero. The default state supports the device requirement that it provide a default configuration descriptor set with zero endpoint descriptors. As the host configures individual endpoints

(see VEN_Change_EP) it adds or changes the endpoint descriptors to match the *valid* endpoints configured in this table.

```
typedef struct _TransferTypeCapabilities {
    unsigned char    bValid;             /* support this transfer type */
    unsigned char    bMaxExtraBuffer;    /* max for this transfer type */
    unsigned short   wMaxPacketSize;     /* max for this transfer type */
} EpTransTypeCap_t, *pEpTransTypeCap_t;

typedef struct _EpCapabilities {
    unsigned char     bf_EpAddress;      /* [3:0] number [7] direction     */
    unsigned char     bLBEpIndex;        /* index of loop back partner     */
    unsigned char     bf_Attributes;     /* capable of doing,              */
                                         /*   bit  meaning                 */
                                         /*   [0]   high-bandwidth capable */
                                         /*   [1]   loop back capable      */
                                         /*   [2]   perfect source/sink    */
                                         /*           i.e.: never-nak      */
                                         /* [3:4]   reserved               */
                                         /*   [5]   Transaction Counter    */
                                         /* [7:6]   reserved               */
    unsigned char     bFixedRespCaps;    /* for setting fixed response mode*/
                                         /*   mode encodings are:          */
                                         /*   00H   normal response mode   */
                                         /*   01H   Always Stall handshake */
                                         /*   02H   Always Nak handshake   */
                                         /*   04H   Always TimeOut response*/
                                         /*   08H   Always CRCERROR        */
    EpTransTypeCap_t  bulkCaps;          /* capabilities as bulk ep        */
    EpTransTypeCap_t  interruptCaps;     /* capabilities as interrupt ep   */
    EpTransTypeCap_t  isochCaps;         /* capabilities as isoch ep       */
    EpTransTypeCap_t  controlCaps;       /* capabilities as a control ep   */
} EpCapabilities_t, *pEpCapabilities_t;

typedef struct _EpCapabilityTable {
    unsigned char      length;       /* number of table entries */
    pEpCapabilities_t  EpCapTable[]; /* table of endpoints */
} EpCapabilityTable_t;
```

Notes:

1. Only *bFixedRespCaps* encoded values that are applicable to an endpoint configured as an Isochronous transfer type, are: *Normal* and *CRCERR*. The other values apply to endpoints configured as Bulk, Interrupt, or Control.

2. A non-zero value in the field *bMaxExtraBuffer* indicates to the host that the device may have additional buffering available to the endpoint (beyond the required maximum packet size). The host may use the VEN_Get_EP_BufferSize after a VEN_Change_EP to determine how much additional buffering the device has allocated to the endpoint.

3. A control pipe is by definition a bi-directional resource of a USB device. It requires both an IN and OUT endpoint, with identical endpoint numbers. If a device exports control endpoints via the endpoint table, it must do so following the following rules.

   For each control pipe supported by the device, the endpoint table will have one endpoint index with a *controlCaps* record where the *bValid* field is set to a one.

   It is a requirement that each endpoint capability in and endpoint table, with a *controlCaps*.*bValid* set to a one, has a unique endpoint number (least significant 5 bits of the *bf_EpAddress* field.

   A control endpoint can only be established (via the VEN_Change_EP request) when both endpoint numbers needed for the control endpoint are free for use. For example, a device exports a control endpoint capability at endpoint number $1^{IN}$, but also has an endpoint number $1^{OUT}$. If the host turns on endpoint number $1^{OUT}$ (via VEN_Change_EP request), then the device must STALL a VEN_Change_EP rquest to activate endpoint number $1^{IN}$ as a control endpoint, because not all of the resources to support the functionality are available at the device.

### 4.2.4 VEN_Set_PID

This command is used to set the device's PID reported in a **GetDescriptor(Device)** to a specific value.

| bmRequestType | bRequest | wValue | wIndex | wLength | Data |
|---|---|---|---|---|---|
| 01000000B | VEN_Set_PID | PID | zero | zero | None |

The *wValue* field contains the new Product ID value.

### 4.2.5 VEN_Set_RMTWK_TO

This command is used to set the number of seconds the device should wait after detecting a suspend event, before it will automatically initiate a resume.

| bmRequestType | bRequest | wValue | wIndex | wLength | Data |
|---|---|---|---|---|---|
| 01000000B | VEN_Set_RMT_WK_TO | SleepTime | zero | zero | None |

Note, the device will only initiate a timed resume if this request is given and *SleepTime* has a non-zero value. *SleepTime* must be in the range [0, 60]. When this command is 'active' (e.g. non-zero *SleepTime*), the compliance device is not required to go into the *Suspended* device state when it detects a bus suspend. It is allowed to remain at full configured power in order to count the *SleepTime*, then initiate remote wakeup. This request must be re-issued each time the host needs this functionality from the device.

The device must not lose any internal state as a result of a bus Suspend and bus Resume.

### 4.2.6 VEN_Get_Count

This command is used to extract (and reset) an endpoint-specific counter.

| bmRequestType | Brequest | wValue | wIndex | wLength | Data |
|---|---|---|---|---|---|
| 11000010B | VEN_Get_Count | Zero | Endpoint Index | Four | Counter Data |

This request is addressed to a selected endpoint. The index of the endpoint is specified in the low-order byte of the *wIndex* field. In response to this request, the device will return the value of the selected endpoint's counter during the data stage. It will also reset the selected counter to zero.

### 4.2.7 VEN_Get_Version

This command is used to extract the version number of the firmware and hardware interface.

| bmRequestType | bRequest | wValue | wIndex | wLength | Data |
|---|---|---|---|---|---|
| 11000000B | VEN_Get_Version | Zero | Zero | Four | Version Numbers |

This request asks for four bytes. The value of the first two bytes is a binary coded decimal encoding of the version of the hardware interface (FPGA, for now). The value of the second two bytes is a binary coded decimal encoding of the version of the firmware.

### 4.2.8 VEN_Set_Chirp

This command is used to set the parameters, which determine how a high-speed compliance device behaves during a high-speed chirp sequence.

| bmRequestType | bRequest | wValue | wIndex | wLength | Data |
|---|---|---|---|---|---|
| 01000000B | VEN_Set_Chirp | Chirp Delay | Chirp Duration | Zero | None |

This request sends two values to the high-speed compliance device that affect the behavior of the device during a high-speed chirp sequence. The value specified in the *wValue* field is the *Chirp Delay*. This is the amount of time after detecting a reset that the device will wait before asserting chirp. The value specified in the *wIndex* field is the *Chirp Duration*. This is the amount of time the device will hold the chirp-K active. Both values are units of microseconds. The ranges for each parameter specified below, allow the testing across all valid boundaries of the chirp protocol.

The valid range of values for *Chirp Delay* is [1, 6000]

The valid range of values for *Chirp Duration* is: [500, 8000].

### 4.2.9 VEN_Control_Write_Data

This command is used to send one Maximum Packet size worth of data to a control endpoint.

| bmRequestType | brequest | wValue | wIndex | wLength | Data |
|---|---|---|---|---|---|
| 01000000B | VEN_Control_Write_Data | zero | Zero | ≤ Default Endpoint Maximum Packet Size | data |

The device is allowed to provide a STALL response to this request if the value in *wLength* is larger than the size of the control endpoint's maximum packet size. The device is only required to preserve the data until the next control out request (e.g. Host to Device). For example, system software uses VEN_Control_Write_Data to send a packet's worth of data to the device. If the next control request is a VEN_Control_Read_Data, the device is required to respond with the data it received during the VEN_Control_Write_Data command. If any other control request is received after the VEN_Control_Write_Data, the device may overwrite the data.

Note, if the device does not have independent endpoint hardware that can be reconfigured to support a control transfer, this request must be supported on the Default Control Endpoint. In the event the device supports only control transfers on only the Default Control Endpoint, it must provide an entry in the Endpoint Capabilities Table for the Default Control Endpoint.

### 4.2.10 VEN_Control_Read_Data

This command is used read a Maximum Packet's worth of data from a control endpoint.

| bmRequestType | bRequest | wValue | wIndex | wLength | Data |
|---|---|---|---|---|---|
| 11000000B | VEN_Control_Read_Data | zero | Zero | ≤ Default Endpoint Maximum Packet Size | Data |

If the previous command to the device was a VEN_Control_Write_Data, then the data sent to the host during the data stage of the control transfer is the data written during the VEN_Control_Write_Data request. If the previous command was not a VEN_Control_Write_Data request, then the contents of the data returned in this command is undefined.

Note, if the device does not have independent endpoint hardware that can be reconfigured to support a control transfer, this request must be supported on the Default Control Endpoint. In the event the device supports only control transfers on only the Default Control Endpoint, it must provide an entry in the Endpoint Capabilities Table for the Default Control Endpoint.

### 4.2.11 VEN_Get_EP_BufferSize

This command is used to query a device endpoint index for the amount of buffer allocated to the endpoint by the device, for the currently selected endpoint configuration.

| bmRequestType | bRequest | wValue | wIndex | wLength | Data |
|---|---|---|---|---|---|
| 11000010B | VEN_Get_EP_BufferSize | zero | Endpoint Index | Four | Data |

This request is addressed to a specific endpoint index. The index is specified in the low-order byte of the *wIndex* field. In response to this request, the device will return the amount of additional buffering currently allocated to the endpoint during the data stage. The value return is in units of bytes. Note that the amount of additional buffer allocated to an endpoint is static for as long as the current endpoint characteristics are valid. The amount of allocated buffer will not decrease or increase as other endpoints are enabled or disabled.