Future Technology Devices International Ltd.

Application Note

AN_172

Vinculum-II

Using the USB Slave Driver

Document Reference No.: FT_000424

Version 1.0

Issue Date: 2011-03-15

This application note provides an example of how to use the FTDI Vinculum-II (VNC2) USB Slave driver. Sample source code is included.

Future Technology Devices International Limited (FTDI)

Unit 1, 2 Seaward Place, Glasgow G41 1HH, United Kingdom
Tel.: +44 (0) 141 429 2777 Fax: + 44 (0) 141 429 2758
E-Mail (Support): **support1@ftdichip.com** Web: http://www.ftdichip.com

Copyright © 2011 Future Technology Devices International Limited

Table of Contents

1 Introduction

The FTDI USB Slave driver is a peripheral driver that controls the USB slave ports on Vinculum II (VNC2). As a peripheral driver, the USB Slave driver exposes the standard device driver interface accessed via the Device Manager to an application [1]. It is the purpose of this application note to describe the USB Slave driver interface.

While it is entirely feasible for an application to call the USB Slave interface directly, it is more likely that applications will be designed to encapsulate USB Slave functionality in a function driver that is layered above, and attached to, the USB Slave driver. This application note contains information of use to developers implementing USB Slave function drivers.

The sample source code contained in this application note is provided as an example and is neither guaranteed nor supported by FTDI.

1.1 Driver Hierarchy

In the application architecture, the USB Slave driver can have a direct interface to the application, or a function driver can be layered between application and USB Slave driver.

1.1.1 USB Slave Interface

The relationship between an application and the USB Slave driver is shown in Figure 1:

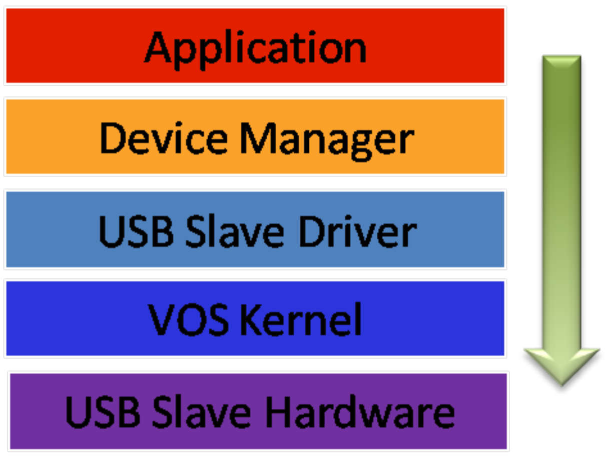

Figure 1: USB Slave Interface

1.1.2 USB Slave Function Driver Interface

The relationship between an application, a USB Slave function driver and the USB Slave driver is shown in Figure 2:

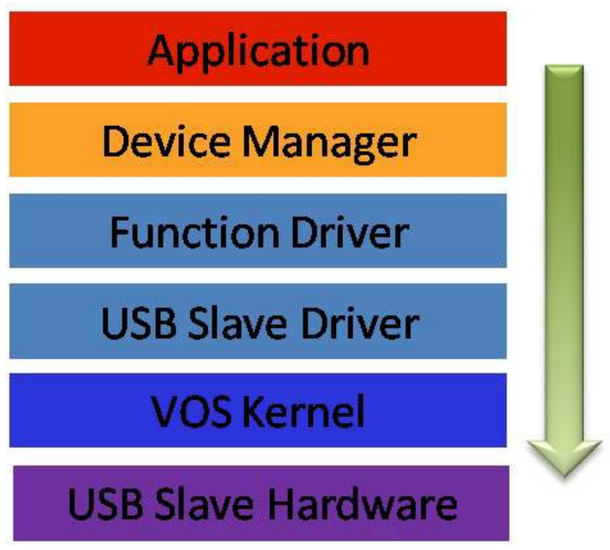

Figure 2: USB Slave Function Driver Interface

2 USB Slave Concepts

VNC2 can be configured with at most 2 USB Slave ports. The USB Slave driver maintains a context for each configured USB Slave port and presents an endpoint-based interface to the application. Requests to an endpoint must be routed through the correct driver handle to the appropriate USB Slave port.

2.1 Initialisation

The *usbslave_init()* function must be called to initialise the driver before the kernel scheduler is started with *vos_start_scheduler()*.

Syntax

```
unsigned char usbslave_init (unsigned char slavePort,  unsigned char devNum);
```

Description

Initialise the USB Slave driver and register the driver with the Device Manager. There are two USB Slave ports, both are controlled from a single instance of the driver. However, the *usbslave_init()* function must be called for each slave port used.

Parameters

slavePort

The slave port to initialise to use the device number passed in devNum. This can be either USBSLAVE_PORT_A or USBSLAVE_PORT_B.

devNum

The device number to use when registering this USB Slave port with the Device Manager is passed in the devNum parameter.

Returns

The function returns zero if successful and non-zero if it could not initialise the driver or allocate memory for the driver.

Comments

The function must be called twice to configure both USB Slave ports. If a port is configured by the USB Host then it cannot be used for the USB Slave. The USB Slave has no thread memory requirements.

2.2 Hardware Configuration

The driver can be configured to control either USB Port 1, USB Port 2 or both USB Ports. A unique VOS_HANDLE and *usbslave_ep_handle_t* handle is required for each endpoint. If a port is configured for use by the USB Host then it cannot be used by the USB Slave.

Once the USB Slave driver is configured it cannot be reconfigured.

2.3 Driver Handles

The USB Slave driver requires a unique device number to register a USB port as a USB Slave device with the Device Manager. Two unique device numbers are required to register both USB ports as USB Slave devices with the Device Manager.

If both USB Ports are configured then the application will have 2 driver handles when both ports are opened, one for each USB Port and effectively 2 device drivers active. They should be treated separately by the application.

2.4 Endpoints

The interface to a USB port is based on operations on endpoints [2]. Each device has a control endpoint and a variable number of IN and OUT endpoints.

Endpoints are accessed via a handle of type *usbslave_ep_handle_t*. The control endpoint (EP0) is treated as a special case. It handles SETUP packets and supports IN and OUT transactions, and separate handles are required for EP0 IN and EP0 OUT

2.5 Device Configuration

A device consists of a control endpoint and up to 7 IN or OUT endpoints. The following IOCTLs are used to set the endpoint configuration for a device:

VOS_IOCTL_USBSLAVE_GET_BULK_IN_ENDPOINT_HANDLE

VOS_IOCTL_USBSLAVE_GET_BULK_OUT_ENDPOINT_HANDLE

VOS_IOCTL_USBSLAVE_GET_INT_IN_ENDPOINT_HANDLE

VOS_IOCTL_USBSLAVE_GET_INT_OUT_ENDPOINT_HANDLE

VOS_IOCTL_USBSLAVE_GET_ISO_IN_ENDPOINT_HANDLE

VOS_IOCTL_USBSLAVE_GET_ISO_OUT_ENDPOINT_HANDLE

The control endpoint is always enabled.

2.6 Obtaining an Endpoint Handle

A handle must be obtained prior to accessing an endpoint. The following IOCTLs are used to obtain a handle to an endpoint of a specific type:

Control Endpoint	VOS_IOCTL_USBSLAVE_GET_CONTROL_ENDPOINT_HANDLE
IN Endpoint	VOS_IOCTL_USBSLAVE_GET_BULK_IN_ENDPOINT_HANDLE
	VOS_IOCTL_USBSLAVE_GET_INT_IN_ENDPOINT_HANDLE
	VOS_IOCTL_USBSLAVE_GET_ISO_IN_ENDPOINT_HANDLE
OUT Endpoint	VOS_IOCTL_USBSLAVE_GET_BULK_OUT_ENDPOINT_HANDLE
	VOS_IOCTL_USBSLAVE_GET_INT_OUT_ENDPOINT_HANDLE
	VOS_IOCTL_USBSLAVE_GET_ISO_OUT_ENDPOINT_HANDLE

Once a handle is obtained then data can be sent to the endpoint (for an OUT endpoint) and received from the endpoint (for IN endpoints).

2.7 Enumeration

To support device enumeration, the following algorithm is required. All IOCTLs must be directed to the Control endpoint.

- Wait for a SETUP packet to be received (VOS_IOCTL_USBSLAVE_WAIT_SETUP_RCVD)
- Decode SETUP packet
- Handle Standard Device Requests - mandatory requests that must be handled are:
    - Set Address, use VOS_IOCTL_USBSLAVE_SET_ADDRESS to change slave address
    - Set Configuration
    - Get Descriptor for both Device and Configuration descriptors
- Handle Class Specific Requests (if any)
- Handle Vendor Specific Requests (if any)
- If SETUP packet has a data phase read or write further data with VOS_IOCTL_USBSLAVE_SETUP_TRANSFER
- Use VOS_IOCTL_USBSLAVE_SETUP_TRANSFER again to acknowledge transaction with ACK phase

2.8 Reading and Writing Data

The VOS_IOCTL_USBSLAVE_TRANSFER IOCTL is used for both IN and OUT endpoints. It must not be used on a Control endpoint.

2.9 Return Codes

All calls to the USB Slave driver will return one of the following status codes.

USBSLAVE_OK

No error.

USBSLAVE_INVALID_PARAMETER

A parameter is incorrect or has a mistake.

USBSLAVE_ERROR

An unspecified error occurred.

3 USB Slave Requests

As defined in USBSlave.h, calls to the IOCTL functions for the USB Slave driver take the form:

```
typedef struct _usbslave_ioctl_cb_t {
  uint8 ioctl_code;
  uint8 ep;
  usbslave_ep_handle_t handle;
  // read buffer
  void *get;
  // write butter
  void *set;
  union {
          struct {
                  uint8 in_mask;
                  int16 out_mask;
          } set_ep_masks;
          struct {
                  uint8 *buffer;
                  int16 size;
                  int16 bytes_transferred;
          } setup_or_bulk_transfer;
  } request;
} usbslave_ioctl_cb_t;
```

As defined in USBSlave.h, the following IOCTL request codes are supported by the USB Host driver:

VOS_IOCTL_USBSLAVE_GET_STATE

VOS_IOCTL_USBSLAVE_GET_CONTROL_ENDPOINT_HANDLE

VOS_IOCTL_USBSLAVE_GET_IN_ENDPOINT_HANDLE

VOS_IOCTL_USBSLAVE_GET_OUT_ENDPOINT_HANDLE

VOS_IOCTL_USBSLAVE_SET_ENDPOINT_MASKS

VOS_IOCTL_USBSLAVE_WAIT_SETUP_RCVD

VOS_IOCTL_USBSLAVE_SETUP_TRANSFER

VOS_IOCTL_USBSLAVE_SET_ADDRESS

VOS_IOCTL_USBSLAVE_TRANSFER

VOS_IOCTL_USBSLAVE_ENDPOINT_STALL

VOS_IOCTL_USBSLAVE_ ENDPOINT_CLEAR

VOS_IOCTL_USBSLAVE_ ENDPOINT_STATE

VOS_IOCTL_USBSLAVE_SET_LOW_SPEED

VOS_IOCTL_USBSLAVE_DISCONNECT

VOS_IOCTL_USBSLAVE_GET_BULK_IN_ENDPOINT_HANDLE

VOS_IOCTL_USBSLAVE_GET_ BULK_OUT_ENDPOINT_HANDLE

VOS_IOCTL_USBSLAVE_GET_ INT_IN_ENDPOINT_HANDLE

VOS_IOCTL_USBSLAVE_GET_ INT_OUT_ENDPOINT_HANDLE

VOS_IOCTL_USBSLAVE_GET_ ISO_IN_ENDPOINT_HANDLE

VOS_IOCTL_USBSLAVE_GET_ ISO_OUT_ENDPOINT_HANDLE

VOS_IOCTL_USBSLAVE_SET_ENDPOINT_MAX_PACKET_SIZE

VOS_IOCTL_USBSLAVE_WAIT_ON_USB_SUSPEND

VOS_IOCTL_USBSLAVE_WAIT_ON_USB_RESUME

VOS_IOCTL_USBSLAVE_ISSUE_REMOTE_WAKEUP

3.1 VOS_IOCTL_USBSLAVE_GET_STATE

Description

Returns the current state of the USB Slave hardware interface.

Parameters

There are no parameters.

Returns

The current state is returned to the location whose address is passed in the *get* field of the *usbslave_ioctl_cb_t* structure. It can be one of the following values:

usbsStateNotAttached	Not attached to a host controller.
usbsStateAttached	Attached to a host controller which is not configured.
usbsStatePowered	Attached to a host controller which is configured. Configuration of device can commence.
usbsStateDefault	Default mode where configuration sequence has performed a device reset operation.
usbsStateAddress	Address has been assigned by host.
usbsStateConfigured	Device is fully configured by host.
usbsStateSuspended	Device has been suspended by host.

Example

```
usbslave_ioctl_cb_t iocb;
unsigned char state;

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_GET_STATE;
iocb.get = &state;
vos_dev_ioctl(hA,&iocb);

if (state == usbsStateConfigured)
{
           // device in action
}
```

3.2 VOS_IOCTL_USBSLAVE_GET_CONTROL_ENDPOINT_HANDLE

Description

Returns a handle for the control endpoint EP0. Separate handles are required for EP0 IN and EP0 OUT.

Parameters

The control endpoint identifier is passed in the *ep* field of the *usbslave_ioctl_cb_t* structure. Control endpoint identifiers are defined as follows:

```
enum {
     USBSLAVE_CONTROL_SETUP,
     USBSLAVE_CONTROL_OUT,
     USBSLAVE_CONTROL_IN
};
```

Returns

The control endpoint handle is returned to the location whose address is passed in the *get* field of the *usbslave_ioctl_cb_t* structure.

Example

```
usbslave_ioctl_cb_t iocb;
usbslave_ep_handle_t in_ep0;
usbslave_ep_handle_t out_ep0;

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_GET_CONTROL_ENDPOINT_HANDLE;
iocb.ep = USBSLAVE_CONTROL_IN;
iocb.get = &in_ep0;
vos_dev_ioctl(hA,&iocb);

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_GET_CONTROL_ENDPOINT_HANDLE;
iocb.ep = USBSLAVE_CONTROL_OUT;
iocb.get = &out_ep0;
vos_dev_ioctl(hA,&iocb);
```

3.3 VOS_IOCTL_USBSLAVE_GET_IN_ENDPOINT_HANDLE

Description

Returns a handle for an IN endpoint.

NOTE: this IOCTL is deprecated. Please use one of the following IOCTLs instead:

VOS_IOCTL_USBSLAVE_GET_BULK_IN_ENDPOINT_HANDLE

VOS_IOCTL_USBSLAVE_GET_INT_IN_ENDPOINT_HANDLE

VOS_IOCTL_USBSLAVE_GET_ISO_IN_ENDPOINT_HANDLE

Parameters

The endpoint address is passed in the *ep* field of the *usbslave_ioctl_cb_t* structure. Valid endpoint addresses are in the range 1-7.

Returns

The IN endpoint handle is returned to the location whose address is passed in the *get* field of the *usbslave_ioctl_cb_t* structure.

Example

```
usbslave_ioctl_cb_t iocb;
usbslave_ep_handle_t in_ep;

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_GET_IN_ENDPOINT_HANDLE;
iocb.ep = 1;
iocb.get = &in_ep;
vos_dev_ioctl(hA,&iocb);
```

3.4 VOS_IOCTL_USBSLAVE_GET_OUT_ENDPOINT_HANDLE

Description

Returns a handle for an OUT endpoint.

NOTE: this IOCTL is deprecated. Please use one of the following IOCTLs instead:

VOS_IOCTL_USBSLAVE_GET_BULK_OUT_ENDPOINT_HANDLE

VOS_IOCTL_USBSLAVE_GET_INT_OUT_ENDPOINT_HANDLE

VOS_IOCTL_USBSLAVE_GET_ISO_OUT_ENDPOINT_HANDLE

Parameters

The endpoint address is passed in the *ep* field of the *usbslave_ioctl_cb_t* structure. Valid endpoint addresses are in the range 1-7.

Returns

The OUT endpoint handle is returned to the location whose address is passed in the *get* field of the *usbslave_ioctl_cb_t* structure.

Example

```
usbslave_ioctl_cb_t iocb;
usbslave_ep_handle_t out_ep;

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_GET_OUT_ENDPOINT_HANDLE;
iocb.ep = 2;
iocb.get = &out_ep;
vos_dev_ioctl(hA,&iocb);
```

3.5 VOS_IOCTL_USBSLAVE_SET_ENDPOINT_MASKS

Description

Sets the endpoint masks for the device. Endpoint masks represent endpoint addresses and types for the device.

NOTE: this IOCTL is deprecated. Please use one of the following IOCTLs instead:

VOS_IOCTL_USBSLAVE_GET_BULK_IN_ENDPOINT_HANDLE

VOS_IOCTL_USBSLAVE_GET_BULK_OUT_ENDPOINT_HANDLE

VOS_IOCTL_USBSLAVE_GET_INT_IN_ENDPOINT_HANDLE

VOS_IOCTL_USBSLAVE_GET_INT_OUT_ENDPOINT_HANDLE

VOS_IOCTL_USBSLAVE_GET_ISO_IN_ENDPOINT_HANDLE

VOS_IOCTL_USBSLAVE_GET_ISO_OUT_ENDPOINT_HANDLE

Parameters

The endpoint mask for IN endpoints is passed in the *set_ep_masks.in_mask* field of the *request* union in the *usbslave_ioctl_cb_t* structure.

The endpoint mask for OUT endpoints is passed in the *set_ep_masks.out_mask* field of the *request* union in the *usbslave_ioctl_cb_t* structure.

Valid endpoint addresses are in the range 1-7. In the mask fields, bits set to 1 correspond to the addresses of the device's endpoints.

Returns

There is no return value.

Example

```
usbslave_ioctl_cb_t iocb;

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_SET_ENDPOINT_MASKS;
iocb.request.set_ep_masks.in_mask = 0x02;       // EP1
iocb.request.set_ep_masks.out_mask = 0x04;      // EP2
vos_dev_ioctl(hA,&iocb);
```

3.6 VOS_IOCTL_USBSLAVE_WAIT_SETUP_RCVD

Description

Receives a SETUP packet. This call blocks until a SETUP packet is received from the host.

Parameters

The address of the buffer to receive the SETUP packet is passed in the *setup_or_bulk_transfer.buffer* field of the *request* union in the *usbslave_ioctl_cb_t* structure.

The size the buffer to receive the SETUP packet is passed in the *setup_or_bulk_transfer.size* field of the *request* union in the *usbslave_ioctl_cb_t* structure. The USB Slave returns a 9-byte SETUP packet.

Returns

The buffer passed in the *setup_or_bulk_transfer.buffer* field of the *request* union in the *usbslave_ioctl_cb_t* structure contains the SETUP packet.

Example

```
usbslave_ioctl_cb_t iocb;
unsigned char setup_buffer[9];

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_WAIT_SETUP_RCVD;
iocb.request.setup_or_bulk_transfer.buffer = setup_buffer;
iocb.request.setup_or_bulk_transfer.size = 9;
vos_dev_ioctl(hA,&iocb);
```

3.7 VOS_IOCTL_USBSLAVE_SETUP_TRANSFER

Description

Performs a data phase or ACK phase for a SETUP transaction.

Parameters

The handle of the control endpoint on which the transaction is being performed is passed in the *handle* field of the *usbslave_ioctl_cb_t* structure.

The address of the buffer containing data for the transfer is passed in the *setup_or_bulk_transfer.buffer* field of the *request* union in the *usbslave_ioctl_cb_t* structure.

The size of the buffer containing data for the transfer is passed in the *setup_or_bulk_transfer.size* field of the *request* union in the *usbslave_ioctl_cb_t* structure.

Returns

There is no return value.

Example

```
usbslave_ioctl_cb_t iocb;

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_SETUP_TRANSFER;
iocb.handle = in_ep0;
iocb.request.setup_or_bulk_transfer.buffer = (void *) 0;
iocb.request.setup_or_bulk_transfer.size = 0;
vos_dev_ioctl(hA,&iocb);
```

3.8 VOS_IOCTL_USBSLAVE_SET_ADDRESS

Description

Sets the USB address for the device. The USB host assigns the device address during enumeration, and this IOCTL is used to set the USB slave port hardware to respond to that address. This IOCTL should be used when processing the USB standard device request, SET_ADDRESS.

Parameters

The address is passed in the *set* field of the *usbslave_ioctl_cb_t* structure.

Returns

There is no return value.

Example

```
void set_address_request(uint8 addr)
{
   usbslave_ioctl_cb_t iocb;

   iocb.ioctl_code = VOS_IOCTL_USBSLAVE_SET_ADDRESS;
   iocb.set = (void *) addr;
   vos_dev_ioctl(hA,&iocb);
}
```

3.9 VOS_IOCTL_USBSLAVE_TRANSFER

Description

Performs a transfer to a non-control endpoint. This IOCTL is used for bulk transfers on both IN and OUT endpoints. When used on an OUT endpoint, this IOCTL blocks until data is received from the host. When used on an IN endpoint, this IOCTL blocks until data is sent to the host (in response to an IN request sent from the host).

Parameters

The handle of the endpoint on which the transaction is being performed is passed in the *handle* field of the *usbslave_ioctl_cb_t* structure.

The address of the buffer for the transfer is passed in the *setup_or_bulk_transfer.buffer* field of the *request* union in the *usbslave_ioctl_cb_t* structure.

The size of the buffer containing data for the transfer is passed in the *setup_or_bulk_transfer.size* field of the *request* union in the *usbslave_ioctl_cb_t* structure.

Returns

The number of bytes transferred is returned in the *setup_or_bulk_transfer.bytes_transferred* field of the request union in the *usbslave_ioctl_cb_t* structure.

For bulk transfer requests on OUT endpoints, the data is returned in the buffer whose address was passed in the *setup_or_bulk_transfer.buffer* field of the *request* union in the *usbslave_ioctl_cb_t* structure.

Example

```
usbslave_ioctl_cb_t iocb;
usbslave_ep_handle_t in_ep;
usbslave_ep_handle_t out_ep;
char *str = "hello, world";
uint8 out_buffer[64];

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_BULK_TRANSFER;
iocb.handle = in_ep;
iocb.request.setup_or_bulk_transfer.buffer = (unsigned char *) str;
iocb.request.setup_or_bulk_transfer.size = 12;
vos_dev_ioctl(hA,&iocb);

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_BULK_TRANSFER;
iocb.handle = out_ep;
iocb.request.setup_or_bulk_transfer.buffer = out_buffer;
iocb.request.setup_or_bulk_transfer.size = 64;
iocb.request.setup_or_bulk_transfer.bytes_transferred = 0;
vos_dev_ioctl(hA,&iocb);

while (iocb.request.setup_or_bulk_transfer.bytes_transferred) {
          // process bytes received from host
}
```

3.10 VOS_IOCTL_USBSLAVE_ENDPOINT_STALL

Description

Force an endpoint to stall on the USB Slave device. An IN, OUT or control endpoint may be stalled. This may be used on the control endpoint when a device does not support a certain SETUP request or on other endpoints as required. If an endpoint it halted then it will return a STALL to a request from the host.

Parameters

The endpoint identifier is passed in the *ep* field of the *usbslave_ioctl_cb_t* structure.

Returns

There is no return value.

Example

```
usbslave_ioctl_cb_t iocb;

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_ENDPOINT_STALL;
iocb.ep = 1;
vos_dev_ioctl(hA,&iocb);
```

3.11 VOS_IOCTL_USBSLAVE_ENDPOINT_CLEAR

Description

Remove a halt state on the USB Slave device. An IN, OUT or control endpoint may be stalled but only IN and OUT endpoints can be cleared by this IOCTL.

Parameters

The endpoint identifier is passed in the *ep* field of the *usbslave_ioctl_cb_t* structure.

Returns

There is no return value.

Example

```
usbslave_ioctl_cb_t iocb;

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_ENDPOINT_CLEAR;
iocb.ep = 1;
vos_dev_ioctl(hA,&iocb);
```

3.12 VOS_IOCTL_USBSLAVE_ENDPOINT_STATE

Description

Returns the halt state of an endpoint on the USB Slave device. If an endpoint it halted then it will return a STALL to a request from the host.

Parameters

The endpoint identifier is passed in the *ep* field of the *usbslave_ioctl_cb_t* structure.

Returns

The return value is zero if the endpoint it not halted and non-zero if it is halted.

Example

```
usbslave_ioctl_cb_t iocb;
char x;

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_ENDPOINT_STATE;
iocb.ep = 1;
iocb.get = &x;
vos_dev_ioctl(hA,&iocb);
```

3.13 VOS_IOCTL_USBSLAVE_SET_LOW_SPEED

Description

Sets the USB Slave device to Low Speed. This is non-reversible. This should be performed as soon as possible after opening the USB Slave device and before host enumeration occurs.

Parameters

There are no parameters.

Returns

There is no return value.

Example

```
usbslave_ioctl_cb_t iocb;

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_SET_LOW_SPEED;
vos_dev_ioctl(hA,&iocb);
```

3.14 VOS_IOCTL_USBSLAVE_DISCONNECT

Description

Sets the USB slave into an un-addressed state and resets the hardware needed to allow the device to be reconnected to a USB host at a later time. This IOCTL can also be used to force a disconnect from a USB host. In order to detect a disconnect for a USB host, a GPIO line must be used. The GPIO may be polled or use an interrupt. When a disconnect from the host is detected, this IOCTL should be called from the application.

Parameters

The set field of the IOCTL control block should be set to 0.

Returns

There is no return value.

Example

```
void handle_disconnect()
{
   // call this function when a disconnect from the USB host has been detected
   usbslave_ioctl_cb_t iocb;

   iocb.ioctl_code = VOS_IOCTL_USBSLAVE_DISCONNECT;
   iocb.set = (void *) 0;
   vos_dev_ioctl(hA,&iocb);
}
```

3.15 VOS_IOCTL_USBSLAVE_GET_BULK_IN_ENDPOINT_HANDLE

Description

Returns a handle for a bulk IN endpoint.

Parameters

The endpoint address is passed in the *ep* field of the *usbslave_ioctl_cb_t* structure. Valid endpoint addresses are in the range 1-7.

Returns

The bulk IN endpoint handle is returned to the location whose address is passed in the *get* field of the *usbslave_ioctl_cb_t* structure.

Example

```
usbslave_ioctl_cb_t iocb;
usbslave_ep_handle_t in_ep;

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_GET_BULK_IN_ENDPOINT_HANDLE;
iocb.ep = 1;
iocb.get = &in_ep;
vos_dev_ioctl(hA,&iocb);
```

3.16 VOS_IOCTL_USBSLAVE_GET_BULK_OUT_ENDPOINT_HANDLE

Description

Returns a handle for a bulk OUT endpoint.

Parameters

The endpoint address is passed in the *ep* field of the *usbslave_ioctl_cb_t* structure. Valid endpoint addresses are in the range 1-7.

Returns

The bulk OUT endpoint handle is returned to the location whose address is passed in the *get* field of the *usbslave_ioctl_cb_t* structure.

Example

```
usbslave_ioctl_cb_t iocb;
usbslave_ep_handle_t out_ep;

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_GET_BULK_OUT_ENDPOINT_HANDLE;
iocb.ep = 2;
iocb.get = &out_ep;
vos_dev_ioctl(hA,&iocb);
```

3.17 VOS_IOCTL_USBSLAVE_GET_INT_IN_ENDPOINT_HANDLE

Description

Returns a handle for an interrupt IN endpoint.

Parameters

The endpoint address is passed in the *ep* field of the *usbslave_ioctl_cb_t* structure. Valid endpoint addresses are in the range 1-7.

Returns

The interrupt IN endpoint handle is returned to the location whose address is passed in the *get* field of the *usbslave_ioctl_cb_t* structure.

Example

```
usbslave_ioctl_cb_t iocb;
usbslave_ep_handle_t in_ep;

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_GET_INT_IN_ENDPOINT_HANDLE;
iocb.ep = 1;
iocb.get = &in_ep;
vos_dev_ioctl(hA,&iocb);
```

3.18 VOS_IOCTL_USBSLAVE_GET_INT_OUT_ENDPOINT_HANDLE

Description

Returns a handle for an interrupt OUT endpoint.

Parameters

The endpoint address is passed in the *ep* field of the *usbslave_ioctl_cb_t* structure. Valid endpoint addresses are in the range 1-7.

Returns

The interrupt OUT endpoint handle is returned to the location whose address is passed in the *get* field of the *usbslave_ioctl_cb_t* structure.

Example

```
usbslave_ioctl_cb_t iocb;
usbslave_ep_handle_t out_ep;

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_GET_INT_OUT_ENDPOINT_HANDLE;
iocb.ep = 2;
iocb.get = &out_ep;
vos_dev_ioctl(hA,&iocb);
```

3.19 VOS_IOCTL_USBSLAVE_GET_ISO_IN_ENDPOINT_HANDLE

Description

Returns a handle for an isochronous IN endpoint.

Parameters

The endpoint address is passed in the *ep* field of the *usbslave_ioctl_cb_t* structure. Valid endpoint addresses are in the range 1-7.

Returns

The isochronous IN endpoint handle is returned to the location whose address is passed in the *get* field of the *usbslave_ioctl_cb_t* structure.

Example

```
usbslave_ioctl_cb_t iocb;
usbslave_ep_handle_t in_ep;

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_GET_ISO_IN_ENDPOINT_HANDLE;
iocb.ep = 1;
iocb.get = &in_ep;
vos_dev_ioctl(hA,&iocb);
```

3.20 VOS_IOCTL_USBSLAVE_GET_ISO_OUT_ENDPOINT_HANDLE

Description

Returns a handle for an isochronous OUT endpoint.

Parameters

The endpoint address is passed in the *ep* field of the *usbslave_ioctl_cb_t* structure. Valid endpoint addresses are in the range 1-7.

Returns

The isochronous OUT endpoint handle is returned to the location whose address is passed in the *get* field of the *usbslave_ioctl_cb_t* structure.

Example

```
usbslave_ioctl_cb_t iocb;
usbslave_ep_handle_t out_ep;

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_GET_ISO_OUT_ENDPOINT_HANDLE;
iocb.ep = 2;
iocb.get = &out_ep;
vos_dev_ioctl(hA,&iocb);
```

3.21 VOS_IOCTL_USBSLAVE_SET_ENDPOINT_MAX_PACKET_SIZE

Description

Set the max packet size for the specified endpoint. The endpoint max packet size can be set to 8, 16, 32 or 64 bytes for a bulk IN, bulk OUT, interrupt IN or interrupt OUT endpoint. Isochronous endpoints do not use the max packet size field.

Parameters

The handle of the endpoint is passed in the *handle* field of the *usbslave_ioctl_cb_t* structure.

The desired maximum packet size is passed in the *ep_max_packet_size* field of the *set* union in the *usbslave_ioctl_cb_t* structure.

Returns

If an invalid endpoint maximum packet size is requested the function will return USBSLAVE_INVALID_PARAMETER. Otherwise, USBSLAVE_OK will be returned.

Example

```
usbslave_ioctl_cb_t iocb;
usbslave_ep_handle_t in_ep;

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_SET_ENDPOINT_MAX_PACKET_SIZE;
iocb.handle = in_ep;
iocb.request.ep_max_packet_size = USBSLAVE_MAX_PACKET_SIZE_64;
vos_dev_ioctl(hA,&iocb);
```

3.22 VOS_IOCTL_USBSLAVE_WAIT_ON_USB_SUSPEND

Description

This call blocks until a SUSPEND signal is received from the host.

Parameters

There are no parameters.

Returns

USBSLAVE_OK will always be returned.

Example

```
usbslave_ioctl_cb_t iocb;

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_WAIT_ON_USB_SUSPEND;
vos_dev_ioctl(hA,&iocb);
```

3.23 VOS_IOCTL_USBSLAVE_WAIT_ON_USB_RESUME

Description

This call blocks until a RESUME signal is received from the host.

Parameters

There are no parameters.

Returns

USBSLAVE_OK will always be returned.

Example

```
usbslave_ioctl_cb_t iocb;

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_WAIT_ON_USB_RESUME;
vos_dev_ioctl(hA,&iocb);
```

3.24 VOS_IOCTL_USBSLAVE_ISSUE_REMOTE_WAKEUP

Description

Issues a remote wakeup request to the host. This IOCTL should not be used unless the USB configuration descriptor indicates that the device is remote wakeup enabled.

Parameters

There are no parameters.

Returns

USBSLAVE_OK will always be returned.

Example

```
usbslave_ioctl_cb_t iocb;

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_ISSUE_REMOTE_WAKEUP;
vos_dev_ioctl(hA,&iocb);
```

4 Contact Information

Head Office – Glasgow, UK

Future Technology Devices International Limited
Unit 1, 2 Seaward Place, Centurion Business Park
Glasgow G41 1HH
United Kingdom
Tel: +44 (0) 141 429 2777
Fax: +44 (0) 141 429 2758

E-mail (Sales)	sales1@ftdichip.com
E-mail (Support)	support1@ftdichip.com
E-mail (General Enquiries)	admin1@ftdichip.com
Web Site URL	http://www.ftdichip.com
Web Shop URL	http://www.ftdichip.com

Branch Office – Taipei, Taiwan

Future Technology Devices International Limited (Taiwan)
2F, No. 516, Sec. 1, NeiHu Road
Taipei 114
Taiwan , R.O.C.
Tel: +886 (0) 2 8791 3570
Fax: +886 (0) 2 8791 3576

E-mail (Sales)	tw.sales1@ftdichip.com
E-mail (Support)	tw.support1@ftdichip.com
E-mail (General Enquiries)	tw.admin1@ftdichip.com
Web Site URL	http://www.ftdichip.com

Branch Office – Hillsboro, Oregon, USA

Future Technology Devices International Limited (USA)
7235 NW Evergreen Parkway, Suite 600
Hillsboro, OR 97123-5803
USA
Tel: +1 (503) 547 0988
Fax: +1 (503) 547 0987

E-Mail (Sales)	us.sales@ftdichip.com
E-Mail (Support)	us.support@ftdichip.com
E-Mail (General Enquiries)	us.admin@ftdichip.com
Web Site URL	http://www.ftdichip.com

Branch Office – Shanghai, China

Future Technology Devices International Limited (China)
Room 408, 317 Xianxia Road,
Shanghai, 200051
China
Tel: +86 21 62351596
Fax: +86 21 62351595

E-mail (Sales)	cn.sales@ftdichip.com
E-mail (Support)	cn.support@ftdichip.com
E-mail (General Enquiries)	cn.admin@ftdichip.com
Web Site URL	http://www.ftdichip.com

Distributor and Sales Representatives

Please visit the Sales Network page of the FTDI Web site for the contact details of our distributor(s) and sales representative(s) in your country.

System and equipment manufacturers and designers are responsible to ensure that their systems, and any Future Technology Devices International Ltd (FTDI) devices incorporated in their systems, meet all applicable safety, regulatory and system-level performance requirements. All application-related information in this document (including application descriptions, suggested FTDI devices and other materials) is provided for reference only. While FTDI has taken care to assure it is accurate, this information is subject to customer confirmation, and FTDI disclaims all liability for system designs and for any applications assistance provided by FTDI. Use of FTDI devices in life support and/or safety applications is entirely at the user's risk, and the user agrees to defend, indemnify and hold harmless FTDI from any and all damages, claims, suits or expense resulting from such use. This document is subject to change without notice. No freedom to use patents or other intellectual property rights is implied by the publication of this document. Neither the whole nor any part of the information contained in, or the product described in this document, may be adapted or reproduced in any material or electronic form without the prior written consent of the copyright holder. Future Technology Devices International Ltd, Unit 1, 2 Seaward Place, Centurion Business Park, Glasgow G41 1HH, United Kingdom. Scotland Registered Company Number: SC136640

5 Appendix A – References

Document References

[1] FTDI Application Note AN_151, *Vinculum II User Guide*, FTDI, 2010. Available from http://www.ftdichip.com/Support/Documents/AppNotes.htm

[2] *Universal Serial Bus Specification Revision 2.0*, USB Implementers Forum, 2000. Available from http://www.usb.org/developers/docs/

Acronyms and Abbreviations

Terms	Description
VNC2	Vinculum II
VOS	Vinculum Operating System

6 Appendix B – Revision History

Revision	Changes	Date
1.0	Initial Release	2011-03-15