Future Technology Devices International Ltd.

Application Note

AN_164

Vinculum-II USB Slave

Writing a Function Driver

Document Reference No.: FT_000373

Version 1.0

Issue Date: 2011-03-15

This application note provides an example of how to implement a function driver for an FTDI Vinculum-II (VNC2) USB Slave device. 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

Table of Contents

1 Introduction

FTDI provides drivers for the hardware peripherals on Vinculum-II (VNC2), and function drivers which enhance the basic hardware driver functionality for a specific purpose [1]. Function drivers are layered over hardware drivers using a standard interface, and are fully integrated with the Device Manager [2].

This application note describes how to implement a function driver for the VNC2 USB Slave [3]. FTDI provides function drivers that allow VNC2 to appear to a host system as a HID device and as an FT232 device, and it is aspects of the implementation of these drivers that provide the basis of this discussion. However, this information can be applied to the implementation of any function driver for the VNC2 USB Slave.

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

This application note should be read in conjunction with [3] which contains full details of all USB Slave IOCTLs referenced here.

1.1 Function Driver Architecture

The relationship between a function driver, its underlying peripheral driver, and the Device Manager is shown in Figure 1.

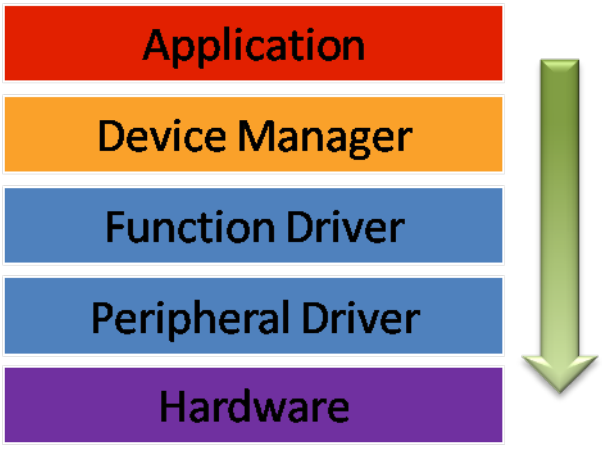

Figure 1: Function Driver Architecture

2 Function Driver Implementation

A function driver has the standard device driver format that includes functions for *init()*, *open()*, *close()*, *read()*, *write()*, *ioctl()* and *interrupt()*. This section describes device-independent implementation of these functions.

2.1 Initialisation

In order to provide access to its device, a function driver has to signal its presence to DeviceManager. The *init()* function typically allocates a context that is used with calls to the function driver, and registers the function driver entry points. The following code fragment shows how *init()* is implemented for the FT232 driver.

```
unsigned char usbslaveft232_init(uint8 vos_dev_num)
{
        vos_driver_t *usbSlaveFt232_cb;
        usbSlaveFt232_context *ctx;

        usbSlaveFt232_cb = vos_malloc(sizeof(vos_driver_t));

        ctx = vos_malloc(sizeof(usbSlaveFt232_context));

        // initialise context

        // Set up function pointers for the driver
        usbSlaveFt232_cb->flags = 0;
        usbSlaveFt232_cb->read = usbSlaveFt232_read;
        usbSlaveFt232_cb->write = usbSlaveFt232_write;
        usbSlaveFt232_cb->ioctl = usbSlaveFt232_ioctl;
        usbSlaveFt232_cb->interrupt = (PF_INT) NULL;
        usbSlaveFt232_cb->open = (PF_OPEN) NULL;
        usbSlaveFt232_cb->close = (PF_CLOSE) NULL;

        // register with device manager
        vos_dev_init(vos_dev_num, usbSlaveFt232_cb, ctx);

        return USBSLAVEFT232_OK;
}
```

Note that driver entry points are optional; it is permissible to pass NULL pointers for functions that are not supported in the function driver. For example, interrupts are handled in the underlying USB Slave driver, so the *interrupt* entry point should be NULL for a USB function driver.

2.2 Open

This function performs device-specific processing. It is called from DeviceManager as part of the handling of a *vos_dev_open()* request. It is rare that this entry point is enabled for a device – indeed, FT232 doesn't support it – but it could be used for context initialisation.

```
void functionDriver_open(functionDriver_context *ctx)
```

2.3 Close

This function performs device-specific processing. It is called from DeviceManager as part of the handling of a *vos_dev_close()* request. This function is more common in function drivers than *open()* – although FT232 doesn't support it either – and it is normally used to close the device gracefully by putting it into a known state.

```
void functionDriver_close(functionDriver_context *ctx)
```

2.4 Read

This function performs a read operation on the device. Its implementation in a function driver depends on the device configuration. FT232 has a relatively simple configuration with one IN and one OUT endpoint, and the *read()* function returns data from the OUT endpoint.

```
unsigned char usbSlaveFt232_read (
        char *xfer,
        unsigned short num_to_read,
        unsigned short *num_read,
        usbSlaveFt232_context *ctx)
{
        *num_read = 0;

        while (num_to_read--) {

               // copy character from OUT endpoint to xfer buffer

               ++*num_read;
        }

        return USBSLAVEFT232_OK;
}
```

2.5 Write

This function performs a write operation on the device. Its implementation in a function driver depends on the device configuration. FT232 has a relatively simple configuration with one IN and one OUT endpoint, and the w*rite()* function passes data to the IN endpoint.

```
unsigned char usbSlaveFt232_write (
        char *xfer,
        unsigned short num_to_write,
        unsigned short *num_written,
        usbSlaveFt232_context *ctx)
{
        *num_written = 0;

        while (num_to_write--) {

               // copy status (if necessary) and character
               // from xfer buffer to IN endpoint

               ++*num_written;
        }

        return USBSLAVEFT232_OK;
}
```

For FT232, note that the copy to the IN endpoint must take into account the proprietary format of IN data.

2.6 Ioctl

This function performs device-specific request processing. The *ioctl()* function handles a different set of requests for each device type, but some requests, for example *attach()* and *detach()*, are common to all function drivers. In addition, by using the *common_ioctl_cb_t* type, a function driver can utilise a set of requests that are common to all drivers, as the following example for FT232 demonstrates.

```
unsigned char usbSlaveFt232_ioctl(common_ioctl_cb_t *cb,usbSlaveFt232_context *ctx)
{
        unsigned char status = USBSLAVEFT232_INVALID_PARAMETER;

        switch (cb->ioctl_code) {

        case VOS_IOCTL_USBSLAVEFT232_ATTACH:
               status = usbSlaveFt232_attach((VOS_HANDLE)cb->set.data, ctx);
               break;

        case VOS_IOCTL_USBSLAVEFT232_DETACH:
               usbSlaveFt232_detach(ctx);
               status = USBSLAVEFT232_OK;
               break;

        case VOS_IOCTL_USBSLAVEFT232_SET_LATENCY:
               status = usbSlaveFt232_set_latency((unsigned char)cb->set.data, ctx);
               break;

        case VOS_IOCTL_COMMON_GET_RX_QUEUE_STATUS:
               cb->get.queue_stat = usbSlaveFt232_get_rx_queue_status(ctx);
               status = USBSLAVEFT232_OK;
               break;

        default:
               break;
        }

        return status;
}
```

VOS_IOCTL_USBSLAVEFT232_ATTACH is an example of a function driver *attach()* function, and *VOS_IOCTL_USBSLAVEFT232_DETACH* is an example of a function driver *detach()* function. *attach()* and *detach()* requests are described in detail in the next section.

VOS_IOCTL_USBSLAVEFT232_SET_LATENCY is an example of a device-specific request.

VOS_IOCTL_COMMON_GET_RX_QUEUE_STATUS is an example of a common request.

2.7 Interrupt

No interrupt entry point needs to be set for USB Slave function drivers. USB Slave interrupts are handled in the underlying USB Slave driver.

3 Function Driver Specifics

This section deals with implementation issues specific to function drivers.

By definition, a function driver is layered over another driver, so a function driver must be able to both establish a connection to its underlying driver, and close that connection down. For these purposes, function drivers implement *attach()* and *detach()* functions respectively, and these functions are implemented as *ioctl* requests.

3.1 Attach

This function attaches a function driver to its underlying driver thus establishing a connection between the two drivers. Although every function driver will have an *attach()* function, its implementation is device-specific.

For FT232, the function driver is attached to the USB Slave driver using the *ioctl* request *VOS_IOCTL_USBSLAVEFT232_ATTACH*. This function communicates with the USB Slave driver to obtain handles for the function device endpoints; these handles are required by other requests that access the function device endpoints. As a result, the device state is set to attached.

```
unsigned char usbSlaveFt232_attach(VOS_HANDLE handle, usbSlaveFt232_context *ctx)
{
        usbslave_ioctl_cb_t iocb;
        unsigned char status = USBSLAVEFT232_OK;

        // save usb slave handle
        ctx->handle = handle;

        if (!ctx->attached) {

              iocb.ioctl_code = VOS_IOCTL_USBSLAVE_GET_CONTROL_ENDPOINT_HANDLE;
              iocb.ep = USBSLAVE_CONTROL_IN;
              iocb.get = &ctx->in_ep0;
              vos_dev_ioctl(ctx->handle,&iocb);

              iocb.ioctl_code = VOS_IOCTL_USBSLAVE_GET_CONTROL_ENDPOINT_HANDLE;
              iocb.ep = USBSLAVE_CONTROL_OUT;
              iocb.get = &ctx->out_ep0;
              vos_dev_ioctl(ctx->handle,&iocb);

              iocb.ioctl_code = VOS_IOCTL_USBSLAVE_GET_BULK_IN_ENDPOINT_HANDLE;
              iocb.ep = 1;
              iocb.get = &ctx->in_ep;
              vos_dev_ioctl(ctx->handle,&iocb);

              iocb.ioctl_code = VOS_IOCTL_USBSLAVE_GET_BULK_OUT_ENDPOINT_HANDLE;
              iocb.ep = 2;
              iocb.get = &ctx->out_ep;
              vos_dev_ioctl(ctx->handle,&iocb);

              ctx->attached = 1;
        }

        return status;
}
```

Before attaching a function driver to its underlying driver, a handle to the underlying driver must be obtained. For FT232, the application code opens the required USB Slave port (A or B), and passes the handle thus obtained in the *VOS_IOCTL_USBSLAVEFT232_ATTACH* request:

```
VOS_HANDLE hA;
VOS_HANDLE hFT232;

void main(void)
{
    common_ioctl_cb_t ft232Attach;

    …

    init_devices();

    // open USB Slave port A
    hA = vos_dev_open(USBSA);

    // open FT232BM
    hFT232 = vos_dev_open(USBSFT232);

    // attach FT232BM to USB Slave port A
    ft232Attach.ioctl_code = VOS_IOCTL_USBSLAVEFT232_ATTACH;
    ft232Attach.set.data = hA;
    vos_dev_ioctl(hFT232,&ft232Attach);

    …

    vos_start_scheduler();

main_loop:

    goto main_loop;
}
```

3.2 Detach

This function detaches a function driver from its underlying driver thus closing down the connection between the two drivers. Although every function driver will have an *detach()* function, its implementation is device-specific.

For FT232, the function driver is detached from the USB Slave driver using the *ioctl* request *VOS_IOCTL_USBSLAVEFT232_DETACH*. The implementation is very simple in this case – the device state is set to not attached.

```
void usbSlaveFt232_detach(usbSlaveFt232_context *ctx)
{
        ctx->attached = 0;

        return;
}
```

4 USB Specifics

This section deals with general implementation issues concerning USB devices.

4.1 Descriptors

The function of a USB device is defined by its set of standard USB descriptors. Descriptor types are defined in the header file USB.h in the VNC2 toolchain, and this section contains example descriptors for an FT232 device.

4.1.1 Device Descriptor

The device descriptor is defined in USB.h as a structure of type *usb_deviceDescriptor_t*. For further details, see Section 9.6.1 in [4].

For an FT232 device, the default device descriptor is:

```
usb_deviceDescriptor_t
FT232_device_descriptor = {
	18,			// bLength
	1,			// bDescriptorType
	0x0200,		// bcdUSB
	0,			// bDeviceClass
	0,			// bDeviceSubClass
	0,			// bDeviceProtocol
	8,			// bMaxPacketSize0
	0x0403,		// idVendor
	0x6001,		// idProduct
	0x0400,		// bcdDevice
	1,			// iManufacturer
	2,			// iProduct
	3,			// iSerialNumber
	1			// bNumConfigurations
};
```

4.1.2 Configuration Descriptor

The configuration descriptor is defined in USB.h as a structure of type *usb_deviceConfigurationDescriptor_t*. For further details, see Section 9.6.3 in [4].

For an FT232 device, the default configuration descriptor is:

```
usb_deviceConfigurationDescriptor_t
FT232_configuration_descriptor = {
	9,                                  // bLength
	USB_DESCRIPTOR_TYPE_CONFIGURATION,  // bDescriptorType
	0x0020,                             // wTotalLength
	1,                                  // bNumInterfaces
	1,                                  // bConfigurationValue
	0,                                  // iConfiguration
	0x80,                               // bmAttributes
	45                                  // bMaxPower
};
```

4.1.3 Interface Descriptor

The interface descriptor is defined in USB.h as a structure of type *usb_deviceInterfaceDescriptor_t*. For further details, see Section 9.6.5 in [4].

For an FT232 device, the default interface descriptor is:

```
usb_deviceInterfaceDescriptor_t
FT232_interface_descriptor = {
        9,                                  // bLength
        USB_DESCRIPTOR_TYPE_INTERFACE,      // bDescriptorType
        0,                                  // bInterfaceNumber
        0,                                  // bAlternateSetting
        2,                                  // bNumEndpoints
        USB_CLASS_VENDOR,                   // bInterfaceClass
        USB_SUBCLASS_ANY,                   // bInterfaceSubClass
        USB_PROTOCOL_ANY,                   // bInterfaceProtocol
        2                                   // iInterface
};
```

4.1.4 Endpoint Descriptor

The endpoint descriptor is defined in USB.h as a structure of type *usb_deviceEndpointDescriptor_t*. For further details, see Section 9.6.6 in [4].

For an FT232 device, the default endpoint descriptor for its BULK OUT endpoint is:

```
usb_deviceEndpointDescriptor_t
FT232_out_endpoint_descriptor = {
        7,                                  // bLength
        USB_DESCRIPTOR_TYPE_ENDPOINT,       // bDescriptorType
        0x02,                               // bEndpointAddress
        2,                                  // bmAttributes
        0x0040,                             // wMaxPacketSize
        0                                   // bInterval
};
```

4.1.5 Zero String Descriptor

The zero string descriptor is defined in USB.h as a structure of type *usb_deviceStringDescriptorZero_t*. For further details, see Section 9.6.7 in [4].

For an FT232 device, the default zero string descriptor is:

```
usb_deviceStringDescriptorZero_t
FT232_zero_string_descriptor = {
        4,                                  // bLength
        USB_DESCRIPTOR_TYPE_STRING,         // bDescriptorType
        USB_LANGID_ENGLISH_UNITED_STATES    // LANGID code zero
};
```

4.1.6 String Descriptor

The string descriptor is defined in USB.h as a structure of type *usb_deviceStringDescriptor_t*. For further details, see Section 9.6.7 in [4].

For an FT232 device, an example string descriptor is the manufacturer string descriptor, and its default value is:

```
usb_deviceStringDescriptor_t
FT232_manufacturer_string_descriptor = {
       10,                                  // bLength
       USB_DESCRIPTOR_TYPE_STRING,          // bDescriptorType
       0x0046,                              // bString
       0x0054,
       0x004d,
       0x0049
};
```

4.2 Transaction Types

A USB Slave function driver must handle each of the USB transaction formats.

4.2.1 Control Transfers

Control transfers have a minimum of two stages: Setup and Status; a Data stage between Setup and Status is optional. During the Setup phase, the host sends a SETUP packet to the control endpoint. The Data phase, if present, consists of one or more IN or OUT transactions to EP0. The Status stage consists of a single transaction on EP0: if there is a Data phase, this transaction follows a change of direction of the data flow; if there was no Data phase, this is a single IN transaction.

4.2.1.1 Receiving a SETUP Packet

Typically, the function driver includes a dedicated thread that handles SETUP packets. The thread sends a *VOS_IOCTL_USBSLAVE_WAIT_SETUP_RCVD* request to the USB Slave driver. This is a blocking request, and returns when a SETUP packet has been received on the USB Slave port. The thread parses the SETUP packet and processes it accordingly.

The SETUP packet may represent a Standard, Class or Vendor request. Different devices handle their own specific combination of requests. The implementation of a thread that demonstrates the general case is shown in the following code:

```
void function_driver_setup(usbSlaveFt232_context *ctx)
{
        usbslave_ioctl_cb_t iocb;
        uint8 bmRequestType;
        uint8 status;

        while (1) {

              iocb.ioctl_code = VOS_IOCTL_USBSLAVE_WAIT_SETUP_RCVD;
              iocb.request.setup_or_bulk_transfer.buffer = &ctx->setup_buffer[0];
              iocb.request.setup_or_bulk_transfer.size = 9;
              vos_dev_ioctl(ctx->handle,&iocb);

              bmRequestType = ctx->setup_buffer[0] & 0x60;
              if (bmRequestType == USB_BMREQUESTTYPE_STANDARD) {
```

```
                        status = standard_request(ctx);
                }
                else if (bmRequestType == USB_BMREQUESTTYPE_CLASS) {
                        status = class_request(ctx);
                }
                else if (bmRequestType == USB_BMREQUESTTYPE_VENDOR) {
                        status = vendor_request(ctx);
                }
        }

        return;
}
```

4.2.1.2 Data Stage

If present, the Data stage consists of a data transfer on either the control IN endpoint or the control OUT endpoint. For example, here is a code fragment that demonstrates sending two bytes on the control IN endpoint in response to a SETUP packet:

```
void send_response(usbSlaveFt232_context *ctx)
{
        uint8 b[2] = { 0xff, 0xff };
        usbslave_ioctl_cb_t iocb;

        iocb.ioctl_code = VOS_IOCTL_USBSLAVE_SETUP_TRANSFER;
        iocb.handle = ctx->in_ep0;
        iocb.request.setup_or_bulk_transfer.buffer = b;
        iocb.request.setup_or_bulk_transfer.size = 2;
        vos_dev_ioctl(ctx->handle,&iocb);
}
```

4.2.1.3 Status Stage

If the SETUP transaction has no Data stage, the Status stage consists of a single IN transaction. This takes the form of a zero-length data packet to the control IN endpoint (EP0), as shown in the following code fragment:

```
void send_zldp(usbSlaveFt232_context *ctx)
{
        usbslave_ioctl_cb_t iocb;

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

If the Data stage consists of OUTs, the Status stage changes the direction of the data flow and consists of a single IN transaction.

If the Data stage consists of IN, the Status stage changes the direction of the data flow and consists of a single OUT transaction.

4.2.2 Bulk Transactions

Handles to the bulk endpoints are obtained. This example is for a device with two bulk endpoints, one IN endpoint and one OUT endpoint; the endpoint addresses are 0x81 and 0x02 respectively. Note that

VOS_IOCTL_USBSLAVE_GET_BULK_IN_ENDPOINT_HANDLE does not require the Direction bit of the endpoint address.

```
usbslave_ioctl_cb_t iocb;

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_GET_BULK_IN_ENDPOINT_HANDLE;
iocb.ep = 1;
iocb.get = &ctx->in_ep;
vos_dev_ioctl(ctx->handle,&iocb);

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_GET_BULK_OUT_ENDPOINT_HANDLE;
iocb.ep = 2;
iocb.get = &ctx->out_ep;
vos_dev_ioctl(ctx->handle,&iocb);
```

Data transfer is performed by a call to *VOS_IOCTL_USBSLAVE_TRANSFER*. For bulk IN transfers:

```
iocb.ioctl_code = VOS_IOCTL_USBSLAVE_TRANSFER;
iocb.handle = ctx->in_ep;
iocb.request.setup_or_bulk_transfer.buffer = &ctx->ep1_in_buffer[0];
iocb.request.setup_or_bulk_transfer.size = (int16) ctx->in_cnt;
vos_dev_ioctl(ctx->handle,&iocb);
```

For bulk OUT transfers:

```
iocb.ioctl_code = VOS_IOCTL_USBSLAVE_TRANSFER;
iocb.handle = ctx->out_ep;
iocb.request.setup_or_bulk_transfer.buffer = ctx->ep2_out_buffer;
iocb.request.setup_or_bulk_transfer.size = OUT_EP_BUF_LEN;
iocb.request.setup_or_bulk_transfer.bytes_transferred = 0;
vos_dev_ioctl(ctx->handle,&iocb);
```

4.2.3 Interrupt Transactions

Handles to the interrupt endpoints are obtained. This example is for a device with an interrupt IN endpoint at address 0x81. Note that *VOS_IOCTL_USBSLAVE_GET_INT_IN_ENDPOINT_HANDLE* does not require the Direction bit of the endpoint address.

```
usbslave_ioctl_cb_t iocb;

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_GET_INT_IN_ENDPOINT_HANDLE;
iocb.ep = 1;
iocb.get = &ctx->in_ep;
vos_dev_ioctl(ctx->handle,&iocb);
```

Data transfer is performed by a call to *VOS_IOCTL_USBSLAVE_TRANSFER*:

```
iocb.ioctl_code = VOS_IOCTL_USBSLAVE_TRANSFER;
iocb.handle = ctx->in_ep;
iocb.request.setup_or_bulk_transfer.buffer = &ctx->ep_in_buffer[0];
iocb.request.setup_or_bulk_transfer.size = (int16) 8; // Report length
vos_dev_ioctl(ctx->handle,&iocb);
```

4.2.4 Isochronous Transactions

Handles to the isochronous endpoints are obtained. This example is for a device with two isochronous endpoints, one IN endpoint and one OUT endpoint; the endpoint addresses are 0x81 and 0x02 respectively. Note that *VOS_IOCTL_USBSLAVE_GET_ISO_IN_ENDPOINT_HANDLE* does not require the Direction bit of the endpoint address.

```
usbslave_ioctl_cb_t iocb;

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_GET_ISO_IN_ENDPOINT_HANDLE;
iocb.ep = 1;
iocb.get = &ctx->in_ep;
vos_dev_ioctl(ctx->handle,&iocb);

iocb.ioctl_code = VOS_IOCTL_USBSLAVE_GET_ISO_OUT_ENDPOINT_HANDLE;
iocb.ep = 2;
iocb.get = &ctx->out_ep;
vos_dev_ioctl(ctx->handle,&iocb);
```

Data transfer is performed by a call to *VOS_IOCTL_USBSLAVE_TRANSFER*. For isochronous IN transfers:

```
iocb.ioctl_code = VOS_IOCTL_USBSLAVE_TRANSFER;
iocb.handle = ctx->in_ep;
iocb.request.setup_or_bulk_transfer.buffer = &ctx->ep1_in_buffer[0];
iocb.request.setup_or_bulk_transfer.size = IN_TRANSFER_SIZE;
vos_dev_ioctl(ctx->handle,&iocb);
```

For isochronous OUT transfers:

```
iocb.ioctl_code = VOS_IOCTL_USBSLAVE_TRANSFER;
iocb.handle = ctx->out_ep;
iocb.request.setup_or_bulk_transfer.buffer = ctx->ep2_out_buffer;
iocb.request.setup_or_bulk_transfer.size = OUT_TRANSFER_SIZE;
iocb.request.setup_or_bulk_transfer.bytes_transferred = 0;
vos_dev_ioctl(ctx->handle,&iocb);
```

4.3 Device Requests

USB device requests are handled in a USB Slave function driver. The reception of SETUP packets was described in the previous section. This section deals with processing the SETUP packets and handling device requests.

The SETUP packet may represent a Standard, Class or Vendor request. For FT232, there are no Class requests, so its thread only handles Standard and Vendor requests; for HID, there are no Vendor requests, so its thread only handles Standard and Class requests. The implementation of a thread that demonstrates the general case is shown in the following code:

```
void function_driver_setup(usbSlaveFt232_context *ctx)
{
        usbslave_ioctl_cb_t iocb;
        uint8 bmRequestType;
        uint8 status;

        while (1) {
               iocb.ioctl_code = VOS_IOCTL_USBSLAVE_WAIT_SETUP_RCVD;
               iocb.request.setup_or_bulk_transfer.buffer = &ctx->setup_buffer[0];
               iocb.request.setup_or_bulk_transfer.size = 9;
```

```
                vos_dev_ioctl(ctx->handle,&iocb);

                bmRequestType = ctx->setup_buffer[0] & 0x60;
                if (bmRequestType == USB_BMREQUESTTYPE_STANDARD) {
                        status = standard_request(ctx);
                }
                else if (bmRequestType == USB_BMREQUESTTYPE_CLASS) {
                        status = class_request(ctx);
                }
                else if (bmRequestType == USB_BMREQUESTTYPE_VENDOR) {
                        status = vendor_request(ctx);
                }
        }

        return;
}
```

4.3.1 Standard Requests

Standard requests are defined in Chapter 9 of the USB specification [4]. Devices must respond to Standard requests. The Standard requests that FT232 responds to are shown in the following code:

```
unsigned char standard_request(usbSlaveFt232_context *ctx)
{
        unsigned char status = USBSLAVE_OK;
        unsigned char *p;
        unsigned char bReq;

        p = ctx->setup_buffer;
        bReq = ctx->setup_buffer[1];

        switch (bReq) {

        case USB_REQUEST_CODE_SET_ADDRESS :
                set_address_request(ctx,*(p+2));
                break;

        case USB_REQUEST_CODE_GET_DESCRIPTOR :
                get_descriptor_request(ctx);
                break;

        case USB_REQUEST_CODE_SET_CONFIGURATION :
                set_configuration_request(ctx,*(p+2));
                break;

        case USB_REQUEST_CODE_CLEAR_FEATURE :
                clear_feature_request(ctx);
                break;

        default:
                break;
        }

        return status;
}
```

4.3.2 Class Requests

Each USB device class supports its own set of class requests. For FT232, there are no Class requests; HID class requests are defined in [5]. The class requests handler for a HID device is shown in the following code:

```
unsigned char HID_class_request(usbSlaveHIDKbd_context *ctx)
{
```

```
    usb_deviceRequest_t *devReq;
    usbslave_ioctl_cb_t iocb;
    unsigned char status = USBSLAVE_OK;
    unsigned char bReq;

    devReq = (usb_deviceRequest_t *)ctx->setup_buffer;
    bReq = devReq->bRequest;

    switch (bReq) {

    case USB_HID_REQUEST_CODE_SET_IDLE:
          class_ack(ctx);
          break;

    case USB_HID_REQUEST_CODE_SET_PROTOCOL:
          class_ack(ctx);
          break;

    case USB_HID_REQUEST_CODE_SET_REPORT:
          // dummy read of one byte
          class_control_out(ctx, (signed)&status, 1);
          ctx->report = 1;
          break;

    default:
          // force a protocol stall
          set_control_ep_halt(ctx);
          break;
    }

    return status;
}
```

4.3.3 Vendor Requests

Vendor requests are device-specific requests. Each device type has its own set of Vendor requests. For FT232, its Vendor request implementation is as follows:

```
// vendor command codes
…
#define FTDI_GET_MODEM_STATUS   0x05
…
#define FTDI_READ_EE            0x90
…


void ft232_zldp(usbSlaveFt232_context *ctx)
{
        usbslave_ioctl_cb_t iocb;

        iocb.ioctl_code = VOS_IOCTL_USBSLAVE_SETUP_TRANSFER;
        iocb.handle = ctx->in_ep0;
        iocb.request.setup_or_bulk_transfer.buffer = (void *) 0;
        iocb.request.setup_or_bulk_transfer.size = 0;
        vos_dev_ioctl(ctx->handle,&iocb);
}

void ft232_read_ee(usbSlaveFt232_context *ctx)
{
        uint8 b[2] = { 0xff, 0xff };
        usbslave_ioctl_cb_t iocb;
```

```
        iocb.ioctl_code = VOS_IOCTL_USBSLAVE_SETUP_TRANSFER;
        iocb.handle = ctx->in_ep0;
        iocb.request.setup_or_bulk_transfer.buffer = b;
        iocb.request.setup_or_bulk_transfer.size = 2;
        vos_dev_ioctl(ctx->handle,&iocb);
}

void ft232_get_modem_status(usbSlaveFt232_context *ctx)
{
        uint8 b[2] = { 0x31, 0x60 };
        usbslave_ioctl_cb_t iocb;

        iocb.ioctl_code = VOS_IOCTL_USBSLAVE_SETUP_TRANSFER;
        iocb.handle = ctx->in_ep0;
        iocb.request.setup_or_bulk_transfer.buffer = b;
        iocb.request.setup_or_bulk_transfer.size = 2;
        vos_dev_ioctl(ctx->handle,&iocb);
}

uint8 ft232_vendor_request(usbSlaveFt232_context *ctx)
{
        switch (ctx->setup_buffer[1]) {

        case FTDI_READ_EE :
               ft232_read_ee(ctx);
               break;

        case FTDI_GET_MODEM_STATUS :
               ft232_get_modem_status(ctx);
               break;

        default:
               ft232_zldp(ctx);
               break;
        }

        return USBSLAVEFT232_OK;
}
```

The functions *ft232_read_ee()* and *ft232_get_modem_status()* show how to send data to the host by performing a VOS_IOCTL_USBSLAVE_SETUP_TRANSFER request on the control IN endpoint.

The function *ft232_zdlp()* shows how to send a zero-length data packet to the host using a VOS_IOCTL_USBSLAVE_SETUP_TRANSFER request on the control IN endpoint.

5 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.

6 Appendix A – References

Document References

[1] FTDI Software Application Development, *Vinculum II Firmware Overview*, FTDI, 2009. Available from http://www.ftdichip.com/Support/Documents/AppNotes.htm

[2] FTDI Software Application Development, *Vinculum II Driver Architecture*, FTDI, 2009. Available from http://www.ftdichip.com/Support/Documents/AppNotes.htm

[3] FTDI Application Note AN_172, *Vinculum II Using the USB Slave Driver*, FTDI, 2011. Available from http://www.ftdichip.com/Support/Documents/AppNotes.htm

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

[5] *Device Class Definitions for Human Interface Devices (HID) Version 1.11*, USB Implementers Forum, 2001. Available from http://www.usb.org/developers/hidpage/

Acronyms and Abbreviations

Terms	Description
FT232	VNC2 USB Slave FT232 function driver.
VOS	Vinculum Operating System
IDE	Integrated Development Environment
VNC2	Vinculum II

7 Appendix B – Revision History

Revision	Changes	Date
1.0	Initial Release	2011-03-15