libusb Developers Guide

Johannes Erdfelt

Table of Contents

List of Tables

Preface

This document's purpose is to explain the API for libusb and how to use it to make a USB aware application

Any suggestions, corrections and comments regarding this document can be sent to the author: Johannes Erdfelt or the libusb developers mailing list.

I. Introduction

Table of Contents

Chapter 1. Overview

This documentation will give an overview of how the v0.1 libusb API works and relates to USB. Work is rapidly progressing on a newer version of libusb, to be v1.0, which will be a redesigned API and is intended to obsolete v0.1. You may want to check the libusb website to see if it is stable and recommended.

This documentation assumes that you have a good understanding of USB and how it works. If you don't have a good understanding of USB, it is recommended you obtain the USB v2.0 specs and read them.

libusb is geared towards USB 1.1, however from the perspective of libusb, USB 2.0 won't be a significant change for libusb

Chapter 2. Current OS support

- Linux (2.2, 2.4 and on)
- FreeBSD, NetBSD and OpenBSD
- Darwin/MacOS X

II. API

This is the external API for applications to use.

The API is relatively lean and designed to have close analogies to the USB specification. The v0.1 API was mostly hacked together and kludged together without much forethought and as a result, it's missing quite a few features. v1.0 is intended to rectify this.

Table of Contents

Chapter 3. Devices and interfaces

The libusb API ties an open device to a specific interface. This means that if you want to claim multiple interfaces on a device, you should open the device multiple times to receive one usb_dev_handle for each interface you want to communicate with. Don't forget to call usb_claim_interface.

Chapter 4. Timeouts

Timeout's in libusb are always specified in milliseconds.

Chapter 5. Data Types

libusb uses both abstracted and non abstracted structures to maintain portability.

Chapter 6. Synchronous

All functions in libusb v0.1 are synchronous, meaning the functions block and wait for the operation to finish or timeout before returning execution to the calling application. Asynchronous operation will be supported in v1.0, but not v0.1.

Chapter 7. Return values

There are two types of return values used in libusb v0.1. The first is a handle returned by usb_open. The second is an int. In all cases where an int is returned, >= 0 is a success and < 0 is an error condition.

III. Functions

Table of Contents

I. Core

These functions comprise the core of libusb. They are used by all applications that utilize libusb.

Table of Contents

usb_init

Name

usb_init -- Initialize libusb

Description

```
void usb_init(void);
```

Just like the name implies, `usb_init` sets up some internal structures. `usb_init` *must* be called before any other libusb functions.

usb_find_busses

Name

usb_find_busses -- Finds all USB busses on system

Description

```
int usb_find_busses(void);
```

`usb_find_busses` will find all of the busses on the system. Returns the number of changes since previous call to this function (total of new busses and busses removed).

usb_find_devices

Name

usb_find_devices -- Find all devices on all USB devices

Description

int usb_find_devices(void);

usb_find_devices will find all of the devices on each bus. This should be called after usb_find_busses. Returns the number of changes since the previous call to this function (total of new device and devices removed).

usb_get_busses

Name

usb_get_busses -- Return the list of USB busses found

Description

struct usb_bus *usb_get_busses(void);

usb_get_busses simply returns the value of the global variable *usb_busses*. This was implemented for those languages that support C calling convention and can use shared libraries, but don't support C global variables (like Delphi).

II. Device operations

This group of functions deal with the device. It allows you to open and close the device as well standard USB operations like setting the configuration, alternate settings, clearing halts and resetting the device. It also provides OS level operations such as claiming and releasing interfaces.

Table of Contents

usb_open

Name

usb_open -- Opens a USB device

Description

```
usb_dev_handle *usb_open(struct *usb_device dev);
```

usb_open is to be used to open up a device for use. usb_open must be called before attempting to perform any operations to the device. Returns a handle used in future communication with the device.

usb_close

Name

usb_close -- Closes a USB device

Description

```
int usb_close(usb_dev_handle *dev);
```

usb_close closes a device opened with usb_open. No further operations may be performed on the handle after usb_close is called. Returns 0 on success or < 0 on error.

usb_set_configuration

Name

usb_set_configuration -- Sets the active configuration of a device

Description

```
int usb_set_configuration(usb_dev_handle *dev, int configuration);
```

usb_set_configuration sets the active configuration of a device. The *configuration* parameter is the value as specified in the descriptor field bConfigurationValue. Returns 0 on success or < 0 on error.

usb_set_altinterface

Name

usb_set_altinterface -- Sets the active alternate setting of the current interface

Description

```
int usb_set_altinterface(usb_dev_handle *dev, int alternate);
```

usb_set_altinterface sets the active alternate setting of the current interface. The *alternate* parameter is the value as specified in the descriptor field bAlternateSetting. Returns 0 on success or < 0 on error.

usb_resetep

Name

usb_resetep -- Resets state for an endpoint

Description

```
int usb_resetep(usb_dev_handle *dev, unsigned int ep);
```

usb_resetep resets all state (like toggles) for the specified endpoint. The *ep* parameter is the value specified in the descriptor field bEndpointAddress. Returns 0 on success or < 0 on error.

Deprecated: usb_resetep is deprecated. You probably want to use usb_clear_halt.

usb_clear_halt

Name

usb_clear_halt -- Clears any halt status on an endpoint

Description

```
int usb_clear_halt(usb_dev_handle *dev, unsigned int ep);
```

usb_clear_halt clears any halt status on the specified endpoint. The *ep* parameter is the value specified in the descriptor field bEndpointAddress. Returns 0 on success or < 0 on error.

usb_reset

Name

usb_reset -- Resets a device

Description

```
int usb_reset(usb_dev_handle *dev);
```

usb_reset resets the specified device by sending a RESET down the port it is connected to. Returns 0 on success or < 0 on error.

Causes re-enumeration: After calling usb_reset, the device will need to re-enumerate and thusly, requires you to find the new device and open a new handle. The handle used to call usb_reset will no longer work.

usb_claim_interface

Name

usb_claim_interface -- Claim an interface of a device

Description

int usb_claim_interface(usb_dev_handle *dev, int interface);

usb_claim_interface claims the interface with the Operating System. The interface parameter is the value as specified in the descriptor field bInterfaceNumber. Returns 0 on success or < 0 on error.

Must be called!: usb_claim_interface *must* be called before you perform any operations related to this interface (like usb_set_altinterface, usb_bulk_write, etc).

Table 1. Return Codes

code	description
-EBUSY	Interface is not available to be claimed
-ENOMEM	Insufficient memory

usb_release_interface

Name

usb_release_interface -- Releases a previously claimed interface

Description

int usb_release_interface(usb_dev_handle *dev, int interface);

usb_release_interface releases an interface previously claimed with usb_claim_interface. The interface parameter is the value as specified in the descriptor field bInterfaceNumber. Returns 0 on success or < 0 on error.

III. Control Transfers

This group of functions allow applications to send messages to the default control pipe.

Table of Contents

usb_control_msg

Name

usb_control_msg -- Send a control message to a device

Description

```
int usb_control_msg(usb_dev_handle *dev, int requesttype, int request, int value, int index, char *bytes, int size, int timeout);
```

usb_control_msg performs a control request to the default control pipe on a device. The parameters mirror the types of the same name in the USB specification. Returns number of bytes written/read or < 0 on error.

usb_get_string

Name

usb_get_string -- Retrieves a string descriptor from a device

Description

```
int usb_get_string(usb_dev_handle *dev, int index, int langid, char *buf, size_t buflen);
```

usb_get_string retrieves the string descriptor specified by index and langid from a device. The string will be returned in Unicode as specified by the USB specification. Returns the number of bytes returned in *buf* or < 0 on error.

usb_get_string_simple

Name

usb_get_string_simple -- Retrieves a string descriptor from a device using the first language

Description

int usb_get_string_simple(usb_dev_handle *dev, int index, char *buf, size_t buflen);

usb_get_string_simple is a wrapper around usb_get_string that retrieves the string description specified by index in the first language for the descriptor and converts it into C style ASCII. Returns number of bytes returned in *buf* or < 0 on error.

usb_get_descriptor

Name

usb_get_descriptor -- Retrieves a descriptor from a device's default control pipe

Description

int usb_get_descriptor(usb_dev_handle *dev, unsigned char type, unsigned char index, void *buf, int size);

usb_get_descriptor retrieves a descriptor from the device identified by the *type* and *index* of the descriptor from the default control pipe. Returns number of bytes read for the descriptor or < 0 on error.

See usb_get_descriptor_by_endpoint for a function that allows the control endpoint to be specified.

usb_get_descriptor_by_endpoint

Name

usb_get_descriptor_by_endpoint -- Retrieves a descriptor from a device

Description

int usb_get_descriptor_by_endpoint(usb_dev_handle *dev, int ep, unsigned char type, unsigned char index, void *buf, int size);

usb_get_descriptor_by_endpoint retrieves a descriptor from the device identified by the *type* and *index* of the descriptor from the control pipe identified by *ep*. Returns number of bytes read for the descriptor or < 0 on error.

IV. Bulk Transfers

This group of functions allow applications to send and receive data via bulk pipes.

Table of Contents

usb_bulk_write

Name

usb_bulk_write -- Write data to a bulk endpoint

Description

```
int usb_bulk_write(usb_dev_handle *dev, int ep, char *bytes, int size, int timeout);
```

usb_bulk_write performs a bulk write request to the endpoint specified by *ep*. Returns number of bytes written on success or < 0 on error.

usb_bulk_read

Name

usb_bulk_read -- Read data from a bulk endpoint

Description

```
int usb_bulk_read(usb_dev_handle *dev, int ep, char *bytes, int size, int timeout);
```

usb_bulk_read performs a bulk read request to the endpoint specified by *ep*. Returns number of bytes read on success or < 0 on error.

V. Interrupt Transfers

This group of functions allow applications to send and receive data via interrupt pipes.

Table of Contents

usb_interrupt_write

Name

usb_interrupt_write -- Write data to an interrupt endpoint

Description

```
int usb_interrupt_write(usb_dev_handle *dev, int ep, char *bytes, int size, int timeout);
```

usb_interrupt_write performs an interrupt write request to the endpoint specified by *ep*. Returns number of bytes written on success or < 0 on error.

usb_interrupt_read

Name

usb_interrupt_read -- Read data from a interrupt endpoint

Description

```
int usb_interrupt_read(usb_dev_handle *dev, int ep, char *bytes, int size, int timeout);
```

usb_interrupt_read performs a interrupt read request to the endpoint specified by *ep*. Returns number of bytes read on success or < 0 on error.

VI. Non Portable

These functions are non portable. They may expose some part of the USB API on one OS or perhaps a couple, but not all. They are all marked with the string _np at the end of the function name.

A C preprocessor macro will be defined if the function is implemented. The form is LIBUSB_HAS_ prepended to the function name, without the leading "usb_", in all caps. For example, if usb_get_driver_np is implemented, LIBUSB_HAS_GET_DRIVER_NP will be defined.

Table of Contents

usb_get_driver_np

Name

usb_get_driver_np -- Get driver name bound to interface

Description

```
int usb_get_driver_np(usb_dev_handle *dev, int interface, char *name, int namelen);
```

This function will obtain the name of the driver bound to the interface specified by the parameter *interface* and place it into the buffer named *name* limited to *namelen* characters. Returns 0 on success or < 0 on error.

Implemented on Linux only.

usb_detach_kernel_driver_np

Name

usb_detach_kernel_driver_np -- Detach kernel driver from interface

Description

```
int usb_detach_kernel_driver_np(usb_dev_handle *dev, int interface);
```

This function will detach a kernel driver from the interface specified by parameter *interface*. Applications using libusb can then try claiming the interface. Returns 0 on success or < 0 on error.

Implemented on Linux only.

IV. Examples

There are some nonintuitive parts of libusb v0.1 that aren't difficult, but are probably easier to understand with some examples.

Table of Contents

Chapter 8. Basic Examples

Before any communication can occur with a device, it needs to be found. This is accomplished by finding all of the busses and then finding all of the devices on all of the busses:

```
struct usb_bus *busses;

usb_init();
usb_find_busses();
usb_find_devices();

busses = usb_get_busses();
```

After this, the application should manually loop through all of the busess and all of the devices and matching the device by whatever criteria is needed:

```
struct usb_bus *bus;
int c, i, a;

/* ... */

for (bus = busses; bus; bus = bus->next) {
struct usb_device *dev;

    for (dev = bus->devices; dev; dev = dev->next) {
        /* Check if this device is a printer */
        if (dev->descriptor.bDeviceClass == 7) {
            /* Open the device, claim the interface and do your processing */
            ...
        }

        /* Loop through all of the configurations */
        for (c = 0; c < dev->descriptor.bNumConfigurations; c++) {
            /* Loop through all of the interfaces */
            for (i = 0; i < dev->config[c].bNumInterfaces; i++) {
                /* Loop through all of the alternate settings */
                for (a = 0; a < dev->config[c].interface[i].num_altsetting; a++) {
                    /* Check if this interface is a printer */
                    if (dev->config[c].interface[i].altsetting[a].bInterfaceClass == 7) {
                            /* Open the device, set the alternate setting, claim the
                                        interface and do your processing */
                        ...
                    }
                }
            }
        }
    }
}
```

Chapter 9. Examples in the source distribution

The tests directory has a program called testlibusb.c. It simply calls libusb to find all of the devices, then iterates through all of the devices and prints out the descriptor dump. It's very simple and as a result, it's of limited usefulness in itself. However, it could serve as a starting point for a new program.

Chapter 10. Other Applications

Another source of examples can be obtained from other applications.

- gPhoto uses libusb to communicate with digital still cameras.
- rio500 utils uses libusb to communicate with SONICblue Rio 500 Digital Audio Player.