

NAME

libusb - USB access library

LIBRARY

USB access library (libusb, -lusb)

SYNOPSIS

#include <libusb.h>

DESCRIPTION

The **libusb** library contains interfaces for directly managing a usb device. The current implementation supports v1.0 of the libusb API.

LIBRARY INITIALISATION AND DEINITIALISATION

*const struct libusb_version * **libusb_get_version**(void)* This function returns version information about LibUSB.

*int **libusb_init**(libusb_context **ctx)* Call this function before any other libusb v1.0 API function, to initialise a valid libusb v1.0 context. If the *ctx* argument is non-NULL, a pointer to the libusb context is stored at the given location. This function returns 0 upon success or LIBUSB_ERROR on failure.

*int **libusb_init_context**(libusb_context **ctx, const struct libusb_init_option [], int num_options)* Call this function before any other libusb v1.0 API function, to initialise a valid libusb v1.0 context. If the *ctx* argument is non-NULL, a pointer to the libusb context is stored at the given location. Additional options, like the USB debug level, may be given using the second and third argument. If no options are needed, simply use `libusb_init()`. This function returns 0 upon success or a LIBUSB_ERROR value on failure.

*void **libusb_exit**(libusb_context *ctx)* Deinitialise libusb. Must be called at the end of the application. Other libusb routines may not be called after this function.

*int **libusb_has_capability**(uint32_t capability)* This function checks the runtime capabilities of **libusb**. This function will return non-zero if the given *capability* is supported, 0 if it is not supported. The valid values for *capability* are:

LIBUSB_CAP_HAS_CAPABILITY

libusb supports `libusb_has_capability()`.

LIBUSB_CAP_HAS_HOTPLUG

libusb supports hotplug notifications.

LIBUSB_CAP_HAS_HID_ACCESS

libusb can access HID devices without requiring user intervention.

LIBUSB_CAP_SUPPORTS_DETACH_KERNEL_DRIVER

libusb supports detaching of the default USB driver with **libusb_detach_kernel_driver()**.

*const char ** **libusb_strerror**(*int code*) Get the ASCII representation of the error given by the *code* argument. This function does not return NULL.

*const char ** **libusb_error_name**(*int code*) Get the ASCII representation of the error enum given by the *code* argument. This function does not return NULL.

void **libusb_set_debug**(*libusb_context *ctx, int level*) Set the debug level to *level*.

ssize_t **libusb_get_device_list**(*libusb_context *ctx, libusb_device ***list*) Populate *list* with the list of usb devices available, adding a reference to each device in the list. All the list entries created by this function must have their reference counter decremented when you are done with them, and the list itself must be freed. This function returns the number of devices in the list or a LIBUSB_ERROR code.

void **libusb_free_device_list**(*libusb_device **list, int unref_devices*) Free the list of devices discovered by **libusb_get_device_list**. If *unref_device* is set to 1 all devices in the list have their reference counter decremented once.

uint8_t **libusb_get_bus_number**(*libusb_device *dev*) Returns the number of the bus contained by the device *dev*.

uint8_t **libusb_get_port_number**(*libusb_device *dev*) Returns the port number which the device given by *dev* is attached to.

int **libusb_get_port_numbers**(*libusb_device *dev, uint8_t *buf, uint8_t bufsize*) Stores, in the buffer *buf* of size *bufsize*, the list of all port numbers from root for the device *dev*.

int **libusb_get_port_path**(*libusb_context *ctx, libusb_device *dev, uint8_t *buf, uint8_t bufsize*)
Deprecated function equivalent to **libusb_get_port_numbers**.

uint8_t **libusb_get_device_address**(*libusb_device *dev*) Returns the *device_address* contained by the device *dev*.

enum libusb_speed **libusb_get_device_speed**(*libusb_device *dev*) Returns the wire speed at which the

device is connected. See the LIBUSB_SPEED_XXX enums for more information. LIBUSB_SPEED_UNKNOWN is returned in case of unknown wire speed.

*int libusb_get_max_packet_size(libusb_device *dev, unsigned char endpoint)* Returns the wMaxPacketSize value on success, LIBUSB_ERROR_NOT_FOUND if the endpoint does not exist and LIBUSB_ERROR_OTHERS on other failure.

*int libusb_get_max_iso_packet_size(libusb_device *dev, unsigned char endpoint)* Returns the packet size multiplied by the packet multiplier on success, LIBUSB_ERROR_NOT_FOUND if the endpoint does not exist and LIBUSB_ERROR_OTHERS on other failure.

*libusb_device * libusb_ref_device(libusb_device *dev)* Increment the reference counter of the device *dev*.

*void libusb_unref_device(libusb_device *dev)* Decrement the reference counter of the device *dev*.

*int libusb_open(libusb_device *dev, libusb_device_handle **devh)* Open a device and obtain a device_handle. Returns 0 on success, LIBUSB_ERROR_NO_MEM on memory allocation problems, LIBUSB_ERROR_ACCESS on permissions problems, LIBUSB_ERROR_NO_DEVICE if the device has been disconnected and a LIBUSB_ERROR code on other errors.

*libusb_device_handle * libusb_open_device_with_vid_pid(libusb_context *ctx, uint16_t vid, uint16_t pid)* A convenience function to open a device by vendor and product IDs *vid* and *pid*. Returns NULL on error.

*void libusb_close(libusb_device_handle *devh)* Close a device handle.

*libusb_device * libusb_get_device(libusb_device_handle *devh)* Get the device contained by *devh*. Returns NULL on error.

*int libusb_get_configuration(libusb_device_handle *devh, int *config)* Returns the value of the current configuration. Returns 0 on success, LIBUSB_ERROR_NO_DEVICE if the device has been disconnected and a LIBUSB_ERROR code on error.

*int libusb_set_configuration(libusb_device_handle *devh, int config)* Set the active configuration to *config* for the device contained by *devh*. This function returns 0 on success, LIBUSB_ERROR_NOT_FOUND if the requested configuration does not exist, LIBUSB_ERROR_BUSY if the interfaces are currently claimed, LIBUSB_ERROR_NO_DEVICE if the device has been disconnected and a LIBUSB_ERROR code on failure.

*int libusb_claim_interface(libusb_device_handle *devh, int interface_number)* Claim an interface in a given *libusb_device_handle devh*. This is a non-blocking function. It returns 0 on success, `LIBUSB_ERROR_NOT_FOUND` if the requested interface does not exist, `LIBUSB_ERROR_BUSY` if a program or driver has claimed the interface, `LIBUSB_ERROR_NO_DEVICE` if the device has been disconnected and a `LIBUSB_ERROR` code on failure.

*int libusb_release_interface(libusb_device_handle *devh, int interface_number)* This function releases an interface. All the claimed interfaces on a device must be released before closing the device. Returns 0 on success, `LIBUSB_ERROR_NOT_FOUND` if the interface was not claimed, `LIBUSB_ERROR_NO_DEVICE` if the device has been disconnected and `LIBUSB_ERROR` on failure.

*int libusb_set_interface_alt_setting(libusb_device_handle *dev, int interface_number, int alternate_setting)* Activate an alternate setting for an interface. Returns 0 on success, `LIBUSB_ERROR_NOT_FOUND` if the interface was not claimed or the requested setting does not exist, `LIBUSB_ERROR_NO_DEVICE` if the device has been disconnected and a `LIBUSB_ERROR` code on failure.

*int libusb_clear_halt(libusb_device_handle *devh, unsigned char endpoint)* Clear an halt/stall for a endpoint. Returns 0 on success, `LIBUSB_ERROR_NOT_FOUND` if the endpoint does not exist, `LIBUSB_ERROR_NO_DEVICE` if the device has been disconnected and a `LIBUSB_ERROR` code on failure.

*int libusb_reset_device(libusb_device_handle *devh)* Perform an USB port reset for an usb device. Returns 0 on success, `LIBUSB_ERROR_NOT_FOUND` if re-enumeration is required or if the device has been disconnected and a `LIBUSB_ERROR` code on failure.

*int libusb_check_connected(libusb_device_handle *devh)* Test if the USB device is still connected. Returns 0 on success, `LIBUSB_ERROR_NO_DEVICE` if it has been disconnected and a `LIBUSB_ERROR` code on failure.

*int libusb_kernel_driver_active(libusb_device_handle *devh, int interface)* Determine if a driver is active on a interface. Returns 0 if no kernel driver is active and 1 if a kernel driver is active, `LIBUSB_ERROR_NO_DEVICE` if the device has been disconnected and a `LIBUSB_ERROR` code on failure.

*int libusb_get_driver(libusb_device_handle *devh, int interface, char *name, int namelen)* or *int libusb_get_driver_np(libusb_device_handle *devh, int interface, char *name, int namelen)* Copy the name of the driver attached to the given *device* and *interface* into the buffer *name* of length *namelen*. Returns 0 on success, `LIBUSB_ERROR_NOT_FOUND` if no kernel driver is attached to the given interface and `LIBUSB_ERROR_INVALID_PARAM` if the interface does not exist. This function is

non-portable. The buffer pointed to by *name* is only zero terminated on success.

int libusb_detach_kernel_driver(*libusb_device_handle *devh, int interface*) or *int libusb_detach_kernel_driver_np*(*libusb_device_handle *devh, int interface*) Detach a kernel driver from an interface. This is needed to claim an interface already claimed by a kernel driver. Returns 0 on success, LIBUSB_ERROR_NOT_FOUND if no kernel driver was active, LIBUSB_ERROR_INVALID_PARAM if the interface does not exist, LIBUSB_ERROR_NO_DEVICE if the device has been disconnected and a LIBUSB_ERROR code on failure. This function is non-portable.

int libusb_attach_kernel_driver(*libusb_device_handle *devh, int interface*) Re-attach an interface kernel driver that was previously detached. Returns 0 on success, LIBUSB_ERROR_INVALID_PARAM if the interface does not exist, LIBUSB_ERROR_NO_DEVICE if the device has been disconnected, LIBUSB_ERROR_BUSY if the driver cannot be attached because the interface is claimed by a program or driver and a LIBUSB_ERROR code on failure.

int libusb_set_auto_detach_kernel_driver(*libusb_device_handle *devh, int enable*) This function enables automatic kernel interface driver detach when an interface is claimed. When the interface is restored the kernel driver is allowed to be re-attached. If the *enable* argument is non-zero the feature is enabled. Else disabled. Returns 0 on success and a LIBUSB_ERROR code on failure.

USB DESCRIPTORS

int libusb_get_device_descriptor(*libusb_device *dev, libusb_device_descriptor *desc*) Get the USB device descriptor for the device *dev*. This is a non-blocking function. Returns 0 on success and a LIBUSB_ERROR code on failure.

int libusb_get_active_config_descriptor(*libusb_device *dev, struct libusb_config_descriptor **config*) Get the USB configuration descriptor for the active configuration. Returns 0 on success, LIBUSB_ERROR_NOT_FOUND if the device is in an unconfigured state and a LIBUSB_ERROR code on error.

int libusb_get_config_descriptor(*libusb_device *dev, uint8_t config_index, libusb_config_descriptor **config*) Get a USB configuration descriptor based on its index *idx*. Returns 0 on success, LIBUSB_ERROR_NOT_FOUND if the configuration does not exist and a LIBUSB_ERROR code on error.

int libusb_get_config_descriptor_by_value(*libusb_device *dev, uint8_t bConfigurationValue, libusb_config_descriptor **config*) Get a USB configuration descriptor with a specific *bConfigurationValue*. This is a non-blocking function which does not send a request through the device. Returns 0 on success, LIBUSB_ERROR_NOT_FOUND if the configuration does not exist and a

LIBUSB_ERROR code on failure.

void libusb_free_config_descriptor(*libusb_config_descriptor *config*) Free a configuration descriptor.

int libusb_get_string_descriptor(*libusb_device_handle *devh, uint8_t desc_idx, uint16_t langid, unsigned char *data, int length*) Retrieve a string descriptor in raw format. Returns the number of bytes actually transferred on success or a negative LIBUSB_ERROR code on failure.

int libusb_get_string_descriptor_ascii(*libusb_device_handle *devh, uint8_t desc_idx, unsigned char *data, int length*) Retrieve a string descriptor in C style ASCII. Returns the positive number of bytes in the resulting ASCII string on success and a LIBUSB_ERROR code on failure.

int libusb_parse_ss_endpoint_comp(*const void *buf, int len, libusb_ss_endpoint_companion_descriptor **ep_comp*) This function parses the USB 3.0 endpoint companion descriptor in host endian format pointed to by *buf* and having a length of *len*. Typically these arguments are the extra and extra_length fields of the endpoint descriptor. On success the pointer to resulting descriptor is stored at the location given by *ep_comp*. Returns zero on success and a LIBUSB_ERROR code on failure. On success the parsed USB 3.0 endpoint companion descriptor must be freed using the `libusb_free_ss_endpoint_comp` function.

void libusb_free_ss_endpoint_comp(*libusb_ss_endpoint_companion_descriptor *ep_comp*) This function is NULL safe and frees a parsed USB 3.0 endpoint companion descriptor given by *ep_comp*.

int libusb_get_ss_endpoint_companion_descriptor(*struct libusb_context *ctx, const struct libusb_endpoint_descriptor *endpoint, struct libusb_ss_endpoint_companion_descriptor **ep_comp*) This function finds and parses the USB 3.0 endpoint companion descriptor given by *endpoint*. Returns zero on success and a LIBUSB_ERROR code on failure. On success the parsed USB 3.0 endpoint companion descriptor must be freed using the `libusb_free_ss_endpoint_companion_descriptor` function.

void libusb_free_ss_endpoint_companion_descriptor(*struct libusb_ss_endpoint_companion_descriptor *ep_comp*) This function is NULL safe and frees a parsed USB 3.0 endpoint companion descriptor given by *ep_comp*.

int libusb_get_bos_descriptor(*libusb_device_handle *handle, struct libusb_bos_descriptor **bos*) This function queries the USB device given by *handle* and stores a pointer to a parsed BOS descriptor into *bos*. Returns zero on success and a LIBUSB_ERROR code on failure. On success the parsed BOS descriptor must be freed using the `libusb_free_bos_descriptor` function.

int libusb_parse_bos_descriptor(*const void *buf, int len, libusb_bos_descriptor **bos*) This function parses a Binary Object Store, BOS, descriptor into host endian format pointed to by *buf* and having a

length of *len*. On success the pointer to resulting descriptor is stored at the location given by *bos*. Returns zero on success and a LIBUSB_ERROR code on failure. On success the parsed BOS descriptor must be freed using the `libusb_free_bos_descriptor` function.

`void libusb_free_bos_descriptor`(*libusb_bos_descriptor *bos*) This function is NULL safe and frees a parsed BOS descriptor given by *bos*.

`int libusb_get_usb_2_0_extension_descriptor`(*struct libusb_context *ctx, struct libusb_bos_dev_capability_descriptor *dev_cap, struct libusb_usb_2_0_extension_descriptor **usb_2_0_extension*) This function parses the USB 2.0 extension descriptor from the descriptor given by *dev_cap* and stores a pointer to the parsed descriptor into *usb_2_0_extension*. Returns zero on success and a LIBUSB_ERROR code on failure. On success the parsed USB 2.0 extension descriptor must be freed using the `libusb_free_usb_2_0_extension_descriptor` function.

`void libusb_free_usb_2_0_extension_descriptor`(*struct libusb_usb_2_0_extension_descriptor *usb_2_0_extension*) This function is NULL safe and frees a parsed USB 2.0 extension descriptor given by *usb_2_0_extension*.

`int libusb_get_ss_usb_device_capability_descriptor`(*struct libusb_context *ctx, struct libusb_bos_dev_capability_descriptor *dev_cap, struct libusb_ss_usb_device_capability_descriptor **ss_usb_device_capability*) This function parses the SuperSpeed device capability descriptor from the descriptor given by *dev_cap* and stores a pointer to the parsed descriptor into *ss_usb_device_capability*. Returns zero on success and a LIBUSB_ERROR code on failure. On success the parsed SuperSpeed device capability descriptor must be freed using the `libusb_free_ss_usb_device_capability_descriptor` function.

`void libusb_free_ss_usb_device_capability_descriptor`(*struct libusb_ss_usb_device_capability_descriptor *ss_usb_device_capability*) This function is NULL safe and frees a parsed SuperSpeed device capability descriptor given by *ss_usb_device_capability*.

`int libusb_get_container_id_descriptor`(*struct libusb_context *ctx, struct libusb_bos_dev_capability_descriptor *dev_cap, struct libusb_container_id_descriptor **container_id*) This function parses the container ID descriptor from the descriptor given by *dev_cap* and stores a pointer to the parsed descriptor into *container_id*. Returns zero on success and a LIBUSB_ERROR code on failure. On success the parsed container ID descriptor must be freed using the `libusb_free_container_id_descriptor` function.

`void libusb_free_container_id_descriptor`(*struct libusb_container_id_descriptor *container_id*) This function is NULL safe and frees a parsed container ID descriptor given by *container_id*.

USB ASYNCHRONOUS I/O

struct libusb_transfer * **libusb_alloc_transfer**(*int iso_packets*) Allocate a transfer with the number of isochronous packet descriptors specified by *iso_packets*. Returns NULL on error.

void **libusb_free_transfer**(*struct libusb_transfer* **tr*) Free a transfer.

int **libusb_submit_transfer**(*struct libusb_transfer* **tr*) This function will submit a transfer and returns immediately. Returns 0 on success, LIBUSB_ERROR_NO_DEVICE if the device has been disconnected and a LIBUSB_ERROR code on other failure.

int **libusb_cancel_transfer**(*struct libusb_transfer* **tr*) This function asynchronously cancels a transfer. Returns 0 on success and a LIBUSB_ERROR code on failure.

USB SYNCHRONOUS I/O

int **libusb_control_transfer**(*libusb_device_handle* **devh*, *uint8_t bmRequestType*, *uint8_t bRequest*, *uint16_t wValue*, *uint16_t wIndex*, *unsigned char* **data*, *uint16_t wLength*, *unsigned int timeout*) Perform a USB control transfer. Returns the actual number of bytes transferred on success, in the range from and including zero up to and including *wLength*. On error a LIBUSB_ERROR code is returned, for example LIBUSB_ERROR_TIMEOUT if the transfer timed out, LIBUSB_ERROR_PIPE if the control request was not supported, LIBUSB_ERROR_NO_DEVICE if the device has been disconnected and another LIBUSB_ERROR code on other failures. The LIBUSB_ERROR codes are all negative.

int **libusb_bulk_transfer**(*struct libusb_device_handle* **devh*, *unsigned char endpoint*, *unsigned char* **data*, *int length*, *int* **transferred*, *unsigned int timeout*) Perform an USB bulk transfer. A timeout value of zero means no timeout. The timeout value is given in milliseconds. Returns 0 on success, LIBUSB_ERROR_TIMEOUT if the transfer timed out, LIBUSB_ERROR_PIPE if the control request was not supported, LIBUSB_ERROR_OVERFLOW if the device offered more data, LIBUSB_ERROR_NO_DEVICE if the device has been disconnected and a LIBUSB_ERROR code on other failure.

int **libusb_interrupt_transfer**(*struct libusb_device_handle* **devh*, *unsigned char endpoint*, *unsigned char* **data*, *int length*, *int* **transferred*, *unsigned int timeout*) Perform an USB Interrupt transfer. A timeout value of zero means no timeout. The timeout value is given in milliseconds. Returns 0 on success, LIBUSB_ERROR_TIMEOUT if the transfer timed out, LIBUSB_ERROR_PIPE if the control request was not supported, LIBUSB_ERROR_OVERFLOW if the device offered more data, LIBUSB_ERROR_NO_DEVICE if the device has been disconnected and a LIBUSB_ERROR code on other failure.

USB STREAMS SUPPORT

int **libusb_alloc_streams**(*libusb_device_handle* **dev*, *uint32_t num_streams*, *unsigned char* **endpoints*,

int num_endpoints) This function verifies that the given number of streams using the given number of endpoints is allowed and allocates the resources needed to use so-called USB streams. Currently only a single stream per endpoint is supported to simplify the internals of LibUSB. This function returns 0 on success or a LIBUSB_ERROR code on failure.

*int libusb_free_streams(libusb_device_handle *dev, unsigned char *endpoints, int num_endpoints)* This function release resources needed for streams usage. Returns 0 on success or a LIBUSB_ERROR code on failure.

*void libusb_transfer_set_stream_id(struct libusb_transfer *transfer, uint32_t stream_id)* This function sets the stream ID for the given USB transfer.

*uint32_t libusb_transfer_get_stream_id(struct libusb_transfer *transfer)* This function returns the stream ID for the given USB transfer. If no stream ID is used a value of zero is returned.

USB EVENTS

*int libusb_try_lock_events(libusb_context *ctx)* Try to acquire the event handling lock. Returns 0 if the lock was obtained and 1 if not.

*void libusb_lock_events(libusb_context *ctx)* Acquire the event handling lock. This function is blocking.

*void libusb_unlock_events(libusb_context *ctx)* Release the event handling lock. This will wake up any thread blocked on **libusb_wait_for_event()**.

*int libusb_event_handling_ok(libusb_context *ctx)* Determine if it still OK for this thread to be doing event handling. Returns 1 if event handling can start or continue. Returns 0 if this thread must give up the events lock.

*int libusb_event_handler_active(libusb_context *ctx)* Determine if an active thread is handling events. Returns 1 if there is a thread handling events and 0 if there are no threads currently handling events.

*void libusb_interrupt_event_handler(libusb_context *ctx)* Causes the **libusb_handle_events()** family of functions to return to the caller one time. The **libusb_handle_events()** functions may be called again after calling this function.

*void libusb_lock_event_waiters(libusb_context *ctx)* Acquire the event_waiters lock. This lock is designed to be obtained in the situation where you want to be aware when events are completed, but some other thread is event handling so calling **libusb_handle_events()** is not allowed.

*void libusb_unlock_event_waiters(libusb_context *ctx)* Release the event_waiters lock.

*int libusb_wait_for_event(libusb_context *ctx, struct timeval *tv)* Wait for another thread to signal completion of an event. Must be called with the event waiters lock held, see **libusb_lock_event_waiters()**. This will block until the timeout expires or a transfer completes or a thread releases the event handling lock through **libusb_unlock_events()**. Returns 0 after a transfer completes or another thread stops event handling, and 1 if the timeout expired.

*int libusb_handle_events_timeout_completed(libusb_context *ctx, struct timeval *tv, int *completed)* Handle any pending events by checking if timeouts have expired and by checking the set of file descriptors for activity. If the *completed* argument is not equal to NULL, this function will loop until a transfer completion callback sets the variable pointed to by the *completed* argument to non-zero. If the *tv* argument is not equal to NULL, this function will return LIBUSB_ERROR_TIMEOUT after the given timeout. Returns 0 on success, or a LIBUSB_ERROR code on failure or timeout.

*int libusb_handle_events_completed(libusb_context *ctx, int *completed)* Handle any pending events by checking the set of file descriptors for activity. If the *completed* argument is not equal to NULL, this function will loop until a transfer completion callback sets the variable pointed to by the *completed* argument to non-zero. Returns 0 on success, or a LIBUSB_ERROR code on failure.

*int libusb_handle_events_timeout(libusb_context *ctx, struct timeval *tv)* Handle any pending events by checking if timeouts have expired and by checking the set of file descriptors for activity. Returns 0 on success, or a LIBUSB_ERROR code on failure or timeout.

*int libusb_handle_events(libusb_context *ctx)* Handle any pending events in blocking mode with a sensible timeout. Returns 0 on success and a LIBUSB_ERROR code on failure.

*int libusb_handle_events_locked(libusb_context *ctx, struct timeval *tv)* Handle any pending events by polling file descriptors, without checking if another thread is already doing so. Must be called with the event lock held.

*int libusb_get_next_timeout(libusb_context *ctx, struct timeval *tv)* Determine the next internal timeout that libusb needs to handle. Returns 0 if there are no pending timeouts, 1 if a timeout was returned, or a LIBUSB_ERROR code on failure or timeout.

*void libusb_set_pollfd_notifiers(libusb_context *ctx, libusb_pollfd_added_cb added_cb, libusb_pollfd_removed_cb remove_cb, void *user_data)* Register notification functions for file descriptor additions/removals. These functions will be invoked for every new or removed file descriptor that libusb uses as an event source.

*const struct libusb_pollfd ** libusb_get_pollfds(libusb_context *ctx)* Retrieve a list of file descriptors that should be polled by your main loop as libusb event sources. Returns a NULL-terminated list on success or NULL on failure.

*int libusb_hotplug_register_callback(libusb_context *ctx, libusb_hotplug_event events, libusb_hotplug_flag flags, int vendor_id, int product_id, int dev_class, libusb_hotplug_callback_fn cb_fn, void *user_data, libusb_hotplug_callback_handle *handle)* This function registers a hotplug filter. The *events* argument select which events makes the hotplug filter trigger. Available event values are LIBUSB_HOTPLUG_EVENT_DEVICE_ARRIVED and LIBUSB_HOTPLUG_EVENT_DEVICE_LEFT. One or more events must be specified. The *vendor_id*, *product_id* and *dev_class* arguments can be set to LIBUSB_HOTPLUG_MATCH_ANY to match any value in the USB device descriptor. Else the specified value is used for matching. If the *flags* argument is set to LIBUSB_HOTPLUG_ENUMERATE, all currently attached and matching USB devices will be passed to the hotplug filter, given by the *cb_fn* argument. Else the *flags* argument should be set to LIBUSB_HOTPLUG_NO_FLAGS. This function returns 0 upon success or a LIBUSB_ERROR code on failure.

*int libusb_hotplug_callback_fn(libusb_context *ctx, libusb_device *device, libusb_hotplug_event event, void *user_data)* The hotplug filter function. If this function returns non-zero, the filter is removed. Else the filter is kept and can receive more events. The *user_data* argument is the same as given when the filter was registered. The *event* argument can be either of LIBUSB_HOTPLUG_EVENT_DEVICE_ARRIVED or LIBUSB_HOTPLUG_EVENT_DEVICE_LEFT.

*void libusb_hotplug_deregister_callback(libusb_context *ctx, libusb_hotplug_callback_handle handle)* This function unregisters a hotplug filter.

LIBUSB VERSION 0.1 COMPATIBILITY

The library is also compliant with LibUSB version 0.1.12.

usb_open() **usb_close()** **usb_get_string()** **usb_get_string_simple()** **usb_get_descriptor_by_endpoint()**
usb_get_descriptor() **usb_parse_descriptor()** **usb_parse_configuration()** **usb_destroy_configuration()**
usb_fetch_and_parse_descriptors() **usb_bulk_write()** **usb_bulk_read()** **usb_interrupt_write()**
usb_interrupt_read() **usb_control_msg()** **usb_set_configuration()** **usb_claim_interface()**
usb_release_interface() **usb_set_altinterface()** **usb_resetep()** **usb_clear_halt()** **usb_reset()** **usb_strerror()**
usb_init() **usb_set_debug()** **usb_find_busses()** **usb_find_devices()** **usb_device()** **usb_get_busses()**
usb_check_connected() **usb_get_driver_np()** **usb_detach_kernel_driver_np()**

SEE ALSO

libusb20(3), usb(4), usbconfig(8), usbdump(8)

<https://libusb.info/>

HISTORY

libusb support first appeared in FreeBSD 8.0.