NAME

usb_fifo_alloc_buffer, **usb_fifo_attach**, **usb_fifo_detach**, **usb_fifo_free_buffer**, **usb_fifo_get_data**, **usb_fifo_get_data_buffer**, **usb_fifo_get_data_error**, **usb_fifo_get_data_linear**, **usb_fifo_put_bytes_max**, **usb_fifo_put_data**, **usb_fifo_put_data_buffer**, **usb_fifo_put_data_error**, **usb_fifo_put_data_linear**, **usb_fifo_reset**, **usb_fifo_softc**, **usb_fifo_wakeup**, **usbd_do_request**, **usbd_do_request_flags**, **usbd_errstr**, **usbd_lookup_id_by_info**, **usbd_lookup_id_by_uaa**, **usbd_transfer_clear_stall**, **usbd_transfer_drain**, **usbd_transfer_pending**, **usbd_transfer_poll**, **usbd_transfer_setup**, **usbd_transfer_start**, **usbd_transfer_stop**, **usbd_transfer_submit**, **usbd_transfer_unsetup**, **usbd_xfer_clr_flag**, **usbd_xfer_frame_data**, **usbd_xfer_frame_len**, **usbd_xfer_get_frame**, **usbd_xfer_get_priv**, **usbd_xfer_is_stalled**, **usbd_xfer_max_framelen**, **usbd_xfer_max_frames**, **usbd_xfer_max_len**, **usbd_xfer_set_flag**, **usbd_xfer_set_frame_data**, **usbd_xfer_set_frame_len**, **usbd_xfer_set_frame_offset**, **usbd_xfer_set_frames**, **usbd_xfer_set_interval**, **usbd_xfer_set_priv**, **usbd_xfer_set_stall**, **usbd_xfer_set_timeout**, **usbd_xfer_softc**, **usbd_xfer_state**, **usbd_xfer_status** - Universal Serial Bus driver programming interface

SYNOPSIS

#include <dev/usb/usb.h>
#include <dev/usb/usbdi.h>
#include <dev/usb/usbdi_util.h>

usb_error_t
usbd_transfer_setup(*struct usb_device *udev, const uint8_t *ifaces, struct usb_xfer **pxfer, const struct usb_config *setup_start, uint16_t n_setup, void *priv_sc, struct mtx *priv_mtx*);

void
usbd_transfer_unsetup(*struct usb_xfer **pxfer, uint16_t n_setup*);

void
usbd_transfer_start(*struct usb_xfer *xfer*);

void
usbd_transfer_stop(*struct usb_xfer *xfer*);

void
usbd_transfer_drain(*struct usb_xfer *xfer*);

DESCRIPTION

The Universal Serial Bus (USB) driver programming interface provides USB peripheral drivers with a host controller independent API for controlling and communicating with USB peripherals. The **usb** module supports both USB Host and USB Device side mode.

USB TRANSFER MANAGEMENT FUNCTIONS

The USB standard defines four types of USB transfers. Control transfers, Bulk transfers, Interrupt transfers and Isochronous transfers. All the transfer types are managed using the following five functions:

usbd_transfer_setup() This function will allocate memory for and initialise an array of USB transfers and all required DMA memory. This function can sleep or block waiting for resources to become available. *udev* is a pointer to "struct usb_device". *ifaces* is an array of interface index numbers to use. See "if_index". *pxfer* is a pointer to an array of USB transfer pointers that are initialized to NULL, and then pointed to allocated USB transfers. *setup_start* is a pointer to an array of USB config structures. *n_setup* is a number telling the USB system how many USB transfers should be setup. *priv_sc* is the private softc pointer, which will be used to initialize "xfer->priv_sc". *priv_mtx* is the private mutex protecting the transfer structure and the softc. This pointer is used to initialize "xfer->priv_mtx". This function returns zero upon success. A non-zero return value indicates failure.

usbd_transfer_unsetup() This function will release the given USB transfers and all allocated resources associated with these USB transfers. *pxfer* is a pointer to an array of USB transfer pointers, that may be NULL, that should be freed by the USB system. *n_setup* is a number telling the USB system how many USB transfers should be unsetup. This function can sleep waiting for USB transfers to complete. This function is NULL safe with regard to the USB transfer structure pointer. It is not allowed to call this function from the USB transfer callback.

usbd_transfer_start() This function will start the USB transfer pointed to by *xfer*, if not already started. This function is always non-blocking and must be called with the so-called private USB mutex locked. This function is NULL safe with regard to the USB transfer structure pointer.

usbd_transfer_stop() This function will stop the USB transfer pointed to by *xfer*, if not already stopped. This function is always non-blocking and must be called with the so-called private USB mutex locked. This function can return before the USB callback has been called. This function is NULL safe with regard to the USB transfer structure pointer. If the transfer was in progress, the callback will called with "USB_ST_ERROR" and "error = USB_ERR_CANCELLED".

usbd_transfer_drain() This function will stop an USB transfer, if not already stopped and wait for any additional USB hardware operations to complete. Buffers that are loaded into DMA using "usbd_xfer_set_frame_data()" can safely be freed after that this function has returned. This function can block the caller and will not return before the USB callback has been called. This function is NULL safe with regard to the USB transfer structure pointer.

USB TRANSFER CALLBACK

The USB callback has three states. USB_ST_SETUP, USB_ST_TRANSFERRED and

USB_ST_ERROR. USB_ST_SETUP is the initial state. After the callback has been called with this state it will always be called back at a later stage in one of the other two states. The USB callback should not restart the USB transfer in case the error cause is USB_ERR_CANCELLED. The USB callback is protected from recursion. That means one can start and stop whatever transfer from the callback of another transfer one desires. Also the transfer that is currently called back. Recursion is handled like this that when the callback that wants to recurse returns it is called one more time.

usbd_transfer_submit() This function should only be called from within the USB callback and is used to start the USB hardware. An USB transfer can have multiple frames consisting of one or more USB packets making up an I/O vector for all USB transfer types.

```
void
usb_default_callback(struct usb_xfer *xfer, usb_error_t error)
{
        int actlen;

        usbd_xfer_status(xfer, &actlen, NULL, NULL, NULL);

        switch (USB_GET_STATE(xfer)) {
        case USB_ST_SETUP:
                /*
                 * Setup xfer frame lengths/count and data
                 */
                usbd_transfer_submit(xfer);
                break;

        case USB_ST_TRANSFERRED:
                /*
                 * Read    usb frame data, if any.
                 * "actlen" has the total length for all frames
                 * transferred.
                 */
                break;

        default: /* Error */
                /*
                 * Print error message and clear stall
                 * for example.
                 */
                break;
```

```
        }
        /*
         * Here it is safe to do something without the private
         * USB mutex locked.
         */
        return;
}
```

USB CONTROL TRANSFERS

An USB control transfer has three parts. First the SETUP packet, then DATA packet(s) and then a STATUS packet. The SETUP packet is always pointed to by frame 0 and the length is set by **usbd_xfer_frame_len**() also if there should not be sent any SETUP packet! If an USB control transfer has no DATA stage, then the number of frames should be set to 1. Else the default number of frames is 2.

Example1: SETUP + STATUS

```
usbd_xfer_set_frames(xfer, 1);
usbd_xfer_set_frame_len(xfer, 0, 8);
usbd_transfer_submit(xfer);
```

Example2: SETUP + DATA + STATUS

```
usbd_xfer_set_frames(xfer, 2);
usbd_xfer_set_frame_len(xfer, 0, 8);
usbd_xfer_set_frame_len(xfer, 1, 1);
usbd_transfer_submit(xfer);
```

Example3: SETUP + DATA + STATUS - split

1st callback:

```
usbd_xfer_set_frames(xfer, 1);
usbd_xfer_set_frame_len(xfer, 0, 8);
usbd_transfer_submit(xfer);
```

2nd callback:

```
/* IMPORTANT: frbuffers[0] must still point at the setup packet! */
usbd_xfer_set_frames(xfer, 2);
usbd_xfer_set_frame_len(xfer, 0, 0);
usbd_xfer_set_frame_len(xfer, 1, 1);
usbd_transfer_submit(xfer);
```

```
Example4: SETUP + STATUS - split
1st callback:
 usbd_xfer_set_frames(xfer, 1);
 usbd_xfer_set_frame_len(xfer, 0, 8);
 usbd_xfer_set_flag(xfer, USB_MANUAL_STATUS);
 usbd_transfer_submit(xfer);

2nd callback:
 usbd_xfer_set_frames(xfer, 1);
 usbd_xfer_set_frame_len(xfer, 0, 0);
 usbd_xfer_clr_flag(xfer, USB_MANUAL_STATUS);
 usbd_transfer_submit(xfer);
```

USB TRANSFER CONFIG

To simply the search for endpoints the **usb** module defines a USB config structure where it is possible to specify the characteristics of the wanted endpoint.

```
struct usb_config {
        bufsize,
        callback
        direction,
        endpoint,
        frames,
        index flags,
        interval,
        timeout,
        type,
};
```

type field selects the USB pipe type. Valid values are: UE_INTERRUPT, UE_CONTROL, UE_BULK, UE_ISOCHRONOUS. The special value UE_BULK_INTR will select BULK and INTERRUPT pipes. This field is mandatory.

endpoint field selects the USB endpoint number. A value of 0xFF, "-1" or "UE_ADDR_ANY" will select the first matching endpoint. This field is mandatory.

direction field selects the USB endpoint direction. A value of "UE_DIR_ANY" will select the first

matching endpoint. Else valid values are: "UE_DIR_IN" and "UE_DIR_OUT". "UE_DIR_IN" and "UE_DIR_OUT" can be binary OR'ed by "UE_DIR_SID" which means that the direction will be swapped in case of USB_MODE_DEVICE. Note that "UE_DIR_IN" refers to the data transfer direction of the "IN" tokens and "UE_DIR_OUT" refers to the data transfer direction of the "OUT" tokens. This field is mandatory.

interval field selects the interrupt interval. The value of this field is given in milliseconds and is independent of device speed. Depending on the endpoint type, this field has different meaning:

UE_INTERRUPT "0" use the default interrupt interval based on endpoint descriptor. "Else" use the given value for polling rate.

UE_ISOCHRONOUS "0" use default. "Else" the value is ignored.

UE_BULK

UE_CONTROL "0" no transfer pre-delay. "Else" a delay as given by this field in milliseconds is inserted before the hardware is started when "usbd_transfer_submit()" is called.

NOTE: The transfer timeout, if any, is started after that the pre-delay has elapsed!

timeout field, if non-zero, will set the transfer timeout in milliseconds. If the "timeout" field is zero and the transfer type is ISOCHRONOUS a timeout of 250ms will be used.

frames field sets the maximum number of frames. If zero is specified it will yield the following results:

UE_BULK xfer->nframes = 1;

UE_INTERRUPT xfer->nframes = 1;

UE_CONTROL xfer->nframes = 2;

UE_ISOCHRONOUS
Not allowed. Will cause an error.

ep_index field allows you to give a number, in case more endpoints match the description, that selects which matching "ep_index" should be used.

if_index field allows you to select which of the interface numbers in the "ifaces" array parameter passed

to "usbd_transfer_setup" that should be used when setting up the given USB transfer.

flags field has type "struct usb_xfer_flags" and allows one to set initial flags an USB transfer. Valid flags are:

force_short_xfer This flag forces the last transmitted USB packet to be short. A short packet has a length of less than "xfer->max_packet_size", which derives from "wMaxPacketSize". This flag can be changed during operation.

short_xfer_ok This flag allows the received transfer length, "xfer->actlen" to be less than "xfer->sumlen" upon completion of a transfer. This flag can be changed during operation.

short_frames_ok
This flag allows the reception of multiple short USB frames. This flag only has effect for BULK and INTERRUPT endpoints and if the number of frames received is greater than 1. This flag can be changed during operation.

pipe_bof This flag causes a failing USB transfer to remain first in the PIPE queue except in the case of "xfer->error" equal to "USB_ERR_CANCELLED". No other USB transfers in the affected PIPE queue will be started until either:

1 The failing USB transfer is stopped using "usbd_transfer_stop()".

2 The failing USB transfer performs a successful transfer.
The purpose of this flag is to avoid races when multiple transfers are queued for execution on an USB endpoint, and the first executing transfer fails leading to the need for clearing of stall for example. In this case this flag is used to prevent the following USB transfers from being executed at the same time the clear-stall command is executed on the USB control endpoint. This flag can be changed during operation.

"BOF" is short for "Block On Failure".

NOTE: This flag should be set on all BULK and INTERRUPT USB transfers which use an endpoint that can be shared between userland and kernel.

proxy_buffer Setting this flag will cause that the total buffer size will be rounded up to the nearest atomic hardware transfer size. The maximum data length of any USB transfer is always stored in the "xfer->max_data_length". For control transfers the USB kernel will allocate additional space for the 8-bytes of SETUP header. These 8-bytes are not

counted by the "xfer->max_data_length" variable. This flag cannot be changed during operation.

ext_buffer Setting this flag will cause that no data buffer will be allocated. Instead the USB client must supply a data buffer. This flag cannot be changed during operation.

manual_status Setting this flag prevents an USB STATUS stage to be appended to the end of the USB control transfer. If no control data is transferred this flag must be cleared. Else an error will be returned to the USB callback. This flag is mostly useful for the USB device side. This flag can be changed during operation.

no_pipe_ok Setting this flag causes the USB_ERR_NO_PIPE error to be ignored. This flag cannot be changed during operation.

stall_pipe

Device Side Mode Setting this flag will cause STALL pids to be sent to the endpoint belonging to this transfer before the transfer is started. The transfer is started at the moment the host issues a clear-stall command on the STALL'ed endpoint. This flag can be changed during operation.

Host Side Mode Setting this flag will cause a clear-stall control request to be executed on the endpoint before the USB transfer is started.

If this flag is changed outside the USB callback function you have to use the "usbd_xfer_set_stall()" and "usbd_transfer_clear_stall()" functions! This flag is automatically cleared after that the stall or clear stall has been executed.

pre_scale_frames

If this flag is set the number of frames specified is assumed to give the buffering time in milliseconds instead of frames. During transfer setup the frames field is pre scaled with the corresponding value for the endpoint and rounded to the nearest number of frames greater than zero. This option only has effect for ISOCHRONOUS transfers.

bufsize field sets the total buffer size in bytes. If this field is zero, "wMaxPacketSize" will be used, multiplied by the "frames" field if the transfer type is ISOCHRONOUS. This is useful for setting up interrupt pipes. This field is mandatory.

NOTE: For control transfers "bufsize" includes the length of the request structure.

callback pointer sets the USB callback. This field is mandatory.

USB LINUX COMPAT LAYER

The **usb** module supports the Linux USB API.

SEE ALSO

libusb(3), usb(4), usbconfig(8)

STANDARDS

The **usb** module complies with the USB 2.0 standard.

HISTORY

The **usb** module has been inspired by the NetBSD USB stack initially written by Lennart Augustsson.

The **usb** module was written by Hans Petter Selasky <*hselasky@FreeBSD.org*>.