## NAME

**usbnet** — common USB Ethernet driver framework

## SYNOPSIS

```
#include <dev/usb/usbnet.h>
```

### Functions offered by usbnet.h

```
void
usbnet_set_link(struct usbnet *un, bool link);

struct ifnet *
usbnet_ifp(struct usbnet *un);

struct ethercom *
usbnet_ec(struct usbnet *un);

struct mii_data *
usbnet_mii(struct usbnet *un);

krndsource_t *
usbnet_rndsrc(struct usbnet *un);

void *
usbnet_softc(struct usbnet *un);

bool
usbnet_havelink(struct usbnet *un);

bool
usbnet_isdying(struct usbnet *un);

void
usbnet_enqueue(struct usbnet *un, uint8_t *buf, size_t buflen,
    int csum_flags, uint32_t csum_data, int mbuf_flags);

void
usbnet_input(struct usbnet *un, uint8_t *buf, size_t buflen);

void
usbnet_attach(struct usbnet *un);

void
usbnet_attach_ifp(struct usbnet *un, unsigned if_flags,
    unsigned if_extflags, const struct usbnet_mii *unm);

int
usbnet_detach(device_t dev, int flags);

int
usbnet_activate(device_t dev, devact_t act);
```

## DESCRIPTION

The **usbnet** framework provides methods usable for USB Ethernet drivers. The framework has support for these features:

- Partial autoconf handling

- USB endpoint pipe handling
- Rx and Tx chain handling
- Generic handlers or support for several struct ifnet callbacks
- Network stack locking protocol
- Interrupt handling

**usbnet** provides many or all of the traditional "softc" members inside *struct usbnet*, which can be used directly as the device softc structure if no additional storage is required. A structure exists for receive and transmit chain management, *struct usbnet_chain*, that tracks the metadata for each transfer descriptor available, minimum of one each for Rx and Tx slot, and will be passed to the Rx and Tx callbacks.

There is a *struct usbnet_ops* structure that provides a number of optional and required callbacks that will be described below.

For autoconfiguration the device attach routine is expected to ensure that this device's *struct usbnet* is the first member of the device softc, if it can not be used directly as the device softc, as well as set up the necessary structure members, find end-points, find the Ethernet address if relevant, call **usbnet_attach**(), set up interface, Ethernet, and MII capabilities, and finally call **usbnet_attach_ifp**(). The device detach routine should free any resources allocated by attach and then call **usbnet_detach**(), possibly directly using **usbnet_detach**() as most consumers have no additional resources not owned and released by the **usbnet** framework itself. The device activate function should be set to **usbnet_activate**().

When bringing an interface up from `if_init`(9), which happens under `IFNET_LOCK`(9), **usbnet** will:

1. call "uno_init" to initialize the hardware for sending and receiving packets,
2. open the USB pipes,
3. allocate Rx and Tx buffers for transfers,
4. call "uno_mcast" to initially program the hardware multicast filter, and finally
5. start the Rx transfers so packets can be received.

See the **RECEIVE AND SEND** section for details on using the chains.

When bringing an interface down, **usbnet** will:

1. abort the USB pipes,
2. call "uno_stop" to stop the hardware from receiving packets (unless the device is detaching),
3. free Rx and Tx buffers for transfers, and
4. close the USB pipes.

For interface ioctl, most of the handling is in the framework. While the interface is running, the optional "uno_mcast" callback is invoked after handling the `SIOCADDMULTI` and `SIOCDELMULTI` ioctl commands to update the hardware's multicast filter from the `ethersubr`(9) lists. The optional "uno_ioctl" callback, which is invoked under `IFNET_LOCK`(9), can be used to program special settings like offload handling.

If ioctl handling requires capturing device-specific ioctls then the "uno_override_ioctl" callback may be used instead to replace the framework's ioctl handler completely (i.e., the replacement should call any generic ioctl handlers such as **ether_ioctl**() as required.) For sending packets, the "uno_tx_prepare" callback must be used to convert an mbuf into a chain buffer ready for transmission.

For devices requiring MII handling there are callbacks for reading and writing registers, and for status change events. Access to all the MII functions is serialized by **usbnet**.

As receive must handle the case of multiple packets in one buffer, the support is split between the driver and the framework. A "uno_rx_loop" callback must be provided that loops over the incoming packet data found in a chain, performs necessary checking and passes the network frame up the stack via either **usbnet_enqueue**() or **usbnet_input**(). Typically Ethernet devices prefer **usbnet_enqueue**().

General accessor functions for *struct usbnet*:

**usbnet_set_link**(*un*, *link*)
Set the link status for this *un* to *link*.

**usbnet_ifp**(*un*)
Returns pointer to this *un*'s *struct ifnet*.

**usbnet_ec**(*un*)
Returns pointer to this *un*'s *struct ethercom*.

**usbnet_mii**(*un*)
Returns pointer to this *un*'s *struct mii_data*.

**usbnet_rndsrc**(*un*)
Returns pointer to this *un*'s *krndsource_t*.

**usbnet_softc**(*un*)
Returns pointer to this *un*'s device softc.

**usbnet_havelink**(*un*)
Returns true if link is active.

**usbnet_isdying**(*un*)
Returns true if device is dying (has been pulled or deactivated, pending detach). This should be used only to abort timeout loops early.

Buffer enqueue handling for *struct usbnet*:

**usbnet_enqueue**(*un*, *buf*, *buflen*, *csum_flags*, *csum_data*, *mbuf_flags*)
Enqueue buffer *buf* for length *buflen* with higher layers, using the provided *csum_flags*, and *csum_data*, which are written directly to the mbuf packet header, and *mbuf_flags*, which is or-ed into the mbuf flags for the created mbuf.

**usbnet_input**(*un*, *buf*, *buflen*)
Enqueue buffer *buf* for length *buflen* with higher layers.

Autoconfiguration handling for *struct usbnet*. See the **AUTOCONFIGURATION** section for more details about these functions.

**usbnet_attach**(*un*)
Initial stage attach of a usb network device. Performs internal initialization and memory allocation only — nothing is published yet.

**usbnet_attach_ifp**(*un*, *if_flags*, *if_extflags*, *unm*)
Final stage attach of usb network device. Publishes the network interface to the rest of the system.

If the passed in *unm* is non-NULL then an MII interface will be created using the values provided in the *struct usbnet_mii* structure, which has these members passed to **mii_attach**():

un_mii_flags Flags.

un_mii_capmask Capability mask.

un_mii_phyloc PHY location.

un_mii_offset PHY offset.

A default *unm* can be set using the **USBNET_MII_DECL_DEFAULT**() macro. The *if_flags* and *if_extflags* will be or-ed into the interface flags and extflags.

**usbnet_detach**(*dev*, *flags*)
Device detach. Stops all activity and frees memory. Usable as driver(9) detach method.

**usbnet_activate**(*dev*, *act*)
Device activate (deactivate) method. Usable as driver(9) activate method.

## AUTOCONFIGURATION

The framework expects the usbnet structure to have these members filled in with valid values or functions:

un_sc Real softc allocated by autoconf and provided to attach, should be set to the usbnet structure if no device-specific softc is needed.

un_dev device_t saved in attach, used for messages mostly.

un_iface
The USB iface handle for data interactions, see **usbd_device2interface_handle**() for more details.

un_udev
The struct usbd_device for this device, provided as the usb_attach_arg's *uaa_device* member.

un_ops Points to a *struct usbnet_ops* structure which contains these members:

*void* (***uno_stop**)(*struct ifnet *ifp*, *int disable*)
Stop hardware activity (optional). Called under IFNET_LOCK(9) when bringing the interface down, except when the device is detaching.

*int* (***uno_ioctl**)(*struct ifnet *ifp*, *u_long cmd*, *void *data*)
Handle driver-specific ioctls (optional). Called under IFNET_LOCK(9).

*void* (***uno_mcast**)(*struct ifnet **)
Program hardware multicast filters from ethersubr(9) lists (optional). Called between, and not during, "uno_init" and "uno_stop".

*int* (***uno_override_ioctl**)(*struct ifnet *ifp*, *u_long cmd*, *void *data*)
Handle all ioctls, including standard ethernet ioctls normally handled internally by **usbnet** (optional). May or may not be called under IFNET_LOCK(9).

*int* (***uno_init**)(*struct ifnet *ifp*)
Initialize hardware activity. Required. Called under IFNET_LOCK(9) when bringing the interface up.

*int* (***uno_read_reg**)(*struct usbnet *un*, *int phy*, *int reg*, *uint16_t *val*)
Read MII register. Required with MII. Serialized with other MII functions, and with "uno_init" and "uno_stop".

*int* (***uno_write_reg**)(*struct usbnet *un*, *int phy*, *int reg*, *uint16_t val*)
Write MII register. Required with MII. Serialized with other MII functions, and with "uno_init" and "uno_stop".

*usbd_status* **(*uno_statchg)**(*struct ifnet *ifp*)
Handle MII status change. Required with MII. Serialized with other MII functions, and with "uno_init" and "uno_stop".

*unsigned* **(*uno_tx_prepare)**(*struct usbnet *un*, *struct mbuf *m*, *struct usbnet_chain *c*)
Prepare an mbuf for transmit. Required. Called sequentially between, and not during, "uno_init". and "uno_stop".

*void* **(*uno_rx_loop)**(*struct usbnet *un*, *struct usbnet_chain *c*, *uint32_t total_len*)
Prepare one or more chain for enqueue. Required. Called sequentially between, and not during, "uno_init" and "uno_stop".

*void* **(*uno_intr)**(*struct usbnet *un*, *usbd_status status*)
Process periodic interrupt (optional). Called sequentially between, and not during, "uno_init" and "uno_stop".

*void* **(*uno_tick)**(*struct usbnet *un*)
Called every second with USB task thread context (optional). Called sequentially between, and not during, "uno_init" and "uno_stop".

un_intr Points to a *struct usbnet_intr* structure which should have these members set:

uni_buf
If non-NULL, points to a buffer passed to **usbd_open_pipe_intr**() in the device init callback, along with the size and interval.

uni_bufsz
Size of interrupt pipe buffer.

uni_interval
Frequency of the interrupt in milliseconds.

un_ed Array of endpoint descriptors. There indexes are provided: USBNET_ENDPT_RX, USBNET_ENDPT_TX, and USBNET_ENDPT_INTR. The Rx and Tx endpoints are required.

un_phyno
MII phy number. Not used by **usbnet**.

un_eaddr
6 bytes of Ethernet address that must be provided before calling **usbnet_attach_ifp**() if the device has Ethernet.

un_flags
Device owned flags word. The **usbnet** framework will not touch this value.

un_rx_xfer_flags
Passed to **usbd_setup_xfer**() for receiving packets.

un_tx_xfer_flags
Passed to **usbd_setup_xfer**() for sending packets.

un_rx_list_cnt
Number of chain elements to allocate for Rx.

un_tx_list_cnt
Number of chain elements to allocate for Tx.

un_rx_bufsz
: Rx buffer size.

un_tx_bufsz
: Tx buffer size.

The device detach and activate callbacks can typically be set to **usbnet_detach**() and **usbnet_activate**() unless device-specific handling is required, in which case, they can be called before or after such handling.

The capabilities described in both *struct ifp* and *struct ethercom* must be set before calling **usbnet_attach_ifp**().

## RECEIVE AND SEND

Receive and send routines are structured around a the *usbnet_cdata* and *usbnet_chain* structures, the `un_ed`, `un_rx_xfer_flags`, and `un_tx_xfer_flags` members, and the **uno_stop**(), **uno_init**(), **uno_tx_prepare**(), and **uno_rx_loop**() callbacks of *usbnet_ops*.

Typically, the device attach routine will fill in members of the *usbnet* structure, as listed in **AUTOCONFIGURATION**. The `un_ed` array should have the `USBNET_ENDPT_RX` and `USBNET_ENDPT_TX` array entries filled in, and optionally the `USBNET_ENDPT_INTR` entry filled in if applicable.

The optional **uno_stop**() callback performs device-specific operations to shutdown the transmit or receive handling.

The **uno_init**() callback both performs device-specific enablement and then calls **usbnet_rx_tx_init**(), which sets up the receive, transmit, and, optionally, the interrupt pipes, as well as starting the receive pipes. All USB transfer setup is handled internally to the framework, and the driver callbacks merely copy data in or out of a chain entry using what is typically a device-specific method.

The **uno_rx_loop**() callback, called sequentially, converts the provided *usbnet_chain* data and length into a series (one or more) of packets that are enqueued with the higher layers using either **usbnet_enqueue**() (for most devices) or **usbnet_input**() for devices that use **if_input**() (This currently relies upon the *struct ifnet* having the "_if_input" member set as well, which is true for current consumers.)

The **uno_tx_prepare**() callback must convert the provided *struct mbuf* into the provided *struct usbnet_chain* performing any device-specific padding, checksum, header or other. Note that this callback must check that it is not attempting to copy more than the chain buffer size, as set in the *usbnet* "un_tx_bufsz" member. This callback is only called once per packet, sequentially.

The *`struct usbnet_chain`* structure which contains a "unc_buf" member which has the chain buffer allocated where data should be copied to or from for receive or transmit operations. It also contains pointers back to the owning *`struct usbnet`*, and the *struct usbd_xfer* associated with this transfer.

## MII

For devices that have MII support these callbacks in *`struct usbnet_ops`* must be provided:

uno_read_reg
: Read an MII register for a particular PHY. Returns standard `errno`(2). Must initialize the result even on failure.

uno_write_reg
: Write an MII register for a particular PHY. Returns standard `errno`(2).

uno_statchg
: Handle a status change event for this interface.

## INTERRUPT PIPE

The interrupt specific callback, "uno_intr", is an optional callback that can be called periodically, registered by **usbnet** using the **usbd_open_pipe_intr**() function (instead of the **usbd_open_pipe**() function.) The **usbnet** framework provides most of the interrupt handling and the callback simply inspects the returned buffer as necessary. To enable the this callback point the *struct usbnet* member "un_intr" to a *struct usbnet_intr* structure with these members set:

uni_buf
: Data buffer for interrupt status relies.

uni_bufsz
: Size of the above buffer.

uni_interval
: Interval in millieconds.

These values will be passed to **usbd_open_pipe_intr**().

## CONVERTING OLD-STYLE DRIVERS

The porting of an older driver to the **usbnet** framework is largely an effort in deleting code. The process involves making these changes:

Headers
: Many headers are included in `usbnet.h` and can be removed from the driver, as well as headers no longer used, such as `callout.h` and `rndsource.h`, etc.

Device softc
: The majority of the driver's existing "softc" structure can likely be replaced with usage of *struct usbnet* and its related functionality. This includes at least the device_t pointer, Ethernet address, the ethercom and mii_data structures, end point descriptors, usbd device, interface, and task and callout structures (both these probably go away entirely) and all the associated watchdog handling, timevals, list size, buffer size and xfer flags for both Rx, and Tx, and interrupt notices, interface flags, device link, PHY number, chain data, locks including Rx, Tx, and MII. There is a driver-only "un_flags" in the *usbnet* structure available for drivers to use.

  Many drivers can use the *usbnet* structure as the device private storage passed to `CFATTACH_DECL_NEW`. Many internal functions to the driver may look better if switched to operate on the device's *usbnet* as, for example, the *usbd_device* value is now available (and must be set by the driver) in the *usbnet*, which may be needed for any call to **usbd_do_request**(). The standard endpoint values must be stored in the **usbnet** "un_ed[]" array.

  As **usbnet** manages xfer chains all code related to the opening, closing, aborting and transferring of data on pipes is performed by the framework based upon the buffer size and more provided in *subnet*, so all code related to them should be deleted.

Interface setup
: The vast majority of interface specific code should be deleted. For device-specific interface values, the *ifnet* flags and exflags can be set, as well as the *ethercom* "ec_capabilities" member, before calling **usbnet_attach_ifp**(). All calls to **ifmedia_init**(), **mii_attach**(), **ifmedia_add**(), **ifmedia_set**(), **if_attach**(), **ether_ifattach**(), **rnd_attach_source**(), and **usbd_add_drv_event**() should be eliminated. The device "ioctl" routine can use the default handling with a callback for additional device specific programming (multicast filters, etc.), which can be empty, or, the override ioctl can be used for heavier requirements. The device "stop" routine is replaced with a simple call that turns off the device-specific transmitter and receiver if necessary, as the framework handles pipes and transfers and buffers.

MII handling
: For devices with MII support the three normal callbacks ( read, write, and status change ) must be converted to *usbnet*. Local "link" variables need to be replaced with accesses to **usbnet_set_link**() and **usbnet_havelink**(). Other ifmedia callbacks that were passed to **ifmedia_init**() should be deleted and any work moved into "uno_statchg".

Receive and Transmit
: The **usbnet** framework handles the majority of handling of both network directions. The interface init routine should keep all of the device specific setup but replace all pipe management with a call to **usbnet_init_rx_tx**(). The typical receive handling will normally be replaced with a receive loop functions that can accept one or more packets, "uno_rx_loop", which can use either **usbnet_enqueue**() or **usbnet_input**() to pass the packets up to higher layers. The typical interface "if_start" function and any additional functions used will normal be replaced with a relatively simple "uno_tx_prepare" function that simply converts an *mbuf* into a *usbnet_chain* useful for this device that will be passed onto **usbd_transfer**(). The framework's handling of the Tx interrupt is all internal.

Interrupt pipe handling
: For devices requiring special handling of the interrupt pipe (i.e., they use the **usbd_open_pipe_intr**() method), most of the interrupt handler should be deleted, leaving only code that inspects the result of the interrupt transfer.

Common errors
: It's common to forget to set link active on devices with MII. Be sure to call **usbent_set_link**() during any status change event.

  Many locking issues are hidden without `LOCKDEBUG`, including hard-hangs. It's highly recommended to develop with `LOCKDEBUG`.

  The *usbnet* "un_ed" array is unsigned and should use "0" as the no-endpoint value.

## SEE ALSO

`usb`(4), `driver`(9), `usbd_status`(9), `usbdi`(9)

## HISTORY

This **usbnet** interface first appeared in NetBSD 9.0. Portions of the original design are based upon ideas from Nick Hudson ⟨`skrll@NetBSD.org`⟩.

## AUTHORS

Matthew R. Green ⟨`mrg@eterna.com.au`⟩