NAME

tap, **vmnet** - Ethernet tunnel software network interface

SYNOPSIS

device tuntap

DESCRIPTION

The **tap** interface is a software loopback mechanism that can be loosely described as the network interface analog of the pty(4), that is, **tap** does for network interfaces what the pty(4) driver does for terminals.

The **tap** driver, like the pty(4) driver, provides two interfaces: an interface like the usual facility it is simulating (an Ethernet network interface in the case of **tap**, or a terminal for pty(4)), and a character-special device "control" interface. A client program transfers Ethernet frames to or from the **tap** "control" interface. The tun(4) interface provides similar functionality at the network layer: a client will transfer IP (by default) packets to or from a tun(4) "control" interface.

The network interfaces are named "tap0", "tap1", etc., one for each control device that has been opened. These Ethernet network interfaces persist until *if_tuntap.ko* module is unloaded, or until removed with "ifconfig destroy" (see below).

tap devices are created using interface cloning. This is done using the "ifconfig tap**N** create" command. This is the preferred method of creating **tap** devices. The same method allows removal of interfaces. For this, use the "ifconfig tap**N** destroy" command.

If the sysctl(8) variable *net.link.tap.devfs_cloning* is non-zero, the **tap** interface permits opens on the special control device */dev/tap*. When this device is opened, **tap** will return a handle for the lowest unused **tap** device (use devname(3) to determine which).

Disabling the legacy devfs cloning functionality may break existing applications which use **tap***, such as VMware and ssh(1). It therefore defaults to being enabled until further notice.*

Control devices (once successfully opened) persist until *if_tuntap.ko* is unloaded or the interface is destroyed.

Each interface supports the usual Ethernet network interface ioctl(2)s and thus can be used with ifconfig(8) like any other Ethernet interface. When the system chooses to transmit an Ethernet frame on the network interface, the frame can be read from the control device (it appears as "input" there); writing an Ethernet frame to the control device generates an input frame on the network interface, as if the (non-existent) hardware had just received it.

The Ethernet tunnel device, normally */dev/tap***N**, is exclusive-open (it cannot be opened if it is already open) and is restricted to the super-user, unless the sysctl(8) variable *net.link.tap.user_open* is non-zero. If the sysctl(8) variable *net.link.tap.up_on_open* is non-zero, the tunnel device will be marked "up" when the control device is opened. A **read**() call will return an error (EHOSTDOWN) if the interface is not "ready". Once the interface is ready, **read**() will return an Ethernet frame if one is available; if not, it will either block until one is or return EWOULDBLOCK, depending on whether non-blocking I/O has been enabled. If the frame is longer than is allowed for in the buffer passed to **read**(), the extra data will be silently dropped.

A write(2) call passes an Ethernet frame in to be "received" on the pseudo-interface. Each **write**() call supplies exactly one frame; the frame length is taken from the amount of data provided to **write**(). Writes will not block; if the frame cannot be accepted for a transient reason (e.g., no buffer space available), it is silently dropped; if the reason is not transient (e.g., frame too large), an error is returned. The following ioctl(2) calls are supported (defined in <*net/if_tap.h*>):

TAPSIFINFO	Set network interface information (line speed and MTU). The type must be the same as returned by TAPGIFINFO or set to IFT_ETHER else the ioctl(2) call will fail. The argument should be a pointer to a *struct tapinfo*.
TAPGIFINFO	Retrieve network interface information (line speed, MTU and type). The argument should be a pointer to a *struct tapinfo*.
TAPSDEBUG	The argument should be a pointer to an *int*; this sets the internal debugging variable to that value. What, if anything, this variable controls is not documented here; see the source code.
TAPGDEBUG	The argument should be a pointer to an *int*; this stores the internal debugging variable’s value into it.
TAPGIFNAME	Retrieve network interface name. The argument should be a pointer to a *struct ifreq*. The interface name will be returned in the *ifr_name* field.
FIONBIO	Turn non-blocking I/O for reads off or on, according as the argument *int*’s value is or is not zero (Writes are always nonblocking).
FIOASYNC	Turn asynchronous I/O for reads (i.e., generation of SIGIO when data is available to be read) off or on, according as the argument *int*’s value is or is not zero.

FIONREAD — If any frames are queued to be read, store the size of the first one into the argument *int*; otherwise, store zero.

TIOCSPGRP — Set the process group to receive SIGIO signals, when asynchronous I/O is enabled, to the argument *int* value.

TIOCGPGRP — Retrieve the process group value for SIGIO signals into the argument *int* value.

SIOCGIFADDR — Retrieve the Media Access Control (MAC) address of the "remote" side. This command is used by the VMware port and expected to be executed on descriptor, associated with control device (usually */dev/vmnet***N** or */dev/tap***N**). The *buffer*, which is passed as the argument, is expected to have enough space to store the MAC address. At the open time both "local" and "remote" MAC addresses are the same, so this command could be used to retrieve the "local" MAC address.

SIOCSIFADDR — Set the Media Access Control (MAC) address of the "remote" side. This command is used by VMware port and expected to be executed on a descriptor, associated with control device (usually */dev/vmnet***N**).

The control device also supports select(2) for read; selecting for write is pointless, and always succeeds, since writes are always non-blocking.

On the last close of the data device, the interface is brought down (as if with "ifconfig tap**N** down") and has all of its configured addresses deleted unless the device is a *VMnet* device, or has IFF_LINK0 flag set. All queued frames are thrown away. If the interface is up when the data device is not open, output frames are thrown away rather than letting them pile up.

The **tap** device can also be used with the VMware port as a replacement for the old *VMnet* device driver. *VMnet* devices do not ifconfig(8) themselves down when the control device is closed. Everything else is the same.

In addition to the above mentioned ioctl(2) calls, there is an additional one for the VMware port.

VMIO_SIOCSIFFLAGS — VMware SIOCSIFFLAGS.

SEE ALSO

inet(4), intro(4), tun(4)