Linux Networking Documentation

Release 4.16.0-rc4+

The kernel development community

March 08, 2018

CONTENTS

Contents:

CHAPTER

ONE

BATMAN-ADV

Batman advanced is a new approach to wireless networking which does no longer operate on the IP basis. Unlike the batman daemon, which exchanges information using UDP packets and sets routing tables, batman-advanced operates on ISO/OSI Layer 2 only and uses and routes (or better: bridges) Ethernet Frames. It emulates a virtual network switch of all nodes participating. Therefore all nodes appear to be link local, thus all higher operating protocols won't be affected by any changes within the network. You can run almost any protocol above batman advanced, prominent examples are: IPv4, IPv6, DHCP, IPX.

Batman advanced was implemented as a Linux kernel driver to reduce the overhead to a minimum. It does not depend on any (other) network driver, and can be used on wifi as well as ethernet lan, vpn, etc ... (anything with ethernet-style layer 2).

Configuration

Load the batman-adv module into your kernel:

```
$ insmod batman-adv.ko
```

The module is now waiting for activation. You must add some interfaces on which batman can operate. After loading the module batman advanced will scan your systems interfaces to search for compatible interfaces. Once found, it will create subfolders in the `/sys` directories of each supported interface, e.g.:

```
$ ls /sys/class/net/eth0/batman_adv/
elp_interval iface_status mesh_iface throughput_override
```

If an interface does not have the `batman_adv` subfolder, it probably is not supported. Not supported interfaces are: loopback, non-ethernet and batman's own interfaces.

Note: After the module was loaded it will continuously watch for new interfaces to verify the compatibility. There is no need to reload the module if you plug your USB wifi adapter into your machine after batman advanced was initially loaded.

The batman-adv soft-interface can be created using the iproute2 tool `ip`:

```
$ ip link add name bat0 type batadv
```

To activate a given interface simply attach it to the `bat0` interface:

```
$ ip link set dev eth0 master bat0
```

Repeat this step for all interfaces you wish to add. Now batman starts using/broadcasting on this/these interface(s).

By reading the "iface_status" file you can check its status:

```
$ cat /sys/class/net/eth0/batman_adv/iface_status
active
```

To deactivate an interface you have to detach it from the "bat0" interface:

```
$ ip link set dev eth0 nomaster
```

All mesh wide settings can be found in batman's own interface folder:

```
$ ls /sys/class/net/bat0/mesh/
aggregated_ogms       fragmentation isolation_mark routing_algo
ap_isolation          gw_bandwidth  log_level      vlan0
bonding               gw_mode       multicast_mode
bridge_loop_avoidance gw_sel_class  network_coding
distributed_arp_table hop_penalty   orig_interval
```

There is a special folder for debugging information:

```
$ ls /sys/kernel/debug/batman_adv/bat0/
bla_backbone_table log         neighbors         transtable_local
bla_claim_table    mcast_flags originators
dat_cache          nc          socket
gateways           nc_nodes    transtable_global
```

Some of the files contain all sort of status information regarding the mesh network. For example, you can view the table of originators (mesh participants) with:

```
$ cat /sys/kernel/debug/batman_adv/bat0/originators
```

Other files allow to change batman's behaviour to better fit your requirements. For instance, you can check the current originator interval (value in milliseconds which determines how often batman sends its broadcast packets):

```
$ cat /sys/class/net/bat0/mesh/orig_interval
1000
```

and also change its value:

```
$ echo 3000 > /sys/class/net/bat0/mesh/orig_interval
```

In very mobile scenarios, you might want to adjust the originator interval to a lower value. This will make the mesh more responsive to topology changes, but will also increase the overhead.

Usage

To make use of your newly created mesh, batman advanced provides a new interface "bat0" which you should use from this point on. All interfaces added to batman advanced are not relevant any longer because batman handles them for you. Basically, one "hands over" the data by using the batman interface and batman will make sure it reaches its destination.

The "bat0" interface can be used like any other regular interface. It needs an IP address which can be either statically configured or dynamically (by using DHCP or similar services):

```
NodeA: ip link set up dev bat0
NodeA: ip addr add 192.168.0.1/24 dev bat0

NodeB: ip link set up dev bat0
NodeB: ip addr add 192.168.0.2/24 dev bat0
NodeB: ping 192.168.0.1
```

Note: In order to avoid problems remove all IP addresses previously assigned to interfaces now used by batman advanced, e.g.:

```
$ ip addr flush dev eth0
```

Logging/Debugging

All error messages, warnings and information messages are sent to the kernel log. Depending on your operating system distribution this can be read in one of a number of ways. Try using the commands: `dmesg`, `logread`, or looking in the files `/var/log/kern.log` or `/var/log/syslog`. All batman-adv messages are prefixed with "batman-adv:" So to see just these messages try:

```
$ dmesg | grep batman-adv
```

When investigating problems with your mesh network, it is sometimes necessary to see more detail debug messages. This must be enabled when compiling the batman-adv module. When building batman-adv as part of kernel, use "make menuconfig" and enable the option `B.A.T.M.A.N. debugging` (`CONFIG_BATMAN_ADV_DEBUG=y`).

Those additional debug messages can be accessed using a special file in debugfs:

```
$ cat /sys/kernel/debug/batman_adv/bat0/log
```

The additional debug output is by default disabled. It can be enabled during run time. Following log_levels are defined:

0	All debug output disabled
1	Enable messages related to routing / flooding / broadcasting
2	Enable messages related to route added / changed / deleted
4	Enable messages related to translation table operations
8	Enable messages related to bridge loop avoidance
16	Enable messages related to DAT, ARP snooping and parsing
32	Enable messages related to network coding
64	Enable messages related to multicast
128	Enable messages related to throughput meter
255	Enable all messages

The debug output can be changed at runtime using the file `/sys/class/net/bat0/mesh/log_level`. e.g.:

```
$ echo 6 > /sys/class/net/bat0/mesh/log_level
```

will enable debug messages for when routes change.

Counters for different types of packets entering and leaving the batman-adv module are available through ethtool:

```
$ ethtool --statistics bat0
```

batctl

As batman advanced operates on layer 2, all hosts participating in the virtual switch are completely transparent for all protocols above layer 2. Therefore the common diagnosis tools do not work as expected. To overcome these problems, batctl was created. At the moment the batctl contains ping, traceroute, tcpdump and interfaces to the kernel module settings.

For more information, please see the manpage (`man batctl`).

batctl is available on https://www.open-mesh.org/

Contact

Please send us comments, experiences, questions, anything :)

IRC: #batman on irc.freenode.org

Mailing-list: b.a.t.m.a.n@open-mesh.org (optional subscription at https://lists.open-mesh.org/mm/listinfo/b.a.t.m.a.n)

You can also contact the Authors:

- Marek Lindner <mareklindner@neomailbox.ch>
- Simon Wunderlich <sw@simonwunderlich.de>

CHAPTER

TWO

SOCKETCAN - CONTROLLER AREA NETWORK

Overview / What is SocketCAN

The socketcan package is an implementation of CAN protocols (Controller Area Network) for Linux. CAN is a networking technology which has widespread use in automation, embedded devices, and automotive fields. While there have been other CAN implementations for Linux based on character devices, SocketCAN uses the Berkeley socket API, the Linux network stack and implements the CAN device drivers as network interfaces. The CAN socket API has been designed as similar as possible to the TCP/IP protocols to allow programmers, familiar with network programming, to easily learn how to use CAN sockets.

Motivation / Why Using the Socket API

There have been CAN implementations for Linux before SocketCAN so the question arises, why we have started another project. Most existing implementations come as a device driver for some CAN hardware, they are based on character devices and provide comparatively little functionality. Usually, there is only a hardware-specific device driver which provides a character device interface to send and receive raw CAN frames, directly to/from the controller hardware. Queueing of frames and higher-level transport protocols like ISO-TP have to be implemented in user space applications. Also, most character-device implementations support only one single process to open the device at a time, similar to a serial interface. Exchanging the CAN controller requires employment of another device driver and often the need for adaption of large parts of the application to the new driver's API.

SocketCAN was designed to overcome all of these limitations. A new protocol family has been implemented which provides a socket interface to user space applications and which builds upon the Linux network layer, enabling use all of the provided queueing functionality. A device driver for CAN controller hardware registers itself with the Linux network layer as a network device, so that CAN frames from the controller can be passed up to the network layer and on to the CAN protocol family module and also vice-versa. Also, the protocol family module provides an API for transport protocol modules to register, so that any number of transport protocols can be loaded or unloaded dynamically. In fact, the can core module alone does not provide any protocol and cannot be used without loading at least one additional protocol module. Multiple sockets can be opened at the same time, on different or the same protocol module and they can listen/send frames on different or the same CAN IDs. Several sockets listening on the same interface for frames with the same CAN ID are all passed the same received matching CAN frames. An application wishing to communicate using a specific transport protocol, e.g. ISO-TP, just selects that protocol when opening the socket, and then can read and write application data byte streams, without having to deal with CAN-IDs, frames, etc.

Similar functionality visible from user-space could be provided by a character device, too, but this would lead to a technically inelegant solution for a couple of reasons:

- **Intricate usage:** Instead of passing a protocol argument to socket(2) and using bind(2) to select a CAN interface and CAN ID, an application would have to do all these operations using ioctl(2)s.
- **Code duplication:** A character device cannot make use of the Linux network queueing code, so all that code would have to be duplicated for CAN networking.

- **Abstraction:** In most existing character-device implementations, the hardware-specific device driver for a CAN controller directly provides the character device for the application to work with. This is at least very unusual in Unix systems for both, char and block devices. For example you don't have a character device for a certain UART of a serial interface, a certain sound chip in your computer, a SCSI or IDE controller providing access to your hard disk or tape streamer device. Instead, you have abstraction layers which provide a unified character or block device interface to the application on the one hand, and a interface for hardware-specific device drivers on the other hand. These abstractions are provided by subsystems like the tty layer, the audio subsystem or the SCSI and IDE subsystems for the devices mentioned above.

  The easiest way to implement a CAN device driver is as a character device without such a (complete) abstraction layer, as is done by most existing drivers. The right way, however, would be to add such a layer with all the functionality like registering for certain CAN IDs, supporting several open file descriptors and (de)multiplexing CAN frames between them, (sophisticated) queueing of CAN frames, and providing an API for device drivers to register with. However, then it would be no more difficult, or may be even easier, to use the networking framework provided by the Linux kernel, and this is what SocketCAN does.

The use of the networking framework of the Linux kernel is just the natural and most appropriate way to implement CAN for Linux.

SocketCAN Concept

As described in *Motivation / Why Using the Socket API* the main goal of SocketCAN is to provide a socket interface to user space applications which builds upon the Linux network layer. In contrast to the commonly known TCP/IP and ethernet networking, the CAN bus is a broadcast-only(!) medium that has no MAC-layer addressing like ethernet. The CAN-identifier (can_id) is used for arbitration on the CAN-bus. Therefore the CAN-IDs have to be chosen uniquely on the bus. When designing a CAN-ECU network the CAN-IDs are mapped to be sent by a specific ECU. For this reason a CAN-ID can be treated best as a kind of source address.

Receive Lists

The network transparent access of multiple applications leads to the problem that different applications may be interested in the same CAN-IDs from the same CAN network interface. The SocketCAN core module - which implements the protocol family CAN - provides several high efficient receive lists for this reason. If e.g. a user space application opens a CAN RAW socket, the raw protocol module itself requests the (range of) CAN-IDs from the SocketCAN core that are requested by the user. The subscription and unsubscription of CAN-IDs can be done for specific CAN interfaces or for all(!) known CAN interfaces with the can_rx_(un)register() functions provided to CAN protocol modules by the SocketCAN core (see *SocketCAN Core Module*). To optimize the CPU usage at runtime the receive lists are split up into several specific lists per device that match the requested filter complexity for a given use-case.

Local Loopback of Sent Frames

As known from other networking concepts the data exchanging applications may run on the same or different nodes without any change (except for the according addressing information):

```
 ___   ___   ___                   _______   ___
| _ | | _ | | _ |                 | _   _ | | _ |
||A|| ||B|| ||C||                 ||A| |B|| ||C||
|___| |___| |___|                 |_______| |___|
  |     |     |                       |       |
-----------------(1)- CAN bus -(2)---------------
```

To ensure that application A receives the same information in the example (2) as it would receive in example (1) there is need for some kind of local loopback of the sent CAN frames on the appropriate node.

The Linux network devices (by default) just can handle the transmission and reception of media dependent frames. Due to the arbitration on the CAN bus the transmission of a low prio CAN-ID may be delayed by the reception of a high prio CAN frame. To reflect the correct *[0] traffic on the node the loopback of the sent data has to be performed right after a successful transmission. If the CAN network interface is not capable of performing the loopback for some reason the SocketCAN core can do this task as a fallback solution. See *Local Loopback of Sent Frames* for details (recommended).

The loopback functionality is enabled by default to reflect standard networking behaviour for CAN applications. Due to some requests from the RT-SocketCAN group the loopback optionally may be disabled for each separate socket. See sockopts from the CAN RAW sockets in *RAW Protocol Sockets with can_filters (SOCK_RAW)*.

Network Problem Notifications

The use of the CAN bus may lead to several problems on the physical and media access control layer. Detecting and logging of these lower layer problems is a vital requirement for CAN users to identify hardware issues on the physical transceiver layer as well as arbitration problems and error frames caused by the different ECUs. The occurrence of detected errors are important for diagnosis and have to be logged together with the exact timestamp. For this reason the CAN interface driver can generate so called Error Message Frames that can optionally be passed to the user application in the same way as other CAN frames. Whenever an error on the physical layer or the MAC layer is detected (e.g. by the CAN controller) the driver creates an appropriate error message frame. Error messages frames can be requested by the user application using the common CAN filter mechanisms. Inside this filter definition the (interested) type of errors may be selected. The reception of error messages is disabled by default. The format of the CAN error message frame is briefly described in the Linux header file “include/uapi/linux/can/error.h”.

How to use SocketCAN

Like TCP/IP, you first need to open a socket for communicating over a CAN network. Since SocketCAN implements a new protocol family, you need to pass PF_CAN as the first argument to the socket(2) system call. Currently, there are two CAN protocols to choose from, the raw socket protocol and the broadcast manager (BCM). So to open a socket, you would write:

```
s = socket(PF_CAN, SOCK_RAW, CAN_RAW);
```

and:

```
s = socket(PF_CAN, SOCK_DGRAM, CAN_BCM);
```

respectively. After the successful creation of the socket, you would normally use the bind(2) system call to bind the socket to a CAN interface (which is different from TCP/IP due to different addressing - see *SocketCAN Concept*). After binding (CAN_RAW) or connecting (CAN_BCM) the socket, you can read(2) and write(2) from/to the socket or use send(2), sendto(2), sendmsg(2) and the recv* counterpart operations on the socket as usual. There are also CAN specific socket options described below.

The basic CAN frame structure and the sockaddr structure are defined in include/linux/can.h:

```
struct can_frame {
        canid_t can_id;  /* 32 bit CAN_ID + EFF/RTR/ERR flags */
        __u8    can_dlc; /* frame payload length in byte (0 .. 8) */
        __u8    __pad;   /* padding */
        __u8    __res0;  /* reserved / padding */
        __u8    __res1;  /* reserved / padding */
```

[0] you really like to have this when you’re running analyser tools like ‘candump’ or ‘cansniffer’ on the (same) node.

```
        __u8    data[8] __attribute__((aligned(8)));
};
```

The alignment of the (linear) payload data[] to a 64bit boundary allows the user to define their own structs and unions to easily access the CAN payload. There is no given byteorder on the CAN bus by default. A read(2) system call on a CAN_RAW socket transfers a struct can_frame to the user space.

The sockaddr_can structure has an interface index like the PF_PACKET socket, that also binds to a specific interface:

```
struct sockaddr_can {
        sa_family_t can_family;
        int         can_ifindex;
        union {
                /* transport protocol class address info (e.g. ISOTP) */
                struct { canid_t rx_id, tx_id; } tp;

                /* reserved for future CAN protocols address information */
        } can_addr;
};
```

To determine the interface index an appropriate ioctl() has to be used (example for CAN_RAW sockets without error checking):

```
int s;
struct sockaddr_can addr;
struct ifreq ifr;

s = socket(PF_CAN, SOCK_RAW, CAN_RAW);

strcpy(ifr.ifr_name, "can0" );
ioctl(s, SIOCGIFINDEX, &ifr);

addr.can_family = AF_CAN;
addr.can_ifindex = ifr.ifr_ifindex;

bind(s, (struct sockaddr *)&addr, sizeof(addr));

(..)
```

To bind a socket to all(!) CAN interfaces the interface index must be 0 (zero). In this case the socket receives CAN frames from every enabled CAN interface. To determine the originating CAN interface the system call recvfrom(2) may be used instead of read(2). To send on a socket that is bound to 'any' interface sendto(2) is needed to specify the outgoing interface.

Reading CAN frames from a bound CAN_RAW socket (see above) consists of reading a struct can_frame:

```
struct can_frame frame;

nbytes = read(s, &frame, sizeof(struct can_frame));

if (nbytes < 0) {
        perror("can raw socket read");
        return 1;
}

/* paranoid check ... */
if (nbytes < sizeof(struct can_frame)) {
        fprintf(stderr, "read: incomplete CAN frame\n");
        return 1;
}

/* do something with the received CAN frame */
```

Writing CAN frames can be done similarly, with the write(2) system call:

```
nbytes = write(s, &frame, sizeof(struct can_frame));
```

When the CAN interface is bound to 'any' existing CAN interface (addr.can_ifindex = 0) it is recommended to use recvfrom(2) if the information about the originating CAN interface is needed:

```
struct sockaddr_can addr;
struct ifreq ifr;
socklen_t len = sizeof(addr);
struct can_frame frame;

nbytes = recvfrom(s, &frame, sizeof(struct can_frame),
                  0, (struct sockaddr*)&addr, &len);

/* get interface name of the received CAN frame */
ifr.ifr_ifindex = addr.can_ifindex;
ioctl(s, SIOCGIFNAME, &ifr);
printf("Received a CAN frame from interface %s", ifr.ifr_name);
```

To write CAN frames on sockets bound to 'any' CAN interface the outgoing interface has to be defined certainly:

```
strcpy(ifr.ifr_name, "can0");
ioctl(s, SIOCGIFINDEX, &ifr);
addr.can_ifindex = ifr.ifr_ifindex;
addr.can_family  = AF_CAN;

nbytes = sendto(s, &frame, sizeof(struct can_frame),
                0, (struct sockaddr*)&addr, sizeof(addr));
```

An accurate timestamp can be obtained with an ioctl(2) call after reading a message from the socket:

```
struct timeval tv;
ioctl(s, SIOCGSTAMP, &tv);
```

The timestamp has a resolution of one microsecond and is set automatically at the reception of a CAN frame.

Remark about CAN FD (flexible data rate) support:

Generally the handling of CAN FD is very similar to the formerly described examples. The new CAN FD capable CAN controllers support two different bitrates for the arbitration phase and the payload phase of the CAN FD frame and up to 64 bytes of payload. This extended payload length breaks all the kernel interfaces (ABI) which heavily rely on the CAN frame with fixed eight bytes of payload (struct can_frame) like the CAN_RAW socket. Therefore e.g. the CAN_RAW socket supports a new socket option CAN_RAW_FD_FRAMES that switches the socket into a mode that allows the handling of CAN FD frames and (legacy) CAN frames simultaneously (see *RAW Socket Option CAN_RAW_FD_FRAMES*).

The struct canfd_frame is defined in include/linux/can.h:

```
struct canfd_frame {
        canid_t can_id;  /* 32 bit CAN_ID + EFF/RTR/ERR flags */
        __u8    len;     /* frame payload length in byte (0 .. 64) */
        __u8    flags;   /* additional flags for CAN FD */
        __u8    __res0;  /* reserved / padding */
        __u8    __res1;  /* reserved / padding */
        __u8    data[64] __attribute__((aligned(8)));
};
```

The struct canfd_frame and the existing struct can_frame have the can_id, the payload length and the payload data at the same offset inside their structures. This allows to handle the different structures very similar. When the content of a struct can_frame is copied into a struct canfd_frame all structure elements can be used as-is - only the data[] becomes extended.

When introducing the struct canfd_frame it turned out that the data length code (DLC) of the struct can_frame was used as a length information as the length and the DLC has a 1:1 mapping in the range of 0 .. 8. To preserve the easy handling of the length information the canfd_frame.len element contains a plain length value from 0 .. 64. So both canfd_frame.len and can_frame.can_dlc are equal and contain a length information and no DLC. For details about the distinction of CAN and CAN FD capable devices and the mapping to the bus-relevant data length code (DLC), see *CAN FD (Flexible Data Rate) Driver Support*.

The length of the two CAN(FD) frame structures define the maximum transfer unit (MTU) of the CAN(FD) network interface and skbuff data length. Two definitions are specified for CAN specific MTUs in include/linux/can.h:

```
#define CAN_MTU   (sizeof(struct can_frame))   == 16  => 'legacy' CAN frame
#define CANFD_MTU (sizeof(struct canfd_frame)) == 72  => CAN FD frame
```

RAW Protocol Sockets with can_filters (SOCK_RAW)

Using CAN_RAW sockets is extensively comparable to the commonly known access to CAN character devices. To meet the new possibilities provided by the multi user SocketCAN approach, some reasonable defaults are set at RAW socket binding time:

- The filters are set to exactly one filter receiving everything
- The socket only receives valid data frames (=> no error message frames)
- The loopback of sent CAN frames is enabled (see *Local Loopback of Sent Frames*)
- The socket does not receive its own sent frames (in loopback mode)

These default settings may be changed before or after binding the socket. To use the referenced definitions of the socket options for CAN_RAW sockets, include <linux/can/raw.h>.

RAW socket option CAN_RAW_FILTER

The reception of CAN frames using CAN_RAW sockets can be controlled by defining 0 .. n filters with the CAN_RAW_FILTER socket option.

The CAN filter structure is defined in include/linux/can.h:

```
struct can_filter {
        canid_t can_id;
        canid_t can_mask;
};
```

A filter matches, when:

```
<received_can_id> & mask == can_id & mask
```

which is analogous to known CAN controllers hardware filter semantics. The filter can be inverted in this semantic, when the CAN_INV_FILTER bit is set in can_id element of the can_filter structure. In contrast to CAN controller hardware filters the user may set 0 .. n receive filters for each open socket separately:

```
struct can_filter rfilter[2];

rfilter[0].can_id   = 0x123;
rfilter[0].can_mask = CAN_SFF_MASK;
rfilter[1].can_id   = 0x200;
rfilter[1].can_mask = 0x700;

setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, &rfilter, sizeof(rfilter));
```

To disable the reception of CAN frames on the selected CAN_RAW socket:

```
setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, NULL, 0);
```

To set the filters to zero filters is quite obsolete as to not read data causes the raw socket to discard the received CAN frames. But having this 'send only' use-case we may remove the receive list in the Kernel to save a little (really a very little!) CPU usage.

CAN Filter Usage Optimisation

The CAN filters are processed in per-device filter lists at CAN frame reception time. To reduce the number of checks that need to be performed while walking through the filter lists the CAN core provides an optimized filter handling when the filter subscription focusses on a single CAN ID.

For the possible 2048 SFF CAN identifiers the identifier is used as an index to access the corresponding subscription list without any further checks. For the 2^29 possible EFF CAN identifiers a 10 bit XOR folding is used as hash function to retrieve the EFF table index.

To benefit from the optimized filters for single CAN identifiers the CAN_SFF_MASK or CAN_EFF_MASK have to be set into can_filter.mask together with set CAN_EFF_FLAG and CAN_RTR_FLAG bits. A set CAN_EFF_FLAG bit in the can_filter.mask makes clear that it matters whether a SFF or EFF CAN ID is subscribed. E.g. in the example from above:

```
rfilter[0].can_id   = 0x123;
rfilter[0].can_mask = CAN_SFF_MASK;
```

both SFF frames with CAN ID 0x123 and EFF frames with 0xXXXXX123 can pass.

To filter for only 0x123 (SFF) and 0x12345678 (EFF) CAN identifiers the filter has to be defined in this way to benefit from the optimized filters:

```
struct can_filter rfilter[2];

rfilter[0].can_id   = 0x123;
rfilter[0].can_mask = (CAN_EFF_FLAG | CAN_RTR_FLAG | CAN_SFF_MASK);
rfilter[1].can_id   = 0x12345678 | CAN_EFF_FLAG;
rfilter[1].can_mask = (CAN_EFF_FLAG | CAN_RTR_FLAG | CAN_EFF_MASK);

setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, &rfilter, sizeof(rfilter));
```

RAW Socket Option CAN_RAW_ERR_FILTER

As described in *Network Problem Notifications* the CAN interface driver can generate so called Error Message Frames that can optionally be passed to the user application in the same way as other CAN frames. The possible errors are divided into different error classes that may be filtered using the appropriate error mask. To register for every possible error condition CAN_ERR_MASK can be used as value for the error mask. The values for the error mask are defined in linux/can/error.h:

```
can_err_mask_t err_mask = ( CAN_ERR_TX_TIMEOUT | CAN_ERR_BUSOFF );

setsockopt(s, SOL_CAN_RAW, CAN_RAW_ERR_FILTER,
           &err_mask, sizeof(err_mask));
```

RAW Socket Option CAN_RAW_LOOPBACK

To meet multi user needs the local loopback is enabled by default (see *Local Loopback of Sent Frames* for details). But in some embedded use-cases (e.g. when only one application uses the CAN bus) this loopback functionality can be disabled (separately for each socket):

```
int loopback = 0; /* 0 = disabled, 1 = enabled (default) */

setsockopt(s, SOL_CAN_RAW, CAN_RAW_LOOPBACK, &loopback, sizeof(loopback));
```

RAW socket option CAN_RAW_RECV_OWN_MSGS

When the local loopback is enabled, all the sent CAN frames are looped back to the open CAN sockets that registered for the CAN frames' CAN-ID on this given interface to meet the multi user needs. The reception of the CAN frames on the same socket that was sending the CAN frame is assumed to be unwanted and therefore disabled by default. This default behaviour may be changed on demand:

```
int recv_own_msgs = 1; /* 0 = disabled (default), 1 = enabled */

setsockopt(s, SOL_CAN_RAW, CAN_RAW_RECV_OWN_MSGS,
           &recv_own_msgs, sizeof(recv_own_msgs));
```

RAW Socket Option CAN_RAW_FD_FRAMES

CAN FD support in CAN_RAW sockets can be enabled with a new socket option CAN_RAW_FD_FRAMES which is off by default. When the new socket option is not supported by the CAN_RAW socket (e.g. on older kernels), switching the CAN_RAW_FD_FRAMES option returns the error -ENOPROTOOPT.

Once CAN_RAW_FD_FRAMES is enabled the application can send both CAN frames and CAN FD frames. OTOH the application has to handle CAN and CAN FD frames when reading from the socket:

```
CAN_RAW_FD_FRAMES enabled:  CAN_MTU and CANFD_MTU are allowed
CAN_RAW_FD_FRAMES disabled: only CAN_MTU is allowed (default)
```

Example:

```
[ remember: CANFD_MTU == sizeof(struct canfd_frame) ]

struct canfd_frame cfd;

nbytes = read(s, &cfd, CANFD_MTU);

if (nbytes == CANFD_MTU) {
        printf("got CAN FD frame with length %d\n", cfd.len);
        /* cfd.flags contains valid data */
} else if (nbytes == CAN_MTU) {
        printf("got legacy CAN frame with length %d\n", cfd.len);
        /* cfd.flags is undefined */
} else {
        fprintf(stderr, "read: invalid CAN(FD) frame\n");
        return 1;
}

/* the content can be handled independently from the received MTU size */

printf("can_id: %X data length: %d data: ", cfd.can_id, cfd.len);
for (i = 0; i < cfd.len; i++)
        printf("%02X ", cfd.data[i]);
```

When reading with size CANFD_MTU only returns CAN_MTU bytes that have been received from the socket a legacy CAN frame has been read into the provided CAN FD structure. Note that the canfd_frame.flags data field is not specified in the struct can_frame and therefore it is only valid in CANFD_MTU sized CAN FD frames.

Implementation hint for new CAN applications:

To build a CAN FD aware application use struct canfd_frame as basic CAN data structure for CAN_RAW based applications. When the application is executed on an older Linux kernel and switching the CAN_RAW_FD_FRAMES socket option returns an error: No problem. You'll get legacy CAN frames or CAN FD frames and can process them the same way.

When sending to CAN devices make sure that the device is capable to handle CAN FD frames by checking if the device maximum transfer unit is CANFD_MTU. The CAN device MTU can be retrieved e.g. with a SIOCGIFMTU ioctl() syscall.

RAW socket option CAN_RAW_JOIN_FILTERS

The CAN_RAW socket can set multiple CAN identifier specific filters that lead to multiple filters in the af_can.c filter processing. These filters are indenpendent from each other which leads to logical OR'ed filters when applied (see *RAW socket option CAN_RAW_FILTER*).

This socket option joines the given CAN filters in the way that only CAN frames are passed to user space that matched *all* given CAN filters. The semantic for the applied filters is therefore changed to a logical AND.

This is useful especially when the filterset is a combination of filters where the CAN_INV_FILTER flag is set in order to notch single CAN IDs or CAN ID ranges from the incoming traffic.

RAW Socket Returned Message Flags

When using recvmsg() call, the msg->msg_flags may contain following flags:

MSG_DONTROUTE: set when the received frame was created on the local host.

MSG_CONFIRM: set when the frame was sent via the socket it is received on. This flag can be interpreted as a 'transmission confirmation' when the CAN driver supports the echo of frames on driver level, see *Local Loopback of Sent Frames* and *Local Loopback of Sent Frames*. In order to receive such messages, CAN_RAW_RECV_OWN_MSGS must be set.

Broadcast Manager Protocol Sockets (SOCK_DGRAM)

The Broadcast Manager protocol provides a command based configuration interface to filter and send (e.g. cyclic) CAN messages in kernel space.

Receive filters can be used to down sample frequent messages; detect events such as message contents changes, packet length changes, and do time-out monitoring of received messages.

Periodic transmission tasks of CAN frames or a sequence of CAN frames can be created and modified at runtime; both the message content and the two possible transmit intervals can be altered.

A BCM socket is not intended for sending individual CAN frames using the struct can_frame as known from the CAN_RAW socket. Instead a special BCM configuration message is defined. The basic BCM configuration message used to communicate with the broadcast manager and the available operations are defined in the linux/can/bcm.h include. The BCM message consists of a message header with a command ('opcode') followed by zero or more CAN frames. The broadcast manager sends responses to user space in the same form:

```
struct bcm_msg_head {
        __u32 opcode;                   /* command */
        __u32 flags;                    /* special flags */
        __u32 count;                    /* run 'count' times with ival1 */
        struct timeval ival1, ival2;    /* count and subsequent interval */
        canid_t can_id;                 /* unique can_id for task */
        __u32 nframes;                  /* number of can_frames following */
        struct can_frame frames[0];
};
```

The aligned payload 'frames' uses the same basic CAN frame structure defined at the beginning of *RAW Socket Option CAN_RAW_FD_FRAMES* and in the include/linux/can.h include. All messages to the broadcast manager from user space have this structure.

Note a CAN_BCM socket must be connected instead of bound after socket creation (example without error checking):

```
int s;
struct sockaddr_can addr;
struct ifreq ifr;

s = socket(PF_CAN, SOCK_DGRAM, CAN_BCM);

strcpy(ifr.ifr_name, "can0");
ioctl(s, SIOCGIFINDEX, &ifr);

addr.can_family = AF_CAN;
addr.can_ifindex = ifr.ifr_ifindex;

connect(s, (struct sockaddr *)&addr, sizeof(addr));

(..)
```

The broadcast manager socket is able to handle any number of in flight transmissions or receive filters concurrently. The different RX/TX jobs are distinguished by the unique can_id in each BCM message. However additional CAN_BCM sockets are recommended to communicate on multiple CAN interfaces. When the broadcast manager socket is bound to 'any' CAN interface (=> the interface index is set to zero) the configured receive filters apply to any CAN interface unless the sendto() syscall is used to overrule the 'any' CAN interface index. When using recvfrom() instead of read() to retrieve BCM socket messages the originating CAN interface is provided in can_ifindex.

Broadcast Manager Operations

The opcode defines the operation for the broadcast manager to carry out, or details the broadcast managers response to several events, including user requests.

Transmit Operations (user space to broadcast manager):

TX_SETUP: Create (cyclic) transmission task.

TX_DELETE: Remove (cyclic) transmission task, requires only can_id.

TX_READ: Read properties of (cyclic) transmission task for can_id.

TX_SEND: Send one CAN frame.

Transmit Responses (broadcast manager to user space):

TX_STATUS: Reply to TX_READ request (transmission task configuration).

TX_EXPIRED: Notification when counter finishes sending at initial interval 'ival1'. Requires the TX_COUNTEVT flag to be set at TX_SETUP.

Receive Operations (user space to broadcast manager):

RX_SETUP: Create RX content filter subscription.

RX_DELETE: Remove RX content filter subscription, requires only can_id.

RX_READ: Read properties of RX content filter subscription for can_id.

Receive Responses (broadcast manager to user space):

RX_STATUS: Reply to RX_READ request (filter task configuration).

RX_TIMEOUT: Cyclic message is detected to be absent (timer ival1 expired).

RX_CHANGED: BCM message with updated CAN frame (detected content change). Sent on first message received or on receipt of revised CAN messages.

Broadcast Manager Message Flags

When sending a message to the broadcast manager the 'flags' element may contain the following flag definitions which influence the behaviour:

SETTIMER: Set the values of ival1, ival2 and count

STARTTIMER: Start the timer with the actual values of ival1, ival2 and count. Starting the timer leads simultaneously to emit a CAN frame.

TX_COUNTEVT: Create the message TX_EXPIRED when count expires

TX_ANNOUNCE: A change of data by the process is emitted immediately.

TX_CP_CAN_ID: Copies the can_id from the message header to each subsequent frame in frames. This is intended as usage simplification. For TX tasks the unique can_id from the message header may differ from the can_id(s) stored for transmission in the subsequent struct can_frame(s).

RX_FILTER_ID: Filter by can_id alone, no frames required (nframes=0).

RX_CHECK_DLC: A change of the DLC leads to an RX_CHANGED.

RX_NO_AUTOTIMER: Prevent automatically starting the timeout monitor.

RX_ANNOUNCE_RESUME: If passed at RX_SETUP and a receive timeout occurred, a RX_CHANGED message will be generated when the (cyclic) receive restarts.

TX_RESET_MULTI_IDX: Reset the index for the multiple frame transmission.

RX_RTR_FRAME: Send reply for RTR-request (placed in op->frames[0]).

Broadcast Manager Transmission Timers

Periodic transmission configurations may use up to two interval timers. In this case the BCM sends a number of messages ('count') at an interval 'ival1', then continuing to send at another given interval 'ival2'. When only one timer is needed 'count' is set to zero and only 'ival2' is used. When SET_TIMER and START_TIMER flag were set the timers are activated. The timer values can be altered at runtime when only SET_TIMER is set.

Broadcast Manager message sequence transmission

Up to 256 CAN frames can be transmitted in a sequence in the case of a cyclic TX task configuration. The number of CAN frames is provided in the 'nframes' element of the BCM message head. The defined number of CAN frames are added as array to the TX_SETUP BCM configuration message:

```
/* create a struct to set up a sequence of four CAN frames */
struct {
        struct bcm_msg_head msg_head;
        struct can_frame frame[4];
} mytxmsg;

(..)
mytxmsg.msg_head.nframes = 4;
(..)

write(s, &mytxmsg, sizeof(mytxmsg));
```

With every transmission the index in the array of CAN frames is increased and set to zero at index overflow.

Broadcast Manager Receive Filter Timers

The timer values ival1 or ival2 may be set to non-zero values at RX_SETUP. When the SET_TIMER flag is set the timers are enabled:

ival1: Send RX_TIMEOUT when a received message is not received again within the given time. When START_TIMER is set at RX_SETUP the timeout detection is activated directly - even without a former CAN frame reception.

ival2: Throttle the received message rate down to the value of ival2. This is useful to reduce messages for the application when the signal inside the CAN frame is stateless as state changes within the ival2 periode may get lost.

Broadcast Manager Multiplex Message Receive Filter

To filter for content changes in multiplex message sequences an array of more than one CAN frames can be passed in a RX_SETUP configuration message. The data bytes of the first CAN frame contain the mask of relevant bits that have to match in the subsequent CAN frames with the received CAN frame. If one of the subsequent CAN frames is matching the bits in that frame data mark the relevant content to be compared with the previous received content. Up to 257 CAN frames (multiplex filter bit mask CAN frame plus 256 CAN filters) can be added as array to the TX_SETUP BCM configuration message:

```
/* usually used to clear CAN frame data[] - beware of endian problems! */
#define U64_DATA(p) (*(unsigned long long*)(p)->data)

struct {
        struct bcm_msg_head msg_head;
        struct can_frame frame[5];
} msg;

msg.msg_head.opcode  = RX_SETUP;
msg.msg_head.can_id  = 0x42;
msg.msg_head.flags   = 0;
msg.msg_head.nframes = 5;
U64_DATA(&msg.frame[0]) = 0xFF00000000000000ULL; /* MUX mask */
U64_DATA(&msg.frame[1]) = 0x01000000000000FFULL; /* data mask (MUX 0x01) */
U64_DATA(&msg.frame[2]) = 0x0200FFFF000000FFULL; /* data mask (MUX 0x02) */
U64_DATA(&msg.frame[3]) = 0x330000FFFFFF0003ULL; /* data mask (MUX 0x33) */
U64_DATA(&msg.frame[4]) = 0x4F07FC0FF0000000ULL; /* data mask (MUX 0x4F) */

write(s, &msg, sizeof(msg));
```

Broadcast Manager CAN FD Support

The programming API of the CAN_BCM depends on struct can_frame which is given as array directly behind the bcm_msg_head structure. To follow this schema for the CAN FD frames a new flag 'CAN_FD_FRAME' in the bcm_msg_head flags indicates that the concatenated CAN frame structures behind the bcm_msg_head are defined as struct canfd_frame:

```
struct {
        struct bcm_msg_head msg_head;
        struct canfd_frame frame[5];
} msg;

msg.msg_head.opcode  = RX_SETUP;
msg.msg_head.can_id  = 0x42;
msg.msg_head.flags   = CAN_FD_FRAME;
msg.msg_head.nframes = 5;
(..)
```

When using CAN FD frames for multiplex filtering the MUX mask is still expected in the first 64 bit of the struct canfd_frame data section.

Connected Transport Protocols (SOCK_SEQPACKET)

(to be written)

Unconnected Transport Protocols (SOCK_DGRAM)

(to be written)

SocketCAN Core Module

The SocketCAN core module implements the protocol family PF_CAN. CAN protocol modules are loaded by the core module at runtime. The core module provides an interface for CAN protocol modules to subscribe needed CAN IDs (see *Receive Lists*).

can.ko Module Params

- **stats_timer**: To calculate the SocketCAN core statistics (e.g. current/maximum frames per second) this 1 second timer is invoked at can.ko module start time by default. This timer can be disabled by using stattimer=0 on the module commandline.
- **debug**: (removed since SocketCAN SVN r546)

procfs content

As described in *Receive Lists* the SocketCAN core uses several filter lists to deliver received CAN frames to CAN protocol modules. These receive lists, their filters and the count of filter matches can be checked in the appropriate receive list. All entries contain the device and a protocol module identifier:

```
foo@bar:~$ cat /proc/net/can/rcvlist_all

receive list 'rx_all':
  (vcan3: no entry)
  (vcan2: no entry)
  (vcan1: no entry)
  device   can_id   can_mask  function  userdata   matches  ident
   vcan0     000    00000000  f88e6370  f6c6f400         0  raw
  (any: no entry)
```

In this example an application requests any CAN traffic from vcan0:

```
rcvlist_all - list for unfiltered entries (no filter operations)
rcvlist_eff - list for single extended frame (EFF) entries
rcvlist_err - list for error message frames masks
rcvlist_fil - list for mask/value filters
rcvlist_inv - list for mask/value filters (inverse semantic)
rcvlist_sff - list for single standard frame (SFF) entries
```

Additional procfs files in /proc/net/can:

```
stats       - SocketCAN core statistics (rx/tx frames, match ratios, ...)
reset_stats - manual statistic reset
version     - prints the SocketCAN core version and the ABI version
```

Writing Own CAN Protocol Modules

To implement a new protocol in the protocol family PF_CAN a new protocol has to be defined in include/linux/can.h . The prototypes and definitions to use the SocketCAN core can be accessed by including include/linux/can/core.h . In addition to functions that register the CAN protocol and the CAN device notifier chain there are functions to subscribe CAN frames received by CAN interfaces and to send CAN frames:

```
can_rx_register   - subscribe CAN frames from a specific interface
can_rx_unregister - unsubscribe CAN frames from a specific interface
can_send          - transmit a CAN frame (optional with local loopback)
```

For details see the kerneldoc documentation in net/can/af_can.c or the source code of net/can/raw.c or net/can/bcm.c .

CAN Network Drivers

Writing a CAN network device driver is much easier than writing a CAN character device driver. Similar to other known network device drivers you mainly have to deal with:

- TX: Put the CAN frame from the socket buffer to the CAN controller.
- RX: Put the CAN frame from the CAN controller to the socket buffer.

See e.g. at Documentation/networking/netdevices.txt . The differences for writing CAN network device driver are described below:

General Settings

```
dev->type  = ARPHRD_CAN; /* the netdevice hardware type */
dev->flags = IFF_NOARP;  /* CAN has no arp */

dev->mtu = CAN_MTU; /* sizeof(struct can_frame) -> legacy CAN interface */

or alternative, when the controller supports CAN with flexible data rate:
dev->mtu = CANFD_MTU; /* sizeof(struct canfd_frame) -> CAN FD interface */
```

The struct can_frame or struct canfd_frame is the payload of each socket buffer (skbuff) in the protocol family PF_CAN.

Local Loopback of Sent Frames

As described in *Local Loopback of Sent Frames* the CAN network device driver should support a local loopback functionality similar to the local echo e.g. of tty devices. In this case the driver flag IFF_ECHO has to be set to prevent the PF_CAN core from locally echoing sent frames (aka loopback) as fallback solution:

```
dev->flags = (IFF_NOARP | IFF_ECHO);
```

CAN Controller Hardware Filters

To reduce the interrupt load on deep embedded systems some CAN controllers support the filtering of CAN IDs or ranges of CAN IDs. These hardware filter capabilities vary from controller to controller and have to be identified as not feasible in a multi-user networking approach. The use of the very controller specific hardware filters could make sense in a very dedicated use-case, as a filter on driver level would affect all users in the multi-user system. The high efficient filter sets inside the PF_CAN core allow to set different multiple filters for each socket separately. Therefore the use of hardware filters goes to the category 'handmade tuning on deep embedded systems'. The author is running a MPC603e @133MHz with four SJA1000 CAN controllers from 2002 under heavy bus load without any problems ...

The Virtual CAN Driver (vcan)

Similar to the network loopback devices, vcan offers a virtual local CAN interface. A full qualified address on CAN consists of

- a unique CAN Identifier (CAN ID)
- the CAN bus this CAN ID is transmitted on (e.g. can0)

so in common use cases more than one virtual CAN interface is needed.

The virtual CAN interfaces allow the transmission and reception of CAN frames without real CAN controller hardware. Virtual CAN network devices are usually named 'vcanX', like vcan0 vcan1 vcan2 ... When compiled as a module the virtual CAN driver module is called vcan.ko

Since Linux Kernel version 2.6.24 the vcan driver supports the Kernel netlink interface to create vcan network devices. The creation and removal of vcan network devices can be managed with the ip(8) tool:

```
- Create a virtual CAN network interface:
     $ ip link add type vcan

- Create a virtual CAN network interface with a specific name 'vcan42':
     $ ip link add dev vcan42 type vcan

- Remove a (virtual CAN) network interface 'vcan42':
     $ ip link del vcan42
```

The CAN Network Device Driver Interface

The CAN network device driver interface provides a generic interface to setup, configure and monitor CAN network devices. The user can then configure the CAN device, like setting the bit-timing parameters, via the netlink interface using the program "ip" from the "IPROUTE2" utility suite. The following chapter describes briefly how to use it. Furthermore, the interface uses a common data structure and exports a set of common functions, which all real CAN network device drivers should use. Please have a look to the SJA1000 or MSCAN driver to understand how to use them. The name of the module is can-dev.ko.

Netlink interface to set/get devices properties

The CAN device must be configured via netlink interface. The supported netlink message types are defined and briefly described in "include/linux/can/netlink.h". CAN link support for the program "ip" of the IPROUTE2 utility suite is available and it can be used as shown below:

Setting CAN device properties:

```
$ ip link set can0 type can help
Usage: ip link set DEVICE type can
    [ bitrate BITRATE [ sample-point SAMPLE-POINT] ] |
    [ tq TQ prop-seg PROP_SEG phase-seg1 PHASE-SEG1
      phase-seg2 PHASE-SEG2 [ sjw SJW ] ]

    [ dbitrate BITRATE [ dsample-point SAMPLE-POINT] ] |
    [ dtq TQ dprop-seg PROP_SEG dphase-seg1 PHASE-SEG1
      dphase-seg2 PHASE-SEG2 [ dsjw SJW ] ]

    [ loopback { on | off } ]
    [ listen-only { on | off } ]
```

```
    [ triple-sampling { on | off } ]
    [ one-shot { on | off } ]
    [ berr-reporting { on | off } ]
    [ fd { on | off } ]
    [ fd-non-iso { on | off } ]
    [ presume-ack { on | off } ]

    [ restart-ms TIME-MS ]
    [ restart ]

    Where: BITRATE       := { 1..1000000 }
           SAMPLE-POINT  := { 0.000..0.999 }
           TQ            := { NUMBER }
           PROP-SEG      := { 1..8 }
           PHASE-SEG1    := { 1..8 }
           PHASE-SEG2    := { 1..8 }
           SJW           := { 1..4 }
           RESTART-MS    := { 0 | NUMBER }
```

Display CAN device details and statistics:

```
$ ip -details -statistics link show can0
2: can0: <NOARP,UP,LOWER_UP,ECHO> mtu 16 qdisc pfifo_fast state UP qlen 10
  link/can
  can <TRIPLE-SAMPLING> state ERROR-ACTIVE restart-ms 100
  bitrate 125000 sample_point 0.875
  tq 125 prop-seg 6 phase-seg1 7 phase-seg2 2 sjw 1
  sja1000: tseg1 1..16 tseg2 1..8 sjw 1..4 brp 1..64 brp-inc 1
  clock 8000000
  re-started bus-errors arbit-lost error-warn error-pass bus-off
  41         17457      0          41         42         41
  RX: bytes  packets  errors  dropped overrun mcast
  140859     17608    17457   0       0       0
  TX: bytes  packets  errors  dropped carrier collsns
  861        112      0       41      0       0
```

More info to the above output:

“<TRIPLE-SAMPLING>” Shows the list of selected CAN controller modes: LOOPBACK, LISTEN-ONLY, or TRIPLE-SAMPLING.

“state ERROR-ACTIVE” The current state of the CAN controller: “ERROR-ACTIVE”, “ERROR-WARNING”, “ERROR-PASSIVE”, “BUS-OFF” or “STOPPED”

“restart-ms 100” Automatic restart delay time. If set to a non-zero value, a restart of the CAN controller will be triggered automatically in case of a bus-off condition after the specified delay time in milliseconds. By default it’s off.

“bitrate 125000 sample-point 0.875” Shows the real bit-rate in bits/sec and the sample-point in the range 0.000..0.999. If the calculation of bit-timing parameters is enabled in the kernel (CONFIG_CAN_CALC_BITTIMING=y), the bit-timing can be defined by setting the “bitrate” argument. Optionally the “sample-point” can be specified. By default it’s 0.000 assuming CIA-recommended sample-points.

“tq 125 prop-seg 6 phase-seg1 7 phase-seg2 2 sjw 1” Shows the time quanta in ns, propagation segment, phase buffer segment 1 and 2 and the synchronisation jump width in units of tq. They allow to define the CAN bit-timing in a hardware independent format as proposed by the Bosch CAN 2.0 spec (see chapter 8 of http://www.semiconductors.bosch.de/pdf/can2spec.pdf).

“sja1000: tseg1 1..16 tseg2 1..8 sjw 1..4 brp 1..64 brp-inc 1 clock 8000000” Shows the bit-timing constants of the CAN controller, here the “sja1000”. The minimum and maximum values of the time segment 1 and 2, the synchronisation jump width in units of tq, the bitrate pre-scaler and the CAN system clock frequency in Hz. These constants could be used for user-defined (non-standard) bit-timing calculation algorithms in user-space.

"re-started bus-errors arbit-lost error-warn error-pass bus-off" Shows the number of restarts, bus and arbitration lost errors, and the state changes to the error-warning, error-passive and bus-off state. RX overrun errors are listed in the "overrun" field of the standard network statistics.

Setting the CAN Bit-Timing

The CAN bit-timing parameters can always be defined in a hardware independent format as proposed in the Bosch CAN 2.0 specification specifying the arguments "tq", "prop_seg", "phase_seg1", "phase_seg2" and "sjw":

```
$ ip link set canX type can tq 125 prop-seg 6 \
                            phase-seg1 7 phase-seg2 2 sjw 1
```

If the kernel option CONFIG_CAN_CALC_BITTIMING is enabled, CIA recommended CAN bit-timing parameters will be calculated if the bit- rate is specified with the argument "bitrate":

```
$ ip link set canX type can bitrate 125000
```

Note that this works fine for the most common CAN controllers with standard bit-rates but may *fail* for exotic bit-rates or CAN system clock frequencies. Disabling CONFIG_CAN_CALC_BITTIMING saves some space and allows user-space tools to solely determine and set the bit-timing parameters. The CAN controller specific bit-timing constants can be used for that purpose. They are listed by the following command:

```
$ ip -details link show can0
...
  sja1000: clock 8000000 tseg1 1..16 tseg2 1..8 sjw 1..4 brp 1..64 brp-inc 1
```

Starting and Stopping the CAN Network Device

A CAN network device is started or stopped as usual with the command "ifconfig canX up/down" or "ip link set canX up/down". Be aware that you *must* define proper bit-timing parameters for real CAN devices before you can start it to avoid error-prone default settings:

```
$ ip link set canX up type can bitrate 125000
```

A device may enter the "bus-off" state if too many errors occurred on the CAN bus. Then no more messages are received or sent. An automatic bus-off recovery can be enabled by setting the "restart-ms" to a non-zero value, e.g.:

```
$ ip link set canX type can restart-ms 100
```

Alternatively, the application may realize the "bus-off" condition by monitoring CAN error message frames and do a restart when appropriate with the command:

```
$ ip link set canX type can restart
```

Note that a restart will also create a CAN error message frame (see also *Network Problem Notifications*).

CAN FD (Flexible Data Rate) Driver Support

CAN FD capable CAN controllers support two different bitrates for the arbitration phase and the payload phase of the CAN FD frame. Therefore a second bit timing has to be specified in order to enable the CAN FD bitrate.

Additionally CAN FD capable CAN controllers support up to 64 bytes of payload. The representation of this length in can_frame.can_dlc and canfd_frame.len for userspace applications and inside the Linux network layer is a plain value from 0 .. 64 instead of the CAN 'data length code'. The data length code was a 1:1 mapping to the payload length in the legacy CAN frames anyway. The payload length to the bus-relevant

DLC mapping is only performed inside the CAN drivers, preferably with the helper functions can_dlc2len() and can_len2dlc().

The CAN netdevice driver capabilities can be distinguished by the network devices maximum transfer unit (MTU):

```
MTU = 16 (CAN_MTU)   => sizeof(struct can_frame)   => 'legacy' CAN device
MTU = 72 (CANFD_MTU) => sizeof(struct canfd_frame) => CAN FD capable device
```

The CAN device MTU can be retrieved e.g. with a SIOCGIFMTU ioctl() syscall. N.B. CAN FD capable devices can also handle and send legacy CAN frames.

When configuring CAN FD capable CAN controllers an additional 'data' bitrate has to be set. This bitrate for the data phase of the CAN FD frame has to be at least the bitrate which was configured for the arbitration phase. This second bitrate is specified analogue to the first bitrate but the bitrate setting keywords for the 'data' bitrate start with 'd' e.g. dbitrate, dsample-point, dsjw or dtq and similar settings. When a data bitrate is set within the configuration process the controller option "fd on" can be specified to enable the CAN FD mode in the CAN controller. This controller option also switches the device MTU to 72 (CANFD_MTU).

The first CAN FD specification presented as whitepaper at the International CAN Conference 2012 needed to be improved for data integrity reasons. Therefore two CAN FD implementations have to be distinguished today:

- ISO compliant: The ISO 11898-1:2015 CAN FD implementation (default)
- non-ISO compliant: The CAN FD implementation following the 2012 whitepaper

Finally there are three types of CAN FD controllers:

1. ISO compliant (fixed)
2. non-ISO compliant (fixed, like the M_CAN IP core v3.0.1 in m_can.c)
3. ISO/non-ISO CAN FD controllers (switchable, like the PEAK PCAN-USB FD)

The current ISO/non-ISO mode is announced by the CAN controller driver via netlink and displayed by the 'ip' tool (controller option FD-NON-ISO). The ISO/non-ISO-mode can be altered by setting 'fd-non-iso {on|off}' for switchable CAN FD controllers only.

Example configuring 500 kbit/s arbitration bitrate and 4 Mbit/s data bitrate:

```
$ ip link set can0 up type can bitrate 500000 sample-point 0.75 \
                               dbitrate 4000000 dsample-point 0.8 fd on
$ ip -details link show can0
5: can0: <NOARP,UP,LOWER_UP,ECHO> mtu 72 qdisc pfifo_fast state UNKNOWN \
         mode DEFAULT group default qlen 10
link/can  promiscuity 0
can <FD> state ERROR-ACTIVE (berr-counter tx 0 rx 0) restart-ms 0
      bitrate 500000 sample-point 0.750
      tq 50 prop-seg 14 phase-seg1 15 phase-seg2 10 sjw 1
      pcan_usb_pro_fd: tseg1 1..64 tseg2 1..16 sjw 1..16 brp 1..1024 \
      brp-inc 1
      dbitrate 4000000 dsample-point 0.800
      dtq 12 dprop-seg 7 dphase-seg1 8 dphase-seg2 4 dsjw 1
      pcan_usb_pro_fd: dtseg1 1..16 dtseg2 1..8 dsjw 1..4 dbrp 1..1024 \
      dbrp-inc 1
      clock 80000000
```

Example when 'fd-non-iso on' is added on this switchable CAN FD adapter:

```
can <FD,FD-NON-ISO> state ERROR-ACTIVE (berr-counter tx 0 rx 0) restart-ms 0
```

Supported CAN Hardware

Please check the "Kconfig" file in "drivers/net/can" to get an actual list of the support CAN hardware. On the SocketCAN project website (see *SocketCAN Resources*) there might be further drivers available, also for older kernel versions.

SocketCAN Resources

The Linux CAN / SocketCAN project resources (project site / mailing list) are referenced in the MAINTAINERS file in the Linux source tree. Search for CAN NETWORK [LAYERS|DRIVERS].

Credits

- Oliver Hartkopp (PF_CAN core, filters, drivers, bcm, SJA1000 driver)
- Urs Thuermann (PF_CAN core, kernel integration, socket interfaces, raw, vcan)
- Jan Kizka (RT-SocketCAN core, Socket-API reconciliation)
- Wolfgang Grandegger (RT-SocketCAN core & drivers, Raw Socket-API reviews, CAN device driver interface, MSCAN driver)
- Robert Schwebel (design reviews, PTXdist integration)
- Marc Kleine-Budde (design reviews, Kernel 2.6 cleanups, drivers)
- Benedikt Spranger (reviews)
- Thomas Gleixner (LKML reviews, coding style, posting hints)
- Andrey Volkov (kernel subtree structure, ioctls, MSCAN driver)
- Matthias Brukner (first SJA1000 CAN netdevice implementation Q2/2003)
- Klaus Hitschler (PEAK driver integration)
- Uwe Koppe (CAN netdevices with PF_PACKET approach)
- Michael Schulze (driver layer loopback requirement, RT CAN drivers review)
- Pavel Pisa (Bit-timing calculation)
- Sascha Hauer (SJA1000 platform driver)
- Sebastian Haas (SJA1000 EMS PCI driver)
- Markus Plessing (SJA1000 EMS PCI driver)
- Per Dalen (SJA1000 Kvaser PCI driver)
- Sam Ravnborg (reviews, coding style, kbuild help)

CHAPTER

THREE

LINUX NETWORKING AND NETWORK DEVICES APIS

Linux Networking

Networking Base Types

enum **sock_type**
Socket types

Constants

SOCK_STREAM stream (connection) socket

SOCK_DGRAM datagram (conn.less) socket

SOCK_RAW raw socket

SOCK_RDM reliably-delivered message

SOCK_SEQPACKET sequential packet socket

SOCK_DCCP Datagram Congestion Control Protocol socket

SOCK_PACKET linux specific way of getting packets at the dev level. For writing rarp and other similar things on the user level.

Description

When adding some new socket type please grep ARCH_HAS_SOCKET_TYPE include/asm-* /socket.h, at least MIPS overrides this enum for binary compat reasons.

struct **socket**
general BSD socket

Definition

```
struct socket {
  socket_state state;
  short type;
  unsigned long           flags;
  struct socket_wq __rcu  *wq;
  struct file             *file;
  struct sock             *sk;
  const struct proto_ops  *ops;
};
```

Members

state socket state (SS_CONNECTED, etc)

type socket type (SOCK_STREAM, etc)

flags socket flags (SOCK_NOSPACE, etc)

wq wait queue for several uses

file File back pointer for gc

sk internal networking protocol agnostic socket representation

ops protocol specific socket operations

Socket Buffer Functions

skb_frag_foreach_page(*f*, *f_off*, *f_len*, *p*, *p_off*, *p_len*, *copied*)
loop over pages in a fragment

Parameters

f skb frag to operate on

f_off offset from start of f->page.p

f_len length from f_off to loop over

p (temp var) current page

p_off (temp var) offset from start of current page, non-zero only on first page.

p_len (temp var) length in current page, < PAGE_SIZE only on first and last page.

copied (temp var) length so far, excluding current p_len.

Description

A fragment can hold a compound page, in which case per-page operations, notably kmap_atomic, must be called for each regular page.

struct **skb_shared_hwtstamps**
hardware time stamps

Definition

```
struct skb_shared_hwtstamps {
  ktime_t hwtstamp;
};
```

Members

hwtstamp hardware time stamp transformed into duration since arbitrary point in time

Description

Software time stamps generated by `ktime_get_real()` are stored in skb->tstamp.

hwtstamps can only be compared against other hwtstamps from the same device.

This structure is attached to packets as part of the `skb_shared_info`. Use `skb_hwtstamps()` to get a pointer.

struct **sk_buff**
socket buffer

Definition

```
struct sk_buff {
  union {
    struct {
      struct sk_buff          *next;
      struct sk_buff          *prev;
      union {
        struct net_device       *dev;
        unsigned long           dev_scratch;
      };
    };
```

```
    struct rb_node  rbnode;
  };
  struct sock             *sk;
  union {
    ktime_t tstamp;
    u64 skb_mstamp;
  };
  char cb[48] ;
  union {
    struct {
      unsigned long   _skb_refdst;
      void (*destructor)(struct sk_buff *skb);
    };
    struct list_head        tcp_tsorted_anchor;
  };
#ifdef CONFIG_XFRM;
  struct sec_path        *sp;
#endif;
#if defined(CONFIG_NF_CONNTRACK) || defined(CONFIG_NF_CONNTRACK_MODULE);
  unsigned long            _nfct;
#endif;
#if IS_ENABLED(CONFIG_BRIDGE_NETFILTER);
  struct nf_bridge_info   *nf_bridge;
#endif;
  unsigned int            len, data_len;
  __u16 mac_len, hdr_len;
  __u16 queue_mapping;
#ifdef __BIG_ENDIAN_BITFIELD;
#define CLONED_MASK     (1 << 7);
#else;
#define CLONED_MASK     1;
#endif;
#define CLONED_OFFSET()         offsetof(struct sk_buff, __cloned_offset);
  __u8 __cloned_offset[0];
  __u8 cloned:1,nohdr:1,fclone:2,peeked:1,head_frag:1,xmit_more:1, __unused:1;
#ifdef __BIG_ENDIAN_BITFIELD;
#define PKT_TYPE_MAX    (7 << 5);
#else;
#define PKT_TYPE_MAX    7;
#endif;
#define PKT_TYPE_OFFSET()       offsetof(struct sk_buff, __pkt_type_offset);
  __u8 __pkt_type_offset[0];
  __u8 pkt_type:3;
  __u8 pfmemalloc:1;
  __u8 ignore_df:1;
  __u8 nf_trace:1;
  __u8 ip_summed:2;
  __u8 ooo_okay:1;
  __u8 l4_hash:1;
  __u8 sw_hash:1;
  __u8 wifi_acked_valid:1;
  __u8 wifi_acked:1;
  __u8 no_fcs:1;
  __u8 encapsulation:1;
  __u8 encap_hdr_csum:1;
  __u8 csum_valid:1;
  __u8 csum_complete_sw:1;
  __u8 csum_level:2;
  __u8 csum_not_inet:1;
  __u8 dst_pending_confirm:1;
#ifdef CONFIG_IPV6_NDISC_NODETYPE;
  __u8 ndisc_nodetype:2;
#endif;
```

```
  __u8 ipvs_property:1;
  __u8 inner_protocol_type:1;
  __u8 remcsum_offload:1;
#ifdef CONFIG_NET_SWITCHDEV;
  __u8 offload_fwd_mark:1;
  __u8 offload_mr_fwd_mark:1;
#endif;
#ifdef CONFIG_NET_CLS_ACT;
  __u8 tc_skip_classify:1;
  __u8 tc_at_ingress:1;
  __u8 tc_redirected:1;
  __u8 tc_from_ingress:1;
#endif;
#ifdef CONFIG_NET_SCHED;
  __u16 tc_index;
#endif;
  union {
    __wsum csum;
    struct {
      __u16 csum_start;
      __u16 csum_offset;
    };
  };
  __u32 priority;
  int skb_iif;
  __u32 hash;
  __be16 vlan_proto;
  __u16 vlan_tci;
#if defined(CONFIG_NET_RX_BUSY_POLL) || defined(CONFIG_XPS);
  union {
    unsigned int    napi_id;
    unsigned int    sender_cpu;
  };
#endif;
#ifdef CONFIG_NETWORK_SECMARK;
  __u32 secmark;
#endif;
  union {
    __u32 mark;
    __u32 reserved_tailroom;
  };
  union {
    __be16 inner_protocol;
    __u8 inner_ipproto;
  };
  __u16 inner_transport_header;
  __u16 inner_network_header;
  __u16 inner_mac_header;
  __be16 protocol;
  __u16 transport_header;
  __u16 network_header;
  __u16 mac_header;
  sk_buff_data_t tail;
  sk_buff_data_t end;
  unsigned char          *head, *data;
  unsigned int           truesize;
  refcount_t users;
};
```

Members

{unnamed_union} anonymous

{unnamed_struct} anonymous

next Next buffer in list

prev Previous buffer in list

{unnamed_union} anonymous

dev Device we arrived on/are leaving by

rbnode RB tree node, alternative to next/prev for netem/tcp

sk Socket we are owned by

{unnamed_union} anonymous

tstamp Time we arrived/left

cb Control buffer. Free for use by every layer. Put private vars here

{unnamed_union} anonymous

{unnamed_struct} anonymous

_skb_refdst destination entry (with norefcount bit)

destructor Destruct function

tcp_tsorted_anchor list structure for TCP (tp->tsorted_sent_queue)

sp the security path, used for xfrm

_nfct Associated connection, if any (with nfctinfo bits)

nf_bridge Saved data about a bridged frame - see br_netfilter.c

len Length of actual data

data_len Data length

mac_len Length of link layer header

hdr_len writable header length of cloned skb

queue_mapping Queue mapping for multiqueue devices

cloned Head may be cloned (check refcnt to be sure)

nohdr Payload reference only, must not modify header

fclone skbuff clone status

peeked this packet has been seen already, so stats have been done for it, don't do them again

xmit_more More SKBs are pending for this queue

pkt_type Packet class

ignore_df allow local fragmentation

nf_trace netfilter packet trace flag

ip_summed Driver fed us an IP checksum

ooo_okay allow the mapping of a socket to a queue to be changed

l4_hash indicate hash is a canonical 4-tuple hash over transport ports.

sw_hash indicates hash was computed in software stack

wifi_acked_valid wifi_acked was set

wifi_acked whether frame was acked on wifi or not

no_fcs Request NIC to treat last 4 bytes as Ethernet FCS

csum_not_inet use CRC32c to resolve CHECKSUM_PARTIAL

dst_pending_confirm need to confirm neighbour

ndisc_nodetype router type (from link layer)

ipvs_property skbuff is owned by ipvs

tc_skip_classify do not classify packet. set by IFB device

tc_at_ingress used within tc_classify to distinguish in/egress

tc_redirected packet was redirected by a tc action

tc_from_ingress if tc_redirected, tc_at_ingress at time of redirect

tc_index Traffic control index

{unnamed_union} anonymous

csum Checksum (must include start/offset pair)

{unnamed_struct} anonymous

csum_start Offset from skb->head where checksumming should start

csum_offset Offset from csum_start where checksum should be stored

priority Packet queueing priority

skb_iif ifindex of device we arrived on

hash the packet hash

vlan_proto vlan encapsulation protocol

vlan_tci vlan tag control information

{unnamed_union} anonymous

napi_id id of the NAPI struct this skb came from

secmark security marking

{unnamed_union} anonymous

mark Generic packet mark

{unnamed_union} anonymous

inner_protocol Protocol (encapsulation)

inner_transport_header Inner transport layer header (encapsulation)

inner_network_header Network layer header (encapsulation)

inner_mac_header Link layer header (encapsulation)

protocol Packet protocol from driver

transport_header Transport layer header

network_header Network layer header

mac_header Link layer header

tail Tail pointer

end End pointer

head Head of buffer

data Data head pointer

truesize Buffer size

users User count - see {datagram,tcp}.c

struct dst_entry * **skb_dst**(const struct *sk_buff* * *skb*)
returns skb dst_entry

Parameters

const struct sk_buff * skb buffer

Description

Returns skb dst_entry, regardless of reference taken or not.

void **skb_dst_set**(struct *sk_buff* * *skb*, struct dst_entry * *dst*)
sets skb dst

Parameters

struct sk_buff * skb buffer

struct dst_entry * dst dst entry

Description

Sets skb dst, assuming a reference was taken on dst and should be released by `skb_dst_drop()`

void **skb_dst_set_noref**(struct *sk_buff* * *skb*, struct dst_entry * *dst*)
sets skb dst, hopefully, without taking reference

Parameters

struct sk_buff * skb buffer

struct dst_entry * dst dst entry

Description

Sets skb dst, assuming a reference was not taken on dst. If dst entry is cached, we do not take reference and dst_release will be avoided by refdst_drop. If dst entry is not cached, we take reference, so that last dst_release can destroy the dst immediately.

bool **skb_dst_is_noref**(const struct *sk_buff* * *skb*)
Test if skb dst isn't refcounted

Parameters

const struct sk_buff * skb buffer

bool **skb_fclone_busy**(const struct *sock* * *sk*, const struct *sk_buff* * *skb*)
check if fclone is busy

Parameters

const struct sock * sk socket

const struct sk_buff * skb buffer

Description

Returns true if skb is a fast clone, and its clone is not freed. Some drivers call *skb_orphan()* in their `ndo_start_xmit()`, so we also check that this didnt happen.

int **skb_pad**(struct *sk_buff* * *skb*, int *pad*)
zero pad the tail of an skb

Parameters

struct sk_buff * skb buffer to pad

int pad space to pad

Description

Ensure that a buffer is followed by a padding area that is zero filled. Used by network drivers which may DMA or transfer data beyond the buffer end onto the wire.

May return error in out of memory cases. The skb is freed on error.

int **skb_queue_empty**(const struct sk_buff_head * *list*)
check if a queue is empty

Parameters

const struct sk_buff_head * list queue head

Description

Returns true if the queue is empty, false otherwise.

bool **skb_queue_is_last**(const struct sk_buff_head * *list*, const struct *sk_buff* * *skb*)
check if skb is the last entry in the queue

Parameters

const struct sk_buff_head * list queue head

const struct sk_buff * skb buffer

Description

Returns true if **skb** is the last buffer on the list.

bool **skb_queue_is_first**(const struct sk_buff_head * *list*, const struct *sk_buff* * *skb*)
check if skb is the first entry in the queue

Parameters

const struct sk_buff_head * list queue head

const struct sk_buff * skb buffer

Description

Returns true if **skb** is the first buffer on the list.

struct *sk_buff* * **skb_queue_next**(const struct sk_buff_head * *list*, const struct *sk_buff* * *skb*)
return the next packet in the queue

Parameters

const struct sk_buff_head * list queue head

const struct sk_buff * skb current buffer

Description

Return the next packet in **list** after **skb**. It is only valid to call this if *skb_queue_is_last()* evaluates to false.

struct *sk_buff* * **skb_queue_prev**(const struct sk_buff_head * *list*, const struct *sk_buff* * *skb*)
return the prev packet in the queue

Parameters

const struct sk_buff_head * list queue head

const struct sk_buff * skb current buffer

Description

Return the prev packet in **list** before **skb**. It is only valid to call this if *skb_queue_is_first()* evaluates to false.

struct *sk_buff* * **skb_get**(struct *sk_buff* * *skb*)
reference buffer

Parameters

struct sk_buff * skb buffer to reference

Description

Makes another reference to a socket buffer and returns a pointer to the buffer.

int **skb_cloned**(const struct *sk_buff* * *skb*)
is the buffer a clone

Parameters

const struct sk_buff * skb buffer to check

Description

Returns true if the buffer was generated with *skb_clone()* and is one of multiple shared copies of the buffer. Cloned buffers are shared data so must not be written to under normal circumstances.

int **skb_header_cloned**(const struct *sk_buff* * *skb*)
is the header a clone

Parameters

const struct sk_buff * skb buffer to check

Description

Returns true if modifying the header part of the buffer requires the data to be copied.

void **__skb_header_release**(struct *sk_buff* * *skb*)
release reference to header

Parameters

struct sk_buff * skb buffer to operate on

int **skb_shared**(const struct *sk_buff* * *skb*)
is the buffer shared

Parameters

const struct sk_buff * skb buffer to check

Description

Returns true if more than one person has a reference to this buffer.

struct *sk_buff* * **skb_share_check**(struct *sk_buff* * *skb*, gfp_t *pri*)
check if buffer is shared and if so clone it

Parameters

struct sk_buff * skb buffer to check

gfp_t pri priority for memory allocation

Description

If the buffer is shared the buffer is cloned and the old copy drops a reference. A new clone with a single reference is returned. If the buffer is not shared the original buffer is returned. When being called from interrupt status or with spinlocks held pri must be GFP_ATOMIC.

NULL is returned on a memory allocation failure.

struct *sk_buff* * **skb_unshare**(struct *sk_buff* * *skb*, gfp_t *pri*)
make a copy of a shared buffer

Parameters

struct sk_buff * skb buffer to check

gfp_t pri priority for memory allocation

Description

If the socket buffer is a clone then this function creates a new copy of the data, drops a reference count on the old copy and returns the new copy with the reference count at 1. If the buffer is not a clone the original buffer is returned. When called with a spinlock held or from interrupt state **pri** must be GFP_ATOMIC

NULL is returned on a memory allocation failure.

struct *sk_buff* * **skb_peek**(const struct sk_buff_head * *list_*)
peek at the head of an sk_buff_head

Parameters

const struct sk_buff_head * list_ list to peek at

Description

Peek an *sk_buff*. Unlike most other operations you _MUST_ be careful with this one. A peek leaves the buffer on the list and someone else may run off with it. You must hold the appropriate locks or have a private queue to do this.

Returns NULL for an empty list or a pointer to the head element. The reference count is not incremented and the reference is therefore volatile. Use with caution.

struct *sk_buff* * **skb_peek_next**(struct *sk_buff* * *skb*, const struct sk_buff_head * *list_*)
peek skb following the given one from a queue

Parameters

struct sk_buff * skb skb to start from

const struct sk_buff_head * list_ list to peek at

Description

Returns NULL when the end of the list is met or a pointer to the next element. The reference count is not incremented and the reference is therefore volatile. Use with caution.

struct *sk_buff* * **skb_peek_tail**(const struct sk_buff_head * *list_*)
peek at the tail of an sk_buff_head

Parameters

const struct sk_buff_head * list_ list to peek at

Description

Peek an *sk_buff*. Unlike most other operations you _MUST_ be careful with this one. A peek leaves the buffer on the list and someone else may run off with it. You must hold the appropriate locks or have a private queue to do this.

Returns NULL for an empty list or a pointer to the tail element. The reference count is not incremented and the reference is therefore volatile. Use with caution.

__u32 **skb_queue_len**(const struct sk_buff_head * *list_*)
get queue length

Parameters

const struct sk_buff_head * list_ list to measure

Description

Return the length of an *sk_buff* queue.

void **__skb_queue_head_init**(struct sk_buff_head * *list*)
initialize non-spinlock portions of sk_buff_head

Parameters

struct sk_buff_head * list queue to initialize

Description

This initializes only the list and queue length aspects of an sk_buff_head object. This allows to initialize the list aspects of an sk_buff_head without reinitializing things like the spinlock. It can also be used for on-stack sk_buff_head objects where the spinlock is known to not be used.

void **skb_queue_splice**(const struct sk_buff_head * *list*, struct sk_buff_head * *head*)
join two skb lists, this is designed for stacks

Parameters

const struct sk_buff_head * list the new list to add

struct sk_buff_head * head the place to add it in the first list

void **skb_queue_splice_init**(struct sk_buff_head * *list*, struct sk_buff_head * *head*)
join two skb lists and reinitialise the emptied list

Parameters

struct sk_buff_head * list the new list to add

struct sk_buff_head * head the place to add it in the first list

Description

The list at **list** is reinitialised

void **skb_queue_splice_tail**(const struct sk_buff_head * *list*, struct sk_buff_head * *head*)
join two skb lists, each list being a queue

Parameters

const struct sk_buff_head * list the new list to add

struct sk_buff_head * head the place to add it in the first list

void **skb_queue_splice_tail_init**(struct sk_buff_head * *list*, struct sk_buff_head * *head*)
join two skb lists and reinitialise the emptied list

Parameters

struct sk_buff_head * list the new list to add

struct sk_buff_head * head the place to add it in the first list

Description

Each of the lists is a queue. The list at **list** is reinitialised

void **__skb_queue_after**(struct sk_buff_head * *list*, struct *sk_buff* * *prev*, struct *sk_buff* * *newsk*)
queue a buffer at the list head

Parameters

struct sk_buff_head * list list to use

struct sk_buff * prev place after this buffer

struct sk_buff * newsk buffer to queue

Description

Queue a buffer int the middle of a list. This function takes no locks and you must therefore hold required locks before calling it.

A buffer cannot be placed on two lists at the same time.

void **skb_queue_head**(struct sk_buff_head * *list*, struct *sk_buff* * *newsk*)
queue a buffer at the list head

Parameters

struct sk_buff_head * list list to use

struct sk_buff * newsk buffer to queue

Description

Queue a buffer at the start of a list. This function takes no locks and you must therefore hold required locks before calling it.

A buffer cannot be placed on two lists at the same time.

void **skb_queue_tail**(struct sk_buff_head * *list*, struct *sk_buff* * *newsk*)
queue a buffer at the list tail

Parameters

struct sk_buff_head * list list to use

struct sk_buff * newsk buffer to queue

Description

Queue a buffer at the end of a list. This function takes no locks and you must therefore hold required locks before calling it.

A buffer cannot be placed on two lists at the same time.

struct *sk_buff* * **skb_dequeue**(struct sk_buff_head * *list*)
remove from the head of the queue

Parameters

struct sk_buff_head * list list to dequeue from

Description

Remove the head of the list. This function does not take any locks so must be used with appropriate locks held only. The head item is returned or `NULL` if the list is empty.

struct *sk_buff* * **skb_dequeue_tail**(struct sk_buff_head * *list*)
remove from the tail of the queue

Parameters

struct sk_buff_head * list list to dequeue from

Description

Remove the tail of the list. This function does not take any locks so must be used with appropriate locks held only. The tail item is returned or `NULL` if the list is empty.

void **__skb_fill_page_desc**(struct *sk_buff* * *skb*, int *i*, struct page * *page*, int *off*, int *size*)
initialise a paged fragment in an skb

Parameters

struct sk_buff * skb buffer containing fragment to be initialised

int i paged fragment index to initialise

struct page * page the page to use for this fragment

int off the offset to the data with **page**

int size the length of the data

Description

Initialises the **i**'th fragment of **skb** to point to `size` bytes at offset **off** within **page**.

Does not take any additional reference on the fragment.

void **skb_fill_page_desc**(struct *sk_buff* * *skb*, int *i*, struct page * *page*, int *off*, int *size*)
initialise a paged fragment in an skb

Parameters

struct sk_buff * skb buffer containing fragment to be initialised

int i paged fragment index to initialise

struct page * page the page to use for this fragment

int off the offset to the data with **page**

int size the length of the data

Description

As per *__skb_fill_page_desc()* – initialises the **i**'th fragment of **skb** to point to **size** bytes at offset **off** within **page**. In addition updates **skb** such that **i** is the last fragment.

Does not take any additional reference on the fragment.

unsigned int **skb_headroom**(const struct *sk_buff * skb*)
bytes at buffer head

Parameters

const struct sk_buff * skb buffer to check

Description

Return the number of bytes of free space at the head of an *sk_buff*.

int **skb_tailroom**(const struct *sk_buff * skb*)
bytes at buffer end

Parameters

const struct sk_buff * skb buffer to check

Description

Return the number of bytes of free space at the tail of an sk_buff

int **skb_availroom**(const struct *sk_buff * skb*)
bytes at buffer end

Parameters

const struct sk_buff * skb buffer to check

Description

Return the number of bytes of free space at the tail of an sk_buff allocated by `sk_stream_alloc()`

void **skb_reserve**(struct *sk_buff * skb*, int *len*)
adjust headroom

Parameters

struct sk_buff * skb buffer to alter

int len bytes to move

Description

Increase the headroom of an empty *sk_buff* by reducing the tail room. This is only allowed for an empty buffer.

void **skb_tailroom_reserve**(struct *sk_buff * skb*, unsigned int *mtu*, unsigned int *needed_tailroom*)
adjust reserved_tailroom

Parameters

struct sk_buff * skb buffer to alter

unsigned int mtu maximum amount of headlen permitted

unsigned int needed_tailroom minimum amount of reserved_tailroom

Description

Set reserved_tailroom so that headlen can be as large as possible but not larger than mtu and tailroom cannot be smaller than needed_tailroom. The required headroom should already have been reserved before using this function.

void **pskb_trim_unique**(struct *sk_buff* * *skb*, unsigned int *len*)
remove end from a paged unique (not cloned) buffer

Parameters

struct sk_buff * skb buffer to alter

unsigned int len new length

Description

This is identical to pskb_trim except that the caller knows that the skb is not cloned so we should never get an error due to out- of-memory.

void **skb_orphan**(struct *sk_buff* * *skb*)
orphan a buffer

Parameters

struct sk_buff * skb buffer to orphan

Description

If a buffer currently has an owner then we call the owner's destructor function and make the **skb** unowned. The buffer continues to exist but is no longer charged to its former owner.

int **skb_orphan_frags**(struct *sk_buff* * *skb*, gfp_t *gfp_mask*)
orphan the frags contained in a buffer

Parameters

struct sk_buff * skb buffer to orphan frags from

gfp_t gfp_mask allocation mask for replacement pages

Description

For each frag in the SKB which needs a destructor (i.e. has an owner) create a copy of that frag and release the original page by calling the destructor.

void **skb_queue_purge**(struct sk_buff_head * *list*)
empty a list

Parameters

struct sk_buff_head * list list to empty

Description

Delete all buffers on an *sk_buff* list. Each buffer is removed from the list and one reference dropped. This function does not take the list lock and the caller must hold the relevant locks to use it.

struct *sk_buff* * **netdev_alloc_skb**(struct *net_device* * *dev*, unsigned int *length*)
allocate an skbuff for rx on a specific device

Parameters

struct net_device * dev network device to receive on

unsigned int length length to allocate

Description

Allocate a new *sk_buff* and assign it a usage count of one. The buffer has unspecified headroom built in. Users should allocate the headroom they think they need without accounting for the built in space. The built in space is used for optimisations.

NULL is returned if there is no free memory. Although this function allocates memory it can be called from an interrupt.

struct page * **__dev_alloc_pages**(gfp_t *gfp_mask*, unsigned int *order*)
allocate page for network Rx

Parameters

gfp_t gfp_mask allocation priority. Set __GFP_NOMEMALLOC if not for network Rx

unsigned int order size of the allocation

Description

Allocate a new page.

NULL is returned if there is no free memory.

struct page * **__dev_alloc_page**(gfp_t *gfp_mask*)
allocate a page for network Rx

Parameters

gfp_t gfp_mask allocation priority. Set __GFP_NOMEMALLOC if not for network Rx

Description

Allocate a new page.

NULL is returned if there is no free memory.

void **skb_propagate_pfmemalloc**(struct page * *page*, struct *sk_buff* * *skb*)
Propagate pfmemalloc if skb is allocated after RX page

Parameters

struct page * page The page that was allocated from skb_alloc_page

struct sk_buff * skb The skb that may need pfmemalloc set

struct page * **skb_frag_page**(const skb_frag_t * *frag*)
retrieve the page referred to by a paged fragment

Parameters

const skb_frag_t * frag the paged fragment

Description

Returns the struct page associated with **frag**.

void **__skb_frag_ref**(skb_frag_t * *frag*)
take an addition reference on a paged fragment.

Parameters

skb_frag_t * frag the paged fragment

Description

Takes an additional reference on the paged fragment **frag**.

void **skb_frag_ref**(struct *sk_buff* * *skb*, int *f*)
take an addition reference on a paged fragment of an skb.

Parameters

struct sk_buff * skb the buffer

int f the fragment offset.

Description

Takes an additional reference on the **f**'th paged fragment of **skb**.

void **__skb_frag_unref**(skb_frag_t * *frag*)
release a reference on a paged fragment.

Parameters

skb_frag_t * frag the paged fragment

Description

Releases a reference on the paged fragment **frag**.

void **skb_frag_unref**(struct *sk_buff* * *skb*, int *f*)
release a reference on a paged fragment of an skb.

Parameters

struct sk_buff * skb the buffer

int f the fragment offset

Description

Releases a reference on the **f**'th paged fragment of **skb**.

void * **skb_frag_address**(const skb_frag_t * *frag*)
gets the address of the data contained in a paged fragment

Parameters

const skb_frag_t * frag the paged fragment buffer

Description

Returns the address of the data within **frag**. The page must already be mapped.

void * **skb_frag_address_safe**(const skb_frag_t * *frag*)
gets the address of the data contained in a paged fragment

Parameters

const skb_frag_t * frag the paged fragment buffer

Description

Returns the address of the data within **frag**. Checks that the page is mapped and returns `NULL` otherwise.

void **__skb_frag_set_page**(skb_frag_t * *frag*, struct page * *page*)
sets the page contained in a paged fragment

Parameters

skb_frag_t * frag the paged fragment

struct page * page the page to set

Description

Sets the fragment **frag** to contain **page**.

void **skb_frag_set_page**(struct *sk_buff* * *skb*, int *f*, struct page * *page*)
sets the page contained in a paged fragment of an skb

Parameters

struct sk_buff * skb the buffer

int f the fragment offset

struct page * page the page to set

Description

Sets the **f**'th fragment of **skb** to contain **page**.

dma_addr_t **skb_frag_dma_map**(struct device * *dev*, const skb_frag_t * *frag*, size_t *offset*, size_t *size*, enum dma_data_direction *dir*)
maps a paged fragment via the DMA API

Parameters

struct device * dev the device to map the fragment to

const skb_frag_t * frag the paged fragment to map

size_t offset the offset within the fragment (starting at the fragment's own offset)

size_t size the number of bytes to map

enum dma_data_direction dir the direction of the mapping (PCI_DMA_*)

Description

Maps the page associated with **frag** to **device**.

int **skb_clone_writable**(const struct *sk_buff* * *skb*, unsigned int *len*)
is the header of a clone writable

Parameters

const struct sk_buff * skb buffer to check

unsigned int len length up to which to write

Description

Returns true if modifying the header part of the cloned buffer does not requires the data to be copied.

int **skb_cow**(struct *sk_buff* * *skb*, unsigned int *headroom*)
copy header of skb when it is required

Parameters

struct sk_buff * skb buffer to cow

unsigned int headroom needed headroom

Description

If the skb passed lacks sufficient headroom or its data part is shared, data is reallocated. If reallocation fails, an error is returned and original skb is not changed.

The result is skb with writable area skb->head...skb->tail and at least **headroom** of space at head.

int **skb_cow_head**(struct *sk_buff* * *skb*, unsigned int *headroom*)
skb_cow but only making the head writable

Parameters

struct sk_buff * skb buffer to cow

unsigned int headroom needed headroom

Description

This function is identical to skb_cow except that we replace the skb_cloned check by skb_header_cloned. It should be used when you only need to push on some header and do not need to modify the data.

int **skb_padto**(struct *sk_buff* * *skb*, unsigned int *len*)
pad an skbuff up to a minimal size

Parameters

struct sk_buff * skb buffer to pad

unsigned int len minimal length

Description

Pads up a buffer to ensure the trailing bytes exist and are blanked. If the buffer already contains sufficient data it is untouched. Otherwise it is extended. Returns zero on success. The skb is freed on error.

int **__skb_put_padto**(struct *sk_buff* * *skb*, unsigned int *len*, bool *free_on_error*)
increase size and pad an skbuff up to a minimal size

Parameters

struct sk_buff * skb buffer to pad

unsigned int len minimal length

bool free_on_error free buffer on error

Description

Pads up a buffer to ensure the trailing bytes exist and are blanked. If the buffer already contains sufficient data it is untouched. Otherwise it is extended. Returns zero on success. The skb is freed on error if **free_on_error** is true.

int **skb_put_padto**(struct *sk_buff* * *skb*, unsigned int *len*)
increase size and pad an skbuff up to a minimal size

Parameters

struct sk_buff * skb buffer to pad

unsigned int len minimal length

Description

Pads up a buffer to ensure the trailing bytes exist and are blanked. If the buffer already contains sufficient data it is untouched. Otherwise it is extended. Returns zero on success. The skb is freed on error.

int **skb_linearize**(struct *sk_buff* * *skb*)
convert paged skb to linear one

Parameters

struct sk_buff * skb buffer to linarize

Description

If there is no free memory -ENOMEM is returned, otherwise zero is returned and the old skb data released.

bool **skb_has_shared_frag**(const struct *sk_buff* * *skb*)
can any frag be overwritten

Parameters

const struct sk_buff * skb buffer to test

Description

Return true if the skb has at least one frag that might be modified by an external entity (as in `vmsplice()`/`sendfile()`)

int **skb_linearize_cow**(struct *sk_buff* * *skb*)
make sure skb is linear and writable

Parameters

struct sk_buff * skb buffer to process

Description

If there is no free memory -ENOMEM is returned, otherwise zero is returned and the old skb data released.

void **skb_postpull_rcsum**(struct *sk_buff* * *skb*, const void * *start*, unsigned int *len*)
update checksum for received skb after pull

Parameters

struct sk_buff * skb buffer to update

const void * start start of data before pull

unsigned int len length of data pulled

Description

After doing a pull on a received packet, you need to call this to update the CHECKSUM_COMPLETE checksum, or set ip_summed to CHECKSUM_NONE so that it can be recomputed from scratch.

void **skb_postpush_rcsum**(struct *sk_buff* * *skb*, const void * *start*, unsigned int *len*)
update checksum for received skb after push

Parameters

struct sk_buff * skb buffer to update

const void * start start of data after push

unsigned int len length of data pushed

Description

After doing a push on a received packet, you need to call this to update the CHECKSUM_COMPLETE checksum.

void * **skb_push_rcsum**(struct *sk_buff* * *skb*, unsigned int *len*)
push skb and update receive checksum

Parameters

struct sk_buff * skb buffer to update

unsigned int len length of data pulled

Description

This function performs an skb_push on the packet and updates the CHECKSUM_COMPLETE checksum. It should be used on receive path processing instead of skb_push unless you know that the checksum difference is zero (e.g., a valid IP header) or you are setting ip_summed to CHECKSUM_NONE.

int **pskb_trim_rcsum**(struct *sk_buff* * *skb*, unsigned int *len*)
trim received skb and update checksum

Parameters

struct sk_buff * skb buffer to trim

unsigned int len new length

Description

This is exactly the same as pskb_trim except that it ensures the checksum of received packets are still valid after the operation.

bool **skb_needs_linearize**(struct *sk_buff* * *skb*, netdev_features_t *features*)
check if we need to linearize a given skb depending on the given device features.

Parameters

struct sk_buff * skb socket buffer to check

netdev_features_t features net device features

Description

Returns true if either: 1. skb has frag_list and the device doesn't support FRAGLIST, or 2. skb is fragmented and the device does not support SG.

void **skb_get_timestamp**(const struct *sk_buff* * *skb*, struct timeval * *stamp*)
get timestamp from a skb

Parameters

const struct sk_buff * skb skb to get stamp from

struct timeval * stamp pointer to struct timeval to store stamp in

Description

Timestamps are stored in the skb as offsets to a base timestamp. This function converts the offset back to a struct timeval and stores it in stamp.

void **skb_complete_tx_timestamp**(struct *sk_buff* * *skb*, struct *skb_shared_hwtstamps* * *hwtstamps*)
deliver cloned skb with tx timestamps

Parameters

struct sk_buff * skb clone of the the original outgoing packet

struct skb_shared_hwtstamps * hwtstamps hardware time stamps

Description

PHY drivers may accept clones of transmitted packets for timestamping via their phy_driver.txtstamp method. These drivers must call this function to return the skb back to the stack with a timestamp.

void **skb_tstamp_tx**(struct *sk_buff* * *orig_skb*, struct *skb_shared_hwtstamps* * *hwtstamps*)
queue clone of skb with send time stamps

Parameters

struct sk_buff * orig_skb the original outgoing packet

struct skb_shared_hwtstamps * hwtstamps hardware time stamps, may be NULL if not available

Description

If the skb has a socket associated, then this function clones the skb (thus sharing the actual data and optional structures), stores the optional hardware time stamping information (if non NULL) or generates a software time stamp (otherwise), then queues the clone to the error queue of the socket. Errors are silently ignored.

void **skb_tx_timestamp**(struct *sk_buff* * *skb*)
Driver hook for transmit timestamping

Parameters

struct sk_buff * skb A socket buffer.

Description

Ethernet MAC Drivers should call this function in their `hard_xmit()` function immediately before giving the sk_buff to the MAC hardware.

Specifically, one should make absolutely sure that this function is called before TX completion of this packet can trigger. Otherwise the packet could potentially already be freed.

void **skb_complete_wifi_ack**(struct *sk_buff* * *skb*, bool *acked*)
deliver skb with wifi status

Parameters

struct sk_buff * skb the original outgoing packet

bool acked ack status

__sum16 **skb_checksum_complete**(struct *sk_buff* * *skb*)
Calculate checksum of an entire packet

Parameters

struct sk_buff * skb packet to process

Description

This function calculates the checksum over the entire packet plus the value of skb->csum. The latter can be used to supply the checksum of a pseudo header as used by TCP/UDP. It returns the checksum.

For protocols that contain complete checksums such as ICMP/TCP/UDP, this function can be used to verify that checksum on received packets. In that case the function should return zero if the checksum is correct. In particular, this function will return zero if skb->ip_summed is CHECKSUM_UNNECESSARY which indicates that the hardware has already verified the correctness of the checksum.

void **skb_checksum_none_assert**(const struct *sk_buff* * *skb*)
make sure skb ip_summed is CHECKSUM_NONE

Parameters

const struct sk_buff * skb skb to check

Description

fresh skbs have their ip_summed set to CHECKSUM_NONE. Instead of forcing ip_summed to CHECKSUM_NONE, we can use this helper, to document places where we make this assertion.

bool **skb_head_is_locked**(const struct *sk_buff* * *skb*)
Determine if the skb->head is locked down

Parameters

const struct sk_buff * skb skb to check

Description

The head on skbs build around a head frag can be removed if they are not cloned. This function returns true if the skb head is locked down due to either being allocated via kmalloc, or by being a clone with multiple references to the head.

struct **sock_common**
minimal network layer representation of sockets

Definition

```
struct sock_common {
  union {
    __addrpair skc_addrpair;
    struct {
      __be32 skc_daddr;
      __be32 skc_rcv_saddr;
    };
  };
  union {
    unsigned int    skc_hash;
    __u16 skc_u16hashes[2];
  };
  union {
    __portpair skc_portpair;
    struct {
      __be16 skc_dport;
      __u16 skc_num;
    };
  };
```

```
  unsigned short          skc_family;
  volatile unsigned char  skc_state;
  unsigned char           skc_reuse:4;
  unsigned char           skc_reuseport:1;
  unsigned char           skc_ipv6only:1;
  unsigned char           skc_net_refcnt:1;
  int skc_bound_dev_if;
  union {
    struct hlist_node       skc_bind_node;
    struct hlist_node       skc_portaddr_node;
  };
  struct proto            *skc_prot;
  possible_net_t skc_net;
#if IS_ENABLED(CONFIG_IPV6);
  struct in6_addr         skc_v6_daddr;
  struct in6_addr         skc_v6_rcv_saddr;
#endif;
  atomic64_t skc_cookie;
  union {
    unsigned long   skc_flags;
    struct sock     *skc_listener;
    struct inet_timewait_death_row *skc_tw_dr;
  };
  union {
    struct hlist_node       skc_node;
    struct hlist_nulls_node skc_nulls_node;
  };
  int skc_tx_queue_mapping;
  union {
    int skc_incoming_cpu;
    u32 skc_rcv_wnd;
    u32 skc_tw_rcv_nxt;
  };
  refcount_t skc_refcnt;
};
```

Members

{unnamed_union} anonymous

{unnamed_struct} anonymous

skc_daddr Foreign IPv4 addr

skc_rcv_saddr Bound local IPv4 addr

{unnamed_union} anonymous

skc_hash hash value used with various protocol lookup tables

skc_u16hashes two u16 hash values used by UDP lookup tables

{unnamed_union} anonymous

{unnamed_struct} anonymous

skc_dport placeholder for inet_dport/tw_dport

skc_num placeholder for inet_num/tw_num

skc_family network address family

skc_state Connection state

skc_reuse SO_REUSEADDR setting

skc_reuseport SO_REUSEPORT setting

skc_bound_dev_if bound device index if != 0

{unnamed_union} anonymous

skc_bind_node bind hash linkage for various protocol lookup tables

skc_portaddr_node second hash linkage for UDP/UDP-Lite protocol

skc_prot protocol handlers inside a network family

skc_net reference to the network namespace of this socket

{unnamed_union} anonymous

skc_flags place holder for sk_flags SO_LINGER (l_onoff), SO_BROADCAST, SO_KEEPALIVE, SO_OOBINLINE settings, SO_TIMESTAMPING settings

{unnamed_union} anonymous

skc_node main hash linkage for various protocol lookup tables

skc_nulls_node main hash linkage for TCP/UDP/UDP-Lite protocol

skc_tx_queue_mapping tx queue number for this connection

{unnamed_union} anonymous

skc_incoming_cpu record/match cpu processing incoming packets

skc_refcnt reference count

Description

This is the minimal network layer representation of sockets, the header for struct sock and struct inet_timewait_sock.

struct **sock**
network layer representation of sockets

Definition

```
struct sock {
  struct sock_common      __sk_common;
#define sk_node                 __sk_common.skc_node;
#define sk_nulls_node           __sk_common.skc_nulls_node;
#define sk_refcnt               __sk_common.skc_refcnt;
#define sk_tx_queue_mapping     __sk_common.skc_tx_queue_mapping;
#define sk_dontcopy_begin       __sk_common.skc_dontcopy_begin;
#define sk_dontcopy_end         __sk_common.skc_dontcopy_end;
#define sk_hash                 __sk_common.skc_hash;
#define sk_portpair             __sk_common.skc_portpair;
#define sk_num                  __sk_common.skc_num;
#define sk_dport                __sk_common.skc_dport;
#define sk_addrpair             __sk_common.skc_addrpair;
#define sk_daddr                __sk_common.skc_daddr;
#define sk_rcv_saddr            __sk_common.skc_rcv_saddr;
#define sk_family               __sk_common.skc_family;
#define sk_state                __sk_common.skc_state;
#define sk_reuse                __sk_common.skc_reuse;
#define sk_reuseport            __sk_common.skc_reuseport;
#define sk_ipv6only             __sk_common.skc_ipv6only;
#define sk_net_refcnt           __sk_common.skc_net_refcnt;
#define sk_bound_dev_if         __sk_common.skc_bound_dev_if;
#define sk_bind_node            __sk_common.skc_bind_node;
#define sk_prot                 __sk_common.skc_prot;
#define sk_net                  __sk_common.skc_net;
#define sk_v6_daddr             __sk_common.skc_v6_daddr;
#define sk_v6_rcv_saddr __sk_common.skc_v6_rcv_saddr;
#define sk_cookie               __sk_common.skc_cookie;
#define sk_incoming_cpu         __sk_common.skc_incoming_cpu;
#define sk_flags                __sk_common.skc_flags;
```

```
#define sk_rxhash               __sk_common.skc_rxhash;
  socket_lock_t sk_lock;
  atomic_t sk_drops;
  int sk_rcvlowat;
  struct sk_buff_head     sk_error_queue;
  struct sk_buff_head     sk_receive_queue;
  struct {
    atomic_t rmem_alloc;
    int len;
    struct sk_buff  *head;
    struct sk_buff  *tail;
  } sk_backlog;
#define sk_rmem_alloc sk_backlog.rmem_alloc;
  int sk_forward_alloc;
#ifdef CONFIG_NET_RX_BUSY_POLL;
  unsigned int            sk_ll_usec;
  unsigned int            sk_napi_id;
#endif;
  int sk_rcvbuf;
  struct sk_filter __rcu  *sk_filter;
  union {
    struct socket_wq __rcu  *sk_wq;
    struct socket_wq        *sk_wq_raw;
  };
#ifdef CONFIG_XFRM;
  struct xfrm_policy __rcu *sk_policy[2];
#endif;
  struct dst_entry        *sk_rx_dst;
  struct dst_entry __rcu  *sk_dst_cache;
  atomic_t sk_omem_alloc;
  int sk_sndbuf;
  int sk_wmem_queued;
  refcount_t sk_wmem_alloc;
  unsigned long           sk_tsq_flags;
  union {
    struct sk_buff  *sk_send_head;
    struct rb_root  tcp_rtx_queue;
  };
  struct sk_buff_head     sk_write_queue;
  __s32 sk_peek_off;
  int sk_write_pending;
  __u32 sk_dst_pending_confirm;
  u32 sk_pacing_status;
  long sk_sndtimeo;
  struct timer_list       sk_timer;
  __u32 sk_priority;
  __u32 sk_mark;
  u32 sk_pacing_rate;
  u32 sk_max_pacing_rate;
  struct page_frag        sk_frag;
  netdev_features_t sk_route_caps;
  netdev_features_t sk_route_nocaps;
  int sk_gso_type;
  unsigned int            sk_gso_max_size;
  gfp_t sk_allocation;
  __u32 sk_txhash;
  unsigned int            __sk_flags_offset[0];
#ifdef __BIG_ENDIAN_BITFIELD;
#define SK_FL_PROTO_SHIFT  16;
#define SK_FL_PROTO_MASK   0x00ff0000;
#define SK_FL_TYPE_SHIFT   0;
#define SK_FL_TYPE_MASK    0x0000ffff;
#else;
```

```
#define SK_FL_PROTO_SHIFT  8;
#define SK_FL_PROTO_MASK   0x0000ff00;
#define SK_FL_TYPE_SHIFT   16;
#define SK_FL_TYPE_MASK    0xffff0000;
#endif;
  unsigned int           sk_padding : 1,sk_kern_sock : 1,sk_no_check_tx : 1,sk_no_check_rx : 1,sk_userl
#define SK_PROTOCOL_MAX U8_MAX;
  u16 sk_gso_max_segs;
  u8 sk_pacing_shift;
  unsigned long          sk_lingertime;
  struct proto           *sk_prot_creator;
  rwlock_t sk_callback_lock;
  int sk_err, sk_err_soft;
  u32 sk_ack_backlog;
  u32 sk_max_ack_backlog;
  kuid_t sk_uid;
  struct pid             *sk_peer_pid;
  const struct cred      *sk_peer_cred;
  long sk_rcvtimeo;
  ktime_t sk_stamp;
  u16 sk_tsflags;
  u8 sk_shutdown;
  u32 sk_tskey;
  atomic_t sk_zckey;
  struct socket          *sk_socket;
  void *sk_user_data;
#ifdef CONFIG_SECURITY;
  void *sk_security;
#endif;
  struct sock_cgroup_data sk_cgrp_data;
  struct mem_cgroup      *sk_memcg;
  void (*sk_state_change)(struct sock *sk);
  void (*sk_data_ready)(struct sock *sk);
  void (*sk_write_space)(struct sock *sk);
  void (*sk_error_report)(struct sock *sk);
  int (*sk_backlog_rcv)(struct sock *sk, struct sk_buff *skb);
  void (*sk_destruct)(struct sock *sk);
  struct sock_reuseport __rcu     *sk_reuseport_cb;
  struct rcu_head        sk_rcu;
};
```

Members

`__sk_common` shared layout with inet_timewait_sock

`sk_lock` synchronizer

`sk_drops` raw/udp drops counter

`sk_rcvlowat` SO_RCVLOWAT setting

`sk_error_queue` rarely used

`sk_receive_queue` incoming packets

`sk_backlog` always used with the per-socket spinlock held

`sk_forward_alloc` space allocated forward

`sk_ll_usec` usecs to busypoll when there is no data

`sk_napi_id` id of the last napi context to receive data for sk

`sk_rcvbuf` size of receive buffer in bytes

`sk_filter` socket filtering instructions

`{unnamed_union}` anonymous

sk_wq sock wait queue and async head

sk_policy flow policy

sk_rx_dst receive input route used by early demux

sk_dst_cache destination cache

sk_omem_alloc "o" is "option" or "other"

sk_sndbuf size of send buffer in bytes

sk_wmem_queued persistent queue size

sk_wmem_alloc transmit queue bytes committed

sk_tsq_flags TCP Small Queues flags

{unnamed_union} anonymous

sk_send_head front of stuff to transmit

sk_write_queue Packet sending queue

sk_peek_off current peek_offset value

sk_write_pending a write to stream socket waits to start

sk_dst_pending_confirm need to confirm neighbour

sk_pacing_status Pacing status (requested, handled by sch_fq)

sk_sndtimeo SO_SNDTIMEO setting

sk_timer sock cleanup timer

sk_priority SO_PRIORITY setting

sk_mark generic packet mark

sk_pacing_rate Pacing rate (if supported by transport/packet scheduler)

sk_max_pacing_rate Maximum pacing rate (SO_MAX_PACING_RATE)

sk_frag cached page frag

sk_route_caps route capabilities (e.g. NETIF_F_TSO)

sk_route_nocaps forbidden route capabilities (e.g NETIF_F_GSO_MASK)

sk_gso_type GSO type (e.g. SKB_GSO_TCPV4)

sk_gso_max_size Maximum GSO segment size to build

sk_allocation allocation mode

sk_txhash computed flow hash for use on transmit

__sk_flags_offset empty field used to determine location of bitfield

sk_padding unused element for alignment

sk_kern_sock True if sock is using kernel lock classes

sk_no_check_tx SO_NO_CHECK setting, set checksum in TX packets

sk_no_check_rx allow zero checksum in RX packets

sk_userlocks SO_SNDBUF and SO_RCVBUF settings

sk_protocol which protocol this socket belongs in this network family

sk_type socket type (SOCK_STREAM, etc)

sk_gso_max_segs Maximum number of GSO segments

sk_pacing_shift scaling factor for TCP Small Queues

sk_lingertime SO_LINGER l_linger setting

sk_prot_creator sk_prot of original sock creator (see ipv6_setsockopt, IPV6_ADDRFORM for instance)

sk_callback_lock used with the callbacks in the end of this struct

sk_err last error

sk_err_soft errors that don't cause failure but are the cause of a persistent failure not just 'timed out'

sk_ack_backlog current listen backlog

sk_max_ack_backlog listen backlog set in `listen()`

sk_uid user id of owner

sk_peer_pid `struct pid` for this socket's peer

sk_peer_cred `SO_PEERCRED` setting

sk_rcvtimeo `SO_RCVTIMEO` setting

sk_stamp time stamp of last packet received

sk_tsflags SO_TIMESTAMPING socket options

sk_shutdown mask of `SEND_SHUTDOWN` and/or `RCV_SHUTDOWN`

sk_tskey counter to disambiguate concurrent tstamp requests

sk_zckey counter to order MSG_ZEROCOPY notifications

sk_socket Identd and reporting IO signals

sk_user_data RPC layer private data

sk_security used by security modules

sk_cgrp_data cgroup data for this cgroup

sk_memcg this socket's memory cgroup association

sk_state_change callback to indicate change in the state of the sock

sk_data_ready callback to indicate there is data to be processed

sk_write_space callback to indicate there is bf sending space available

sk_error_report callback to indicate errors (e.g. `MSG_ERRQUEUE`)

sk_backlog_rcv callback to process the backlog

sk_destruct called at sock freeing time, i.e. when all refcnt == 0

sk_reuseport_cb reuseport group container

sk_rcu used during RCU grace period

sk_for_each_entry_offset_rcu(*tpos*, *pos*, *head*, *offset*)
iterate over a list at a given struct offset

Parameters

tpos the type * to use as a loop cursor.

pos the `struct hlist_node` to use as a loop cursor.

head the head for your list.

offset offset of hlist_node within the struct.

void **unlock_sock_fast**(struct *sock* * *sk*, bool *slow*)
complement of lock_sock_fast

Parameters

struct sock * sk socket

bool slow slow mode

Description

fast unlock socket for user context. If slow mode is on, we call regular `release_sock()`

int **sk_wmem_alloc_get**(const struct *sock* * *sk*)
returns write allocations

Parameters

const struct sock * sk socket

Description

Returns sk_wmem_alloc minus initial offset of one

int **sk_rmem_alloc_get**(const struct *sock* * *sk*)
returns read allocations

Parameters

const struct sock * sk socket

Description

Returns sk_rmem_alloc

bool **sk_has_allocations**(const struct *sock* * *sk*)
check if allocations are outstanding

Parameters

const struct sock * sk socket

Description

Returns true if socket has write or read allocations

bool **skwq_has_sleeper**(struct socket_wq * *wq*)
check if there are any waiting processes

Parameters

struct socket_wq * wq struct socket_wq

Description

Returns true if socket_wq has waiting processes

The purpose of the skwq_has_sleeper and sock_poll_wait is to wrap the memory barrier call. They were added due to the race found within the tcp code.

Consider following tcp code paths:

```
CPU1                    CPU2
sys_select              receive packet
...                     ...
__add_wait_queue        update tp->rcv_nxt
...                     ...
tp->rcv_nxt check       sock_def_readable
...                     {
schedule                   :c:func:`rcu_read_lock()`;
                           wq = rcu_dereference(sk->sk_wq);
                           if (wq && waitqueue_active(:c:type:`wq->wait <wq>`))
                               wake_up_interruptible(:c:type:`wq->wait <wq>`)
                           ...
                        }
```

The race for tcp fires when the __add_wait_queue changes done by CPU1 stay in its cache, and so does the tp->rcv_nxt update on CPU2 side. The CPU1 could then endup calling schedule and sleep forever if there are no more data on the socket.

void **sock_poll_wait**(struct file * *filp*, wait_queue_head_t * *wait_address*, poll_table * *p*)
place memory barrier behind the poll_wait call.

Parameters

struct file * filp file

wait_queue_head_t * wait_address socket wait queue

poll_table * p poll_table

Description

See the comments in the wq_has_sleeper function.

struct page_frag * **sk_page_frag**(struct *sock* * *sk*)
return an appropriate page_frag

Parameters

struct sock * sk socket

Description

If socket allocation mode allows current thread to sleep, it means its safe to use the per task page_frag instead of the per socket one.

void **sock_tx_timestamp**(const struct *sock* * *sk*, __u16 *tsflags*, __u8 * *tx_flags*)
checks whether the outgoing packet is to be time stamped

Parameters

const struct sock * sk socket sending this packet

__u16 tsflags timestamping flags to use

__u8 * tx_flags completed with instructions for time stamping

Note

callers should take care of initial `*tx_flags` value (usually 0)

void **sk_eat_skb**(struct *sock* * *sk*, struct *sk_buff* * *skb*)
Release a skb if it is no longer needed

Parameters

struct sock * sk socket to eat this skb from

struct sk_buff * skb socket buffer to eat

Description

This routine must be called with interrupts disabled or with the socket locked so that the sk_buff queue operation is ok.

struct *socket* * **sockfd_lookup**(int *fd*, int * *err*)
Go from a file number to its socket slot

Parameters

int fd file handle

int * err pointer to an error code return

Description

The file handle passed in is locked and the socket it is bound to is returned. If an error occurs the err pointer is overwritten with a negative errno code and NULL is returned. The function checks for both invalid handles and passing a handle which is not a socket.

On a success the socket object pointer is returned.

struct *socket* * **sock_alloc**(void)
allocate a socket

Parameters

void no arguments

Description

Allocate a new inode and socket object. The two are bound together and initialised. The socket is then returned. If we are out of inodes NULL is returned.

void **sock_release**(struct *socket* * *sock*)
close a socket

Parameters

struct socket * sock socket to close

Description

The socket is released from the protocol stack if it has a release callback, and the inode is then released if the socket is bound to an inode not a file.

int **kernel_recvmsg**(struct *socket* * *sock*, struct msghdr * *msg*, struct kvec * *vec*, size_t *num*, size_t *size*, int *flags*)
Receive a message from a socket (kernel space)

Parameters

struct socket * sock The socket to receive the message from

struct msghdr * msg Received message

struct kvec * vec Input s/g array for message data

size_t num Size of input s/g array

size_t size Number of bytes to read

int flags Message flags (MSG_DONTWAIT, etc...)

Description

On return the msg structure contains the scatter/gather array passed in the vec argument. The array is modified so that it consists of the unfilled portion of the original array.

The returned value is the total number of bytes received, or an error.

int **sock_register**(const struct net_proto_family * *ops*)
add a socket protocol handler

Parameters

const struct net_proto_family * ops description of protocol

Description

This function is called by a protocol handler that wants to advertise its address family, and have it linked into the socket interface. The value ops->family corresponds to the socket system call protocol family.

void **sock_unregister**(int *family*)
remove a protocol handler

Parameters

int family protocol family to remove

Description

This function is called by a protocol handler that wants to remove its address family, and have it unlinked from the new socket creation.

If protocol handler is a module, then it can use module reference counts to protect against new references. If protocol handler is not a module then it needs to provide its own protection in the ops->create routine.

struct *sk_buff* * **__alloc_skb**(unsigned int *size*, gfp_t *gfp_mask*, int *flags*, int *node*)
allocate a network buffer

Parameters

unsigned int size size to allocate

gfp_t gfp_mask allocation mask

int flags If SKB_ALLOC_FCLONE is set, allocate from fclone cache instead of head cache and allocate a cloned (child) skb. If SKB_ALLOC_RX is set, __GFP_MEMALLOC will be used for allocations in case the data is required for writeback

int node numa node to allocate memory on

Description

Allocate a new *sk_buff*. The returned buffer has no headroom and a tail room of at least size bytes. The object has a reference count of one. The return is the buffer. On a failure the return is `NULL`.

Buffers may only be allocated from interrupts using a **gfp_mask** of `GFP_ATOMIC`.

void * **netdev_alloc_frag**(unsigned int *fragsz*)
allocate a page fragment

Parameters

unsigned int fragsz fragment size

Description

Allocates a frag from a page for receive buffer. Uses GFP_ATOMIC allocations.

struct *sk_buff* * **__netdev_alloc_skb**(struct *net_device* * *dev*, unsigned int *len*, gfp_t *gfp_mask*)
allocate an skbuff for rx on a specific device

Parameters

struct net_device * dev network device to receive on

unsigned int len length to allocate

gfp_t gfp_mask get_free_pages mask, passed to alloc_skb

Description

Allocate a new *sk_buff* and assign it a usage count of one. The buffer has NET_SKB_PAD headroom built in. Users should allocate the headroom they think they need without accounting for the built in space. The built in space is used for optimisations.

`NULL` is returned if there is no free memory.

struct *sk_buff* * **__napi_alloc_skb**(struct napi_struct * *napi*, unsigned int *len*, gfp_t *gfp_mask*)
allocate skbuff for rx in a specific NAPI instance

Parameters

struct napi_struct * napi napi instance this buffer was allocated for

unsigned int len length to allocate

gfp_t gfp_mask get_free_pages mask, passed to alloc_skb and alloc_pages

Description

Allocate a new sk_buff for use in NAPI receive. This buffer will attempt to allocate the head from a special reserved region used only for NAPI Rx allocation. By doing this we can save several CPU cycles by avoiding having to disable and re-enable IRQs.

NULL is returned if there is no free memory.

void **__kfree_skb**(struct *sk_buff* * *skb*)
private function

Parameters

struct sk_buff * skb buffer

Description

Free an sk_buff. Release anything attached to the buffer. Clean the state. This is an internal helper function. Users should always call kfree_skb

void **kfree_skb**(struct *sk_buff* * *skb*)
free an sk_buff

Parameters

struct sk_buff * skb buffer to free

Description

Drop a reference to the buffer and free it if the usage count has hit zero.

void **skb_tx_error**(struct *sk_buff* * *skb*)
report an sk_buff xmit error

Parameters

struct sk_buff * skb buffer that triggered an error

Description

Report xmit error if a device callback is tracking this skb. skb must be freed afterwards.

void **consume_skb**(struct *sk_buff* * *skb*)
free an skbuff

Parameters

struct sk_buff * skb buffer to free

Description

Drop a ref to the buffer and free it if the usage count has hit zero Functions identically to kfree_skb, but kfree_skb assumes that the frame is being dropped after a failure and notes that

struct *sk_buff* * **skb_morph**(struct *sk_buff* * *dst*, struct *sk_buff* * *src*)
morph one skb into another

Parameters

struct sk_buff * dst the skb to receive the contents

struct sk_buff * src the skb to supply the contents

Description

This is identical to skb_clone except that the target skb is supplied by the user.

The target skb is returned upon exit.

int **skb_copy_ubufs**(struct *sk_buff* * *skb*, gfp_t *gfp_mask*)
copy userspace skb frags buffers to kernel

Parameters

struct sk_buff * skb the skb to modify

gfp_t gfp_mask allocation priority

Description

This must be called on SKBTX_DEV_ZEROCOPY skb. It will copy all frags into kernel and drop the reference to userspace pages.

If this function is called from an interrupt `gfp_mask()` must be `GFP_ATOMIC`.

Returns 0 on success or a negative error code on failure to allocate kernel memory to copy to.

struct *sk_buff* * **skb_clone**(struct *sk_buff* * *skb*, gfp_t *gfp_mask*)
duplicate an sk_buff

Parameters

struct sk_buff * skb buffer to clone

gfp_t gfp_mask allocation priority

Description

Duplicate an *sk_buff*. The new one is not owned by a socket. Both copies share the same packet data but not structure. The new buffer has a reference count of 1. If the allocation fails the function returns `NULL` otherwise the new buffer is returned.

If this function is called from an interrupt `gfp_mask()` must be `GFP_ATOMIC`.

struct *sk_buff* * **skb_copy**(const struct *sk_buff* * *skb*, gfp_t *gfp_mask*)
create private copy of an sk_buff

Parameters

const struct sk_buff * skb buffer to copy

gfp_t gfp_mask allocation priority

Description

Make a copy of both an *sk_buff* and its data. This is used when the caller wishes to modify the data and needs a private copy of the data to alter. Returns `NULL` on failure or the pointer to the buffer on success. The returned buffer has a reference count of 1.

As by-product this function converts non-linear *sk_buff* to linear one, so that *sk_buff* becomes completely private and caller is allowed to modify all the data of returned buffer. This means that this function is not recommended for use in circumstances when only header is going to be modified. Use `pskb_copy()` instead.

struct *sk_buff* * **__pskb_copy_fclone**(struct *sk_buff* * *skb*, int *headroom*, gfp_t *gfp_mask*, bool *fclone*)
create copy of an sk_buff with private head.

Parameters

struct sk_buff * skb buffer to copy

int headroom headroom of new skb

gfp_t gfp_mask allocation priority

bool fclone if true allocate the copy of the skb from the fclone cache instead of the head cache; it is recommended to set this to true for the cases where the copy will likely be cloned

Description

Make a copy of both an *sk_buff* and part of its data, located in header. Fragmented data remain shared. This is used when the caller wishes to modify only header of *sk_buff* and needs private copy of the header to alter. Returns `NULL` on failure or the pointer to the buffer on success. The returned buffer has a reference count of 1.

int **pskb_expand_head**(struct *sk_buff* * *skb*, int *nhead*, int *ntail*, gfp_t *gfp_mask*)
reallocate header of *sk_buff*

Parameters

struct sk_buff * skb buffer to reallocate

int nhead room to add at head

int ntail room to add at tail

gfp_t gfp_mask allocation priority

Description

Expands (or creates identical copy, if **nhead** and **ntail** are zero) header of **skb**. *sk_buff* itself is not changed. *sk_buff* MUST have reference count of 1. Returns zero in the case of success or error, if expansion failed. In the last case, *sk_buff* is not changed.

All the pointers pointing into skb header may change and must be reloaded after call to this function.

struct *sk_buff* * **skb_copy_expand**(const struct *sk_buff* * *skb*, int *newheadroom*, int *newtailroom*, gfp_t *gfp_mask*)
copy and expand sk_buff

Parameters

const struct sk_buff * skb buffer to copy

int newheadroom new free bytes at head

int newtailroom new free bytes at tail

gfp_t gfp_mask allocation priority

Description

Make a copy of both an *sk_buff* and its data and while doing so allocate additional space.

This is used when the caller wishes to modify the data and needs a private copy of the data to alter as well as more space for new fields. Returns `NULL` on failure or the pointer to the buffer on success. The returned buffer has a reference count of 1.

You must pass `GFP_ATOMIC` as the allocation priority if this function is called from an interrupt.

int **__skb_pad**(struct *sk_buff* * *skb*, int *pad*, bool *free_on_error*)
zero pad the tail of an skb

Parameters

struct sk_buff * skb buffer to pad

int pad space to pad

bool free_on_error free buffer on error

Description

Ensure that a buffer is followed by a padding area that is zero filled. Used by network drivers which may DMA or transfer data beyond the buffer end onto the wire.

May return error in out of memory cases. The skb is freed on error if **free_on_error** is true.

void * **pskb_put**(struct *sk_buff* * *skb*, struct *sk_buff* * *tail*, int *len*)
add data to the tail of a potentially fragmented buffer

Parameters

struct sk_buff * skb start of the buffer to use

struct sk_buff * tail tail fragment of the buffer to use

int len amount of data to add

Description

This function extends the used data area of the potentially fragmented buffer. **tail** must be the last fragment of **skb** – or **skb** itself. If this would exceed the total buffer size the kernel will panic. A pointer to the first byte of the extra data is returned.

void * **skb_put**(struct *sk_buff* * *skb*, unsigned int *len*)
add data to a buffer

Parameters

struct sk_buff * skb buffer to use

unsigned int len amount of data to add

Description

This function extends the used data area of the buffer. If this would exceed the total buffer size the kernel will panic. A pointer to the first byte of the extra data is returned.

void * **skb_push**(struct *sk_buff* * *skb*, unsigned int *len*)
add data to the start of a buffer

Parameters

struct sk_buff * skb buffer to use

unsigned int len amount of data to add

Description

This function extends the used data area of the buffer at the buffer start. If this would exceed the total buffer headroom the kernel will panic. A pointer to the first byte of the extra data is returned.

void * **skb_pull**(struct *sk_buff* * *skb*, unsigned int *len*)
remove data from the start of a buffer

Parameters

struct sk_buff * skb buffer to use

unsigned int len amount of data to remove

Description

This function removes data from the start of a buffer, returning the memory to the headroom. A pointer to the next data in the buffer is returned. Once the data has been pulled future pushes will overwrite the old data.

void **skb_trim**(struct *sk_buff* * *skb*, unsigned int *len*)
remove end from a buffer

Parameters

struct sk_buff * skb buffer to alter

unsigned int len new length

Description

Cut the length of a buffer down by removing data from the tail. If the buffer is already under the length specified it is not modified. The skb must be linear.

void * **__pskb_pull_tail**(struct *sk_buff* * *skb*, int *delta*)
advance tail of skb header

Parameters

struct sk_buff * skb buffer to reallocate

int delta number of bytes to advance tail

Description

The function makes a sense only on a fragmented *sk_buff*, it expands header moving its tail forward and copying necessary data from fragmented part.

sk_buff MUST have reference count of 1.

Returns NULL (and *sk_buff* does not change) if pull failed or value of new tail of skb in the case of success.

All the pointers pointing into skb header may change and must be reloaded after call to this function.

int **skb_copy_bits**(const struct *sk_buff* * *skb*, int *offset*, void * *to*, int *len*)
copy bits from skb to kernel buffer

Parameters

const struct sk_buff * skb source skb

int offset offset in source

void * to destination buffer

int len number of bytes to copy

Description

Copy the specified number of bytes from the source skb to the destination buffer.

CAUTION ! : If its prototype is ever changed, check arch/{*}/net/{*}.S files, since it is called from BPF assembly code.

int **skb_store_bits**(struct *sk_buff* * *skb*, int *offset*, const void * *from*, int *len*)
store bits from kernel buffer to skb

Parameters

struct sk_buff * skb destination buffer

int offset offset in destination

const void * from source buffer

int len number of bytes to copy

Description

Copy the specified number of bytes from the source buffer to the destination skb. This function handles all the messy bits of traversing fragment lists and such.

int **skb_zerocopy**(struct *sk_buff* * *to*, struct *sk_buff* * *from*, int *len*, int *hlen*)
Zero copy skb to skb

Parameters

struct sk_buff * to destination buffer

struct sk_buff * from source buffer

int len number of bytes to copy from source buffer

int hlen size of linear headroom in destination buffer

Description

Copies up to *len* bytes from *from* to *to* by creating references to the frags in the source buffer.

The *hlen* as calculated by `skb_zerocopy_headlen()` specifies the headroom in the *to* buffer.

Return value: 0: everything is OK -ENOMEM: couldn't orphan frags of **from** due to lack of memory -EFAULT: *skb_copy_bits()* found some problem with skb geometry

struct *sk_buff* * **skb_dequeue**(struct sk_buff_head * *list*)
remove from the head of the queue

Parameters

struct sk_buff_head * list list to dequeue from

Description

Remove the head of the list. The list lock is taken so the function may be used safely with other locking list functions. The head item is returned or NULL if the list is empty.

struct *sk_buff* * **skb_dequeue_tail**(struct sk_buff_head * *list*)
remove from the tail of the queue

Parameters

struct sk_buff_head * list list to dequeue from

Description

Remove the tail of the list. The list lock is taken so the function may be used safely with other locking list functions. The tail item is returned or NULL if the list is empty.

void **skb_queue_purge**(struct sk_buff_head * *list*)
empty a list

Parameters

struct sk_buff_head * list list to empty

Description

Delete all buffers on an *sk_buff* list. Each buffer is removed from the list and one reference dropped. This function takes the list lock and is atomic with respect to other list locking functions.

void **skb_queue_head**(struct sk_buff_head * *list*, struct *sk_buff* * *newsk*)
queue a buffer at the list head

Parameters

struct sk_buff_head * list list to use

struct sk_buff * newsk buffer to queue

Description

Queue a buffer at the start of the list. This function takes the list lock and can be used safely with other locking *sk_buff* functions safely.

A buffer cannot be placed on two lists at the same time.

void **skb_queue_tail**(struct sk_buff_head * *list*, struct *sk_buff* * *newsk*)
queue a buffer at the list tail

Parameters

struct sk_buff_head * list list to use

struct sk_buff * newsk buffer to queue

Description

Queue a buffer at the tail of the list. This function takes the list lock and can be used safely with other locking *sk_buff* functions safely.

A buffer cannot be placed on two lists at the same time.

void **skb_unlink**(struct *sk_buff* * *skb*, struct sk_buff_head * *list*)
remove a buffer from a list

Parameters

struct sk_buff * skb buffer to remove

struct sk_buff_head * list list to use

Description

Remove a packet from a list. The list locks are taken and this function is atomic with respect to other list locked calls

You must know what list the SKB is on.

void **skb_append**(struct *sk_buff* * *old*, struct *sk_buff* * *newsk*, struct sk_buff_head * *list*)
append a buffer

Parameters

struct sk_buff * old buffer to insert after

struct sk_buff * newsk buffer to insert

struct sk_buff_head * list list to use

Description

Place a packet after a given packet in a list. The list locks are taken and this function is atomic with respect to other list locked calls. A buffer cannot be placed on two lists at the same time.

void **skb_insert**(struct *sk_buff* * *old*, struct *sk_buff* * *newsk*, struct sk_buff_head * *list*)
insert a buffer

Parameters

struct sk_buff * old buffer to insert before

struct sk_buff * newsk buffer to insert

struct sk_buff_head * list list to use

Description

Place a packet before a given packet in a list. The list locks are taken and this function is atomic with respect to other list locked calls.

A buffer cannot be placed on two lists at the same time.

void **skb_split**(struct *sk_buff* * *skb*, struct *sk_buff* * *skb1*, const u32 *len*)
Split fragmented skb to two parts at length len.

Parameters

struct sk_buff * skb the buffer to split

struct sk_buff * skb1 the buffer to receive the second part

const u32 len new length for skb

void **skb_prepare_seq_read**(struct *sk_buff* * *skb*, unsigned int *from*, unsigned int *to*, struct skb_seq_state * *st*)
Prepare a sequential read of skb data

Parameters

struct sk_buff * skb the buffer to read

unsigned int from lower offset of data to be read

unsigned int to upper offset of data to be read

struct skb_seq_state * st state variable

Description

Initializes the specified state variable. Must be called before invoking *skb_seq_read()* for the first time.

unsigned int **skb_seq_read**(unsigned int *consumed*, const u8 ** *data*, struct skb_seq_state * *st*)
Sequentially read skb data

Parameters

unsigned int consumed number of bytes consumed by the caller so far

const u8 ** data destination pointer for data to be returned

struct skb_seq_state * st state variable

Description

Reads a block of skb data at **consumed** relative to the lower offset specified to *skb_prepare_seq_read()*. Assigns the head of the data block to **data** and returns the length of the block or 0 if the end of the skb data or the upper offset has been reached.

The caller is not required to consume all of the data returned, i.e. **consumed** is typically set to the number of bytes already consumed and the next call to *skb_seq_read()* will return the remaining part of the block.

Note 1: The size of each block of data returned can be arbitrary, this limitation is the cost for zerocopy sequential reads of potentially non linear data.

Note 2: Fragment lists within fragments are not implemented at the moment, state->root_skb could be replaced with a stack for this purpose.

void **skb_abort_seq_read**(struct skb_seq_state * *st*)
Abort a sequential read of skb data

Parameters

struct skb_seq_state * st state variable

Description

Must be called if *skb_seq_read()* was not called until it returned 0.

unsigned int **skb_find_text**(struct *sk_buff* * *skb*, unsigned int *from*, unsigned int *to*, struct ts_config * *config*)
Find a text pattern in skb data

Parameters

struct sk_buff * skb the buffer to look in

unsigned int from search offset

unsigned int to search limit

struct ts_config * config textsearch configuration

Description

Finds a pattern in the skb data according to the specified textsearch configuration. Use `textsearch_next()` to retrieve subsequent occurrences of the pattern. Returns the offset to the first occurrence or UINT_MAX if no match was found.

int **skb_append_datato_frags**(struct *sock* * *sk*, struct *sk_buff* * *skb*, int (*getfrag) (void **from*, char **to*, int *offset*, int *len*, int *odd*, struct *sk_buff* **skb*, void * *from*, int *length*)
append the user data to a skb

Parameters

struct sock * sk sock structure

struct sk_buff * skb skb structure to be appended with user data.

int (*)(void *from, char *to, int offset, int len, int odd, struct sk_buff *skb) getfrag
call back function to be used for getting the user data

void * from pointer to user message iov

int length length of the iov message

Description

This procedure append the user data in the fragment part of the skb if any page alloc fails user this procedure returns -ENOMEM

void * **skb_pull_rcsum**(struct *sk_buff* * *skb*, unsigned int *len*)
pull skb and update receive checksum

Parameters

struct sk_buff * skb buffer to update

unsigned int len length of data pulled

Description

This function performs an skb_pull on the packet and updates the CHECKSUM_COMPLETE checksum. It should be used on receive path processing instead of skb_pull unless you know that the checksum difference is zero (e.g., a valid IP header) or you are setting ip_summed to CHECKSUM_NONE.

struct *sk_buff* * **skb_segment**(struct *sk_buff* * *head_skb*, netdev_features_t *features*)
Perform protocol segmentation on skb.

Parameters

struct sk_buff * head_skb buffer to segment

netdev_features_t features features for the output path (see dev->features)

Description

This function performs segmentation on the given skb. It returns a pointer to the first in a list of new skbs for the segments. In case of error it returns ERR_PTR(err).

int **skb_to_sgvec**(struct *sk_buff* * *skb*, struct scatterlist * *sg*, int *offset*, int *len*)
Fill a scatter-gather list from a socket buffer

Parameters

struct sk_buff * skb Socket buffer containing the buffers to be mapped

struct scatterlist * sg The scatter-gather list to map into

int offset The offset into the buffer's contents to start mapping

int len Length of buffer space to be mapped

Description

Fill the specified scatter-gather list with mappings/pointers into a region of the buffer space attached to a socket buffer. Returns either the number of scatterlist items used, or -EMSGSIZE if the contents could not fit.

int **skb_cow_data**(struct *sk_buff* * *skb*, int *tailbits*, struct *sk_buff* ** *trailer*)
Check that a socket buffer's data buffers are writable

Parameters

struct sk_buff * skb The socket buffer to check.

int tailbits Amount of trailing space to be added

struct sk_buff ** trailer Returned pointer to the skb where the **tailbits** space begins

Description

Make sure that the data buffers attached to a socket buffer are writable. If they are not, private copies are made of the data buffers and the socket buffer is set to use these instead.

If **tailbits** is given, make sure that there is space to write **tailbits** bytes of data beyond current end of socket buffer. **trailer** will be set to point to the skb in which this space begins.

The number of scatterlist elements required to completely map the COW'd and extended socket buffer will be returned.

struct *sk_buff* * **skb_clone_sk**(struct *sk_buff* * *skb*)
create clone of skb, and take reference to socket

Parameters

struct sk_buff * skb the skb to clone

Description

This function creates a clone of a buffer that holds a reference on sk_refcnt. Buffers created via this function are meant to be returned using sock_queue_err_skb, or free via kfree_skb.

When passing buffers allocated with this function to sock_queue_err_skb it is necessary to wrap the call with sock_hold/sock_put in order to prevent the socket from being released prior to being enqueued on the sk_error_queue.

bool **skb_partial_csum_set**(struct *sk_buff* * *skb*, u16 *start*, u16 *off*)
set up and verify partial csum values for packet

Parameters

struct sk_buff * skb the skb to set

u16 start the number of bytes after skb->data to start checksumming.

u16 off the offset from start to place the checksum.

Description

For untrusted partially-checksummed packets, we need to make sure the values for skb->csum_start and skb->csum_offset are valid so we don't oops.

This function checks and sets those values and skb->ip_summed: if this returns false you should drop the packet.

int **skb_checksum_setup**(struct *sk_buff* * *skb*, bool *recalculate*)
set up partial checksum offset

Parameters

struct sk_buff * skb the skb to set up

bool recalculate if true the pseudo-header checksum will be recalculated

struct *sk_buff* * **skb_checksum_trimmed**(struct *sk_buff* * *skb*, unsigned int *transport_len*, __sum16(*skb_chkf) (struct *sk_buff* **skb*)
validate checksum of an skb

Parameters

struct sk_buff * skb the skb to check

unsigned int transport_len the data length beyond the network header

__sum16(*)(struct sk_buff *skb) skb_chkf checksum function to use

Description

Applies the given checksum function skb_chkf to the provided skb. Returns a checked and maybe trimmed skb. Returns NULL on error.

If the skb has data beyond the given transport length, then a trimmed & cloned skb is checked and returned.

Caller needs to set the skb transport header and free any returned skb if it differs from the provided skb.

bool **skb_try_coalesce**(struct *sk_buff* * *to*, struct *sk_buff* * *from*, bool * *fragstolen*, int * *delta_truesize*)
try to merge skb to prior one

Parameters

struct sk_buff * to prior buffer

struct sk_buff * from buffer to add

bool * fragstolen pointer to boolean

int * delta_truesize how much more was allocated than was requested

void **skb_scrub_packet**(struct *sk_buff* * *skb*, bool *xnet*)
scrub an skb

Parameters

struct sk_buff * skb buffer to clean

bool xnet packet is crossing netns

Description

skb_scrub_packet can be used after encapsulating or decapsulting a packet into/from a tunnel. Some information have to be cleared during these operations. skb_scrub_packet can also be used to clean a skb before injecting it in another namespace (**xnet** == true). We have to clear all information in the skb that could impact namespace isolation.

bool **skb_gso_validate_network_len**(const struct *sk_buff* * *skb*, unsigned int *mtu*)
Will a split GSO skb fit into a given MTU?

Parameters

const struct sk_buff * skb GSO skb

unsigned int mtu MTU to validate against

Description

skb_gso_validate_network_len validates if a given skb will fit a wanted MTU once split. It considers L3 headers, L4 headers, and the payload.

bool **skb_gso_validate_mac_len**(const struct *sk_buff* * *skb*, unsigned int *len*)
Will a split GSO skb fit in a given length?

Parameters

const struct sk_buff * skb GSO skb

unsigned int len length to validate against

Description

skb_gso_validate_mac_len validates if a given skb will fit a wanted length once split, including L2, L3 and L4 headers and the payload.

struct *sk_buff* * **alloc_skb_with_frags**(unsigned long *header_len*, unsigned long *data_len*, int *max_page_order*, int * *errcode*, gfp_t *gfp_mask*)
allocate skb with page frags

Parameters

unsigned long header_len size of linear part

unsigned long data_len needed length in frags

int max_page_order max page order desired.

int * errcode pointer to error code if any

gfp_t gfp_mask allocation mask

Description

This can be used to allocate a paged skb, given a maximal order for frags.

bool **sk_ns_capable**(const struct *sock* * *sk*, struct user_namespace * *user_ns*, int *cap*)
General socket capability test

Parameters

const struct sock * sk Socket to use a capability on or through

struct user_namespace * user_ns The user namespace of the capability to use

int cap The capability to use

Description

Test to see if the opener of the socket had when the socket was created and the current process has the capability **cap** in the user namespace **user_ns**.

bool **sk_capable**(const struct *sock* * *sk*, int *cap*)
Socket global capability test

Parameters

const struct sock * sk Socket to use a capability on or through

int cap The global capability to use

Description

Test to see if the opener of the socket had when the socket was created and the current process has the capability **cap** in all user namespaces.

bool **sk_net_capable**(const struct *sock* * *sk*, int *cap*)
Network namespace socket capability test

Parameters

const struct sock * sk Socket to use a capability on or through

int cap The capability to use

Description

Test to see if the opener of the socket had when the socket was created and the current process has the capability **cap** over the network namespace the socket is a member of.

void **sk_set_memalloc**(struct *sock* * *sk*)
sets `SOCK_MEMALLOC`

Parameters

struct sock * sk socket to set it on

Description

Set `SOCK_MEMALLOC` on a socket for access to emergency reserves. It's the responsibility of the admin to adjust min_free_kbytes to meet the requirements

struct *sock* * **sk_alloc**(struct net * *net*, int *family*, gfp_t *priority*, struct proto * *prot*, int *kern*)
All socket objects are allocated here

Parameters

struct net * net the applicable net namespace

int family protocol family

gfp_t priority for allocation (`GFP_KERNEL`, `GFP_ATOMIC`, etc)

struct proto * prot struct proto associated with this new sock instance

int kern is this to be a kernel socket?

struct *sock* * **sk_clone_lock**(const struct *sock* * *sk*, const gfp_t *priority*)
clone a socket, and lock its clone

Parameters

const struct sock * sk the socket to clone

const gfp_t priority for allocation (GFP_KERNEL, GFP_ATOMIC, etc)

Description

Caller must unlock socket even in error path (bh_unlock_sock(newsk))

bool **skb_page_frag_refill**(unsigned int *sz*, struct page_frag * *pfrag*, gfp_t *gfp*)
check that a page_frag contains enough room

Parameters

unsigned int sz minimum size of the fragment we want to get

struct page_frag * pfrag pointer to page_frag

gfp_t gfp priority for memory allocation

Note

While this allocator tries to use high order pages, there is no guarantee that allocations succeed. Therefore, **sz** MUST be less or equal than PAGE_SIZE.

int **sk_wait_data**(struct *sock* * *sk*, long * *timeo*, const struct *sk_buff* * *skb*)
wait for data to arrive at sk_receive_queue

Parameters

struct sock * sk sock to wait on

long * timeo for how long

const struct sk_buff * skb last skb seen on sk_receive_queue

Description

Now socket state including sk->sk_err is changed only under lock, hence we may omit checks after joining wait queue. We check receive queue before `schedule()` only as optimization; it is very likely that `release_sock()` added new data.

int **__sk_mem_raise_allocated**(struct *sock* * *sk*, int *size*, int *amt*, int *kind*)
increase memory_allocated

Parameters

struct sock * sk socket

int size memory size to allocate

int amt pages to allocate

int kind allocation type

Description

Similar to *__sk_mem_schedule()*, but does not update sk_forward_alloc

int **__sk_mem_schedule**(struct *sock* * *sk*, int *size*, int *kind*)
increase sk_forward_alloc and memory_allocated

Parameters

struct sock * sk socket

int size memory size to allocate

int kind allocation type

Description

If kind is SK_MEM_SEND, it means wmem allocation. Otherwise it means rmem allocation. This function assumes that protocols which have memory_pressure use sk_wmem_queued as write buffer accounting.

void **__sk_mem_reduce_allocated**(struct *sock* * *sk*, int *amount*)
reclaim memory_allocated

Parameters

struct sock * sk socket

int amount number of quanta

Description

Similar to *__sk_mem_reclaim()*, but does not update sk_forward_alloc

void **__sk_mem_reclaim**(struct *sock* * *sk*, int *amount*)
reclaim sk_forward_alloc and memory_allocated

Parameters

struct sock * sk socket

int amount number of bytes (rounded down to a SK_MEM_QUANTUM multiple)

bool **lock_sock_fast**(struct *sock* * *sk*)
fast version of lock_sock

Parameters

struct sock * sk socket

Description

This version should be used for very small section, where process wont block return false if fast path is taken:

sk_lock.slock locked, owned = 0, BH disabled

return true if slow path is taken:

sk_lock.slock unlocked, owned = 1, BH enabled

struct *sk_buff* * **__skb_try_recv_datagram**(struct *sock* * *sk*, unsigned int *flags*, void (*destructor) (struct *sock* **sk*, struct *sk_buff* **skb*, int * *peeked*, int * *off*, int * *err*, struct *sk_buff* ** *last*)
Receive a datagram skbuff

Parameters

struct sock * sk socket

unsigned int flags MSG_ flags

void (*)(struct sock *sk, struct sk_buff *skb) destructor invoked under the receive lock on successful dequeue

int * peeked returns non-zero if this packet has been seen before

int * off an offset in bytes to peek skb from. Returns an offset within an skb where data actually starts

int * err error code returned

struct sk_buff ** last set to last peeked message to inform the wait function what to look for when peeking

Description

Get a datagram skbuff, understands the peeking, nonblocking wakeups and possible races. This replaces identical code in packet, raw and udp, as well as the IPX AX.25 and Appletalk. It also finally fixes the long standing peek and read race for datagram sockets. If you alter this routine remember it must be re-entrant.

This function will lock the socket if a skb is returned, so the caller needs to unlock the socket in that case (usually by calling skb_free_datagram). Returns NULL with **err** set to -EAGAIN if no data was available or to some other value if an error was detected.

- It does not lock socket since today. This function is
- free of race conditions. This measure should/can improve
- significantly datagram socket latencies at high loads,
- when data copying to user space takes lots of time.
- (BTW I've just killed the last `cli()` in IP/IPv6/core/netlink/packet
- 8. Great win.)
- –ANK (980729)

The order of the tests when we find no data waiting are specified quite explicitly by POSIX 1003.1g, don't change them without having the standard around please.

int **skb_kill_datagram**(struct *sock* * *sk*, struct *sk_buff* * *skb*, unsigned int *flags*)
Free a datagram skbuff forcibly

Parameters

struct sock * sk socket

struct sk_buff * skb datagram skbuff

unsigned int flags MSG_ flags

Description

This function frees a datagram skbuff that was received by skb_recv_datagram. The flags argument must match the one used for skb_recv_datagram.

If the MSG_PEEK flag is set, and the packet is still on the receive queue of the socket, it will be taken off the queue before it is freed.

This function currently only disables BH when acquiring the sk_receive_queue lock. Therefore it must not be used in a context where that lock is acquired in an IRQ context.

It returns 0 if the packet was removed by us.

int **skb_copy_datagram_iter**(const struct *sk_buff* * *skb*, int *offset*, struct iov_iter * *to*, int *len*)
Copy a datagram to an iovec iterator.

Parameters

const struct sk_buff * skb buffer to copy

int offset offset in the buffer to start copying from

struct iov_iter * to iovec iterator to copy to

int len amount of data to copy from buffer to iovec

int **skb_copy_datagram_from_iter**(struct *sk_buff* * *skb*, int *offset*, struct iov_iter * *from*, int *len*)
Copy a datagram from an iov_iter.

Parameters

struct sk_buff * skb buffer to copy

int offset offset in the buffer to start copying to

struct iov_iter * from the copy source

int len amount of data to copy to buffer from iovec

Description

Returns 0 or -EFAULT.

int **zerocopy_sg_from_iter**(struct *sk_buff* * *skb*, struct iov_iter * *from*)
Build a zerocopy datagram from an iov_iter

Parameters

struct sk_buff * skb buffer to copy

struct iov_iter * from the source to copy from

Description

The function will first copy up to headlen, and then pin the userspace pages and build frags through them.

Returns 0, -EFAULT or -EMSGSIZE.

int **skb_copy_and_csum_datagram_msg**(struct *sk_buff* * *skb*, int *hlen*, struct msghdr * *msg*)
Copy and checksum skb to user iovec.

Parameters

struct sk_buff * skb skbuff

int hlen hardware length

struct msghdr * msg destination

Description

Caller _must_ check that skb will fit to this iovec.

Return

0 - success. -EINVAL - checksum failure. -EFAULT - fault during copy.

__poll_t **datagram_poll**(struct file * *file*, struct *socket* * *sock*, poll_table * *wait*)
generic datagram poll

Parameters

struct file * file file struct

struct socket * sock socket

poll_table * wait poll table

Description

Datagram poll: Again totally generic. This also handles sequenced packet sockets providing the socket receive queue is only ever holding data ready to receive.

Note

when you *don't* use this routine for this protocol, and you use a different write policy from `sock_writeable()` then please supply your own write_space callback.

int **sk_stream_wait_connect**(struct *sock* * *sk*, long * *timeo_p*)
Wait for a socket to get into the connected state

Parameters

struct sock * sk sock to wait on

long * timeo_p for how long to wait

Description

Must be called with the socket locked.

int **sk_stream_wait_memory**(struct *sock* * *sk*, long * *timeo_p*)
Wait for more memory for a socket

Parameters

struct sock * sk socket to wait for memory

long * timeo_p for how long

Socket Filter

int **sk_filter_trim_cap**(struct *sock* * *sk*, struct *sk_buff* * *skb*, unsigned int *cap*)
run a packet through a socket filter

Parameters

struct sock * sk sock associated with *sk_buff*

struct sk_buff * skb buffer to filter

unsigned int cap limit on how short the eBPF program may trim the packet

Description

Run the eBPF program and then cut skb->data to correct size returned by the program. If pkt_len is 0 we toss packet. If skb->len is smaller than pkt_len we keep whole skb->data. This is the socket level wrapper to BPF_PROG_RUN. It returns 0 if the packet should be accepted or -EPERM if the packet should be tossed.

int **bpf_prog_create**(struct bpf_prog ** *pfp*, struct sock_fprog_kern * *fprog*)
create an unattached filter

Parameters

struct bpf_prog ** pfp the unattached filter that is created

struct sock_fprog_kern * fprog the filter program

Description

Create a filter independent of any socket. We first run some sanity checks on it to make sure it does not explode on us later. If an error occurs or there is insufficient memory for the filter a negative errno code is returned. On success the return is zero.

int **bpf_prog_create_from_user**(struct bpf_prog ** *pfp*, struct sock_fprog * *fprog*, bpf_aux_classic_check_t *trans*, bool *save_orig*)
create an unattached filter from user buffer

Parameters

struct bpf_prog ** pfp the unattached filter that is created

struct sock_fprog * fprog the filter program

bpf_aux_classic_check_t trans post-classic verifier transformation handler

bool save_orig save classic BPF program

Description

This function effectively does the same as *bpf_prog_create()*, only that it builds up its insns buffer from user space provided buffer. It also allows for passing a bpf_aux_classic_check_t handler.

int **sk_attach_filter**(struct sock_fprog * *fprog*, struct *sock* * *sk*)
attach a socket filter

Parameters

struct sock_fprog * fprog the filter program

struct sock * sk the socket to use

Description

Attach the user's filter code. We first run some sanity checks on it to make sure it does not explode on us later. If an error occurs or there is insufficient memory for the filter a negative errno code is returned. On success the return is zero.

Generic Network Statistics

struct **gnet_stats_basic**
byte/packet throughput statistics

Definition

```
struct gnet_stats_basic {
  __u64 bytes;
  __u32 packets;
};
```

Members

bytes number of seen bytes

packets number of seen packets

struct **gnet_stats_rate_est**
rate estimator

Definition

```
struct gnet_stats_rate_est {
  __u32 bps;
  __u32 pps;
};
```

Members

bps current byte rate

pps current packet rate

struct **gnet_stats_rate_est64**
rate estimator

Definition

```
struct gnet_stats_rate_est64 {
  __u64 bps;
  __u64 pps;
};
```

Members

bps current byte rate

pps current packet rate

struct **gnet_stats_queue**
queuing statistics

Definition

```
struct gnet_stats_queue {
  __u32 qlen;
  __u32 backlog;
  __u32 drops;
  __u32 requeues;
  __u32 overlimits;
};
```

Members

qlen queue length

backlog backlog size of queue

drops number of dropped packets

requeues number of requeues

overlimits number of enqueues over the limit

struct **gnet_estimator**
rate estimator configuration

Definition

```
struct gnet_estimator {
  signed char     interval;
  unsigned char   ewma_log;
};
```

Members

interval sampling period

ewma_log the log of measurement window weight

int **gnet_stats_start_copy_compat**(struct *sk_buff* * *skb*, int *type*, int *tc_stats_type*, int *xstats_type*, spinlock_t * *lock*, struct gnet_dump * *d*, int *padattr*)
start dumping procedure in compatibility mode

Parameters

struct sk_buff * skb socket buffer to put statistics TLVs into

int type TLV type for top level statistic TLV

int tc_stats_type TLV type for backward compatibility struct tc_stats TLV

int xstats_type TLV type for backward compatibility xstats TLV

spinlock_t * lock statistics lock

struct gnet_dump * d dumping handle

int padattr padding attribute

Description

Initializes the dumping handle, grabs the statistic lock and appends an empty TLV header to the socket buffer for use a container for all other statistic TLVS.

The dumping handle is marked to be in backward compatibility mode telling all `gnet_stats_copy_XXX()` functions to fill a local copy of struct tc_stats.

Returns 0 on success or -1 if the room in the socket buffer was not sufficient.

int **gnet_stats_start_copy**(struct *sk_buff* * *skb*, int *type*, spinlock_t * *lock*, struct gnet_dump * *d*, int *padattr*)
start dumping procedure in compatibility mode

Parameters

struct sk_buff * skb socket buffer to put statistics TLVs into

int type TLV type for top level statistic TLV

spinlock_t * lock statistics lock

struct gnet_dump * d dumping handle

int padattr padding attribute

Description

Initializes the dumping handle, grabs the statistic lock and appends an empty TLV header to the socket buffer for use a container for all other statistic TLVS.

Returns 0 on success or -1 if the room in the socket buffer was not sufficient.

int **gnet_stats_copy_basic**(const seqcount_t * *running*, struct gnet_dump * *d*, struct gnet_stats_basic_cpu __percpu * *cpu*, struct gnet_stats_basic_packed * *b*)
copy basic statistics into statistic TLV

Parameters

const seqcount_t * running seqcount_t pointer

struct gnet_dump * d dumping handle

struct gnet_stats_basic_cpu __percpu * cpu copy statistic per cpu

struct gnet_stats_basic_packed * b basic statistics

Description

Appends the basic statistics to the top level TLV created by *gnet_stats_start_copy()*.

Returns 0 on success or -1 with the statistic lock released if the room in the socket buffer was not sufficient.

int **gnet_stats_copy_rate_est**(struct gnet_dump * *d*, struct net_rate_estimator __rcu ** *rate_est*)
copy rate estimator statistics into statistics TLV

Parameters

struct gnet_dump * d dumping handle

struct net_rate_estimator __rcu ** rate_est rate estimator

Description

Appends the rate estimator statistics to the top level TLV created by *gnet_stats_start_copy()*.

Returns 0 on success or -1 with the statistic lock released if the room in the socket buffer was not sufficient.

int **gnet_stats_copy_queue**(struct gnet_dump * *d*, struct *gnet_stats_queue* __percpu * *cpu_q*, struct *gnet_stats_queue* * *q*, __u32 *qlen*)
copy queue statistics into statistics TLV

Parameters

struct gnet_dump * d dumping handle

struct gnet_stats_queue __percpu * cpu_q per cpu queue statistics

struct gnet_stats_queue * q queue statistics

__u32 qlen queue length statistics

Description

Appends the queue statistics to the top level TLV created by *gnet_stats_start_copy()*. Using per cpu queue statistics if they are available.

Returns 0 on success or -1 with the statistic lock released if the room in the socket buffer was not sufficient.

int **gnet_stats_copy_app**(struct gnet_dump * *d*, void * *st*, int *len*)
copy application specific statistics into statistics TLV

Parameters

struct gnet_dump * d dumping handle

void * st application specific statistics data

int len length of data

Description

Appends the application specific statistics to the top level TLV created by *gnet_stats_start_copy()* and remembers the data for XSTATS if the dumping handle is in backward compatibility mode.

Returns 0 on success or -1 with the statistic lock released if the room in the socket buffer was not sufficient.

int **gnet_stats_finish_copy**(struct gnet_dump * *d*)
finish dumping procedure

Parameters

struct gnet_dump * d dumping handle

Description

Corrects the length of the top level TLV to include all TLVs added by `gnet_stats_copy_XXX()` calls. Adds the backward compatibility TLVs if *gnet_stats_start_copy_compat()* was used and releases the statistics lock.

Returns 0 on success or -1 with the statistic lock released if the room in the socket buffer was not sufficient.

int **gen_new_estimator**(struct gnet_stats_basic_packed * *bstats*, struct gnet_stats_basic_cpu __percpu * *cpu_bstats*, struct net_rate_estimator __rcu ** *rate_est*, spinlock_t * *stats_lock*, seqcount_t * *running*, struct nlattr * *opt*)
create a new rate estimator

Parameters

struct gnet_stats_basic_packed * bstats basic statistics

struct gnet_stats_basic_cpu __percpu * cpu_bstats bstats per cpu

struct net_rate_estimator __rcu ** rate_est rate estimator statistics

spinlock_t * stats_lock statistics lock

seqcount_t * running qdisc running seqcount

struct nlattr * opt rate estimator configuration TLV

Description

Creates a new rate estimator with `bstats` as source and `rate_est` as destination. A new timer with the interval specified in the configuration TLV is created. Upon each interval, the latest statistics will be read from `bstats` and the estimated rate will be stored in `rate_est` with the statistics lock grabbed during this period.

Returns 0 on success or a negative error code.

void **gen_kill_estimator**(struct net_rate_estimator __rcu ** *rate_est*)
remove a rate estimator

Parameters

struct net_rate_estimator __rcu ** rate_est rate estimator

Description

Removes the rate estimator.

int **gen_replace_estimator**(struct gnet_stats_basic_packed * *bstats*, struct gnet_stats_basic_cpu __percpu * *cpu_bstats*, struct net_rate_estimator __rcu ** *rate_est*, spinlock_t * *stats_lock*, seqcount_t * *running*, struct nlattr * *opt*)
replace rate estimator configuration

Parameters

struct gnet_stats_basic_packed * bstats basic statistics

struct gnet_stats_basic_cpu __percpu * cpu_bstats bstats per cpu

struct net_rate_estimator __rcu ** rate_est rate estimator statistics

spinlock_t * stats_lock statistics lock

seqcount_t * running qdisc running seqcount (might be NULL)

struct nlattr * opt rate estimator configuration TLV

Description

Replaces the configuration of a rate estimator by calling *gen_kill_estimator()* and *gen_new_estimator()*.

Returns 0 on success or a negative error code.

bool **gen_estimator_active**(struct net_rate_estimator __rcu ** *rate_est*)
test if estimator is currently in use

Parameters

struct net_rate_estimator __rcu ** rate_est rate estimator

Description

Returns true if estimator is active, and false if not.

SUN RPC subsystem

__be32 * **xdr_encode_opaque_fixed**(__be32 * *p*, const void * *ptr*, unsigned int *nbytes*)
Encode fixed length opaque data

Parameters

__be32 * p pointer to current position in XDR buffer.

const void * ptr pointer to data to encode (or NULL)

unsigned int nbytes size of data.

Description

Copy the array of data of length nbytes at ptr to the XDR buffer at position p, then align to the next 32-bit boundary by padding with zero bytes (see RFC1832).

Note

if ptr is NULL, only the padding is performed.

Returns the updated current XDR buffer position

__be32 * **xdr_encode_opaque**(__be32 * *p*, const void * *ptr*, unsigned int *nbytes*)
Encode variable length opaque data

Parameters

__be32 * p pointer to current position in XDR buffer.

const void * ptr pointer to data to encode (or NULL)

unsigned int nbytes size of data.

Description

Returns the updated current XDR buffer position

void **xdr_terminate_string**(struct xdr_buf * *buf*, const u32 *len*)
'0'-terminate a string residing in an xdr_buf

Parameters

struct xdr_buf * buf XDR buffer where string resides

const u32 len length of string, in bytes

void **_copy_from_pages**(char * *p*, struct page ** *pages*, size_t *pgbase*, size_t *len*)

Parameters

char * p pointer to destination

struct page ** pages array of pages

size_t pgbase offset of source data

size_t len length

Description

Copies data into an arbitrary memory location from an array of pages The copy is assumed to be non-overlapping.

unsigned int **xdr_stream_pos**(const struct xdr_stream * *xdr*)
Return the current offset from the start of the xdr_stream

Parameters

const struct xdr_stream * xdr pointer to struct xdr_stream

void **xdr_init_encode**(struct xdr_stream * *xdr*, struct xdr_buf * *buf*, __be32 * *p*)
Initialize a struct xdr_stream for sending data.

Parameters

struct xdr_stream * xdr pointer to xdr_stream struct

struct xdr_buf * buf pointer to XDR buffer in which to encode data

__be32 * p current pointer inside XDR buffer

Note

at the moment the RPC client only passes the length of our scratch buffer in the xdr_buf's header kvec. Previously this meant we needed to call `xdr_adjust_iovec()` after encoding the data. With the new scheme, the xdr_stream manages the details of the buffer length, and takes care of adjusting the kvec length for us.

void **xdr_commit_encode**(struct xdr_stream * *xdr*)
Ensure all data is written to buffer

Parameters

struct xdr_stream * xdr pointer to xdr_stream

Description

We handle encoding across page boundaries by giving the caller a temporary location to write to, then later copying the data into place; xdr_commit_encode does that copying.

Normally the caller doesn't need to call this directly, as the following xdr_reserve_space will do it. But an explicit call may be required at the end of encoding, or any other time when the xdr_buf data might be read.

__be32 * **xdr_reserve_space**(struct xdr_stream * *xdr*, size_t *nbytes*)
Reserve buffer space for sending

Parameters

struct xdr_stream * xdr pointer to xdr_stream

size_t nbytes number of bytes to reserve

Description

Checks that we have enough buffer space to encode 'nbytes' more bytes of data. If so, update the total xdr_buf length, and adjust the length of the current kvec.

void **xdr_truncate_encode**(struct xdr_stream * *xdr*, size_t *len*)
truncate an encode buffer

Parameters

struct xdr_stream * xdr pointer to xdr_stream

size_t len new length of buffer

Description

Truncates the xdr stream, so that xdr->buf->len == len, and xdr->p points at offset len from the start of the buffer, and head, tail, and page lengths are adjusted to correspond.

If this means moving xdr->p to a different buffer, we assume that that the end pointer should be set to the end of the current page, except in the case of the head buffer when we assume the head buffer's current length represents the end of the available buffer.

This is *not* safe to use on a buffer that already has inlined page cache pages (as in a zero-copy server read reply), except for the simple case of truncating from one position in the tail to another.

int **xdr_restrict_buflen**(struct xdr_stream * *xdr*, int *newbuflen*)
decrease available buffer space

Parameters

struct xdr_stream * xdr pointer to xdr_stream

int newbuflen new maximum number of bytes available

Description

Adjust our idea of how much space is available in the buffer. If we've already used too much space in the buffer, returns -1. If the available space is already smaller than newbuflen, returns 0 and does nothing. Otherwise, adjusts xdr->buf->buflen to newbuflen and ensures xdr->end is set at most offset newbuflen from the start of the buffer.

void **xdr_write_pages**(struct xdr_stream * *xdr*, struct page ** *pages*, unsigned int *base*, unsigned int *len*)
Insert a list of pages into an XDR buffer for sending

Parameters

struct xdr_stream * xdr pointer to xdr_stream

struct page ** pages list of pages

unsigned int base offset of first byte

unsigned int len length of data in bytes

void **xdr_init_decode**(struct xdr_stream * *xdr*, struct xdr_buf * *buf*, __be32 * *p*)
Initialize an xdr_stream for decoding data.

Parameters

struct xdr_stream * xdr pointer to xdr_stream struct

struct xdr_buf * buf pointer to XDR buffer from which to decode data

__be32 * p current pointer inside XDR buffer

void **xdr_init_decode_pages**(struct xdr_stream * *xdr*, struct xdr_buf * *buf*, struct page ** *pages*, unsigned int *len*)
Initialize an xdr_stream for decoding into pages

Parameters

struct xdr_stream * xdr pointer to xdr_stream struct

struct xdr_buf * buf pointer to XDR buffer from which to decode data

struct page ** pages list of pages to decode into

unsigned int len length in bytes of buffer in pages

void **xdr_set_scratch_buffer**(struct xdr_stream * *xdr*, void * *buf*, size_t *buflen*)
Attach a scratch buffer for decoding data.

Parameters

struct xdr_stream * xdr pointer to xdr_stream struct

void * buf pointer to an empty buffer

size_t buflen size of 'buf'

Description

The scratch buffer is used when decoding from an array of pages. If an *xdr_inline_decode()* call spans across page boundaries, then we copy the data into the scratch buffer in order to allow linear access.

__be32 * **xdr_inline_decode**(struct xdr_stream * *xdr*, size_t *nbytes*)
Retrieve XDR data to decode

Parameters

struct xdr_stream * xdr pointer to xdr_stream struct

size_t nbytes number of bytes of data to decode

Description

Check if the input buffer is long enough to enable us to decode 'nbytes' more bytes of data starting at the current position. If so return the current pointer, then update the current pointer position.

unsigned int **xdr_read_pages**(struct xdr_stream * *xdr*, unsigned int *len*)
Ensure page-based XDR data to decode is aligned at current pointer position

Parameters

struct xdr_stream * xdr pointer to xdr_stream struct

unsigned int len number of bytes of page data

Description

Moves data beyond the current pointer position from the XDR head[] buffer into the page list. Any data that lies beyond current position + "len" bytes is moved into the XDR tail[].

Returns the number of XDR encoded bytes now contained in the pages

void **xdr_enter_page**(struct xdr_stream * *xdr*, unsigned int *len*)
decode data from the XDR page

Parameters

struct xdr_stream * xdr pointer to xdr_stream struct

unsigned int len number of bytes of page data

Description

Moves data beyond the current pointer position from the XDR head[] buffer into the page list. Any data that lies beyond current position + "len" bytes is moved into the XDR tail[]. The current pointer is then repositioned at the beginning of the first XDR page.

int **xdr_buf_subsegment**(struct xdr_buf * *buf*, struct xdr_buf * *subbuf*, unsigned int *base*, unsigned int *len*)
set subbuf to a portion of buf

Parameters

struct xdr_buf * buf an xdr buffer

struct xdr_buf * subbuf the result buffer

unsigned int base beginning of range in bytes

unsigned int len length of range in bytes

Description

sets **subbuf** to an xdr buffer representing the portion of **buf** of length **len** starting at offset **base**.

buf and **subbuf** may be pointers to the same struct xdr_buf.

Returns -1 if base of length are out of bounds.

void **xdr_buf_trim**(struct xdr_buf * *buf*, unsigned int *len*)
lop at most "len" bytes off the end of "buf"

Parameters

struct xdr_buf * buf buf to be trimmed

unsigned int len number of bytes to reduce "buf" by

Description

Trim an xdr_buf by the given number of bytes by fixing up the lengths. Note that it's possible that we'll trim less than that amount if the xdr_buf is too small, or if (for instance) it's all in the head and the parser has already read too far into it.

ssize_t **xdr_stream_decode_string_dup**(struct xdr_stream * *xdr*, char ** *str*, size_t *maxlen*, gfp_t *gfp_flags*)
Decode and duplicate variable length string

Parameters

struct xdr_stream * xdr pointer to xdr_stream

char ** str location to store pointer to string

size_t maxlen maximum acceptable string length

gfp_t gfp_flags GFP mask to use

Description

Return values: On success, returns length of NUL-terminated string stored in ***ptr** `-EBADMSG` on XDR buffer overflow `-EMSGSIZE` if the size of the string would exceed **maxlen** `-ENOMEM` on memory allocation failure

char * **svc_print_addr**(struct svc_rqst * *rqstp*, char * *buf*, size_t *len*)
Format rq_addr field for printing

Parameters

struct svc_rqst * rqstp svc_rqst struct containing address to print

char * buf target buffer for formatted address

size_t len length of target buffer

void **svc_reserve**(struct svc_rqst * *rqstp*, int *space*)
change the space reserved for the reply to a request.

Parameters

struct svc_rqst * rqstp The request in question

int space new max space to reserve

Description

Each request reserves some space on the output queue of the transport to make sure the reply fits. This function reduces that reserved space to be the amount of space used already, plus **space**.

struct svc_xprt * **svc_find_xprt**(struct svc_serv * *serv*, const char * *xcl_name*, struct net * *net*, const sa_family_t *af*, const unsigned short *port*)
find an RPC transport instance

Parameters

struct svc_serv * serv pointer to svc_serv to search

const char * xcl_name C string containing transport's class name

struct net * net owner net pointer

const sa_family_t af Address family of transport's local address

const unsigned short port transport's IP port number

Description

Return the transport instance pointer for the endpoint accepting connections/peer traffic from the specified transport class, address family and port.

Specifying 0 for the address family or port is effectively a wild-card, and will result in matching the first transport in the service's list that has a matching class name.

int **svc_xprt_names**(struct svc_serv * *serv*, char * *buf*, const int *buflen*)
format a buffer with a list of transport names

Parameters

struct svc_serv * serv pointer to an RPC service

char * buf pointer to a buffer to be filled in

const int buflen length of buffer to be filled in

Description

Fills in **buf** with a string containing a list of transport names, each name terminated with 'n'.

Returns positive length of the filled-in string on success; otherwise a negative errno value is returned if an error occurs.

int **xprt_register_transport**(struct xprt_class * *transport*)
register a transport implementation

Parameters

struct xprt_class * transport transport to register

Description

If a transport implementation is loaded as a kernel module, it can call this interface to make itself known to the RPC client.

Return

0: transport successfully registered -EEXIST: transport already registered -EINVAL: transport module being unloaded

int **xprt_unregister_transport**(struct xprt_class * *transport*)
unregister a transport implementation

Parameters

struct xprt_class * transport transport to unregister

Return

0: transport successfully unregistered -ENOENT: transport never registered

int **xprt_load_transport**(const char * *transport_name*)
load a transport implementation

Parameters

const char * transport_name transport to load

Return

0: transport successfully loaded -ENOENT: transport module not available

int **xprt_reserve_xprt**(struct rpc_xprt * *xprt*, struct rpc_task * *task*)
serialize write access to transports

Parameters

struct rpc_xprt * xprt pointer to the target transport

struct rpc_task * task task that is requesting access to the transport

Description

This prevents mixing the payload of separate requests, and prevents transport connects from colliding with writes. No congestion control is provided.

void **xprt_release_xprt**(struct rpc_xprt * *xprt*, struct rpc_task * *task*)
allow other requests to use a transport

Parameters

struct rpc_xprt * xprt transport with other tasks potentially waiting

struct rpc_task * task task that is releasing access to the transport

Description

Note that "task" can be NULL. No congestion control is provided.

void **xprt_release_xprt_cong**(struct rpc_xprt * *xprt*, struct rpc_task * *task*)
allow other requests to use a transport

Parameters

struct rpc_xprt * xprt transport with other tasks potentially waiting

struct rpc_task * task task that is releasing access to the transport

Description

Note that "task" can be NULL. Another task is awoken to use the transport if the transport's congestion window allows it.

void **xprt_release_rqst_cong**(struct rpc_task * *task*)
housekeeping when request is complete

Parameters

struct rpc_task * task RPC request that recently completed

Description

Useful for transports that require congestion control.

void **xprt_adjust_cwnd**(struct rpc_xprt * *xprt*, struct rpc_task * *task*, int *result*)
adjust transport congestion window

Parameters

struct rpc_xprt * xprt pointer to xprt

struct rpc_task * task recently completed RPC request used to adjust window

int result result code of completed RPC request

Description

The transport code maintains an estimate on the maximum number of out- standing RPC requests, using a smoothed version of the congestion avoidance implemented in 44BSD. This is basically the Van Jacobson congestion algorithm: If a retransmit occurs, the congestion window is halved; otherwise, it is incremented by 1/cwnd when

- a reply is received and
- a full number of requests are outstanding and
- the congestion window hasn't been updated recently.

void **xprt_wake_pending_tasks**(struct rpc_xprt * *xprt*, int *status*)
wake all tasks on a transport's pending queue

Parameters

struct rpc_xprt * xprt transport with waiting tasks

int status result code to plant in each task before waking it

void **xprt_wait_for_buffer_space**(struct rpc_task * *task*, rpc_action *action*)
wait for transport output buffer to clear

Parameters

struct rpc_task * task task to be put to sleep

rpc_action action function pointer to be executed after wait

Description

Note that we only set the timer for the case of `RPC_IS_SOFT()`, since we don't in general want to force a socket disconnection due to an incomplete RPC call transmission.

void **xprt_write_space**(struct rpc_xprt * *xprt*)
wake the task waiting for transport output buffer space

Parameters

struct rpc_xprt * xprt transport with waiting tasks

Description

Can be called in a soft IRQ context, so xprt_write_space never sleeps.

void **xprt_set_retrans_timeout_def**(struct rpc_task * *task*)
set a request's retransmit timeout

Parameters

struct rpc_task * task task whose timeout is to be set

Description

Set a request's retransmit timeout based on the transport's default timeout parameters. Used by transports that don't adjust the retransmit timeout based on round-trip time estimation.

void **xprt_set_retrans_timeout_rtt**(struct rpc_task * *task*)
set a request's retransmit timeout

Parameters

struct rpc_task * task task whose timeout is to be set

Description

Set a request's retransmit timeout using the RTT estimator.

void **xprt_disconnect_done**(struct rpc_xprt * *xprt*)
mark a transport as disconnected

Parameters

struct rpc_xprt * xprt transport to flag for disconnect

void **xprt_force_disconnect**(struct rpc_xprt * *xprt*)
force a transport to disconnect

Parameters

struct rpc_xprt * xprt transport to disconnect

struct rpc_rqst * **xprt_lookup_rqst**(struct rpc_xprt * *xprt*, __be32 *xid*)
find an RPC request corresponding to an XID

Parameters

struct rpc_xprt * xprt transport on which the original request was transmitted

__be32 xid RPC XID of incoming reply

void **xprt_pin_rqst**(struct rpc_rqst * *req*)
Pin a request on the transport receive list

Parameters

struct rpc_rqst * req Request to pin

Description

Caller must ensure this is atomic with the call to *xprt_lookup_rqst()* so should be holding the xprt transport lock.

void **xprt_unpin_rqst**(struct rpc_rqst * *req*)
Unpin a request on the transport receive list

Parameters

struct rpc_rqst * req Request to pin

Description

Caller should be holding the xprt transport lock.

void **xprt_complete_rqst**(struct rpc_task * *task*, int *copied*)
called when reply processing is complete

Parameters

struct rpc_task * task RPC request that recently completed

int copied actual number of bytes received from the transport

Description

Caller holds transport lock.

struct rpc_xprt * **xprt_get**(struct rpc_xprt * *xprt*)
return a reference to an RPC transport.

Parameters

struct rpc_xprt * xprt pointer to the transport

void **xprt_put**(struct rpc_xprt * *xprt*)
release a reference to an RPC transport.

Parameters

struct rpc_xprt * xprt pointer to the transport

void **rpc_wake_up**(struct rpc_wait_queue * *queue*)
wake up all rpc_tasks

Parameters

struct rpc_wait_queue * queue rpc_wait_queue on which the tasks are sleeping

Description

Grabs queue->lock

void **rpc_wake_up_status**(struct rpc_wait_queue * *queue*, int *status*)
wake up all rpc_tasks and set their status value.

Parameters

struct rpc_wait_queue * queue rpc_wait_queue on which the tasks are sleeping

int status status value to set

Description

Grabs queue->lock

int **rpc_malloc**(struct rpc_task * *task*)
allocate RPC buffer resources

Parameters

struct rpc_task * task RPC task

Description

A single memory region is allocated, which is split between the RPC call and RPC reply that this task is being used for. When this RPC is retired, the memory is released by calling rpc_free.

To prevent rpciod from hanging, this allocator never sleeps, returning -ENOMEM and suppressing warning if the request cannot be serviced immediately. The caller can arrange to sleep in a way that is safe for rpciod.

Most requests are 'small' (under 2KiB) and can be serviced from a mempool, ensuring that NFS reads and writes can always proceed, and that there is good locality of reference for these buffers.

In order to avoid memory starvation triggering more writebacks of NFS requests, we avoid using GFP_KERNEL.

void **rpc_free**(struct rpc_task * *task*)
free RPC buffer resources allocated via rpc_malloc

Parameters

struct rpc_task * task RPC task

size_t **xdr_skb_read_bits**(struct xdr_skb_reader * *desc*, void * *to*, size_t *len*)
copy some data bits from skb to internal buffer

Parameters

struct xdr_skb_reader * desc sk_buff copy helper

void * to copy destination

size_t len number of bytes to copy

Description

Possibly called several times to iterate over an sk_buff and copy data out of it.

ssize_t **xdr_partial_copy_from_skb**(struct xdr_buf * *xdr*, unsigned int *base*, struct xdr_skb_reader * *desc*, xdr_skb_read_actor *copy_actor*)
copy data out of an skb

Parameters

struct xdr_buf * xdr target XDR buffer

unsigned int base starting offset

struct xdr_skb_reader * desc sk_buff copy helper

xdr_skb_read_actor copy_actor virtual method for copying data

int **csum_partial_copy_to_xdr**(struct xdr_buf * *xdr*, struct sk_buff * *skb*)
checksum and copy data

Parameters

struct xdr_buf * xdr target XDR buffer

struct sk_buff * skb source skb

Description

We have set things up such that we perform the checksum of the UDP packet in parallel with the copies into the RPC client iovec. -DaveM

struct rpc_iostats * **rpc_alloc_iostats**(struct rpc_clnt * *clnt*)
allocate an rpc_iostats structure

Parameters

struct rpc_clnt * clnt RPC program, version, and xprt

void **rpc_free_iostats**(struct rpc_iostats * *stats*)
release an rpc_iostats structure

Parameters

struct rpc_iostats * stats doomed rpc_iostats structure

void **rpc_count_iostats_metrics**(const struct rpc_task * *task*, struct rpc_iostats * *op_metrics*)
tally up per-task stats

Parameters

const struct rpc_task * task completed rpc_task

struct rpc_iostats * op_metrics stat structure for OP that will accumulate stats from **task**

void **rpc_count_iostats**(const struct rpc_task * *task*, struct rpc_iostats * *stats*)
tally up per-task stats

Parameters

const struct rpc_task * task completed rpc_task

struct rpc_iostats * stats array of stat structures

Description

Uses the statidx from **task**

int **rpc_queue_upcall**(struct rpc_pipe * *pipe*, struct rpc_pipe_msg * *msg*)
queue an upcall message to userspace

Parameters

struct rpc_pipe * pipe upcall pipe on which to queue given message

struct rpc_pipe_msg * msg message to queue

Description

Call with an **inode** created by `rpc_mkpipe()` to queue an upcall. A userspace process may then later read the upcall by performing a read on an open file for this inode. It is up to the caller to initialize the fields of **msg** (other than **msg**->list) appropriately.

struct dentry * **rpc_mkpipe_dentry**(struct dentry * *parent*, const char * *name*, void * *private*, struct rpc_pipe * *pipe*)
make an rpc_pipefs file for kernel<->userspace communication

Parameters

struct dentry * parent dentry of directory to create new "pipe" in

const char * name name of pipe

void * private private data to associate with the pipe, for the caller's use

struct rpc_pipe * pipe `rpc_pipe` containing input parameters

Description

Data is made available for userspace to read by calls to *rpc_queue_upcall()*. The actual reads will result in calls to **ops**->upcall, which will be called with the file pointer, message, and userspace buffer to copy to.

Writes can come at any time, and do not necessarily have to be responses to upcalls. They will result in calls to **msg**->downcall.

The **private** argument passed here will be available to all these methods from the file pointer, via RPC_I(file_inode(file))->private.

int **rpc_unlink**(struct dentry * *dentry*)
remove a pipe

Parameters

struct dentry * dentry dentry for the pipe, as returned from rpc_mkpipe

Description

After this call, lookups will no longer find the pipe, and any attempts to read or write using preexisting opens of the pipe will return -EPIPE.

void **rpc_init_pipe_dir_head**(struct rpc_pipe_dir_head * *pdh*)
initialise a struct rpc_pipe_dir_head

Parameters

struct rpc_pipe_dir_head * pdh pointer to struct rpc_pipe_dir_head

void **rpc_init_pipe_dir_object**(struct rpc_pipe_dir_object * *pdo*, const struct rpc_pipe_dir_object_ops * *pdo_ops*, void * *pdo_data*)
initialise a struct rpc_pipe_dir_object

Parameters

struct rpc_pipe_dir_object * pdo pointer to struct rpc_pipe_dir_object

const struct rpc_pipe_dir_object_ops * pdo_ops pointer to const struct rpc_pipe_dir_object_ops

void * pdo_data pointer to caller-defined data

int **rpc_add_pipe_dir_object**(struct net * *net*, struct rpc_pipe_dir_head * *pdh*, struct rpc_pipe_dir_object * *pdo*)
associate a rpc_pipe_dir_object to a directory

Parameters

struct net * net pointer to struct net

struct rpc_pipe_dir_head * pdh pointer to struct rpc_pipe_dir_head

struct rpc_pipe_dir_object * pdo pointer to struct rpc_pipe_dir_object

void **rpc_remove_pipe_dir_object**(struct net * *net*, struct rpc_pipe_dir_head * *pdh*, struct rpc_pipe_dir_object * *pdo*)
remove a rpc_pipe_dir_object from a directory

Parameters

struct net * net pointer to struct net

struct rpc_pipe_dir_head * pdh pointer to struct rpc_pipe_dir_head

struct rpc_pipe_dir_object * pdo pointer to struct rpc_pipe_dir_object struct rpc_pipe_dir_object * **rpc_find_or_alloc_pipe_dir_object**(struct net * *net*, struct rpc_pipe_dir_head * *pdh*, int (*match) (struct rpc_pipe_dir_object *, void *, struct rpc_pipe_dir_object *(*alloc) (void *, void * *data*)

Parameters

struct net * net pointer to struct net

struct rpc_pipe_dir_head * pdh pointer to struct rpc_pipe_dir_head

int (*)(struct rpc_pipe_dir_object *, void *) match match struct rpc_pipe_dir_object to data

struct rpc_pipe_dir_object *(*)(void *) alloc allocate a new struct rpc_pipe_dir_object

void * data user defined data for `match()` and `alloc()`

void **rpcb_getport_async**(struct rpc_task * *task*)
obtain the port for a given RPC service on a given host

Parameters

struct rpc_task * task task that is waiting for portmapper request

Description

This one can be called for an ongoing RPC request, and can be used in an async (rpciod) context.

struct rpc_clnt * **rpc_create**(struct rpc_create_args * *args*)
create an RPC client and transport with one call

Parameters

struct rpc_create_args * args rpc_clnt create argument structure

Description

Creates and initializes an RPC transport and an RPC client.

It can ping the server in order to determine if it is up, and to see if it supports this program and version. RPC_CLNT_CREATE_NOPING disables this behavior so asynchronous tasks can also use rpc_create.

struct rpc_clnt * **rpc_clone_client**(struct rpc_clnt * *clnt*)
Clone an RPC client structure

Parameters

struct rpc_clnt * clnt RPC client whose parameters are copied

Description

Returns a fresh RPC client or an ERR_PTR.

struct rpc_clnt * **rpc_clone_client_set_auth**(struct rpc_clnt * *clnt*, rpc_authflavor_t *flavor*)
Clone an RPC client structure and set its auth

Parameters

struct rpc_clnt * clnt RPC client whose parameters are copied

rpc_authflavor_t flavor security flavor for new client

Description

Returns a fresh RPC client or an ERR_PTR.

int **rpc_switch_client_transport**(struct rpc_clnt * *clnt*, struct xprt_create * *args*, const struct rpc_timeout * *timeout*)

Parameters

struct rpc_clnt * clnt pointer to a struct rpc_clnt

struct xprt_create * args pointer to the new transport arguments

const struct rpc_timeout * timeout pointer to the new timeout parameters

Description

This function allows the caller to switch the RPC transport for the rpc_clnt structure 'clnt' to allow it to connect to a mirrored NFS server, for instance. It assumes that the caller has ensured that there are no active RPC tasks by using some form of locking.

Returns zero if "clnt" is now using the new xprt. Otherwise a negative errno is returned, and "clnt" continues to use the old xprt.

int **rpc_clnt_iterate_for_each_xprt**(struct rpc_clnt * *clnt*, int (*fn) (struct rpc_clnt *, struct rpc_xprt *, void *, void * *data*)
Apply a function to all transports

Parameters

struct rpc_clnt * clnt pointer to client

int (*)(struct rpc_clnt *, struct rpc_xprt *, void *) fn function to apply

void * data void pointer to function data

Description

Iterates through the list of RPC transports currently attached to the client and applies the function fn(clnt, xprt, data).

On error, the iteration stops, and the function returns the error value.

struct rpc_clnt * **rpc_bind_new_program**(struct rpc_clnt * *old*, const struct rpc_program * *program*, u32 *vers*)
bind a new RPC program to an existing client

Parameters

struct rpc_clnt * old old rpc_client

const struct rpc_program * program rpc program to set

u32 vers rpc program version

Description

Clones the rpc client and sets up a new RPC program. This is mainly of use for enabling different RPC programs to share the same transport. The Sun NFSv2/v3 ACL protocol can do this.

struct rpc_task * **rpc_run_task**(const struct rpc_task_setup * *task_setup_data*)
Allocate a new RPC task, then run rpc_execute against it

Parameters

const struct rpc_task_setup * task_setup_data pointer to task initialisation data

int **rpc_call_sync**(struct rpc_clnt * *clnt*, const struct rpc_message * *msg*, int *flags*)
Perform a synchronous RPC call

Parameters

struct rpc_clnt * clnt pointer to RPC client

const struct rpc_message * msg RPC call parameters

int flags RPC call flags

int **rpc_call_async**(struct rpc_clnt * *clnt*, const struct rpc_message * *msg*, int *flags*, const struct rpc_call_ops * *tk_ops*, void * *data*)
Perform an asynchronous RPC call

Parameters

struct rpc_clnt * clnt pointer to RPC client

const struct rpc_message * msg RPC call parameters

int flags RPC call flags

const struct rpc_call_ops * tk_ops RPC call ops

void * data user call data

size_t **rpc_peeraddr**(struct rpc_clnt * *clnt*, struct sockaddr * *buf*, size_t *bufsize*)
extract remote peer address from clnt's xprt

Parameters

struct rpc_clnt * clnt RPC client structure

struct sockaddr * buf target buffer

size_t bufsize length of target buffer

Description

Returns the number of bytes that are actually in the stored address.

const char * **rpc_peeraddr2str**(struct rpc_clnt * *clnt*, enum rpc_display_format_t *format*)
return remote peer address in printable format

Parameters

struct rpc_clnt * clnt RPC client structure

enum rpc_display_format_t format address format

Description

NB: the lifetime of the memory referenced by the returned pointer is the same as the rpc_xprt itself. As long as the caller uses this pointer, it must hold the RCU read lock.

int **rpc_localaddr**(struct rpc_clnt * *clnt*, struct sockaddr * *buf*, size_t *buflen*)
discover local endpoint address for an RPC client

Parameters

struct rpc_clnt * clnt RPC client structure

struct sockaddr * buf target buffer

size_t buflen size of target buffer, in bytes

Description

Returns zero and fills in "buf" and "buflen" if successful; otherwise, a negative errno is returned.

This works even if the underlying transport is not currently connected, or if the upper layer never previously provided a source address.

The result of this function call is transient: multiple calls in succession may give different results, depending on how local networking configuration changes over time.

struct net * **rpc_net_ns**(struct rpc_clnt * *clnt*)
Get the network namespace for this RPC client

Parameters

struct rpc_clnt * clnt RPC client to query

size_t **rpc_max_payload**(struct rpc_clnt * *clnt*)
Get maximum payload size for a transport, in bytes

Parameters

struct rpc_clnt * clnt RPC client to query

Description

For stream transports, this is one RPC record fragment (see RFC 1831), as we don't support multi-record requests yet. For datagram transports, this is the size of an IP packet minus the IP, UDP, and RPC header sizes.

size_t **rpc_max_bc_payload**(struct rpc_clnt * *clnt*)
Get maximum backchannel payload size, in bytes

Parameters

struct rpc_clnt * clnt RPC client to query

void **rpc_force_rebind**(struct rpc_clnt * *clnt*)
force transport to check that remote port is unchanged

Parameters

struct rpc_clnt * clnt client to rebind

int **rpc_clnt_test_and_add_xprt**(struct rpc_clnt * *clnt*, struct rpc_xprt_switch * *xps*, struct rpc_xprt * *xprt*, void * *dummy*)
Test and add a new transport to a rpc_clnt

Parameters

struct rpc_clnt * clnt pointer to struct rpc_clnt

struct rpc_xprt_switch * xps pointer to struct rpc_xprt_switch,

struct rpc_xprt * xprt pointer struct rpc_xprt

void * dummy unused

int **rpc_clnt_setup_test_and_add_xprt**(struct rpc_clnt * *clnt*, struct rpc_xprt_switch * *xps*, struct rpc_xprt * *xprt*, void * *data*)

Parameters

struct rpc_clnt * clnt struct rpc_clnt to get the new transport

struct rpc_xprt_switch * xps the rpc_xprt_switch to hold the new transport

struct rpc_xprt * xprt the rpc_xprt to test

void * data a struct rpc_add_xprt_test pointer that holds the test function and test function call data

Description

This is an rpc_clnt_add_xprt setup() function which returns 1 so: 1) caller of the test function must dereference the rpc_xprt_switch and the rpc_xprt. 2) test function must call rpc_xprt_switch_add_xprt, usually in the rpc_call_done routine.

Upon success (return of 1), the test function adds the new transport to the rpc_clnt xprt switch

int **rpc_clnt_add_xprt**(struct rpc_clnt * *clnt*, struct xprt_create * *xprtargs*, int (*setup) (struct rpc_clnt *, struct rpc_xprt_switch *, struct rpc_xprt *, void *, void * *data*)
Add a new transport to a rpc_clnt

Parameters

struct rpc_clnt * clnt pointer to struct rpc_clnt

struct xprt_create * xprtargs pointer to struct xprt_create

int (*)(struct rpc_clnt *, struct rpc_xprt_switch *, struct rpc_xprt *, void *) setup
callback to test and/or set up the connection

void * data pointer to setup function data

Description

Creates a new transport using the parameters set in args and adds it to clnt. If ping is set, then test that connectivity succeeds before adding the new transport.

WiMAX

struct *sk_buff* * **wimax_msg_alloc**(struct *wimax_dev* * *wimax_dev*, const char * *pipe_name*, const void * *msg*, size_t *size*, gfp_t *gfp_flags*)
Create a new skb for sending a message to userspace

Parameters

struct wimax_dev * wimax_dev WiMAX device descriptor

const char * pipe_name "named pipe" the message will be sent to

const void * msg pointer to the message data to send

size_t size size of the message to send (in bytes), including the header.

gfp_t gfp_flags flags for memory allocation.

Return

`0` if ok, negative errno code on error

Description

Allocates an skb that will contain the message to send to user space over the messaging pipe and initializes it, copying the payload.

Once this call is done, you can deliver it with *wimax_msg_send()*.

IMPORTANT:

Don't use *skb_push()*/*skb_pull()*/*skb_reserve()* on the skb, as *wimax_msg_send()* depends on skb->data being placed at the beginning of the user message.

Unlike other WiMAX stack calls, this call can be used way early, even before *wimax_dev_add()* is called, as long as the wimax_dev->net_dev pointer is set to point to a proper net_dev. This is so that drivers can use it early in case they need to send stuff around or communicate with user space.

const void * **wimax_msg_data_len**(struct *sk_buff* * *msg*, size_t * *size*)
Return a pointer and size of a message's payload

Parameters

struct sk_buff * msg Pointer to a message created with *wimax_msg_alloc()*

size_t * size Pointer to where to store the message's size

Description

Returns the pointer to the message data.

const void * **wimax_msg_data**(struct *sk_buff* * *msg*)
Return a pointer to a message's payload

Parameters

struct sk_buff * msg Pointer to a message created with *wimax_msg_alloc()*

ssize_t **wimax_msg_len**(struct *sk_buff* * *msg*)
Return a message's payload length

Parameters

struct sk_buff * msg Pointer to a message created with *wimax_msg_alloc()*

int **wimax_msg_send**(struct *wimax_dev* * *wimax_dev*, struct *sk_buff* * *skb*)
Send a pre-allocated message to user space

Parameters

struct wimax_dev * wimax_dev WiMAX device descriptor

struct sk_buff * skb *struct sk_buff* returned by *wimax_msg_alloc()*. Note the ownership of **skb** is transferred to this function.

Return

0 if ok, < 0 errno code on error

Description

Sends a free-form message that was preallocated with *wimax_msg_alloc()* and filled up.

Assumes that once you pass an skb to this function for sending, it owns it and will release it when done (on success).

IMPORTANT:

Don't use *skb_push()*/*skb_pull()*/*skb_reserve()* on the skb, as *wimax_msg_send()* depends on skb->data being placed at the beginning of the user message.

Unlike other WiMAX stack calls, this call can be used way early, even before *wimax_dev_add()* is called, as long as the wimax_dev->net_dev pointer is set to point to a proper net_dev. This is so that drivers can use it early in case they need to send stuff around or communicate with user space.

int **wimax_msg**(struct *wimax_dev* * *wimax_dev*, const char * *pipe_name*, const void * *buf*, size_t *size*, gfp_t *gfp_flags*)
Send a message to user space

Parameters

struct wimax_dev * wimax_dev WiMAX device descriptor (properly referenced)

const char * pipe_name "named pipe" the message will be sent to

const void * buf pointer to the message to send.

size_t size size of the buffer pointed to by **buf** (in bytes).

gfp_t gfp_flags flags for memory allocation.

Return

`0` if ok, negative errno code on error.

Description

Sends a free-form message to user space on the device **wimax_dev**.

NOTES

Once the **skb** is given to this function, who will own it and will release it when done (unless it returns error).

int **wimax_reset**(struct *wimax_dev* * *wimax_dev*)
Reset a WiMAX device

Parameters

struct wimax_dev * wimax_dev WiMAX device descriptor

Return

`0` if ok and a warm reset was done (the device still exists in the system).

-`ENODEV` if a cold/bus reset had to be done (device has disconnected and reconnected, so current handle is not valid any more).

-`EINVAL` if the device is not even registered.

Any other negative error code shall be considered as non-recoverable.

Description

Called when wanting to reset the device for any reason. Device is taken back to power on status.

This call blocks; on successful return, the device has completed the reset process and is ready to operate.

void **wimax_report_rfkill_hw**(struct *wimax_dev* * *wimax_dev*, enum wimax_rf_state *state*)
Reports changes in the hardware RF switch

Parameters

struct wimax_dev * wimax_dev WiMAX device descriptor

enum wimax_rf_state state New state of the RF Kill switch. `WIMAX_RF_ON` radio on, `WIMAX_RF_OFF` radio off.

Description

When the device detects a change in the state of thehardware RF switch, it must call this function to let the WiMAX kernel stack know that the state has changed so it can be properly propagated.

The WiMAX stack caches the state (the driver doesn't need to). As well, as the change is propagated it will come back as a request to change the software state to mirror the hardware state.

If the device doesn't have a hardware kill switch, just report it on initialization as always on (`WIMAX_RF_ON`, radio on).

void **wimax_report_rfkill_sw**(struct *wimax_dev* * *wimax_dev*, enum wimax_rf_state *state*)
Reports changes in the software RF switch

Parameters

struct wimax_dev * wimax_dev WiMAX device descriptor

enum wimax_rf_state state New state of the RF kill switch. `WIMAX_RF_ON` radio on, `WIMAX_RF_OFF` radio off.

Description

Reports changes in the software RF switch state to the WiMAX stack.

The main use is during initialization, so the driver can query the device for its current software radio kill switch state and feed it to the system.

On the side, the device does not change the software state by itself. In practice, this can happen, as the device might decide to switch (in software) the radio off for different reasons.

int **wimax_rfkill**(struct *wimax_dev* * *wimax_dev*, enum wimax_rf_state *state*)
Set the software RF switch state for a WiMAX device

Parameters

struct wimax_dev * wimax_dev WiMAX device descriptor

enum wimax_rf_state state New RF state.

Return

>= 0 toggle state if ok, < 0 errno code on error. The toggle state is returned as a bitmap, bit 0 being the hardware RF state, bit 1 the software RF state.

0 means disabled (`WIMAX_RF_ON`, radio on), 1 means enabled radio off (`WIMAX_RF_OFF`).

Description

Called by the user when he wants to request the WiMAX radio to be switched on (`WIMAX_RF_ON`) or off (`WIMAX_RF_OFF`). With `WIMAX_RF_QUERY`, just the current state is returned.

NOTE

This call will block until the operation is complete.

void **wimax_state_change**(struct *wimax_dev* * *wimax_dev*, enum *wimax_st* *new_state*)
Set the current state of a WiMAX device

Parameters

struct wimax_dev * wimax_dev WiMAX device descriptor (properly referenced)

enum wimax_st new_state New state to switch to

Description

This implements the state changes for the wimax devices. It will

- verify that the state transition is legal (for now it'll just print a warning if not) according to the table in linux/wimax.h's documentation for 'enum wimax_st'.
- perform the actions needed for leaving the current state and whichever are needed for entering the new state.
- issue a report to user space indicating the new state (and an optional payload with information about the new state).

NOTE

wimax_dev must be locked

enum *wimax_st* **wimax_state_get**(struct *wimax_dev* * *wimax_dev*)
Return the current state of a WiMAX device

Parameters

struct wimax_dev * wimax_dev WiMAX device descriptor

Return

Current state of the device according to its driver.

void **wimax_dev_init**(struct *wimax_dev* * *wimax_dev*)
initialize a newly allocated instance

Parameters

struct wimax_dev * wimax_dev WiMAX device descriptor to initialize.

Description

Initializes fields of a freshly allocated **wimax_dev** instance. This function assumes that after allocation, the memory occupied by **wimax_dev** was zeroed.

int **wimax_dev_add**(struct *wimax_dev* * *wimax_dev*, struct *net_device* * *net_dev*)
Register a new WiMAX device

Parameters

struct wimax_dev * wimax_dev WiMAX device descriptor (as embedded in your **net_dev**'s priv data). You must have called *wimax_dev_init()* on it before.

struct net_device * net_dev net device the **wimax_dev** is associated with. The function expects SET_NETDEV_DEV() and *register_netdev()* were already called on it.

Description

Registers the new WiMAX device, sets up the user-kernel control interface (generic netlink) and common WiMAX infrastructure.

Note that the parts that will allow interaction with user space are setup at the very end, when the rest is in place, as once that happens, the driver might get user space control requests via netlink or from debugfs that might translate into calls into wimax_dev->op_*().

void **wimax_dev_rm**(struct *wimax_dev* * *wimax_dev*)
Unregister an existing WiMAX device

Parameters

struct wimax_dev * wimax_dev WiMAX device descriptor

Description

Unregisters a WiMAX device previously registered for use with `wimax_add_rm()`.

IMPORTANT! Must call before calling *unregister_netdev()*.

After this function returns, you will not get any more user space control requests (via netlink or debugfs) and thus to wimax_dev->ops.

Reentrancy control is ensured by setting the state to `__WIMAX_ST_QUIESCING`. rfkill operations coming through wimax_*rfkill*() will be stopped by the quiescing state; ops coming from the rfkill subsystem will be stopped by the support being removed by `wimax_rfkill_rm()`.

struct **wimax_dev**
Generic WiMAX device

Definition

```
struct wimax_dev {
  struct net_device *net_dev;
  struct list_head id_table_node;
  struct mutex mutex;
  struct mutex mutex_reset;
  enum wimax_st state;
  int (*op_msg_from_user)(struct wimax_dev *wimax_dev,const char *,const void *, size_t, const struct ge
  int (*op_rfkill_sw_toggle)(struct wimax_dev *wimax_dev, enum wimax_rf_state);
  int (*op_reset)(struct wimax_dev *wimax_dev);
  struct rfkill *rfkill;
  unsigned int rf_hw;
  unsigned int rf_sw;
  char name[32];
  struct dentry *debugfs_dentry;
};
```

Members

net_dev [fill] Pointer to the *struct net_device* this WiMAX device implements.

id_table_node [private] link to the list of wimax devices kept by id-table.c. Protected by it's own spinlock.

mutex [private] Serializes all concurrent access and execution of operations.

mutex_reset [private] Serializes reset operations. Needs to be a different mutex because as part of the reset operation, the driver has to call back into the stack to do things such as state change, that require wimax_dev->mutex.

state [private] Current state of the WiMAX device.

op_msg_from_user [fill] Driver-specific operation to handle a raw message from user space to the driver. The driver can send messages to user space using with `wimax_msg_to_user()`.

op_rfkill_sw_toggle [fill] Driver-specific operation to act on userspace (or any other agent) requesting the WiMAX device to change the RF Kill software switch (WIMAX_RF_ON or WIMAX_RF_OFF). If such hardware support is not present, it is assumed the radio cannot be switched off and it is always on (and the stack will error out when trying to switch it off). In such case, this function pointer can be left as NULL.

op_reset [fill] Driver specific operation to reset the device. This operation should always attempt first a warm reset that does not disconnect the device from the bus and return 0. If that fails, it should resort to some sort of cold or bus reset (even if it implies a bus disconnection and device disappearance). In that case, -ENODEV should be returned to indicate the device is gone. This operation has to be synchronous, and return only when the reset is complete. In case of having had to resort to bus/cold reset implying a device disconnection, the call is allowed to return immediately.

rfkill [private] integration into the RF-Kill infrastructure.

rf_hw [private] State of the hardware radio switch (OFF/ON)

rf_sw [private] State of the software radio switch (OFF/ON)

name [fill] A way to identify this device. We need to register a name with many subsystems (rfkill, workqueue creation, etc). We can't use the network device name as that might change and in some instances we don't know it yet (until we don't call *register_netdev()*). So we generate an unique one using the driver name and device bus id, place it here and use it across the board. Recommended naming: DRIVERNAME-BUSNAME:BUSID (dev->bus->name, dev->bus_id).

debugfs_dentry [private] Used to hook up a debugfs entry. This shows up in the debugfs root as wimax:DEVICENAME.

NOTE

wimax_dev->mutex is NOT locked when this op is being called; however, wimax_dev->mutex_reset IS locked to ensure serialization of calls to *wimax_reset()*. See *wimax_reset()*'s documentation.

Description

This structure defines a common interface to access all WiMAX devices from different vendors and provides a common API as well as a free-form device-specific messaging channel.

Usage:

1. Embed a *struct wimax_dev* at *the beginning* the network device structure so that *netdev_priv()* points to it.
2. `memset()` it to zero
3. Initialize with *wimax_dev_init()*. This will leave the WiMAX device in the `__WIMAX_ST_NULL` state.
4. Fill all the fields marked with [fill]; once called *wimax_dev_add()*, those fields CANNOT be modified.
5. Call *wimax_dev_add()* *after* registering the network device. This will leave the WiMAX device in the `WIMAX_ST_DOWN` state. Protect the driver's net_device->:c:func:*open()* against succeeding if the wimax device state is lower than `WIMAX_ST_DOWN`.
6. Select when the device is going to be turned on/initialized; for example, it could be initialized on 'ifconfig up' (when the netdev op '`open()`' is called on the driver).

When the device is initialized (at *ifconfig up* time, or right after calling *wimax_dev_add()* from `_probe()`, make sure the following steps are taken

1. Move the device to `WIMAX_ST_UNINITIALIZED`. This is needed so some API calls that shouldn't work until the device is ready can be blocked.
2. Initialize the device. Make sure to turn the SW radio switch off and move the device to state `WIMAX_ST_RADIO_OFF` when done. When just initialized, a device should be left in RADIO OFF state until user space devices to turn it on.
3. Query the device for the state of the hardware rfkill switch and call `wimax_rfkill_report_hw()` and `wimax_rfkill_report_sw()` as needed. See below.

wimax_dev_rm() undoes before unregistering the network device. Once *wimax_dev_add()* is called, the driver can get called on the wimax_dev->op_* function pointers

CONCURRENCY:

The stack provides a mutex for each device that will disallow API calls happening concurrently; thus, op calls into the driver through the wimax_dev->op*() function pointers will always be serialized and *never* concurrent.

For locking, take wimax_dev->mutex is taken; (most) operations in the API have to check for `wimax_dev_is_ready()` to return 0 before continuing (this is done internally).

REFERENCE COUNTING:

The WiMAX device is reference counted by the associated network device. The only operation that can be used to reference the device is `wimax_dev_get_by_genl_info()`, and the reference it acquires has to be released with dev_put(wimax_dev->net_dev).

RFKILL:

At startup, both HW and SW radio switchess are assumed to be off.

At initialization time [after calling *wimax_dev_add()*], have the driver query the device for the status of the software and hardware RF kill switches and call *wimax_report_rfkill_hw()* and `wimax_rfkill_report_sw()` to indicate their state. If any is missing, just call it to indicate it is ON (radio always on).

Whenever the driver detects a change in the state of the RF kill switches, it should call *wimax_report_rfkill_hw()* or *wimax_report_rfkill_sw()* to report it to the stack.

enum **wimax_st**
The different states of a WiMAX device

Constants

__WIMAX_ST_NULL The device structure has been allocated and zeroed, but still *wimax_dev_add()* hasn't been called. There is no state.

WIMAX_ST_DOWN The device has been registered with the WiMAX and networking stacks, but it is not initialized (normally that is done with 'ifconfig DEV up' [or equivalent], which can upload firmware and enable communications with the device). In this state, the device is powered down and using as less power as possible. This state is the default after a call to *wimax_dev_add()*. It is ok to have drivers move directly to `WIMAX_ST_UNINITIALIZED` or `WIMAX_ST_RADIO_OFF` in `_probe()` after the call to *wimax_dev_add()*. It is recommended that the driver leaves this state when calling 'ifconfig DEV up' and enters it back on 'ifconfig DEV down'.

__WIMAX_ST_QUIESCING The device is being torn down, so no API operations are allowed to proceed except the ones needed to complete the device clean up process.

WIMAX_ST_UNINITIALIZED [optional] Communication with the device is setup, but the device still requires some configuration before being operational. Some WiMAX API calls might work.

WIMAX_ST_RADIO_OFF The device is fully up; radio is off (wether by hardware or software switches). It is recommended to always leave the device in this state after initialization.

WIMAX_ST_READY The device is fully up and radio is on.

WIMAX_ST_SCANNING [optional] The device has been instructed to scan. In this state, the device cannot be actively connected to a network.

WIMAX_ST_CONNECTING The device is connecting to a network. This state exists because in some devices, the connect process can include a number of negotiations between user space, kernel space and the device. User space needs to know what the device is doing. If the connect sequence in a device is atomic and fast, the device can transition directly to CONNECTED

WIMAX_ST_CONNECTED The device is connected to a network.

__WIMAX_ST_INVALID This is an invalid state used to mark the maximum numeric value of states.

Description

Transitions from one state to another one are atomic and can only be caused in kernel space with *wimax_state_change()*. To read the state, use *wimax_state_get()*.

States starting with __ are internal and shall not be used or referred to by drivers or userspace. They look ugly, but that's the point – if any use is made non-internal to the stack, it is easier to catch on review.

All API operations [with well defined exceptions] will take the device mutex before starting and then check the state. If the state is `__WIMAX_ST_NULL`, `WIMAX_ST_DOWN`, `WIMAX_ST_UNINITIALIZED` or `__WIMAX_ST_QUIESCING`, it will drop the lock and quit with `-EINVAL`, `-ENOMEDIUM`, `-ENOTCONN` or `-ESHUTDOWN`.

The order of the definitions is important, so we can do numerical comparisons (eg: < `WIMAX_ST_RADIO_OFF` means the device is not ready to operate).

Network device support

Driver Support

void **dev_add_pack**(struct packet_type * *pt*)
add packet handler

Parameters

struct packet_type * pt packet type declaration

Description

Add a protocol handler to the networking stack. The passed `packet_type` is linked into kernel lists and may not be freed until it has been removed from the kernel lists.

This call does not sleep therefore it can not guarantee all CPU's that are in middle of receiving packets will see the new packet type (until the next received packet).

void **__dev_remove_pack**(struct packet_type * *pt*)
remove packet handler

Parameters

struct packet_type * pt packet type declaration

Description

Remove a protocol handler that was previously added to the kernel protocol handlers by *dev_add_pack()*. The passed `packet_type` is removed from the kernel lists and can be freed or reused once this function returns.

The packet type might still be in use by receivers and must not be freed until after all the CPU's have gone through a quiescent state.

void **dev_remove_pack**(struct packet_type * *pt*)
remove packet handler

Parameters

struct packet_type * pt packet type declaration

Description

Remove a protocol handler that was previously added to the kernel protocol handlers by *dev_add_pack()*. The passed `packet_type` is removed from the kernel lists and can be freed or reused once this function returns.

This call sleeps to guarantee that no CPU is looking at the packet type after return.

void **dev_add_offload**(struct packet_offload * *po*)
register offload handlers

Parameters

struct packet_offload * po protocol offload declaration

Description

Add protocol offload handlers to the networking stack. The passed `proto_offload` is linked into kernel lists and may not be freed until it has been removed from the kernel lists.

This call does not sleep therefore it can not guarantee all CPU's that are in middle of receiving packets will see the new offload handlers (until the next received packet).

void **dev_remove_offload**(struct packet_offload * *po*)
remove packet offload handler

Parameters

struct packet_offload * po packet offload declaration

Description

Remove a packet offload handler that was previously added to the kernel offload handlers by *dev_add_offload()*. The passed offload_type is removed from the kernel lists and can be freed or reused once this function returns.

This call sleeps to guarantee that no CPU is looking at the packet type after return.

int **netdev_boot_setup_check**(struct *net_device* * *dev*)
check boot time settings

Parameters

struct net_device * dev the netdevice

Description

Check boot time settings for the device. The found settings are set for the device to be used later in the device probing. Returns 0 if no settings found, 1 if they are.

int **dev_get_iflink**(const struct *net_device* * *dev*)
get 'iflink' value of a interface

Parameters

const struct net_device * dev targeted interface

Description

Indicates the ifindex the interface is linked to. Physical interfaces have the same 'ifindex' and 'iflink' values.

int **dev_fill_metadata_dst**(struct *net_device* * *dev*, struct *sk_buff* * *skb*)
Retrieve tunnel egress information.

Parameters

struct net_device * dev targeted interface

struct sk_buff * skb The packet.

Description

For better visibility of tunnel traffic OVS needs to retrieve egress tunnel information for a packet. Following API allows user to get this info.

struct *net_device* * **__dev_get_by_name**(struct net * *net*, const char * *name*)
find a device by its name

Parameters

struct net * net the applicable net namespace

const char * name name to find

Description

Find an interface by name. Must be called under RTNL semaphore or **dev_base_lock**. If the name is found a pointer to the device is returned. If the name is not found then NULL is returned. The reference counters are not incremented so the caller must be careful with locks.

struct *net_device* * **dev_get_by_name_rcu**(struct net * *net*, const char * *name*)
find a device by its name

Parameters

struct net * net the applicable net namespace

const char * name name to find

Description

Find an interface by name. If the name is found a pointer to the device is returned. If the name is not found then NULL is returned. The reference counters are not incremented so the caller must be careful with locks. The caller must hold RCU lock.

struct *net_device* * **dev_get_by_name**(struct net * *net*, const char * *name*)
find a device by its name

Parameters

struct net * net the applicable net namespace

const char * name name to find

Description

Find an interface by name. This can be called from any context and does its own locking. The returned handle has the usage count incremented and the caller must use *dev_put()* to release it when it is no longer needed. NULL is returned if no matching device is found.

struct *net_device* * **__dev_get_by_index**(struct net * *net*, int *ifindex*)
find a device by its ifindex

Parameters

struct net * net the applicable net namespace

int ifindex index of device

Description

Search for an interface by index. Returns NULL if the device is not found or a pointer to the device. The device has not had its reference counter increased so the caller must be careful about locking. The caller must hold either the RTNL semaphore or **dev_base_lock**.

struct *net_device* * **dev_get_by_index_rcu**(struct net * *net*, int *ifindex*)
find a device by its ifindex

Parameters

struct net * net the applicable net namespace

int ifindex index of device

Description

Search for an interface by index. Returns NULL if the device is not found or a pointer to the device. The device has not had its reference counter increased so the caller must be careful about locking. The caller must hold RCU lock.

struct *net_device* * **dev_get_by_index**(struct net * *net*, int *ifindex*)
find a device by its ifindex

Parameters

struct net * net the applicable net namespace

int ifindex index of device

Description

Search for an interface by index. Returns NULL if the device is not found or a pointer to the device. The device returned has had a reference added and the pointer is safe until the user calls dev_put to indicate they have finished with it.

struct *net_device* * **dev_get_by_napi_id**(unsigned int *napi_id*)
find a device by napi_id

Parameters

unsigned int napi_id ID of the NAPI struct

Description

Search for an interface by NAPI ID. Returns `NULL` if the device is not found or a pointer to the device. The device has not had its reference counter increased so the caller must be careful about locking. The caller must hold RCU lock.

struct *net_device* * **dev_getbyhwaddr_rcu**(struct net * *net*, unsigned short *type*, const char * *ha*)
find a device by its hardware address

Parameters

struct net * net the applicable net namespace

unsigned short type media type of device

const char * ha hardware address

Description

Search for an interface by MAC address. Returns NULL if the device is not found or a pointer to the device. The caller must hold RCU or RTNL. The returned device has not had its ref count increased and the caller must therefore be careful about locking

struct *net_device* * **__dev_get_by_flags**(struct net * *net*, unsigned short *if_flags*, unsigned short *mask*)
find any device with given flags

Parameters

struct net * net the applicable net namespace

unsigned short if_flags IFF_* values

unsigned short mask bitmask of bits in if_flags to check

Description

Search for any interface with the given flags. Returns NULL if a device is not found or a pointer to the device. Must be called inside `rtnl_lock()`, and result refcount is unchanged.

bool **dev_valid_name**(const char * *name*)
check if name is okay for network device

Parameters

const char * name name string

Description

Network device names need to be valid file names to to allow sysfs to work. We also disallow any kind of whitespace.

int **dev_alloc_name**(struct *net_device* * *dev*, const char * *name*)
allocate a name for a device

Parameters

struct net_device * dev device

const char * name name format string

Description

Passed a format string - eg "lt''d''" it will try and find a suitable id. It scans list of devices to build up a free map, then chooses the first empty slot. The caller must hold the dev_base or rtnl lock while allocating the name and adding the device in order to avoid duplicates. Limited to bits_per_byte * page size devices (ie 32K on most platforms). Returns the number of the unit assigned or a negative errno code.

void **netdev_features_change**(struct *net_device* * *dev*)
device changes features

Parameters

struct net_device * dev device to cause notification

Description

Called to indicate a device has changed features.

void **netdev_state_change**(struct *net_device* * *dev*)
device changes state

Parameters

struct net_device * dev device to cause notification

Description

Called to indicate a device has changed state. This function calls the notifier chains for netdev_chain and sends a NEWLINK message to the routing socket.

void **netdev_notify_peers**(struct *net_device* * *dev*)
notify network peers about existence of **dev**

Parameters

struct net_device * dev network device

Description

Generate traffic such that interested network peers are aware of **dev**, such as by generating a gratuitous ARP. This may be used when a device wants to inform the rest of the network about some sort of reconfiguration such as a failover event or virtual machine migration.

int **dev_open**(struct *net_device* * *dev*)
prepare an interface for use.

Parameters

struct net_device * dev device to open

Description

Takes a device from down to up state. The device's private open function is invoked and then the multicast lists are loaded. Finally the device is moved into the up state and a `NETDEV_UP` message is sent to the netdev notifier chain.

Calling this function on an active interface is a nop. On a failure a negative errno code is returned.

void **dev_close**(struct *net_device* * *dev*)
shutdown an interface.

Parameters

struct net_device * dev device to shutdown

Description

This function moves an active device into down state. A `NETDEV_GOING_DOWN` is sent to the netdev notifier chain. The device is then deactivated and finally a `NETDEV_DOWN` is sent to the notifier chain.

void **dev_disable_lro**(struct *net_device* * *dev*)
disable Large Receive Offload on a device

Parameters

struct net_device * dev device

Description

Disable Large Receive Offload (LRO) on a net device. Must be called under RTNL. This is needed if received packets may be forwarded to another interface.

int **register_netdevice_notifier**(struct notifier_block * *nb*)
register a network notifier block

Parameters

struct notifier_block * nb notifier

Description

Register a notifier to be called when network device events occur. The notifier passed is linked into the kernel structures and must not be reused until it has been unregistered. A negative errno code is returned on a failure.

When registered all registration and up events are replayed to the new notifier to allow device to have a race free view of the network device list.

int **unregister_netdevice_notifier**(struct notifier_block * *nb*)
unregister a network notifier block

Parameters

struct notifier_block * nb notifier

Description

Unregister a notifier previously registered by *register_netdevice_notifier()*. The notifier is unlinked into the kernel structures and may then be reused. A negative errno code is returned on a failure.

After unregistering unregister and down device events are synthesized for all devices on the device list to the removed notifier to remove the need for special case cleanup code.

int **call_netdevice_notifiers**(unsigned long *val*, struct net_device * *dev*)
call all network notifier blocks

Parameters

unsigned long val value passed unmodified to notifier function

struct net_device * dev net_device pointer passed unmodified to notifier function

Description

Call all network notifier blocks. Parameters and return value are as for `raw_notifier_call_chain()`.

int **dev_forward_skb**(struct net_device * *dev*, struct sk_buff * *skb*)
loopback an skb to another netif

Parameters

struct net_device * dev destination network device

struct sk_buff * skb buffer to forward

Description

return values: NET_RX_SUCCESS (no congestion) NET_RX_DROP (packet was dropped, but freed)

dev_forward_skb can be used for injecting an skb from the start_xmit function of one device into the receive queue of another device.

The receiving device may be in another namespace, so we have to clear all information in the skb that could impact namespace isolation.

int **netif_set_real_num_rx_queues**(struct net_device * *dev*, unsigned int *rxq*)
set actual number of RX queues used

Parameters

struct net_device * dev Network device

unsigned int rxq Actual number of RX queues

Description

This must be called either with the rtnl_lock held or before registration of the net device. Returns 0 on success, or a negative error code. If called before registration, it always succeeds.

int **netif_get_num_default_rss_queues**(void)
default number of RSS queues

Parameters

void no arguments

Description

This routine should set an upper limit on the number of RSS queues used by default by multiqueue devices.

void **netif_device_detach**(struct *net_device* * *dev*)
mark device as removed

Parameters

struct net_device * dev network device

Description

Mark device as removed from system and therefore no longer available.

void **netif_device_attach**(struct *net_device* * *dev*)
mark device as attached

Parameters

struct net_device * dev network device

Description

Mark device as attached from system and restart if needed.

struct *sk_buff* * **skb_mac_gso_segment**(struct *sk_buff* * *skb*, netdev_features_t *features*)
mac layer segmentation handler.

Parameters

struct sk_buff * skb buffer to segment

netdev_features_t features features for the output path (see dev->features)

struct *sk_buff* * **__skb_gso_segment**(struct *sk_buff* * *skb*, netdev_features_t *features*, bool *tx_path*)
Perform segmentation on skb.

Parameters

struct sk_buff * skb buffer to segment

netdev_features_t features features for the output path (see dev->features)

bool tx_path whether it is called in TX path

Description

This function segments the given skb and returns a list of segments.

It may return NULL if the skb requires no segmentation. This is only possible when GSO is used for verifying header integrity.

Segmentation preserves SKB_SGO_CB_OFFSET bytes of previous skb cb.

int **dev_loopback_xmit**(struct net * *net*, struct *sock* * *sk*, struct *sk_buff* * *skb*)
loop back **skb**

Parameters

struct net * net network namespace this loopback is happening in

struct sock * sk sk needed to be a netfilter okfn

struct sk_buff * skb buffer to transmit

bool **rps_may_expire_flow**(struct *net_device* * *dev*, u16 *rxq_index*, u32 *flow_id*, u16 *filter_id*)
check whether an RFS hardware filter may be removed

Parameters

struct net_device * dev Device on which the filter was set

u16 rxq_index RX queue index

u32 flow_id Flow ID passed to `ndo_rx_flow_steer()`

u16 filter_id Filter ID returned by `ndo_rx_flow_steer()`

Description

Drivers that implement `ndo_rx_flow_steer()` should periodically call this function for each installed filter and remove the filters for which it returns `true`.

int **netif_rx**(struct *sk_buff* * *skb*)
post buffer to the network code

Parameters

struct sk_buff * skb buffer to post

Description

This function receives a packet from a device driver and queues it for the upper (protocol) levels to process. It always succeeds. The buffer may be dropped during processing for congestion control or by the protocol layers.

return values: NET_RX_SUCCESS (no congestion) NET_RX_DROP (packet was dropped)

bool **netdev_is_rx_handler_busy**(struct *net_device* * *dev*)
check if receive handler is registered

Parameters

struct net_device * dev device to check

Description

Check if a receive handler is already registered for a given device. Return true if there one.

The caller must hold the rtnl_mutex.

int **netdev_rx_handler_register**(struct *net_device* * *dev*, rx_handler_func_t * *rx_handler*, void * *rx_handler_data*)
register receive handler

Parameters

struct net_device * dev device to register a handler for

rx_handler_func_t * rx_handler receive handler to register

void * rx_handler_data data pointer that is used by rx handler

Description

Register a receive handler for a device. This handler will then be called from __netif_receive_skb. A negative errno code is returned on a failure.

The caller must hold the rtnl_mutex.

For a general description of rx_handler, see enum rx_handler_result.

void **netdev_rx_handler_unregister**(struct *net_device* * *dev*)
unregister receive handler

Parameters

struct net_device * dev device to unregister a handler from

Description

Unregister a receive handler from a device.

The caller must hold the rtnl_mutex.

int **netif_receive_skb_core**(struct *sk_buff* * *skb*)
special purpose version of netif_receive_skb

Parameters

struct sk_buff * skb buffer to process

Description

More direct receive version of *netif_receive_skb()*. It should only be used by callers that have a need to skip RPS and Generic XDP. Caller must also take care of handling if (**page_is_**)pfmemalloc.

This function may only be called from softirq context and interrupts should be enabled.

Return values (usually ignored): NET_RX_SUCCESS: no congestion NET_RX_DROP: packet was dropped

int **netif_receive_skb**(struct *sk_buff* * *skb*)
process receive buffer from network

Parameters

struct sk_buff * skb buffer to process

Description

netif_receive_skb() is the main receive data processing function. It always succeeds. The buffer may be dropped during processing for congestion control or by the protocol layers.

This function may only be called from softirq context and interrupts should be enabled.

Return values (usually ignored): NET_RX_SUCCESS: no congestion NET_RX_DROP: packet was dropped

void **__napi_schedule**(struct napi_struct * *n*)
schedule for receive

Parameters

struct napi_struct * n entry to schedule

Description

The entry's receive function will be scheduled to run. Consider using *__napi_schedule_irqoff()* if hard irqs are masked.

bool **napi_schedule_prep**(struct napi_struct * *n*)
check if napi can be scheduled

Parameters

struct napi_struct * n napi context

Description

Test if NAPI routine is already running, and if not mark it as running. This is used as a condition variable insure only one NAPI poll instance runs. We also make sure there is no pending NAPI disable.

void **__napi_schedule_irqoff**(struct napi_struct * *n*)
schedule for receive

Parameters

struct napi_struct * n entry to schedule

Description

Variant of *__napi_schedule()* assuming hard irqs are masked

bool **netdev_has_upper_dev**(struct *net_device* * *dev*, struct *net_device* * *upper_dev*)
Check if device is linked to an upper device

Parameters

struct net_device * dev device

struct net_device * upper_dev upper device to check

Description

Find out if a device is linked to specified upper device and return true in case it is. Note that this checks only immediate upper device, not through a complete stack of devices. The caller must hold the RTNL lock.

bool **netdev_has_upper_dev_all_rcu**(struct *net_device* * *dev*, struct *net_device* * *upper_dev*)
Check if device is linked to an upper device

Parameters

struct net_device * dev device

struct net_device * upper_dev upper device to check

Description

Find out if a device is linked to specified upper device and return true in case it is. Note that this checks the entire upper device chain. The caller must hold rcu lock.

bool **netdev_has_any_upper_dev**(struct *net_device* * *dev*)
Check if device is linked to some device

Parameters

struct net_device * dev device

Description

Find out if a device is linked to an upper device and return true in case it is. The caller must hold the RTNL lock.

struct *net_device* * **netdev_master_upper_dev_get**(struct *net_device* * *dev*)
Get master upper device

Parameters

struct net_device * dev device

Description

Find a master upper device and return pointer to it or NULL in case it's not there. The caller must hold the RTNL lock.

struct *net_device* * **netdev_upper_get_next_dev_rcu**(struct *net_device* * *dev*, struct list_head ** *iter*)
Get the next dev from upper list

Parameters

struct net_device * dev device

struct list_head ** iter list_head ** of the current position

Description

Gets the next device from the dev's upper list, starting from iter position. The caller must hold RCU read lock.

void * **netdev_lower_get_next_private**(struct *net_device* * *dev*, struct list_head ** *iter*)
Get the next ->private from the lower neighbour list

Parameters

struct net_device * dev device

struct list_head ** iter list_head ** of the current position

Description

Gets the next netdev_adjacent->private from the dev's lower neighbour list, starting from iter position. The caller must hold either hold the RTNL lock or its own locking that guarantees that the neighbour lower list will remain unchanged.

void * **netdev_lower_get_next_private_rcu**(struct *net_device* * *dev*, struct list_head ** *iter*)
Get the next ->private from the lower neighbour list, RCU variant

Parameters

struct net_device * dev device

struct list_head ** iter list_head ** of the current position

Description

Gets the next netdev_adjacent->private from the dev's lower neighbour list, starting from iter position. The caller must hold RCU read lock.

void * **netdev_lower_get_next**(struct *net_device* * *dev*, struct list_head ** *iter*)
Get the next device from the lower neighbour list

Parameters

struct net_device * dev device

struct list_head ** iter list_head ** of the current position

Description

Gets the next netdev_adjacent from the dev's lower neighbour list, starting from iter position. The caller must hold RTNL lock or its own locking that guarantees that the neighbour lower list will remain unchanged.

void * **netdev_lower_get_first_private_rcu**(struct *net_device* * *dev*)
Get the first ->private from the lower neighbour list, RCU variant

Parameters

struct net_device * dev device

Description

Gets the first netdev_adjacent->private from the dev's lower neighbour list. The caller must hold RCU read lock.

struct *net_device* * **netdev_master_upper_dev_get_rcu**(struct *net_device* * *dev*)
Get master upper device

Parameters

struct net_device * dev device

Description

Find a master upper device and return pointer to it or NULL in case it's not there. The caller must hold the RCU read lock.

int **netdev_upper_dev_link**(struct *net_device* * *dev*, struct *net_device* * *upper_dev*, struct netlink_ext_ack * *extack*)
Add a link to the upper device

Parameters

struct net_device * dev device

struct net_device * upper_dev new upper device

struct netlink_ext_ack * extack netlink extended ack

Description

Adds a link to device which is upper to this one. The caller must hold the RTNL lock. On a failure a negative errno code is returned. On success the reference counts are adjusted and the function returns zero.

int **netdev_master_upper_dev_link**(struct *net_device* * *dev*, struct *net_device* * *upper_dev*, void * *upper_priv*, void * *upper_info*, struct netlink_ext_ack * *extack*)
Add a master link to the upper device

Parameters

struct net_device * dev device

struct net_device * upper_dev new upper device

void * upper_priv upper device private

void * upper_info upper info to be passed down via notifier

struct netlink_ext_ack * extack netlink extended ack

Description

Adds a link to device which is upper to this one. In this case, only one master upper device can be linked, although other non-master devices might be linked as well. The caller must hold the RTNL lock. On a failure a negative errno code is returned. On success the reference counts are adjusted and the function returns zero.

void **netdev_upper_dev_unlink**(struct *net_device* * *dev*, struct *net_device* * *upper_dev*)
Removes a link to upper device

Parameters

struct net_device * dev device

struct net_device * upper_dev new upper device

Description

Removes a link to device which is upper to this one. The caller must hold the RTNL lock.

void **netdev_bonding_info_change**(struct *net_device* * *dev*, struct netdev_bonding_info * *bonding_info*)
Dispatch event about slave change

Parameters

struct net_device * dev device

struct netdev_bonding_info * bonding_info info to dispatch

Description

Send NETDEV_BONDING_INFO to netdev notifiers with info. The caller must hold the RTNL lock.

void **netdev_lower_state_changed**(struct *net_device* * *lower_dev*, void * *lower_state_info*)
Dispatch event about lower device state change

Parameters

struct net_device * lower_dev device

void * lower_state_info state to dispatch

Description

Send NETDEV_CHANGELOWERSTATE to netdev notifiers with info. The caller must hold the RTNL lock.

int **dev_set_promiscuity**(struct *net_device* * *dev*, int *inc*)
update promiscuity count on a device

Parameters

struct net_device * dev device

int inc modifier

Description

Add or remove promiscuity from a device. While the count in the device remains above zero the interface remains promiscuous. Once it hits zero the device reverts back to normal filtering operation. A negative inc value is used to drop promiscuity on the device. Return 0 if successful or a negative errno code on error.

int **dev_set_allmulti**(struct *net_device* * *dev*, int *inc*)
update allmulti count on a device

Parameters

struct net_device * dev device

int inc modifier

Description

Add or remove reception of all multicast frames to a device. While the count in the device remains above zero the interface remains listening to all interfaces. Once it hits zero the device reverts back to normal filtering operation. A negative **inc** value is used to drop the counter when releasing a resource needing all multicasts. Return 0 if successful or a negative errno code on error.

unsigned int **dev_get_flags**(const struct *net_device* * *dev*)
get flags reported to userspace

Parameters

const struct net_device * dev device

Description

Get the combination of flag bits exported through APIs to userspace.

int **dev_change_flags**(struct *net_device* * *dev*, unsigned int *flags*)
change device settings

Parameters

struct net_device * dev device

unsigned int flags device state flags

Description

Change settings on device based state flags. The flags are in the userspace exported format.

int **dev_set_mtu**(struct *net_device* * *dev*, int *new_mtu*)
Change maximum transfer unit

Parameters

struct net_device * dev device

int new_mtu new transfer unit

Description

Change the maximum transfer size of the network device.

void **dev_set_group**(struct *net_device* * *dev*, int *new_group*)
Change group this device belongs to

Parameters

struct net_device * dev device

int new_group group this device should belong to

int **dev_set_mac_address**(struct *net_device* * *dev*, struct sockaddr * *sa*)
Change Media Access Control Address

Parameters

struct net_device * dev device

struct sockaddr * sa new address

Description

Change the hardware (MAC) address of the device

int **dev_change_carrier**(struct *net_device* * *dev*, bool *new_carrier*)
Change device carrier

Parameters

struct net_device * dev device

bool new_carrier new value

Description

Change device carrier

int **dev_get_phys_port_id**(struct *net_device* * *dev*, struct netdev_phys_item_id * *ppid*)
Get device physical port ID

Parameters

struct net_device * dev device

struct netdev_phys_item_id * ppid port ID

Description

Get device physical port ID

int **dev_get_phys_port_name**(struct *net_device* * *dev*, char * *name*, size_t *len*)
Get device physical port name

Parameters

struct net_device * dev device

char * name port name

size_t len limit of bytes to copy to name

Description

Get device physical port name

int **dev_change_proto_down**(struct *net_device* * *dev*, bool *proto_down*)
update protocol port state information

Parameters

struct net_device * dev device

bool proto_down new value

Description

This info can be used by switch drivers to set the phys state of the port.

void **netdev_update_features**(struct *net_device* * *dev*)
recalculate device features

Parameters

struct net_device * dev the device to check

Description

Recalculate dev->features set and send notifications if it has changed. Should be called after driver or hardware dependent conditions might have changed that influence the features.

void **netdev_change_features**(struct *net_device* * *dev*)
recalculate device features

Parameters

struct net_device * dev the device to check

Description

Recalculate dev->features set and send notifications even if they have not changed. Should be called instead of *netdev_update_features()* if also dev->vlan_features might have changed to allow the changes to be propagated to stacked VLAN devices.

void **netif_stacked_transfer_operstate**(const struct *net_device* * *rootdev*, struct *net_device* * *dev*)
transfer operstate

Parameters

const struct net_device * rootdev the root or lower level device to transfer state from

struct net_device * dev the device to transfer operstate to

Description

Transfer operational state from root to device. This is normally called when a stacking relationship exists between the root device and the device(a leaf device).

int **register_netdevice**(struct *net_device* * *dev*)
register a network device

Parameters

struct net_device * dev device to register

Description

Take a completed network device structure and add it to the kernel interfaces. A NETDEV_REGISTER message is sent to the netdev notifier chain. 0 is returned on success. A negative errno code is returned on a failure to set up the device, or if the name is a duplicate.

Callers must hold the rtnl semaphore. You may want *register_netdev()* instead of this.

BUGS: The locking appears insufficient to guarantee two parallel registers will not get the same name.

int **init_dummy_netdev**(struct *net_device* * *dev*)
init a dummy network device for NAPI

Parameters

struct net_device * dev device to init

Description

This takes a network device structure and initialize the minimum amount of fields so it can be used to schedule NAPI polls without registering a full blown interface. This is to be used by drivers that need to tie several hardware interfaces to a single NAPI poll scheduler due to HW limitations.

int **register_netdev**(struct *net_device* * *dev*)
register a network device

Parameters

struct net_device * dev device to register

Description

Take a completed network device structure and add it to the kernel interfaces. A NETDEV_REGISTER message is sent to the netdev notifier chain. 0 is returned on success. A negative errno code is returned on a failure to set up the device, or if the name is a duplicate.

This is a wrapper around register_netdevice that takes the rtnl semaphore and expands the device name if you passed a format string to alloc_netdev.

struct rtnl_link_stats64 * **dev_get_stats**(struct *net_device* * *dev*, struct rtnl_link_stats64 * *storage*)
get network device statistics

Parameters

struct net_device * dev device to get statistics from

struct rtnl_link_stats64 * storage place to store stats

Description

Get network statistics from device. Return **storage**. The device driver may provide its own method by setting dev->netdev_ops->get_stats64 or dev->netdev_ops->get_stats; otherwise the internal statistics structure is used.

struct *net_device* * **alloc_netdev_mqs**(int *sizeof_priv*, const char * *name*, unsigned char *name_assign_type*, void (*setup) (struct *net_device* *, unsigned int *txqs*, unsigned int *rxqs*)
allocate network device

Parameters

int sizeof_priv size of private data to allocate space for

const char * name device name format string

unsigned char name_assign_type origin of device name

void (*)(struct net_device *) setup callback to initialize device

unsigned int txqs the number of TX subqueues to allocate

unsigned int rxqs the number of RX subqueues to allocate

Description

Allocates a struct net_device with private data area for driver use and performs basic initialization. Also allocates subqueue structs for each queue on the device.

void **free_netdev**(struct *net_device* * *dev*)
free network device

Parameters

struct net_device * dev device

Description

This function does the last stage of destroying an allocated device interface. The reference to the device object is released. If this is the last reference then it will be freed.Must be called in process context.

void **synchronize_net**(void)
Synchronize with packet receive processing

Parameters

void no arguments

Description

Wait for packets currently being received to be done. Does not block later packets from starting.

void **unregister_netdevice_queue**(struct *net_device* * *dev*, struct list_head * *head*)
remove device from the kernel

Parameters

struct net_device * dev device

struct list_head * head list

Description

This function shuts down a device interface and removes it from the kernel tables. If head not NULL, device is queued to be unregistered later.

Callers must hold the rtnl semaphore. You may want *unregister_netdev()* instead of this.

void **unregister_netdevice_many**(struct list_head * *head*)
unregister many devices

Parameters

struct list_head * head list of devices

Note

As most callers use a stack allocated list_head, we force a `list_del()` to make sure stack wont be corrupted later.

void **unregister_netdev**(struct *net_device* * *dev*)
remove device from the kernel

Parameters

struct net_device * dev device

Description

This function shuts down a device interface and removes it from the kernel tables.

This is just a wrapper for unregister_netdevice that takes the rtnl semaphore. In general you want to use this and not unregister_netdevice.

int **dev_change_net_namespace**(struct *net_device* * *dev*, struct net * *net*, const char * *pat*)
move device to different nethost namespace

Parameters

struct net_device * dev device

struct net * net network namespace

const char * pat If not NULL name pattern to try if the current device name is already taken in the destination network namespace.

Description

This function shuts down a device interface and moves it to a new network namespace. On success 0 is returned, on a failure a netagive errno code is returned.

Callers must hold the rtnl semaphore.

netdev_features_t **netdev_increment_features**(netdev_features_t *all*, netdev_features_t *one*, netdev_features_t *mask*)
increment feature set by one

Parameters

netdev_features_t all current feature set

netdev_features_t one new feature set

netdev_features_t mask mask feature set

Description

Computes a new feature set after adding a device with feature set **one** to the master device with current feature set **all**. Will not enable anything that is off in **mask**. Returns the new feature set.

int **eth_header**(struct *sk_buff* * *skb*, struct *net_device* * *dev*, unsigned short *type*, const void * *daddr*, const void * *saddr*, unsigned int *len*)
create the Ethernet header

Parameters

struct sk_buff * skb buffer to alter

struct net_device * dev source device

unsigned short type Ethernet type field

const void * daddr destination address (NULL leave destination address)

const void * saddr source address (NULL use device source address)

unsigned int len packet length (<= skb->len)

Description

Set the protocol type. For a packet of type ETH_P_802_3/2 we put the length in here instead.

u32 **eth_get_headlen**(void * *data*, unsigned int *len*)
determine the length of header for an ethernet frame

Parameters

void * data pointer to start of frame

unsigned int len total length of frame

Description

Make a best effort attempt to pull the length for all of the headers for a given frame in a linear buffer.

__be16 **eth_type_trans**(struct *sk_buff* * *skb*, struct *net_device* * *dev*)
determine the packet's protocol ID.

Parameters

struct sk_buff * skb received socket data

struct net_device * dev receiving network device

Description

The rule here is that we assume 802.3 if the type field is short enough to be a length. This is normal practice and works for any 'now in use' protocol.

int **eth_header_parse**(const struct *sk_buff* * *skb*, unsigned char * *haddr*)
extract hardware address from packet

Parameters

const struct sk_buff * skb packet to extract header from

unsigned char * haddr destination buffer

int **eth_header_cache**(const struct neighbour * *neigh*, struct hh_cache * *hh*, __be16 *type*)
fill cache entry from neighbour

Parameters

const struct neighbour * neigh source neighbour

struct hh_cache * hh destination cache entry

__be16 type Ethernet type field

Description

Create an Ethernet header template from the neighbour.

void **eth_header_cache_update**(struct hh_cache * *hh*, const struct *net_device* * *dev*, const unsigned char * *haddr*)
update cache entry

Parameters

struct hh_cache * hh destination cache entry

const struct net_device * dev network device

const unsigned char * haddr new hardware address

Description

Called by Address Resolution module to notify changes in address.

int **eth_prepare_mac_addr_change**(struct *net_device* * *dev*, void * *p*)
prepare for mac change

Parameters

struct net_device * dev network device

void * p socket address

void **eth_commit_mac_addr_change**(struct *net_device* * *dev*, void * *p*)
commit mac change

Parameters

struct net_device * dev network device

void * p socket address

int **eth_mac_addr**(struct *net_device* * *dev*, void * *p*)
set new Ethernet hardware address

Parameters

struct net_device * dev network device

void * p socket address

Description

Change hardware address of device.

This doesn’t change hardware matching, so needs to be overridden for most real devices.

int **eth_change_mtu**(struct *net_device* * *dev*, int *new_mtu*)
set new MTU size

Parameters

struct net_device * dev network device

int new_mtu new Maximum Transfer Unit

Description

Allow changing MTU size. Needs to be overridden for devices supporting jumbo frames.

void **ether_setup**(struct *net_device* * *dev*)
setup Ethernet network device

Parameters

struct net_device * dev network device

Description

Fill in the fields of the device structure with Ethernet-generic values.

struct *net_device* * **alloc_etherdev_mqs**(int *sizeof_priv*, unsigned int *txqs*, unsigned int *rxqs*)
Allocates and sets up an Ethernet device

Parameters

int sizeof_priv Size of additional driver-private structure to be allocated for this Ethernet device

unsigned int txqs The number of TX queues this device has.

unsigned int rxqs The number of RX queues this device has.

Description

Fill in the fields of the device structure with Ethernet-generic values. Basically does everything except registering the device.

Constructs a new net device, complete with a private data area of size (sizeof_priv). A 32-byte (not bit) alignment is enforced for this private data area.

void **netif_carrier_on**(struct *net_device* * *dev*)
set carrier

Parameters

struct net_device * dev network device

Description

Device has detected that carrier.

void **netif_carrier_off**(struct *net_device* * *dev*)
clear carrier

Parameters

struct net_device * dev network device

Description

Device has detected loss of carrier.

bool **is_link_local_ether_addr**(const u8 * *addr*)
Determine if given Ethernet address is link-local

Parameters

const u8 * addr Pointer to a six-byte array containing the Ethernet address

Description

Return true if address is link local reserved addr (01:80:c2:00:00:0X) per IEEE 802.1Q 8.6.3 Frame filtering.

Please note: addr must be aligned to u16.

bool **is_zero_ether_addr**(const u8 * *addr*)
Determine if give Ethernet address is all zeros.

Parameters

const u8 * addr Pointer to a six-byte array containing the Ethernet address

Description

Return true if the address is all zeroes.

Please note: addr must be aligned to u16.

bool **is_multicast_ether_addr**(const u8 * *addr*)
Determine if the Ethernet address is a multicast.

Parameters

const u8 * addr Pointer to a six-byte array containing the Ethernet address

Description

Return true if the address is a multicast address. By definition the broadcast address is also a multicast address.

bool **is_local_ether_addr**(const u8 * *addr*)
Determine if the Ethernet address is locally-assigned one (IEEE 802).

Parameters

const u8 * addr Pointer to a six-byte array containing the Ethernet address

Description

Return true if the address is a local address.

bool **is_broadcast_ether_addr**(const u8 * *addr*)
Determine if the Ethernet address is broadcast

Parameters

const u8 * addr Pointer to a six-byte array containing the Ethernet address

Description

Return true if the address is the broadcast address.

Please note: addr must be aligned to u16.

bool **is_unicast_ether_addr**(const u8 * *addr*)
Determine if the Ethernet address is unicast

Parameters

const u8 * addr Pointer to a six-byte array containing the Ethernet address

Description

Return true if the address is a unicast address.

bool **is_valid_ether_addr**(const u8 * *addr*)
Determine if the given Ethernet address is valid

Parameters

const u8 * addr Pointer to a six-byte array containing the Ethernet address

Description

Check that the Ethernet address (MAC) is not 00:00:00:00:00:00, is not a multicast address, and is not FF:FF:FF:FF:FF:FF.

Return true if the address is valid.

Please note: addr must be aligned to u16.

bool **eth_proto_is_802_3**(__be16 *proto*)
Determine if a given Ethertype/length is a protocol

Parameters

__be16 proto Ethertype/length value to be tested

Description

Check that the value from the Ethertype/length field is a valid Ethertype.

Return true if the valid is an 802.3 supported Ethertype.

void **eth_random_addr**(u8 * *addr*)
    Generate software assigned random Ethernet address

Parameters

u8 * addr Pointer to a six-byte array containing the Ethernet address

Description

Generate a random Ethernet address (MAC) that is not multicast and has the local assigned bit set.

void **eth_broadcast_addr**(u8 * *addr*)
    Assign broadcast address

Parameters

u8 * addr Pointer to a six-byte array containing the Ethernet address

Description

Assign the broadcast address to the given address array.

void **eth_zero_addr**(u8 * *addr*)
    Assign zero address

Parameters

u8 * addr Pointer to a six-byte array containing the Ethernet address

Description

Assign the zero address to the given address array.

void **eth_hw_addr_random**(struct *net_device* * *dev*)
    Generate software assigned random Ethernet and set device flag

Parameters

struct net_device * dev pointer to net_device structure

Description

Generate a random Ethernet address (MAC) to be used by a net device and set addr_assign_type so the state can be read by sysfs and be used by userspace.

void **ether_addr_copy**(u8 * *dst*, const u8 * *src*)
    Copy an Ethernet address

Parameters

u8 * dst Pointer to a six-byte array Ethernet address destination

const u8 * src Pointer to a six-byte array Ethernet address source

Description

Please note: dst & src must both be aligned to u16.

void **eth_hw_addr_inherit**(struct *net_device* * *dst*, struct *net_device* * *src*)
    Copy dev_addr from another net_device

Parameters

struct net_device * dst pointer to net_device to copy dev_addr to

struct net_device * src pointer to net_device to copy dev_addr from

Description

Copy the Ethernet address from one net_device to another along with the address attributes (addr_assign_type).

bool **ether_addr_equal**(const u8 * *addr1*, const u8 * *addr2*)
Compare two Ethernet addresses

Parameters

const u8 * addr1 Pointer to a six-byte array containing the Ethernet address

const u8 * addr2 Pointer other six-byte array containing the Ethernet address

Description

Compare two Ethernet addresses, returns true if equal

Please note: addr1 & addr2 must both be aligned to u16.

bool **ether_addr_equal_64bits**(const u8 *addr1*, const u8 *addr2*)
Compare two Ethernet addresses

Parameters

const u8 addr1 Pointer to an array of 8 bytes

const u8 addr2 Pointer to an other array of 8 bytes

Description

Compare two Ethernet addresses, returns true if equal, false otherwise.

The function doesn't need any conditional branches and possibly uses word memory accesses on CPU allowing cheap unaligned memory reads. arrays = { byte1, byte2, byte3, byte4, byte5, byte6, pad1, pad2 }

Please note that alignment of addr1 & addr2 are only guaranteed to be 16 bits.

bool **ether_addr_equal_unaligned**(const u8 * *addr1*, const u8 * *addr2*)
Compare two not u16 aligned Ethernet addresses

Parameters

const u8 * addr1 Pointer to a six-byte array containing the Ethernet address

const u8 * addr2 Pointer other six-byte array containing the Ethernet address

Description

Compare two Ethernet addresses, returns true if equal

Please note: Use only when any Ethernet address may not be u16 aligned.

bool **ether_addr_equal_masked**(const u8 * *addr1*, const u8 * *addr2*, const u8 * *mask*)
Compare two Ethernet addresses with a mask

Parameters

const u8 * addr1 Pointer to a six-byte array containing the 1st Ethernet address

const u8 * addr2 Pointer to a six-byte array containing the 2nd Ethernet address

const u8 * mask Pointer to a six-byte array containing the Ethernet address bitmask

Description

Compare two Ethernet addresses with a mask, returns true if for every bit set in the bitmask the equivalent bits in the ethernet addresses are equal. Using a mask with all bits set is a slower ether_addr_equal.

u64 **ether_addr_to_u64**(const u8 * *addr*)
Convert an Ethernet address into a u64 value.

Parameters

const u8 * addr Pointer to a six-byte array containing the Ethernet address

Description

Return a u64 value of the address

void **u64_to_ether_addr**(u64 *u*, u8 * *addr*)
Convert a u64 to an Ethernet address.

Parameters

u64 u u64 to convert to an Ethernet MAC address

u8 * addr Pointer to a six-byte array to contain the Ethernet address

void **eth_addr_dec**(u8 * *addr*)
Decrement the given MAC address

Parameters

u8 * addr Pointer to a six-byte array containing Ethernet address to decrement

bool **is_etherdev_addr**(const struct *net_device* * *dev*, const u8 *addr*)
Tell if given Ethernet address belongs to the device.

Parameters

const struct net_device * dev Pointer to a device structure

const u8 addr Pointer to a six-byte array containing the Ethernet address

Description

Compare passed address with all addresses of the device. Return true if the address if one of the device addresses.

Note that this function calls *ether_addr_equal_64bits()* so take care of the right padding.

unsigned long **compare_ether_header**(const void * *a*, const void * *b*)
Compare two Ethernet headers

Parameters

const void * a Pointer to Ethernet header

const void * b Pointer to Ethernet header

Description

Compare two Ethernet headers, returns 0 if equal. This assumes that the network header (i.e., IP header) is 4-byte aligned OR the platform can handle unaligned access. This is the case for all packets coming into netif_receive_skb or similar entry points.

int **eth_skb_pad**(struct *sk_buff* * *skb*)
Pad buffer to mininum number of octets for Ethernet frame

Parameters

struct sk_buff * skb Buffer to pad

Description

An Ethernet frame should have a minimum size of 60 bytes. This function takes short frames and pads them with zeros up to the 60 byte limit.

void **napi_schedule**(struct napi_struct * *n*)
schedule NAPI poll

Parameters

struct napi_struct * n NAPI context

Description

Schedule NAPI poll routine to be called if it is not already running.

void **napi_schedule_irqoff**(struct napi_struct * *n*)
    schedule NAPI poll

Parameters

struct napi_struct * n NAPI context

Description

Variant of *napi_schedule()*, assuming hard irqs are masked.

bool **napi_complete**(struct napi_struct * *n*)
    NAPI processing complete

Parameters

struct napi_struct * n NAPI context

Description

Mark NAPI processing as complete. Consider using `napi_complete_done()` instead. Return false if device should avoid rearming interrupts.

bool **napi_hash_del**(struct napi_struct * *napi*)
    remove a NAPI from global table

Parameters

struct napi_struct * napi NAPI context

Description

Warning: caller must observe RCU grace period before freeing memory containing **napi**, if this function returns true.

Note

core networking stack automatically calls it from *netif_napi_del()*. Drivers might want to call this helper to combine all the needed RCU grace periods into a single one.

void **napi_disable**(struct napi_struct * *n*)
    prevent NAPI from scheduling

Parameters

struct napi_struct * n NAPI context

Description

Stop NAPI from being scheduled on this context. Waits till any outstanding processing completes.

void **napi_enable**(struct napi_struct * *n*)
    enable NAPI scheduling

Parameters

struct napi_struct * n NAPI context

Description

Resume NAPI from being scheduled on this context. Must be paired with napi_disable.

void **napi_synchronize**(const struct napi_struct * *n*)
    wait until NAPI is not running

Parameters

const struct napi_struct * n NAPI context

Description

Wait until NAPI is done being scheduled on this context. Waits till any outstanding processing completes but does not disable future activations.

enum **netdev_priv_flags**
struct net_device priv_flags

Constants

IFF_802_1Q_VLAN 802.1Q VLAN device

IFF_EBRIDGE Ethernet bridging device

IFF_BONDING bonding master or slave

IFF_ISATAP ISATAP interface (RFC4214)

IFF_WAN_HDLC WAN HDLC device

IFF_XMIT_DST_RELEASE `dev_hard_start_xmit()` is allowed to release skb->dst

IFF_DONT_BRIDGE disallow bridging this ether dev

IFF_DISABLE_NETPOLL disable netpoll at run-time

IFF_MACVLAN_PORT device used as macvlan port

IFF_BRIDGE_PORT device used as bridge port

IFF_OVS_DATAPATH device used as Open vSwitch datapath port

IFF_TX_SKB_SHARING The interface supports sharing skbs on transmit

IFF_UNICAST_FLT Supports unicast filtering

IFF_TEAM_PORT device used as team port

IFF_SUPP_NOFCS device supports sending custom FCS

IFF_LIVE_ADDR_CHANGE device supports hardware address change when it's running

IFF_MACVLAN Macvlan device

IFF_XMIT_DST_RELEASE_PERM IFF_XMIT_DST_RELEASE not taking into account underlying stacked devices

IFF_IPVLAN_MASTER IPvlan master device

IFF_IPVLAN_SLAVE IPvlan slave device

IFF_L3MDEV_MASTER device is an L3 master device

IFF_NO_QUEUE device can run without qdisc attached

IFF_OPENVSWITCH device is a Open vSwitch master

IFF_L3MDEV_SLAVE device is enslaved to an L3 master device

IFF_TEAM device is a team device

IFF_RXFH_CONFIGURED device has had Rx Flow indirection table configured

IFF_PHONY_HEADROOM the headroom value is controlled by an external entity (i.e. the master device for bridged veth)

IFF_MACSEC device is a MACsec device

Description

These are the *struct net_device*, they are only set internally by drivers and used in the kernel. These flags are invisible to userspace; this means that the order of these flags can change during any kernel release.

You should have a pretty good reason to be extending these flags.

struct **net_device**
The DEVICE structure.

Definition

```
struct net_device {
  char name[IFNAMSIZ];
  struct hlist_node       name_hlist;
  struct dev_ifalias      __rcu *ifalias;
  unsigned long           mem_end;
  unsigned long           mem_start;
  unsigned long           base_addr;
  int irq;
  unsigned long           state;
  struct list_head        dev_list;
  struct list_head        napi_list;
  struct list_head        unreg_list;
  struct list_head        close_list;
  struct list_head        ptype_all;
  struct list_head        ptype_specific;
  struct {
    struct list_head upper;
    struct list_head lower;
  } adj_list;
  netdev_features_t features;
  netdev_features_t hw_features;
  netdev_features_t wanted_features;
  netdev_features_t vlan_features;
  netdev_features_t hw_enc_features;
  netdev_features_t mpls_features;
  netdev_features_t gso_partial_features;
  int ifindex;
  int group;
  struct net_device_stats stats;
  atomic_long_t rx_dropped;
  atomic_long_t tx_dropped;
  atomic_long_t rx_nohandler;
  atomic_t carrier_up_count;
  atomic_t carrier_down_count;
#ifdef CONFIG_WIRELESS_EXT;
  const struct iw_handler_def *wireless_handlers;
  struct iw_public_data   *wireless_data;
#endif;
  const struct net_device_ops *netdev_ops;
  const struct ethtool_ops *ethtool_ops;
#ifdef CONFIG_NET_SWITCHDEV;
  const struct switchdev_ops *switchdev_ops;
#endif;
#ifdef CONFIG_NET_L3_MASTER_DEV;
  const struct l3mdev_ops *l3mdev_ops;
#endif;
#if IS_ENABLED(CONFIG_IPV6);
  const struct ndisc_ops *ndisc_ops;
#endif;
#ifdef CONFIG_XFRM_OFFLOAD;
  const struct xfrmdev_ops *xfrmdev_ops;
#endif;
  const struct header_ops *header_ops;
  unsigned int            flags;
  unsigned int            priv_flags;
  unsigned short          gflags;
  unsigned short          padded;
  unsigned char           operstate;
  unsigned char           link_mode;
```

```
  unsigned char           if_port;
  unsigned char           dma;
  unsigned int            mtu;
  unsigned int            min_mtu;
  unsigned int            max_mtu;
  unsigned short          type;
  unsigned short          hard_header_len;
  unsigned char           min_header_len;
  unsigned short          needed_headroom;
  unsigned short          needed_tailroom;
  unsigned char           perm_addr[MAX_ADDR_LEN];
  unsigned char           addr_assign_type;
  unsigned char           addr_len;
  unsigned short          neigh_priv_len;
  unsigned short          dev_id;
  unsigned short          dev_port;
  spinlock_t addr_list_lock;
  unsigned char           name_assign_type;
  bool uc_promisc;
  struct netdev_hw_addr_list      uc;
  struct netdev_hw_addr_list      mc;
  struct netdev_hw_addr_list      dev_addrs;
#ifdef CONFIG_SYSFS;
  struct kset             *queues_kset;
#endif;
  unsigned int            promiscuity;
  unsigned int            allmulti;
#if IS_ENABLED(CONFIG_VLAN_8021Q);
  struct vlan_info __rcu  *vlan_info;
#endif;
#if IS_ENABLED(CONFIG_NET_DSA);
  struct dsa_port         *dsa_ptr;
#endif;
#if IS_ENABLED(CONFIG_TIPC);
  struct tipc_bearer __rcu *tipc_ptr;
#endif;
  void *atalk_ptr;
  struct in_device __rcu  *ip_ptr;
  struct dn_dev __rcu     *dn_ptr;
  struct inet6_dev __rcu  *ip6_ptr;
  void *ax25_ptr;
  struct wireless_dev     *ieee80211_ptr;
  struct wpan_dev         *ieee802154_ptr;
#if IS_ENABLED(CONFIG_MPLS_ROUTING);
  struct mpls_dev __rcu   *mpls_ptr;
#endif;
  unsigned char           *dev_addr;
  struct netdev_rx_queue  *_rx;
  unsigned int            num_rx_queues;
  unsigned int            real_num_rx_queues;
  struct bpf_prog __rcu   *xdp_prog;
  unsigned long           gro_flush_timeout;
  rx_handler_func_t __rcu *rx_handler;
  void __rcu              *rx_handler_data;
#ifdef CONFIG_NET_CLS_ACT;
  struct mini_Qdisc __rcu *miniq_ingress;
#endif;
  struct netdev_queue __rcu *ingress_queue;
#ifdef CONFIG_NETFILTER_INGRESS;
  struct nf_hook_entries __rcu *nf_hooks_ingress;
#endif;
  unsigned char           broadcast[MAX_ADDR_LEN];
#ifdef CONFIG_RFS_ACCEL;
```

```
  struct cpu_rmap         *rx_cpu_rmap;
#endif;
  struct hlist_node       index_hlist;
  struct netdev_queue     *_tx ____cacheline_aligned_in_smp;
  unsigned int            num_tx_queues;
  unsigned int            real_num_tx_queues;
  struct Qdisc            *qdisc;
#ifdef CONFIG_NET_SCHED;
  unsigned long qdisc_hash[1 << ((4) - 1)];
#endif;
  unsigned int            tx_queue_len;
  spinlock_t tx_global_lock;
  int watchdog_timeo;
#ifdef CONFIG_XPS;
  struct xps_dev_maps __rcu *xps_maps;
#endif;
#ifdef CONFIG_NET_CLS_ACT;
  struct mini_Qdisc __rcu *miniq_egress;
#endif;
  struct timer_list       watchdog_timer;
  int __percpu            *pcpu_refcnt;
  struct list_head        todo_list;
  struct list_head        link_watch_list;
  enum {
    NETREG_UNINITIALIZED=0,
    NETREG_REGISTERED,
    NETREG_UNREGISTERING,
    NETREG_UNREGISTERED,
    NETREG_RELEASED,
    NETREG_DUMMY,
  } reg_state:8;
  bool dismantle;
  enum {
    RTNL_LINK_INITIALIZED,
    RTNL_LINK_INITIALIZING,
  } rtnl_link_state:16;
  bool needs_free_netdev;
  void (*priv_destructor)(struct net_device *dev);
#ifdef CONFIG_NETPOLL;
  struct netpoll_info __rcu       *npinfo;
#endif;
  possible_net_t nd_net;
  union {
    void *ml_priv;
    struct pcpu_lstats __percpu             *lstats;
    struct pcpu_sw_netstats __percpu        *tstats;
    struct pcpu_dstats __percpu             *dstats;
    struct pcpu_vstats __percpu             *vstats;
  };
#if IS_ENABLED(CONFIG_GARP);
  struct garp_port __rcu  *garp_port;
#endif;
#if IS_ENABLED(CONFIG_MRP);
  struct mrp_port __rcu   *mrp_port;
#endif;
  struct device           dev;
  const struct attribute_group *sysfs_groups[4];
  const struct attribute_group *sysfs_rx_queue_group;
  const struct rtnl_link_ops *rtnl_link_ops;
#define GSO_MAX_SIZE            65536;
  unsigned int            gso_max_size;
#define GSO_MAX_SEGS            65535;
  u16 gso_max_segs;
```

```
#ifdef CONFIG_DCB;
  const struct dcbnl_rtnl_ops *dcbnl_ops;
#endif;
  u8 num_tc;
  struct netdev_tc_txq    tc_to_txq[TC_MAX_QUEUE];
  u8 prio_tc_map[TC_BITMASK + 1];
#if IS_ENABLED(CONFIG_FCOE);
  unsigned int           fcoe_ddp_xid;
#endif;
#if IS_ENABLED(CONFIG_CGROUP_NET_PRIO);
  struct netprio_map __rcu *priomap;
#endif;
  struct phy_device       *phydev;
  struct lock_class_key   *qdisc_tx_busylock;
  struct lock_class_key   *qdisc_running_key;
  bool proto_down;
};
```

Members

name This is the first field of the "visible" part of this structure (i.e. as seen by users in the "Space.c" file). It is the name of the interface.

name_hlist Device name hash chain, please keep it close to name[]

ifalias SNMP alias

mem_end Shared memory end

mem_start Shared memory start

base_addr Device I/O address

irq Device IRQ number

state Generic network queuing layer state, see netdev_state_t

dev_list The global list of network devices

napi_list List entry used for polling NAPI devices

unreg_list List entry when we are unregistering the device; see the function unregister_netdev

close_list List entry used when we are closing the device

ptype_all Device-specific packet handlers for all protocols

ptype_specific Device-specific, protocol-specific packet handlers

adj_list Directly linked devices, like slaves for bonding

features Currently active device features

hw_features User-changeable features

wanted_features User-requested features

vlan_features Mask of features inheritable by VLAN devices

hw_enc_features Mask of features inherited by encapsulating devices This field indicates what encapsulation offloads the hardware is capable of doing, and drivers will need to set them appropriately.

mpls_features Mask of features inheritable by MPLS

ifindex interface index

group The group the device belongs to

stats Statistics struct, which was left as a legacy, use rtnl_link_stats64 instead

rx_dropped Dropped packets by core network, do not use this in drivers

tx_dropped Dropped packets by core network, do not use this in drivers

rx_nohandler nohandler dropped packets by core network on inactive devices, do not use this in drivers

carrier_up_count Number of times the carrier has been up

carrier_down_count Number of times the carrier has been down

wireless_handlers List of functions to handle Wireless Extensions, instead of ioctl, see <net/iw_handler.h> for details.

wireless_data Instance data managed by the core of wireless extensions

netdev_ops Includes several pointers to callbacks, if one wants to override the ndo_*() functions

ethtool_ops Management operations

ndisc_ops Includes callbacks for different IPv6 neighbour discovery handling. Necessary for e.g. 6LoWPAN.

header_ops Includes callbacks for creating,parsing,caching,etc of Layer 2 headers.

flags Interface flags (a la BSD)

priv_flags Like 'flags' but invisible to userspace, see if.h for the definitions

gflags Global flags (kept as legacy)

padded How much padding added by `alloc_netdev()`

operstate RFC2863 operstate

link_mode Mapping policy to operstate

if_port Selectable AUI, TP, …

dma DMA channel

mtu Interface MTU value

min_mtu Interface Minimum MTU value

max_mtu Interface Maximum MTU value

type Interface hardware type

hard_header_len Maximum hardware header length.

min_header_len Minimum hardware header length

needed_headroom Extra headroom the hardware may need, but not in all cases can this be guaranteed

needed_tailroom Extra tailroom the hardware may need, but not in all cases can this be guaranteed. Some cases also use LL_MAX_HEADER instead to allocate the skb

perm_addr Permanent hw address

addr_assign_type Hw address assignment type

addr_len Hardware address length

neigh_priv_len Used in `neigh_alloc()`

dev_id Used to differentiate devices that share the same link layer address

dev_port Used to differentiate devices that share the same function

addr_list_lock XXX: need comments on this one

uc_promisc Counter that indicates promiscuous mode has been enabled due to the need to listen to additional unicast addresses in a device that does not implement `ndo_set_rx_mode()`

uc unicast mac addresses

mc multicast mac addresses

dev_addrs list of device hw addresses

queues_kset Group of all Kobjects in the Tx and RX queues

promiscuity Number of times the NIC is told to work in promiscuous mode; if it becomes 0 the NIC will exit promiscuous mode

allmulti Counter, enables or disables allmulticast mode

vlan_info VLAN info

dsa_ptr dsa specific data

tipc_ptr TIPC specific data

atalk_ptr AppleTalk link

ip_ptr IPv4 specific data

dn_ptr DECnet specific data

ip6_ptr IPv6 specific data

ax25_ptr AX.25 specific data

ieee80211_ptr IEEE 802.11 specific data, assign before registering

dev_addr Hw address (before bcast, because most packets are unicast)

_rx Array of RX queues

num_rx_queues Number of RX queues allocated at *register_netdev()* time

real_num_rx_queues Number of RX queues currently active in device

rx_handler handler for received packets

rx_handler_data XXX: need comments on this one

miniq_ingress ingress/clsact qdisc specific data for ingress processing

ingress_queue XXX: need comments on this one

broadcast hw bcast address

rx_cpu_rmap CPU reverse-mapping for RX completion interrupts, indexed by RX queue number. Assigned by driver. This must only be set if the ndo_rx_flow_steer operation is defined

index_hlist Device index hash chain

num_tx_queues Number of TX queues allocated at `alloc_netdev_mq()` time

real_num_tx_queues Number of TX queues currently active in device

qdisc Root qdisc from userspace point of view

tx_queue_len Max frames per queue allowed

tx_global_lock XXX: need comments on this one

watchdog_timeo Represents the timeout that is used by the watchdog (see `dev_watchdog()`)

xps_maps XXX: need comments on this one

miniq_egress clsact qdisc specific data for egress processing

watchdog_timer List of timers

pcpu_refcnt Number of references to this device

todo_list Delayed register/unregister

link_watch_list XXX: need comments on this one

reg_state Register/unregister state machine

dismantle Device is going to be freed

rtnl_link_state This enum represents the phases of creating a new link

needs_free_netdev Should unregister perform free_netdev?

priv_destructor Called from unregister

npinfo XXX: need comments on this one

nd_net Network namespace this network device is inside

{unnamed_union} anonymous

ml_priv Mid-layer private

lstats Loopback statistics

tstats Tunnel statistics

dstats Dummy statistics

vstats Virtual ethernet statistics

garp_port GARP

mrp_port MRP

dev Class/net/name entry

sysfs_groups Space for optional device, statistics and wireless sysfs groups

sysfs_rx_queue_group Space for optional per-rx queue attributes

rtnl_link_ops Rtnl_link_ops

gso_max_size Maximum size of generic segmentation offload

gso_max_segs Maximum number of segments that can be passed to the NIC for GSO

dcbnl_ops Data Center Bridging netlink ops

num_tc Number of traffic classes in the net device

tc_to_txq XXX: need comments on this one

prio_tc_map XXX: need comments on this one

fcoe_ddp_xid Max exchange id for FCoE LRO by ddp

priomap XXX: need comments on this one

phydev Physical device may attach itself for hardware timestamping

qdisc_tx_busylock lockdep class annotating Qdisc->busylock spinlock

qdisc_running_key lockdep class annotating Qdisc->running seqcount

proto_down protocol port state information can be sent to the switch driver and used to set the phys state of the switch port.

Description

Actually, this whole structure is a big mistake. It mixes I/O data with strictly "high-level" data, and it has to know about almost every data structure used in the INET module.

interface address info:

FIXME: cleanup struct net_device such that network protocol info moves out.

void * **netdev_priv**(const struct *net_device* * *dev*)
access network device private data

Parameters

const struct net_device * dev network device

Description

Get network device private data

void **netif_napi_add**(struct *net_device* * *dev*, struct napi_struct * *napi*, int (*poll) (struct napi_struct *, int, int *weight*)
initialize a NAPI context

Parameters

struct net_device * dev network device

struct napi_struct * napi NAPI context

int (*)(struct napi_struct *, int) poll polling function

int weight default weight

Description

netif_napi_add() must be used to initialize a NAPI context prior to calling *any* of the other NAPI-related functions.

void **netif_tx_napi_add**(struct *net_device* * *dev*, struct napi_struct * *napi*, int (*poll) (struct napi_struct *, int, int *weight*)
initialize a NAPI context

Parameters

struct net_device * dev network device

struct napi_struct * napi NAPI context

int (*)(struct napi_struct *, int) poll polling function

int weight default weight

Description

This variant of *netif_napi_add()* should be used from drivers using NAPI to exclusively poll a TX queue. This will avoid we add it into napi_hash[], thus polluting this hash table.

void **netif_napi_del**(struct napi_struct * *napi*)
remove a NAPI context

Parameters

struct napi_struct * napi NAPI context

Description

netif_napi_del() removes a NAPI context from the network device NAPI list

void **netif_start_queue**(struct *net_device* * *dev*)
allow transmit

Parameters

struct net_device * dev network device

Description

Allow upper layers to call the device hard_start_xmit routine.

void **netif_wake_queue**(struct *net_device* * *dev*)
restart transmit

Parameters

struct net_device * dev network device

Description

Allow upper layers to call the device hard_start_xmit routine. Used for flow control when transmit resources are available.

void **netif_stop_queue**(struct *net_device* * *dev*)
stop transmitted packets

Parameters

struct net_device * dev network device

Description

Stop upper layers calling the device hard_start_xmit routine. Used for flow control when transmit resources are unavailable.

bool **netif_queue_stopped**(const struct *net_device* * *dev*)
test if transmit queue is flowblocked

Parameters

const struct net_device * dev network device

Description

Test if transmit queue on device is currently unable to send.

void **netdev_txq_bql_enqueue_prefetchw**(struct netdev_queue * *dev_queue*)
prefetch bql data for write

Parameters

struct netdev_queue * dev_queue pointer to transmit queue

Description

BQL enabled drivers might use this helper in their `ndo_start_xmit()`, to give appropriate hint to the CPU.

void **netdev_txq_bql_complete_prefetchw**(struct netdev_queue * *dev_queue*)
prefetch bql data for write

Parameters

struct netdev_queue * dev_queue pointer to transmit queue

Description

BQL enabled drivers might use this helper in their TX completion path, to give appropriate hint to the CPU.

void **netdev_sent_queue**(struct *net_device* * *dev*, unsigned int *bytes*)
report the number of bytes queued to hardware

Parameters

struct net_device * dev network device

unsigned int bytes number of bytes queued to the hardware device queue

Description

Report the number of bytes queued for sending/completion to the network device hardware queue. **bytes** should be a good approximation and should exactly match *netdev_completed_queue()* **bytes**

void **netdev_completed_queue**(struct *net_device* * *dev*, unsigned int *pkts*, unsigned int *bytes*)
report bytes and packets completed by device

Parameters

struct net_device * dev network device

unsigned int pkts actual number of packets sent over the medium

unsigned int bytes actual number of bytes sent over the medium

Description

Report the number of bytes and packets transmitted by the network device hardware queue over the physical medium, **bytes** must exactly match the **bytes** amount passed to *netdev_sent_queue()*

void **netdev_reset_queue**(struct *net_device* * *dev_queue*)
reset the packets and bytes count of a network device

Parameters

struct net_device * dev_queue network device

Description

Reset the bytes and packet count of a network device and clear the software flow control OFF bit for this network device

u16 **netdev_cap_txqueue**(struct *net_device* * *dev*, u16 *queue_index*)
check if selected tx queue exceeds device queues

Parameters

struct net_device * dev network device

u16 queue_index given tx queue index

Description

Returns 0 if given tx queue index >= number of device tx queues, otherwise returns the originally passed tx queue index.

bool **netif_running**(const struct *net_device* * *dev*)
test if up

Parameters

const struct net_device * dev network device

Description

Test if the device has been brought up.

void **netif_start_subqueue**(struct *net_device* * *dev*, u16 *queue_index*)
allow sending packets on subqueue

Parameters

struct net_device * dev network device

u16 queue_index sub queue index

Description

Start individual transmit queue of a device with multiple transmit queues.

void **netif_stop_subqueue**(struct *net_device* * *dev*, u16 *queue_index*)
stop sending packets on subqueue

Parameters

struct net_device * dev network device

u16 queue_index sub queue index

Description

Stop individual transmit queue of a device with multiple transmit queues.

bool **__netif_subqueue_stopped**(const struct *net_device* * *dev*, u16 *queue_index*)
test status of subqueue

Parameters

const struct net_device * dev network device

u16 queue_index sub queue index

Description

Check individual transmit queue of a device with multiple transmit queues.

void **netif_wake_subqueue**(struct *net_device* * *dev*, u16 *queue_index*)
allow sending packets on subqueue

Parameters

struct net_device * dev network device

u16 queue_index sub queue index

Description

Resume individual transmit queue of a device with multiple transmit queues.

bool **netif_is_multiqueue**(const struct *net_device* * *dev*)
test if device has multiple transmit queues

Parameters

const struct net_device * dev network device

Description

Check if device has multiple transmit queues

void **dev_put**(struct *net_device* * *dev*)
release reference to device

Parameters

struct net_device * dev network device

Description

Release reference to device to allow it to be freed.

void **dev_hold**(struct *net_device* * *dev*)
get reference to device

Parameters

struct net_device * dev network device

Description

Hold reference to device to keep it from being freed.

bool **netif_carrier_ok**(const struct *net_device* * *dev*)
test if carrier present

Parameters

const struct net_device * dev network device

Description

Check if carrier is present on device

void **netif_dormant_on**(struct *net_device* * *dev*)
mark device as dormant.

Parameters

struct net_device * dev network device

Description

Mark device as dormant (as per RFC2863).

The dormant state indicates that the relevant interface is not actually in a condition to pass packets (i.e., it is not 'up') but is in a "pending" state, waiting for some external event. For "on- demand" interfaces, this new state identifies the situation where the interface is waiting for events to place it in the up state.

void **netif_dormant_off**(struct *net_device* * *dev*)
set device as not dormant.

Parameters

struct net_device * dev network device

Description

Device is not in dormant state.

bool **netif_dormant**(const struct *net_device* * *dev*)
test if device is dormant

Parameters

const struct net_device * dev network device

Description

Check if device is dormant.

bool **netif_oper_up**(const struct *net_device* * *dev*)
test if device is operational

Parameters

const struct net_device * dev network device

Description

Check if carrier is operational

bool **netif_device_present**(struct *net_device* * *dev*)
is device available or removed

Parameters

struct net_device * dev network device

Description

Check if device has not been removed from system.

void **netif_tx_lock**(struct *net_device* * *dev*)
grab network device transmit lock

Parameters

struct net_device * dev network device

Description

Get network device transmit lock

int **__dev_uc_sync**(struct *net_device* * *dev*, int (*sync) (struct *net_device* *, const unsigned char *, int (*unsync) (struct *net_device* *, const unsigned char *)
Synchonize device's unicast list

Parameters

struct net_device * dev device to sync

int (*)(struct net_device *, const unsigned char *) sync function to call if address should be added

int (*)(struct net_device *, const unsigned char *) unsync function to call if address should be removed

Description

Add newly added addresses to the interface, and release addresses that have been deleted.

void **__dev_uc_unsync**(struct *net_device* * *dev*, int (*unsync) (struct *net_device* *, const unsigned char *)
Remove synchronized addresses from device

Parameters

struct net_device * dev device to sync

int (*)(struct net_device *, const unsigned char *) unsync function to call if address should be removed

Description

Remove all addresses that were added to the device by `dev_uc_sync()`.

int **__dev_mc_sync**(struct *net_device* * *dev*, int (*sync) (struct *net_device* *, const unsigned char *, int (*unsync) (struct *net_device* *, const unsigned char *)
Synchonize device's multicast list

Parameters

struct net_device * dev device to sync

int (*)(struct net_device *, const unsigned char *) sync function to call if address should be added

int (*)(struct net_device *, const unsigned char *) unsync function to call if address should be removed

Description

Add newly added addresses to the interface, and release addresses that have been deleted.

void **__dev_mc_unsync**(struct *net_device* * *dev*, int (*unsync) (struct *net_device* *, const unsigned char *)
Remove synchronized addresses from device

Parameters

struct net_device * dev device to sync

int (*)(struct net_device *, const unsigned char *) unsync function to call if address should be removed

Description

Remove all addresses that were added to the device by `dev_mc_sync()`.

PHY Support

void **phy_print_status**(struct phy_device * *phydev*)
Convenience function to print out the current phy status

Parameters

struct phy_device * phydev the phy_device struct

int **phy_restart_aneg**(struct phy_device * *phydev*)
restart auto-negotiation

Parameters

struct phy_device * phydev target phy_device struct

Description

Restart the autonegotiation on **phydev**. Returns >= 0 on success or negative errno on error.

int **phy_aneg_done**(struct phy_device * *phydev*)
    return auto-negotiation status

Parameters

struct phy_device * phydev target phy_device struct

Description

Return the auto-negotiation status from this **phydev** Returns > 0 on success or < 0 on error. 0 means that auto-negotiation is still pending.

int **phy_ethtool_sset**(struct phy_device * *phydev*, struct ethtool_cmd * *cmd*)
    generic ethtool sset function, handles all the details

Parameters

struct phy_device * phydev target phy_device struct

struct ethtool_cmd * cmd ethtool_cmd

Description

A few notes about parameter checking:

- We don't set port or transceiver, so we don't care what they were set to.
- *phy_start_aneg()* will make sure forced settings are sane, and choose the next best ones from the ones selected, so we don't care if ethtool tries to give us bad values.

int **phy_mii_ioctl**(struct phy_device * *phydev*, struct ifreq * *ifr*, int *cmd*)
    generic PHY MII ioctl interface

Parameters

struct phy_device * phydev the phy_device struct

struct ifreq * ifr `struct ifreq` for socket ioctl's

int cmd ioctl cmd to execute

Description

Note that this function is currently incompatible with the PHYCONTROL layer. It changes registers without regard to current state. Use at own risk.

int **phy_start_aneg**(struct phy_device * *phydev*)
    start auto-negotiation for this PHY device

Parameters

struct phy_device * phydev the phy_device struct

Description

Sanitizes the settings (if we're not autonegotiating them), and then calls the driver's config_aneg
    function. If the PHYCONTROL Layer is operating, we change the state to reflect the beginning of Auto-negotiation or forcing.

void **phy_start_machine**(struct phy_device * *phydev*)
    start PHY state machine tracking

Parameters

struct phy_device * phydev the phy_device struct

Description

The PHY infrastructure can run a state machine which tracks whether the PHY is starting up, negotiating, etc. This function starts the delayed workqueue which tracks the state of the PHY. If you want to maintain your own state machine, do not call this function.

int **phy_start_interrupts**(struct phy_device * *phydev*)
request and enable interrupts for a PHY device

Parameters

struct phy_device * phydev target phy_device struct

Description

Request the interrupt for the given PHY. If this fails, then we set irq to PHY_POLL. Otherwise, we enable the interrupts in the PHY. This should only be called with a valid IRQ number. Returns 0 on success or < 0 on error.

int **phy_stop_interrupts**(struct phy_device * *phydev*)
disable interrupts from a PHY device

Parameters

struct phy_device * phydev target phy_device struct

void **phy_stop**(struct phy_device * *phydev*)
Bring down the PHY link, and stop checking the status

Parameters

struct phy_device * phydev target phy_device struct

void **phy_start**(struct phy_device * *phydev*)
start or restart a PHY device

Parameters

struct phy_device * phydev target phy_device struct

Description

Indicates the attached device's readiness to handle PHY-related work. Used during startup to start the PHY, and after a call to *phy_stop()* to resume operation. Also used to indicate the MDIO bus has cleared an error condition.

void **phy_mac_interrupt**(struct phy_device * *phydev*)
MAC says the link has changed

Parameters

struct phy_device * phydev phy_device struct with changed link

Description

The MAC layer is able to indicate there has been a change in the PHY link status. Trigger the state machine and work a work queue.

int **phy_init_eee**(struct phy_device * *phydev*, bool *clk_stop_enable*)
init and check the EEE feature

Parameters

struct phy_device * phydev target phy_device struct

bool clk_stop_enable PHY may stop the clock during LPI

Description

it checks if the Energy-Efficient Ethernet (EEE) is supported by looking at the MMD registers 3.20 and 7.60/61 and it programs the MMD register 3.0 setting the "Clock stop enable" bit if required.

int **phy_get_eee_err**(struct phy_device * *phydev*)
report the EEE wake error count

Parameters

struct phy_device * phydev target phy_device struct

Description

it is to report the number of time where the PHY failed to complete its normal wake sequence.

int **phy_ethtool_get_eee**(struct phy_device * *phydev*, struct ethtool_eee * *data*)
get EEE supported and status

Parameters

struct phy_device * phydev target phy_device struct

struct ethtool_eee * data ethtool_eee data

Description

it reportes the Supported/Advertisement/LP Advertisement capabilities.

int **phy_ethtool_set_eee**(struct phy_device * *phydev*, struct ethtool_eee * *data*)
set EEE supported and status

Parameters

struct phy_device * phydev target phy_device struct

struct ethtool_eee * data ethtool_eee data

Description

it is to program the Advertisement EEE register.

int **phy_clear_interrupt**(struct phy_device * *phydev*)
Ack the phy device's interrupt

Parameters

struct phy_device * phydev the phy_device struct

Description

If the **phydev** driver has an ack_interrupt function, call it to ack and clear the phy device's interrupt.

Returns 0 on success or < 0 on error.

int **phy_config_interrupt**(struct phy_device * *phydev*, u32 *interrupts*)
configure the PHY device for the requested interrupts

Parameters

struct phy_device * phydev the phy_device struct

u32 interrupts interrupt flags to configure for this **phydev**

Description

Returns 0 on success or < 0 on error.

const struct phy_setting * **phy_find_valid**(int *speed*, int *duplex*, u32 *supported*)
find a PHY setting that matches the requested parameters

Parameters

int speed desired speed

int duplex desired duplex

u32 supported mask of supported link modes

Description

Locate a supported phy setting that is, in priority order: - an exact match for the specified speed and duplex mode - a match for the specified speed, or slower speed - the slowest supported speed Returns the matched phy_setting entry, or NULL if no supported phy settings were found.

unsigned int **phy_supported_speeds**(struct phy_device * *phy*, unsigned int * *speeds*, unsigned int *size*)
return all speeds currently supported by a phy device

Parameters

struct phy_device * phy The phy device to return supported speeds of.

unsigned int * speeds buffer to store supported speeds in.

unsigned int size size of speeds buffer.

Description

Returns the number of supported speeds, and fills the speeds buffer with the supported speeds. If speeds buffer is too small to contain all currently supported speeds, will return as many speeds as can fit.

bool **phy_check_valid**(int *speed*, int *duplex*, u32 *features*)
check if there is a valid PHY setting which matches speed, duplex, and feature mask

Parameters

int speed speed to match

int duplex duplex to match

u32 features A mask of the valid settings

Description

Returns true if there is a valid setting, false otherwise.

void **phy_sanitize_settings**(struct phy_device * *phydev*)
make sure the PHY is set to supported speed and duplex

Parameters

struct phy_device * phydev the target phy_device struct

Description

Make sure the PHY is set to supported speeds and duplexes. Drop down by one in this order: 1000/FULL, 1000/HALF, 100/FULL, 100/HALF, 10/FULL, 10/HALF.

int **phy_start_aneg_priv**(struct phy_device * *phydev*, bool *sync*)
start auto-negotiation for this PHY device

Parameters

struct phy_device * phydev the phy_device struct

bool sync indicate whether we should wait for the workqueue cancelation

Description

Sanitizes the settings (if we're not autonegotiating them), and then calls the driver's config_aneg function. If the PHYCONTROL Layer is operating, we change the state to reflect the beginning of Auto-negotiation or forcing.

void **phy_trigger_machine**(struct phy_device * *phydev*, bool *sync*)
trigger the state machine to run

Parameters

struct phy_device * phydev the phy_device struct

bool sync indicate whether we should wait for the workqueue cancelation

Description

There has been a change in state which requires that the state machine runs.

void **phy_stop_machine**(struct phy_device * *phydev*)
stop the PHY state machine tracking

Parameters

struct phy_device * phydev target phy_device struct

Description

Stops the state machine delayed workqueue, sets the state to UP (unless it wasn't up yet). This function must be called BEFORE phy_detach.

void **phy_error**(struct phy_device * *phydev*)
enter HALTED state for this PHY device

Parameters

struct phy_device * phydev target phy_device struct

Description

Moves the PHY to the HALTED state in response to a read or write error, and tells the controller the link is down. Must not be called from interrupt context, or while the phydev->lock is held.

irqreturn_t **phy_interrupt**(int *irq*, void * *phy_dat*)
PHY interrupt handler

Parameters

int irq interrupt line

void * phy_dat phy_device pointer

Description

When a PHY interrupt occurs, the handler disables interrupts, and uses phy_change to handle the interrupt.

int **phy_enable_interrupts**(struct phy_device * *phydev*)
Enable the interrupts from the PHY side

Parameters

struct phy_device * phydev target phy_device struct

int **phy_disable_interrupts**(struct phy_device * *phydev*)
Disable the PHY interrupts from the PHY side

Parameters

struct phy_device * phydev target phy_device struct

void **phy_change**(struct phy_device * *phydev*)
Called by the phy_interrupt to handle PHY changes

Parameters

struct phy_device * phydev phy_device struct that interrupted

void **phy_change_work**(struct work_struct * *work*)
Scheduled by the phy_mac_interrupt to handle PHY changes

Parameters

struct work_struct * work work_struct that describes the work to be done

void **phy_state_machine**(struct work_struct * *work*)
Handle the state machine

Parameters

struct work_struct * work work_struct that describes the work to be done

int **phy_register_fixup**(const char * *bus_id*, u32 *phy_uid*, u32 *phy_uid_mask*, int (*run) (struct phy_device *)
creates a new phy_fixup and adds it to the list

Parameters

const char * bus_id A string which matches phydev->mdio.dev.bus_id (or PHY_ANY_ID)

u32 phy_uid Used to match against phydev->phy_id (the UID of the PHY) It can also be PHY_ANY_UID

u32 phy_uid_mask Applied to phydev->phy_id and fixup->phy_uid before comparison

int (*)(struct phy_device *) run The actual code to be run when a matching PHY is found

int **phy_unregister_fixup**(const char * *bus_id*, u32 *phy_uid*, u32 *phy_uid_mask*)
remove a phy_fixup from the list

Parameters

const char * bus_id A string matches fixup->bus_id (or PHY_ANY_ID) in phy_fixup_list

u32 phy_uid A phy id matches fixup->phy_id (or PHY_ANY_UID) in phy_fixup_list

u32 phy_uid_mask Applied to phy_uid and fixup->phy_uid before comparison

struct phy_device * **get_phy_device**(struct mii_bus * *bus*, int *addr*, bool *is_c45*)
reads the specified PHY device and returns its **phy_device** struct

Parameters

struct mii_bus * bus the target MII bus

int addr PHY address on the MII bus

bool is_c45 If true the PHY uses the 802.3 clause 45 protocol

Description

Reads the ID registers of the PHY at addr on the bus, then allocates and returns the phy_device to represent it.

int **phy_device_register**(struct phy_device * *phydev*)
Register the phy device on the MDIO bus

Parameters

struct phy_device * phydev phy_device structure to be added to the MDIO bus

void **phy_device_remove**(struct phy_device * *phydev*)
Remove a previously registered phy device from the MDIO bus

Parameters

struct phy_device * phydev phy_device structure to remove

Description

This doesn't free the phy_device itself, it merely reverses the effects of *phy_device_register()*. Use phy_device_free() to free the device after calling this function.

struct phy_device * **phy_find_first**(struct mii_bus * *bus*)
finds the first PHY device on the bus

Parameters

struct mii_bus * bus the target MII bus

int **phy_connect_direct**(struct *net_device* * *dev*, struct phy_device * *phydev*, void (*handler) (struct *net_device* *, phy_interface_t *interface*)
connect an ethernet device to a specific phy_device

Parameters

struct net_device * dev the network device to connect

struct phy_device * phydev the pointer to the phy device

void (*)(struct net_device *) handler callback function for state change notifications

phy_interface_t interface PHY device's interface

struct phy_device * **phy_connect**(struct *net_device* * *dev*, const char * *bus_id*, void (*handler) (struct *net_device* *, phy_interface_t *interface*)
connect an ethernet device to a PHY device

Parameters

struct net_device * dev the network device to connect

const char * bus_id the id string of the PHY device to connect

void (*)(struct net_device *) handler callback function for state change notifications

phy_interface_t interface PHY device's interface

Description

Convenience function for connecting ethernet devices to PHY devices. The default behavior is for the PHY infrastructure to handle everything, and only notify the connected driver when the link status changes. If you don't want, or can't use the provided functionality, you may choose to call only the subset of functions which provide the desired functionality.

void **phy_disconnect**(struct phy_device * *phydev*)
disable interrupts, stop state machine, and detach a PHY device

Parameters

struct phy_device * phydev target phy_device struct

int **phy_attach_direct**(struct *net_device* * *dev*, struct phy_device * *phydev*, u32 *flags*, phy_interface_t *interface*)
attach a network device to a given PHY device pointer

Parameters

struct net_device * dev network device to attach

struct phy_device * phydev Pointer to phy_device to attach

u32 flags PHY device's dev_flags

phy_interface_t interface PHY device's interface

Description

Called by drivers to attach to a particular PHY device. The phy_device is found, and properly hooked up to the phy_driver. If no driver is attached, then a generic driver is used. The phy_device is given a ptr to the attaching device, and given a callback for link status change. The phy_device is returned to the attaching driver. This function takes a reference on the phy device.

struct phy_device * **phy_attach**(struct *net_device* * *dev*, const char * *bus_id*, phy_interface_t *interface*)
attach a network device to a particular PHY device

Parameters

struct net_device * dev network device to attach

const char * bus_id Bus ID of PHY device to attach

phy_interface_t interface PHY device's interface

Description

Same as *phy_attach_direct()* except that a PHY bus_id string is passed instead of a pointer to a struct phy_device.

void **phy_detach**(struct phy_device * *phydev*)
detach a PHY device from its network device

Parameters

struct phy_device * phydev target phy_device struct

Description

This detaches the phy device from its network device and the phy driver, and drops the reference count taken in *phy_attach_direct()*.

int **phy_reset_after_clk_enable**(struct phy_device * *phydev*)
perform a PHY reset if needed

Parameters

struct phy_device * phydev target phy_device struct

Description

Some PHYs are known to need a reset after their refclk was enabled. This function evaluates the flags and perform the reset if it's needed. Returns < 0 on error, 0 if the phy wasn't reset and 1 if the phy was reset.

int **genphy_setup_forced**(struct phy_device * *phydev*)
configures/forces speed/duplex from **phydev**

Parameters

struct phy_device * phydev target phy_device struct

Description

Configures MII_BMCR to force speed/duplex to the values in phydev. Assumes that the values are valid. Please see *phy_sanitize_settings()*.

int **genphy_restart_aneg**(struct phy_device * *phydev*)
Enable and Restart Autonegotiation

Parameters

struct phy_device * phydev target phy_device struct

int **genphy_config_aneg**(struct phy_device * *phydev*)
restart auto-negotiation or write BMCR

Parameters

struct phy_device * phydev target phy_device struct

Description

If auto-negotiation is enabled, we configure the advertising, and then restart auto-negotiation. If it is not enabled, then we write the BMCR.

int **genphy_aneg_done**(struct phy_device * *phydev*)
return auto-negotiation status

Parameters

struct phy_device * phydev target phy_device struct

Description

Reads the status register and returns 0 either if auto-negotiation is incomplete, or if there was an error. Returns BMSR_ANEGCOMPLETE if auto-negotiation is done.

int **genphy_update_link**(struct phy_device * *phydev*)
update link status in **phydev**

Parameters

struct phy_device * phydev target phy_device struct

Description

Update the value in phydev->link to reflect the current link value. In order to do this, we need to read the status register twice, keeping the second value.

int **genphy_read_status**(struct phy_device * *phydev*)
check the link status and update current link state

Parameters

struct phy_device * phydev target phy_device struct

Description

Check the link, then figure out the current state by comparing what we advertise with what the link partner advertises. Start by checking the gigabit possibilities, then move on to 10/100.

int **genphy_soft_reset**(struct phy_device * *phydev*)
software reset the PHY via BMCR_RESET bit

Parameters

struct phy_device * phydev target phy_device struct

Description

Perform a software PHY reset using the standard BMCR_RESET bit and poll for the reset bit to be cleared.

Return

0 on success, < 0 on failure

int **phy_driver_register**(struct phy_driver * *new_driver*, struct module * *owner*)
register a phy_driver with the PHY layer

Parameters

struct phy_driver * new_driver new phy_driver to register

struct module * owner module owning this PHY

int **get_phy_c45_ids**(struct mii_bus * *bus*, int *addr*, u32 * *phy_id*, struct phy_c45_device_ids * *c45_ids*)
reads the specified addr for its 802.3-c45 IDs.

Parameters

struct mii_bus * bus the target MII bus

int addr PHY address on the MII bus

u32 * phy_id where to store the ID retrieved.

struct phy_c45_device_ids * c45_ids where to store the c45 ID information.

Description

If the PHY devices-in-package appears to be valid, it and the corresponding identifiers are stored in **c45_ids**, zero is stored in **phy_id**. Otherwise 0xffffffff is stored in **phy_id**. Returns zero on success.

int **get_phy_id**(struct mii_bus * *bus*, int *addr*, u32 * *phy_id*, bool *is_c45*, struct phy_c45_device_ids * *c45_ids*)
reads the specified addr for its ID.

Parameters

struct mii_bus * bus the target MII bus

int addr PHY address on the MII bus

u32 * phy_id where to store the ID retrieved.

bool is_c45 If true the PHY uses the 802.3 clause 45 protocol

struct phy_c45_device_ids * c45_ids where to store the c45 ID information.

Description

In the case of a 802.3-c22 PHY, reads the ID registers of the PHY at **addr** on the **bus**, stores it in **phy_id** and returns zero on success.

In the case of a 802.3-c45 PHY, *get_phy_c45_ids()* is invoked, and its return value is in turn returned.

void **phy_prepare_link**(struct phy_device * *phydev*, void (*handler) (struct *net_device* *))
prepares the PHY layer to monitor link status

Parameters

struct phy_device * phydev target phy_device struct

void (*)(struct net_device *) handler callback function for link status change notifications

Description

Tells the PHY infrastructure to handle the gory details on monitoring link status (whether through polling or an interrupt), and to call back to the connected device driver when the link status changes. If you want to monitor your own link state, don't call this function.

int **phy_poll_reset**(struct phy_device * *phydev*)
Safely wait until a PHY reset has properly completed

Parameters

struct phy_device * phydev The PHY device to poll

Description

According to IEEE 802.3, Section 2, Subsection 22.2.4.1.1, as published in 2008, a PHY reset may take up to 0.5 seconds. The MII BMCR register must be polled until the BMCR_RESET bit clears.

Furthermore, any attempts to write to PHY registers may have no effect or even generate MDIO bus errors until this is complete.

Some PHYs (such as the Marvell 88E1111) don't entirely conform to the standard and do not fully reset after the BMCR_RESET bit is set, and may even *REQUIRE* a soft-reset to properly restart autonegotiation. In an effort to support such broken PHYs, this function is separate from the standard phy_init_hw() which will zero all the other bits in the BMCR and reapply all driver-specific and board-specific fixups.

int **genphy_config_advert**(struct phy_device * *phydev*)
sanitize and advertise auto-negotiation parameters

Parameters

struct phy_device * phydev target phy_device struct

Description

Writes MII_ADVERTISE with the appropriate values, after sanitizing the values to make sure we only advertise what is supported. Returns < 0 on error, 0 if the PHY's advertisement hasn't changed, and > 0 if it has changed.

int **genphy_config_eee_advert**(struct phy_device * *phydev*)
disable unwanted eee mode advertisement

Parameters

struct phy_device * phydev target phy_device struct

Description

Writes MDIO_AN_EEE_ADV after disabling unsupported energy efficent ethernet modes. Returns 0 if the PHY's advertisement hasn't changed, and 1 if it has changed.

int **phy_probe**(struct device * *dev*)
probe and init a PHY device

Parameters

struct device * dev device to probe and init

Description

Take care of setting up the phy_device structure, set the state to READY (the driver's init function should set it to STARTING if needed).

struct mii_bus * **mdiobus_alloc_size**(size_t *size*)
allocate a mii_bus structure

Parameters

size_t size extra amount of memory to allocate for private storage. If non-zero, then bus->priv is points to that memory.

Description

called by a bus driver to allocate an mii_bus structure to fill in.

struct mii_bus * **devm_mdiobus_alloc_size**(struct device * *dev*, int *sizeof_priv*)
Resource-managed *mdiobus_alloc_size()*

Parameters

struct device * dev Device to allocate mii_bus for

int sizeof_priv Space to allocate for private structure.

Description

Managed mdiobus_alloc_size. mii_bus allocated with this function is automatically freed on driver detach.

If an mii_bus allocated with this function needs to be freed separately, *devm_mdiobus_free()* must be used.

Return

Pointer to allocated mii_bus on success, NULL on failure.

void **devm_mdiobus_free**(struct device * *dev*, struct mii_bus * *bus*)
Resource-managed *mdiobus_free()*

Parameters

struct device * dev Device this mii_bus belongs to

struct mii_bus * bus the mii_bus associated with the device

Description

Free mii_bus allocated with *devm_mdiobus_alloc_size()*.

struct mii_bus * **of_mdio_find_bus**(struct device_node * *mdio_bus_np*)
Given an mii_bus node, find the mii_bus.

Parameters

struct device_node * mdio_bus_np Pointer to the mii_bus.

Description

Returns a reference to the mii_bus, or NULL if none found. The embedded struct device will have its reference count incremented, and this must be put once the bus is finished with.

Because the association of a device_node and mii_bus is made via `of_mdiobus_register()`, the mii_bus cannot be found before it is registered with `of_mdiobus_register()`.

int **__mdiobus_register**(struct mii_bus * *bus*, struct module * *owner*)
bring up all the PHYs on a given bus and attach them to bus

Parameters

struct mii_bus * bus target mii_bus

struct module * owner module containing bus accessor functions

Description

Called by a bus driver to bring up all the PHYs on a given bus, and attach them to the bus. Drivers should use `mdiobus_register()` rather than *`__mdiobus_register()`* unless they need to pass a specific owner module. MDIO devices which are not PHYs will not be brought up by this function. They are expected to to be explicitly listed in DT and instantiated by `of_mdiobus_register()`.

Returns 0 on success or < 0 on error.

void **mdiobus_free**(struct mii_bus * *bus*)
free a struct mii_bus

Parameters

struct mii_bus * bus mii_bus to free

Description

This function releases the reference to the underlying device object in the mii_bus. If this is the last reference, the mii_bus will be freed.

struct phy_device * **mdiobus_scan**(struct mii_bus * *bus*, int *addr*)
scan a bus for MDIO devices.

Parameters

struct mii_bus * bus mii_bus to scan

int addr address on bus to scan

Description

This function scans the MDIO bus, looking for devices which can be identified using a vendor/product ID in registers 2 and 3. Not all MDIO devices have such registers, but PHY devices typically do. Hence this function assumes anything found is a PHY, or can be treated as a PHY. Other MDIO devices, such as switches, will probably not be found during the scan.

int **__mdiobus_read**(struct mii_bus * *bus*, int *addr*, u32 *regnum*)
Unlocked version of the mdiobus_read function

Parameters

struct mii_bus * bus the mii_bus struct

int addr the phy address

u32 regnum register number to read

Description

Read a MDIO bus register. Caller must hold the mdio bus lock.

NOTE

MUST NOT be called from interrupt context.

int **__mdiobus_write**(struct mii_bus * *bus*, int *addr*, u32 *regnum*, u16 *val*)
Unlocked version of the mdiobus_write function

Parameters

struct mii_bus * bus the mii_bus struct

int addr the phy address

u32 regnum register number to write

u16 val value to write to **regnum**

Description

Write a MDIO bus register. Caller must hold the mdio bus lock.

NOTE

MUST NOT be called from interrupt context.

int **mdiobus_read_nested**(struct mii_bus * *bus*, int *addr*, u32 *regnum*)
Nested version of the mdiobus_read function

Parameters

struct mii_bus * bus the mii_bus struct

int addr the phy address

u32 regnum register number to read

Description

In case of nested MDIO bus access avoid lockdep false positives by using `mutex_lock_nested()`.

NOTE

MUST NOT be called from interrupt context, because the bus read/write functions may wait for an interrupt to conclude the operation.

int **mdiobus_read**(struct mii_bus * *bus*, int *addr*, u32 *regnum*)
Convenience function for reading a given MII mgmt register

Parameters

struct mii_bus * bus the mii_bus struct

int addr the phy address

u32 regnum register number to read

NOTE

MUST NOT be called from interrupt context, because the bus read/write functions may wait for an interrupt to conclude the operation.

int **mdiobus_write_nested**(struct mii_bus * *bus*, int *addr*, u32 *regnum*, u16 *val*)
Nested version of the mdiobus_write function

Parameters

struct mii_bus * bus the mii_bus struct

int addr the phy address

u32 regnum register number to write

u16 val value to write to **regnum**

Description

In case of nested MDIO bus access avoid lockdep false positives by using `mutex_lock_nested()`.

NOTE

MUST NOT be called from interrupt context, because the bus read/write functions may wait for an interrupt to conclude the operation.

int **mdiobus_write**(struct mii_bus * *bus*, int *addr*, u32 *regnum*, u16 *val*)
Convenience function for writing a given MII mgmt register

Parameters

struct mii_bus * bus the mii_bus struct

int addr the phy address

u32 regnum register number to write

u16 val value to write to **regnum**

NOTE

MUST NOT be called from interrupt context, because the bus read/write functions may wait for an interrupt to conclude the operation.

void **mdiobus_release**(struct device * *d*)
mii_bus device release callback

Parameters

struct device * d the target struct device that contains the mii_bus

Description

called when the last reference to an mii_bus is dropped, to free the underlying memory.

int **mdiobus_create_device**(struct mii_bus * *bus*, struct mdio_board_info * *bi*)
create a full MDIO device given a mdio_board_info structure

Parameters

struct mii_bus * bus MDIO bus to create the devices on

struct mdio_board_info * bi mdio_board_info structure describing the devices

Description

Returns 0 on success or < 0 on error.

int **mdio_bus_match**(struct device * *dev*, struct device_driver * *drv*)
determine if given MDIO driver supports the given MDIO device

Parameters

struct device * dev target MDIO device

struct device_driver * drv given MDIO driver

Description

Given a MDIO device, and a MDIO driver, return 1 if the driver supports the device. Otherwise, return 0. This may require calling the devices own match function, since different classes of MDIO devices have different match criteria.

PHYLINK

PHYLINK interfaces traditional network drivers with PHYLIB, fixed-links, and SFF modules (eg, hot-pluggable SFP) that may contain PHYs. PHYLINK provides management of the link state and link modes.

struct **phylink_link_state**
link state structure

Definition

```
struct phylink_link_state {
  __ETHTOOL_DECLARE_LINK_MODE_MASK(advertising);
  __ETHTOOL_DECLARE_LINK_MODE_MASK(lp_advertising);
  phy_interface_t interface;
  int speed;
  int duplex;
  int pause;
  unsigned int link:1;
  unsigned int an_enabled:1;
```

```
  unsigned int an_complete:1;
};
```

Members

interface link typedef phy_interface_t mode

speed link speed, one of the SPEED_* constants.

duplex link duplex mode, one of DUPLEX_* constants.

pause link pause state, described by MLO_PAUSE_* constants.

link true if the link is up.

an_enabled true if autonegotiation is enabled/desired.

an_complete true if autonegotiation has completed.

struct **phylink_mac_ops**
MAC operations structure.

Definition

```
struct phylink_mac_ops {
  void (*validate)(struct net_device *ndev, unsigned long *supported, struct phylink_link_state *state);
  int (*mac_link_state)(struct net_device *ndev, struct phylink_link_state *state);
  void (*mac_config)(struct net_device *ndev, unsigned int mode, const struct phylink_link_state *state);
  void (*mac_an_restart)(struct net_device *ndev);
  void (*mac_link_down)(struct net_device *ndev, unsigned int mode);
  void (*mac_link_up)(struct net_device *ndev, unsigned int mode, struct phy_device *phy);
};
```

Members

validate Validate and update the link configuration.

mac_link_state Read the current link state from the hardware.

mac_config configure the MAC for the selected mode and state.

mac_an_restart restart 802.3z BaseX autonegotiation.

mac_link_down take the link down.

mac_link_up allow the link to come up.

Description

The individual methods are described more fully below.

void **validate**(struct *net_device* * *ndev*, unsigned long * *supported*, struct *phylink_link_state* * *state*)
Validate and update the link configuration

Parameters

struct net_device * ndev a pointer to a *struct net_device* for the MAC.

unsigned long * supported ethtool bitmask for supported link modes.

struct phylink_link_state * state a pointer to a *struct phylink_link_state*.

Description

Clear bits in the **supported** and **state**->advertising masks that are not supportable by the MAC.

Note that the PHY may be able to transform from one connection technology to another, so, eg, don't clear 1000BaseX just because the MAC is unable to BaseX mode. This is more about clearing unsupported speeds and duplex settings.

If the **state**->interface mode is PHY_INTERFACE_MODE_1000BASEX or PHY_INTERFACE_MODE_2500BASEX, select the appropriate mode based on **state**->advertising and/or **state**->speed and update **state**->interface accordingly.

int **mac_link_state**(struct *net_device* * *ndev*, struct *phylink_link_state* * *state*)
Read the current link state from the hardware

Parameters

struct net_device * ndev a pointer to a *struct net_device* for the MAC.

struct phylink_link_state * state a pointer to a *struct phylink_link_state*.

Description

Read the current link state from the MAC, reporting the current speed in **state**->speed, duplex mode in **state**->duplex, pause mode in **state**->pause using the MLO_PAUSE_RX and MLO_PAUSE_TX bits, negotiation completion state in **state**->an_complete, and link up state in **state**->link.

void **mac_config**(struct *net_device* * *ndev*, unsigned int *mode*, const struct *phylink_link_state* * *state*)
configure the MAC for the selected mode and state

Parameters

struct net_device * ndev a pointer to a *struct net_device* for the MAC.

unsigned int mode one of MLO_AN_FIXED, MLO_AN_PHY, MLO_AN_INBAND.

const struct phylink_link_state * state a pointer to a *struct phylink_link_state*.

Description

The action performed depends on the currently selected mode:

MLO_AN_FIXED, MLO_AN_PHY: Configure the specified **state**->speed, **state**->duplex and **state**->pause (MLO_PAUSE_TX / MLO_PAUSE_RX) mode.

MLO_AN_INBAND: place the link in an inband negotiation mode (such as 802.3z 1000base-X or Cisco SGMII mode depending on the **state**->interface mode). In both cases, link state management (whether the link is up or not) is performed by the MAC, and reported via the *mac_link_state()* callback. Changes in link state must be made by calling *phylink_mac_change()*.

If in 802.3z mode, the link speed is fixed, dependent on the **state**->interface. Duplex is negotiated, and pause is advertised according to **state**->an_enabled, **state**->pause and **state**->advertising flags. Beware of MACs which only support full duplex at gigabit and higher speeds.

If in Cisco SGMII mode, the link speed and duplex mode are passed in the serial bitstream 16-bit configuration word, and the MAC should be configured to read these bits and acknowledge the configuration word. Nothing is advertised by the MAC. The MAC is responsible for reading the configuration word and configuring itself accordingly.

void **mac_an_restart**(struct *net_device* * *ndev*)
restart 802.3z BaseX autonegotiation

Parameters

struct net_device * ndev a pointer to a *struct net_device* for the MAC.

void **mac_link_down**(struct *net_device* * *ndev*, unsigned int *mode*)
take the link down

Parameters

struct net_device * ndev a pointer to a *struct net_device* for the MAC.

unsigned int mode link autonegotiation mode

Description

If **mode** is not an in-band negotiation mode (as defined by `phylink_autoneg_inband()`), force the link down and disable any Energy Efficient Ethernet MAC configuration.

void **mac_link_up**(struct *net_device* * *ndev*, unsigned int *mode*, struct phy_device * *phy*)
allow the link to come up

Parameters

struct net_device * ndev a pointer to a *struct net_device* for the MAC.

unsigned int mode link autonegotiation mode

struct phy_device * phy any attached phy

Description

If **mode** is not an in-band negotiation mode (as defined by `phylink_autoneg_inband()`), allow the link to come up. If **phy** is non-`NULL`, configure Energy Efficient Ethernet by calling *phy_init_eee()* and perform appropriate MAC configuration for EEE.

struct **phylink**
internal data type for phylink

Definition

```
struct phylink {
};
```

Members

void **phylink_set_port_modes**(unsigned long * *mask*)
set the port type modes in the ethtool mask

Parameters

unsigned long * mask ethtool link mode mask

Description

Sets all the port type modes in the ethtool mask. MAC drivers should use this in their 'validate' callback.

struct *phylink* * **phylink_create**(struct *net_device* * *ndev*, struct fwnode_handle * *fwnode*, phy_interface_t *iface*, const struct *phylink_mac_ops* * *ops*)
create a phylink instance

Parameters

struct net_device * ndev a pointer to the *struct net_device*

struct fwnode_handle * fwnode a pointer to a `struct fwnode_handle` describing the network interface

phy_interface_t iface the desired link mode defined by `typedef phy_interface_t`

const struct phylink_mac_ops * ops a pointer to a *struct phylink_mac_ops* for the MAC.

Description

Create a new phylink instance, and parse the link parameters found in **np**. This will parse in-band modes, fixed-link or SFP configuration.

Returns a pointer to a *struct phylink*, or an error-pointer value. Users must use `IS_ERR()` to check for errors from this function.

void **phylink_destroy**(struct *phylink* * *pl*)
cleanup and destroy the phylink instance

Parameters

struct phylink * pl a pointer to a *struct phylink* returned from *phylink_create()*

Description

Destroy a phylink instance. Any PHY that has been attached must have been cleaned up via *phylink_disconnect_phy()* prior to calling this function.

int **phylink_connect_phy**(struct *phylink * pl*, struct phy_device * *phy*)
connect a PHY to the phylink instance

Parameters

struct phylink * pl a pointer to a *struct phylink* returned from *phylink_create()*

struct phy_device * phy a pointer to a `struct phy_device`.

Description

Connect **phy** to the phylink instance specified by **pl** by calling *phy_attach_direct()*. Configure the **phy** according to the MAC driver's capabilities, start the PHYLIB state machine and enable any interrupts that the PHY supports.

This updates the phylink's ethtool supported and advertising link mode masks.

Returns 0 on success or a negative errno.

int **phylink_of_phy_connect**(struct *phylink * pl*, struct device_node * *dn*, u32 *flags*)
connect the PHY specified in the DT mode.

Parameters

struct phylink * pl a pointer to a *struct phylink* returned from *phylink_create()*

struct device_node * dn a pointer to a `struct device_node`.

u32 flags PHY-specific flags to communicate to the PHY device driver

Description

Connect the phy specified in the device node **dn** to the phylink instance specified by **pl**. Actions specified in *phylink_connect_phy()* will be performed.

Returns 0 on success or a negative errno.

void **phylink_disconnect_phy**(struct *phylink * pl*)
disconnect any PHY attached to the phylink instance.

Parameters

struct phylink * pl a pointer to a *struct phylink* returned from *phylink_create()*

Description

Disconnect any current PHY from the phylink instance described by **pl**.

int **phylink_fixed_state_cb**(struct *phylink * pl*, void (*cb) (struct *net_device* **dev*, struct *phylink_link_state* **state*)
allow setting a fixed link callback

Parameters

struct phylink * pl a pointer to a *struct phylink* returned from *phylink_create()*

void (*)(struct net_device *dev, struct phylink_link_state *state) cb callback to execute to determine the fixed link state.

Description

The MAC driver should call this driver when the state of its link can be determined through e.g: an out of band MMIO register.

void **phylink_mac_change**(struct *phylink * pl*, bool *up*)
notify phylink of a change in MAC state

Parameters

struct phylink * pl a pointer to a *struct phylink* returned from *phylink_create()*

bool up indicates whether the link is currently up.

Description

The MAC driver should call this driver when the state of its link changes (eg, link failure, new negotiation results, etc.)

void **phylink_start**(struct *phylink * pl*)
start a phylink instance

Parameters

struct phylink * pl a pointer to a *struct phylink* returned from *phylink_create()*

Description

Start the phylink instance specified by **pl**, configuring the MAC for the desired link mode(s) and negotiation style. This should be called from the network device driver's `struct net_device_ops` `ndo_open()` method.

void **phylink_stop**(struct *phylink * pl*)
stop a phylink instance

Parameters

struct phylink * pl a pointer to a *struct phylink* returned from *phylink_create()*

Description

Stop the phylink instance specified by **pl**. This should be called from the network device driver's `struct net_device_ops` `ndo_stop()` method. The network device's carrier state should not be changed prior to calling this function.

void **phylink_ethtool_get_wol**(struct *phylink * pl*, struct ethtool_wolinfo * *wol*)
get the wake on lan parameters for the PHY

Parameters

struct phylink * pl a pointer to a *struct phylink* returned from *phylink_create()*

struct ethtool_wolinfo * wol a pointer to `struct ethtool_wolinfo` to hold the read parameters

Description

Read the wake on lan parameters from the PHY attached to the phylink instance specified by **pl**. If no PHY is currently attached, report no support for wake on lan.

int **phylink_ethtool_set_wol**(struct *phylink * pl*, struct ethtool_wolinfo * *wol*)
set wake on lan parameters

Parameters

struct phylink * pl a pointer to a *struct phylink* returned from *phylink_create()*

struct ethtool_wolinfo * wol a pointer to `struct ethtool_wolinfo` for the desired parameters

Description

Set the wake on lan parameters for the PHY attached to the phylink instance specified by **pl**. If no PHY is attached, returns `EOPNOTSUPP` error.

Returns zero on success or negative errno code.

int **phylink_ethtool_ksettings_get**(struct *phylink * pl*, struct ethtool_link_ksettings * *kset*)
get the current link settings

Parameters

struct phylink * pl a pointer to a *struct phylink* returned from *phylink_create()*

struct ethtool_link_ksettings * kset a pointer to a `struct ethtool_link_ksettings` to hold link settings

Description

Read the current link settings for the phylink instance specified by **pl**. This will be the link settings read from the MAC, PHY or fixed link settings depending on the current negotiation mode.

int **phylink_ethtool_ksettings_set**(struct *phylink* * *pl*, const struct ethtool_link_ksettings * *kset*)
set the link settings

Parameters

struct phylink * pl a pointer to a *struct phylink* returned from *phylink_create()*

const struct ethtool_link_ksettings * kset a pointer to a `struct ethtool_link_ksettings` for the desired modes

int **phylink_ethtool_nway_reset**(struct *phylink* * *pl*)
restart negotiation

Parameters

struct phylink * pl a pointer to a *struct phylink* returned from *phylink_create()*

Description

Restart negotiation for the phylink instance specified by **pl**. This will cause any attached phy to restart negotiation with the link partner, and if the MAC is in a BaseX mode, the MAC will also be requested to restart negotiation.

Returns zero on success, or negative error code.

void **phylink_ethtool_get_pauseparam**(struct *phylink* * *pl*, struct ethtool_pauseparam * *pause*)
get the current pause parameters

Parameters

struct phylink * pl a pointer to a *struct phylink* returned from *phylink_create()*

struct ethtool_pauseparam * pause a pointer to a `struct ethtool_pauseparam`

int **phylink_ethtool_set_pauseparam**(struct *phylink* * *pl*, struct ethtool_pauseparam * *pause*)
set the current pause parameters

Parameters

struct phylink * pl a pointer to a *struct phylink* returned from *phylink_create()*

struct ethtool_pauseparam * pause a pointer to a `struct ethtool_pauseparam`

int **phylink_get_eee_err**(struct *phylink* * *pl*)
read the energy efficient ethernet error counter

Parameters

struct phylink * pl a pointer to a *struct phylink* returned from *phylink_create()*.

Description

Read the Energy Efficient Ethernet error counter from the PHY associated with the phylink instance specified by **pl**.

Returns positive error counter value, or negative error code.

int **phylink_ethtool_get_eee**(struct *phylink* * *pl*, struct ethtool_eee * *eee*)
read the energy efficient ethernet parameters

Parameters

struct phylink * pl a pointer to a *struct phylink* returned from *phylink_create()*

struct ethtool_eee * eee a pointer to a `struct ethtool_eee` for the read parameters

int **phylink_ethtool_set_eee**(struct *phylink* * *pl*, struct ethtool_eee * *eee*)
set the energy efficient ethernet parameters

Parameters

struct phylink * pl a pointer to a *struct phylink* returned from *phylink_create()*

struct ethtool_eee * eee a pointer to a struct ethtool_eee for the desired parameters

int **phylink_mii_ioctl**(struct *phylink* * *pl*, struct ifreq * *ifr*, int *cmd*)
generic mii ioctl interface

Parameters

struct phylink * pl a pointer to a *struct phylink* returned from *phylink_create()*

struct ifreq * ifr a pointer to a struct ifreq for socket ioctls

int cmd ioctl cmd to execute

Description

Perform the specified MII ioctl on the PHY attached to the phylink instance specified by **pl**. If no PHY is attached, emulate the presence of the PHY.

Return

zero on success or negative error code.

SIOCGMIIPHY: read register from the current PHY.

SIOCGMIIREG: read register from the specified PHY.

SIOCSMIIREG: set a register on the specified PHY.

SFP support

struct **sfp_bus**
internal representation of a sfp bus

Definition

```
struct sfp_bus {
};
```

Members

struct **sfp_eeprom_id**
raw SFP module identification information

Definition

```
struct sfp_eeprom_id {
  struct sfp_eeprom_base base;
  struct sfp_eeprom_ext ext;
};
```

Members

base base SFP module identification structure

ext extended SFP module identification structure

Description

See the SFF-8472 specification and related documents for the definition of these structure members. This can be obtained from ftp://ftp.seagate.com/sff

struct **sfp_upstream_ops**
upstream operations structure

Definition

```
struct sfp_upstream_ops {
  int (*module_insert)(void *priv, const struct sfp_eeprom_id *id);
  void (*module_remove)(void *priv);
  void (*link_down)(void *priv);
  void (*link_up)(void *priv);
  int (*connect_phy)(void *priv, struct phy_device *);
  void (*disconnect_phy)(void *priv);
};
```

Members

module_insert called after a module has been detected to determine whether the module is supported for the upstream device.

module_remove called after the module has been removed.

link_down called when the link is non-operational for whatever reason.

link_up called when the link is operational.

connect_phy called when an I2C accessible PHY has been detected on the module.

disconnect_phy called when a module with an I2C accessible PHY has been removed.

int **sfp_parse_port**(struct *sfp_bus * bus*, const struct *sfp_eeprom_id * id*, unsigned long * *support*)
Parse the EEPROM base ID, setting the port type

Parameters

struct sfp_bus * bus a pointer to the *struct sfp_bus* structure for the sfp module

const struct sfp_eeprom_id * id a pointer to the module's *struct sfp_eeprom_id*

unsigned long * support optional pointer to an array of unsigned long for the ethtool support mask

Description

Parse the EEPROM identification given in **id**, and return one of PORT_TP, PORT_FIBRE or PORT_OTHER. If **support** is non-NULL, also set the ethtool ETHTOOL_LINK_MODE_xxx_BIT corresponding with the connector type.

If the port type is not known, returns PORT_OTHER.

phy_interface_t **sfp_parse_interface**(struct *sfp_bus * bus*, const struct *sfp_eeprom_id * id*)
Parse the phy_interface_t

Parameters

struct sfp_bus * bus a pointer to the *struct sfp_bus* structure for the sfp module

const struct sfp_eeprom_id * id a pointer to the module's *struct sfp_eeprom_id*

Description

Derive the phy_interface_t mode for the information found in the module's identifying EEPROM. There is no standard or defined way to derive this information, so we use some heuristics.

If the encoding is 64b66b, then the module must be >= 10G, so return PHY_INTERFACE_MODE_10GKR.

If it's 8b10b, then it's 1G or slower. If it's definitely a fibre module, return PHY_INTERFACE_MODE_1000BASEX mode, otherwise return PHY_INTERFACE_MODE_SGMII mode.

If the encoding is not known, return PHY_INTERFACE_MODE_NA.

void **sfp_parse_support**(struct *sfp_bus * bus*, const struct *sfp_eeprom_id * id*, unsigned long * *support*)
Parse the eeprom id for supported link modes

Parameters

struct sfp_bus * bus a pointer to the *struct sfp_bus* structure for the sfp module

const struct sfp_eeprom_id * id a pointer to the module's *struct sfp_eeprom_id*

unsigned long * support pointer to an array of unsigned long for the ethtool support mask

Description

Parse the EEPROM identification information and derive the supported ethtool link modes for the module.

int **sfp_get_module_info**(struct *sfp_bus* * *bus*, struct ethtool_modinfo * *modinfo*)
Get the ethtool_modinfo for a SFP module

Parameters

struct sfp_bus * bus a pointer to the *struct sfp_bus* structure for the sfp module

struct ethtool_modinfo * modinfo a `struct ethtool_modinfo`

Description

Fill in the type and eeprom_len parameters in **modinfo** for a module on the sfp bus specified by **bus**.

Returns 0 on success or a negative errno number.

int **sfp_get_module_eeprom**(struct *sfp_bus* * *bus*, struct ethtool_eeprom * *ee*, u8 * *data*)
Read the SFP module EEPROM

Parameters

struct sfp_bus * bus a pointer to the *struct sfp_bus* structure for the sfp module

struct ethtool_eeprom * ee a `struct ethtool_eeprom`

u8 * data buffer to contain the EEPROM data (must be at least **ee**->len bytes)

Description

Read the EEPROM as specified by the supplied **ee**. See the documentation for `struct ethtool_eeprom` for the region to be read.

Returns 0 on success or a negative errno number.

void **sfp_upstream_start**(struct *sfp_bus* * *bus*)
Inform the SFP that the network device is up

Parameters

struct sfp_bus * bus a pointer to the *struct sfp_bus* structure for the sfp module

Description

Inform the SFP socket that the network device is now up, so that the module can be enabled by allowing TX_DISABLE to be deasserted. This should be called from the network device driver's `struct net_device_ops ndo_open()` method.

void **sfp_upstream_stop**(struct *sfp_bus* * *bus*)
Inform the SFP that the network device is down

Parameters

struct sfp_bus * bus a pointer to the *struct sfp_bus* structure for the sfp module

Description

Inform the SFP socket that the network device is now up, so that the module can be disabled by asserting TX_DISABLE, disabling the laser in optical modules. This should be called from the network device driver's `struct net_device_ops ndo_stop()` method.

struct *sfp_bus* * **sfp_register_upstream**(struct fwnode_handle * *fwnode*, struct *net_device* * *ndev*, void * *upstream*, const struct *sfp_upstream_ops* * *ops*)
Register the neighbouring device

Parameters

struct fwnode_handle * fwnode firmware node for the SFP bus

struct net_device * ndev network device associated with the interface

void * upstream the upstream private data

const struct sfp_upstream_ops * ops the upstream's *struct sfp_upstream_ops*

Description

Register the upstream device (eg, PHY) with the SFP bus. MAC drivers should use phylink, which will call this function for them. Returns a pointer to the allocated *struct sfp_bus*.

On error, returns `NULL`.

void **sfp_unregister_upstream**(struct *sfp_bus * bus*)
Unregister sfp bus

Parameters

struct sfp_bus * bus a pointer to the *struct sfp_bus* structure for the sfp module

Description

Unregister a previously registered upstream connection for the SFP module. **bus** is returned from *sfp_register_upstream()*.

CHAPTER

FOUR

Z8530 PROGRAMMING GUIDE

Author Alan Cox

Introduction

The Z85x30 family synchronous/asynchronous controller chips are used on a large number of cheap network interface cards. The kernel provides a core interface layer that is designed to make it easy to provide WAN services using this chip.

The current driver only support synchronous operation. Merging the asynchronous driver support into this code to allow any Z85x30 device to be used as both a tty interface and as a synchronous controller is a project for Linux post the 2.4 release

Driver Modes

The Z85230 driver layer can drive Z8530, Z85C30 and Z85230 devices in three different modes. Each mode can be applied to an individual channel on the chip (each chip has two channels).

The PIO synchronous mode supports the most common Z8530 wiring. Here the chip is interface to the I/O and interrupt facilities of the host machine but not to the DMA subsystem. When running PIO the Z8530 has extremely tight timing requirements. Doing high speeds, even with a Z85230 will be tricky. Typically you should expect to achieve at best 9600 baud with a Z8C530 and 64Kbits with a Z85230.

The DMA mode supports the chip when it is configured to use dual DMA channels on an ISA bus. The better cards tend to support this mode of operation for a single channel. With DMA running the Z85230 tops out when it starts to hit ISA DMA constraints at about 512Kbits. It is worth noting here that many PC machines hang or crash when the chip is driven fast enough to hold the ISA bus solid.

Transmit DMA mode uses a single DMA channel. The DMA channel is used for transmission as the transmit FIFO is smaller than the receive FIFO. it gives better performance than pure PIO mode but is nowhere near as ideal as pure DMA mode.

Using the Z85230 driver

The Z85230 driver provides the back end interface to your board. To configure a Z8530 interface you need to detect the board and to identify its ports and interrupt resources. It is also your problem to verify the resources are available.

Having identified the chip you need to fill in a struct z8530_dev, which describes each chip. This object must exist until you finally shutdown the board. Firstly zero the active field. This ensures nothing goes off without you intending it. The irq field should be set to the interrupt number of the chip. (Each chip has a single interrupt source rather than each channel). You are responsible for allocating the interrupt line. The interrupt handler should be set to *z8530_interrupt()*. The device id should be set to the z8530_dev structure pointer. Whether the interrupt can be shared or not is board dependent, and up to you to initialise.

The structure holds two channel structures. Initialise chanA.ctrlio and chanA.dataio with the address of the control and data ports. You can or this with Z8530_PORT_SLEEP to indicate your interface needs the 5uS delay for chip settling done in software. The PORT_SLEEP option is architecture specific. Other flags may become available on future platforms, eg for MMIO. Initialise the chanA.irqs to &z8530_nop to start the chip up as disabled and discarding interrupt events. This ensures that stray interrupts will be mopped up and not hang the bus. Set chanA.dev to point to the device structure itself. The private and name field you may use as you wish. The private field is unused by the Z85230 layer. The name is used for error reporting and it may thus make sense to make it match the network name.

Repeat the same operation with the B channel if your chip has both channels wired to something useful. This isn't always the case. If it is not wired then the I/O values do not matter, but you must initialise chanB.dev.

If your board has DMA facilities then initialise the txdma and rxdma fields for the relevant channels. You must also allocate the ISA DMA channels and do any necessary board level initialisation to configure them. The low level driver will do the Z8530 and DMA controller programming but not board specific magic.

Having initialised the device you can then call *z8530_init()*. This will probe the chip and reset it into a known state. An identification sequence is then run to identify the chip type. If the checks fail to pass the function returns a non zero error code. Typically this indicates that the port given is not valid. After this call the type field of the z8530_dev structure is initialised to either Z8530, Z85C30 or Z85230 according to the chip found.

Once you have called z8530_init you can also make use of the utility function *z8530_describe()*. This provides a consistent reporting format for the Z8530 devices, and allows all the drivers to provide consistent reporting.

Attaching Network Interfaces

If you wish to use the network interface facilities of the driver, then you need to attach a network device to each channel that is present and in use. In addition to use the generic HDLC you need to follow some additional plumbing rules. They may seem complex but a look at the example hostess_sv11 driver should reassure you.

The network device used for each channel should be pointed to by the netdevice field of each channel. The hdlc-> priv field of the network device points to your private data - you will need to be able to find your private data from this.

The way most drivers approach this particular problem is to create a structure holding the Z8530 device definition and put that into the private field of the network device. The network device fields of the channels then point back to the network devices.

If you wish to use the generic HDLC then you need to register the HDLC device.

Before you register your network device you will also need to provide suitable handlers for most of the network device callbacks. See the network device documentation for more details on this.

Configuring And Activating The Port

The Z85230 driver provides helper functions and tables to load the port registers on the Z8530 chips. When programming the register settings for a channel be aware that the documentation recommends initialisation orders. Strange things happen when these are not followed.

z8530_channel_load() takes an array of pairs of initialisation values in an array of u8 type. The first value is the Z8530 register number. Add 16 to indicate the alternate register bank on the later chips. The array is terminated by a 255.

The driver provides a pair of public tables. The z8530_hdlc_kilostream table is for the UK 'Kilostream' service and also happens to cover most other end host configurations. The z8530_hdlc_kilostream_85230 table is the same configuration using the enhancements of the 85230 chip. The configuration loaded is standard NRZ encoded synchronous data with HDLC bitstuffing. All of the timing is taken from the other end of the link.

When writing your own tables be aware that the driver internally tracks register values. It may need to reload values. You should therefore be sure to set registers 1-7, 9-11, 14 and 15 in all configurations. Where the register settings depend on DMA selection the driver will update the bits itself when you open or close. Loading a new table with the interface open is not recommended.

There are three standard configurations supported by the core code. In PIO mode the interface is programmed up to use interrupt driven PIO. This places high demands on the host processor to avoid latency. The driver is written to take account of latency issues but it cannot avoid latencies caused by other drivers, notably IDE in PIO mode. Because the drivers allocate buffers you must also prevent MTU changes while the port is open.

Once the port is open it will call the rx_function of each channel whenever a completed packet arrived. This is invoked from interrupt context and passes you the channel and a network buffer (struct sk_buff) holding the data. The data includes the CRC bytes so most users will want to trim the last two bytes before processing the data. This function is very timing critical. When you wish to simply discard data the support code provides the function *z8530_null_rx()* to discard the data.

To active PIO mode sending and receiving the `z8530_sync_open` is called. This expects to be passed the network device and the channel. Typically this is called from your network device open callback. On a failure a non zero error status is returned. The *z8530_sync_close()* function shuts down a PIO channel. This must be done before the channel is opened again and before the driver shuts down and unloads.

The ideal mode of operation is dual channel DMA mode. Here the kernel driver will configure the board for DMA in both directions. The driver also handles ISA DMA issues such as controller programming and the memory range limit for you. This mode is activated by calling the *z8530_sync_dma_open()* function. On failure a non zero error value is returned. Once this mode is activated it can be shut down by calling the *z8530_sync_dma_close()*. You must call the close function matching the open mode you used.

The final supported mode uses a single DMA channel to drive the transmit side. As the Z85C30 has a larger FIFO on the receive channel this tends to increase the maximum speed a little. This is activated by calling the `z8530_sync_txdma_open`. This returns a non zero error code on failure. The *z8530_sync_txdma_close()* function closes down the Z8530 interface from this mode.

Network Layer Functions

The Z8530 layer provides functions to queue packets for transmission. The driver internally buffers the frame currently being transmitted and one further frame (in order to keep back to back transmission running). Any further buffering is up to the caller.

The function *z8530_queue_xmit()* takes a network buffer in sk_buff format and queues it for transmission. The caller must provide the entire packet with the exception of the bitstuffing and CRC. This is normally done by the caller via the generic HDLC interface layer. It returns 0 if the buffer has been queued and non zero values for queue full. If the function accepts the buffer it becomes property of the Z8530 layer and the caller should not free it.

The function `z8530_get_stats()` returns a pointer to an internally maintained per interface statistics block. This provides most of the interface code needed to implement the network layer get_stats callback.

Porting The Z8530 Driver

The Z8530 driver is written to be portable. In DMA mode it makes assumptions about the use of ISA DMA. These are probably warranted in most cases as the Z85230 in particular was designed to glue to PC type machines. The PIO mode makes no real assumptions.

Should you need to retarget the Z8530 driver to another architecture the only code that should need changing are the port I/O functions. At the moment these assume PC I/O port accesses. This may not be appropriate for all platforms. Replacing *z8530_read_port()* and `z8530_write_port` is intended to be all that is required to port this driver layer.

Known Bugs And Assumptions

Interrupt Locking The locking in the driver is done via the global cli/sti lock. This makes for relatively poor SMP performance. Switching this to use a per device spin lock would probably materially improve performance.

Occasional Failures We have reports of occasional failures when run for very long periods of time and the driver starts to receive junk frames. At the moment the cause of this is not clear.

Public Functions Provided

irqreturn_t **z8530_interrupt**(int *irq*, void * *dev_id*)
Handle an interrupt from a Z8530

Parameters

int irq Interrupt number

void * dev_id The Z8530 device that is interrupting.

Description

A Z85[2]30 device has stuck its hand in the air for attention. We scan both the channels on the chip for events and then call the channel specific call backs for each channel that has events. We have to use callback functions because the two channels can be in different modes.

Locking is done for the handlers. Note that locking is done at the chip level (the 5uS delay issue is per chip not per channel). c->lock for both channels points to dev->lock

int **z8530_sync_open**(struct *net_device* * *dev*, struct z8530_channel * *c*)
Open a Z8530 channel for PIO

Parameters

struct net_device * dev The network interface we are using

struct z8530_channel * c The Z8530 channel to open in synchronous PIO mode

Description

Switch a Z8530 into synchronous mode without DMA assist. We raise the RTS/DTR and commence network operation.

int **z8530_sync_close**(struct *net_device* * *dev*, struct z8530_channel * *c*)
Close a PIO Z8530 channel

Parameters

struct net_device * dev Network device to close

struct z8530_channel * c Z8530 channel to disassociate and move to idle

Description

Close down a Z8530 interface and switch its interrupt handlers to discard future events.

int **z8530_sync_dma_open**(struct *net_device* * *dev*, struct z8530_channel * *c*)
Open a Z8530 for DMA I/O

Parameters

struct net_device * dev The network device to attach

struct z8530_channel * c The Z8530 channel to configure in sync DMA mode.

Description

Set up a Z85x30 device for synchronous DMA in both directions. Two ISA DMA channels must be available for this to work. We assume ISA DMA driven I/O and PC limits on access.

int **z8530_sync_dma_close**(struct *net_device* * *dev*, struct z8530_channel * *c*)
Close down DMA I/O

Parameters

struct net_device * dev Network device to detach

struct z8530_channel * c Z8530 channel to move into discard mode

Description

Shut down a DMA mode synchronous interface. Halt the DMA, and free the buffers.

int **z8530_sync_txdma_open**(struct *net_device* * *dev*, struct z8530_channel * *c*)
Open a Z8530 for TX driven DMA

Parameters

struct net_device * dev The network device to attach

struct z8530_channel * c The Z8530 channel to configure in sync DMA mode.

Description

Set up a Z85x30 device for synchronous DMA transmission. One ISA DMA channel must be available for this to work. The receive side is run in PIO mode, but then it has the bigger FIFO.

int **z8530_sync_txdma_close**(struct *net_device* * *dev*, struct z8530_channel * *c*)
Close down a TX driven DMA channel

Parameters

struct net_device * dev Network device to detach

struct z8530_channel * c Z8530 channel to move into discard mode

Description

Shut down a DMA/PIO split mode synchronous interface. Halt the DMA, and free the buffers.

void **z8530_describe**(struct z8530_dev * *dev*, char * *mapping*, unsigned long *io*)
Uniformly describe a Z8530 port

Parameters

struct z8530_dev * dev Z8530 device to describe

char * mapping string holding mapping type (eg "I/O" or "Mem")

unsigned long io the port value in question

Description

Describe a Z8530 in a standard format. We must pass the I/O as the port offset isn't predictable. The main reason for this function is to try and get a common format of report.

int **z8530_init**(struct z8530_dev * *dev*)
Initialise a Z8530 device

Parameters

struct z8530_dev * dev Z8530 device to initialise.

Description

Configure up a Z8530/Z85C30 or Z85230 chip. We check the device is present, identify the type and then program it to hopefully keep quite and behave. This matters a lot, a Z8530 in the wrong state will sometimes get into stupid modes generating 10Khz interrupt streams and the like.

We set the interrupt handler up to discard any events, in case we get them during reset or setp.

Return 0 for success, or a negative value indicating the problem in errno form.

int **z8530_shutdown**(struct z8530_dev * *dev*)
Shutdown a Z8530 device

Parameters

struct z8530_dev * dev The Z8530 chip to shutdown

Description

We set the interrupt handlers to silence any interrupts. We then reset the chip and wait 100uS to be sure the reset completed. Just in case the caller then tries to do stuff.

This is called without the lock held

int **z8530_channel_load**(struct z8530_channel * *c*, u8 * *rtable*)
Load channel data

Parameters

struct z8530_channel * c Z8530 channel to configure

u8 * rtable table of register, value pairs FIXME: ioctl to allow user uploaded tables

Description

Load a Z8530 channel up from the system data. We use +16 to indicate the "prime" registers. The value 255 terminates the table.

void **z8530_null_rx**(struct z8530_channel * *c*, struct *sk_buff* * *skb*)
Discard a packet

Parameters

struct z8530_channel * c The channel the packet arrived on

struct sk_buff * skb The buffer

Description

We point the receive handler at this function when idle. Instead of processing the frames we get to throw them away.

netdev_tx_t **z8530_queue_xmit**(struct z8530_channel * *c*, struct *sk_buff* * *skb*)
Queue a packet

Parameters

struct z8530_channel * c The channel to use

struct sk_buff * skb The packet to kick down the channel

Description

Queue a packet for transmission. Because we have rather hard to hit interrupt latencies for the Z85230 per packet even in DMA mode we do the flip to DMA buffer if needed here not in the IRQ.

Called from the network code. The lock is not held at this point.

Internal Functions

int **z8530_read_port**(unsigned long *p*)
Architecture specific interface function

Parameters

unsigned long p port to read

Description

Provided port access methods. The Comtrol SV11 requires no delays between accesses and uses PC I/O. Some drivers may need a 5uS delay

In the longer term this should become an architecture specific section so that this can become a generic driver interface for all platforms. For now we only handle PC I/O ports with or without the dread 5uS sanity delay.

The caller must hold sufficient locks to avoid violating the horrible 5uS delay rule.

void **z8530_write_port**(unsigned long *p*, u8 *d*)
Architecture specific interface function

Parameters

unsigned long p port to write

u8 d value to write

Description

Write a value to a port with delays if need be. Note that the caller must hold locks to avoid read/writes from other contexts violating the 5uS rule

In the longer term this should become an architecture specific section so that this can become a generic driver interface for all platforms. For now we only handle PC I/O ports with or without the dread 5uS sanity delay.

u8 **read_zsreg**(struct z8530_channel * *c*, u8 *reg*)
Read a register from a Z85230

Parameters

struct z8530_channel * c Z8530 channel to read from (2 per chip)

u8 reg Register to read FIXME: Use a spinlock.

Most of the Z8530 registers are indexed off the control registers. A read is done by writing to the control register and reading the register back. The caller must hold the lock

u8 **read_zsdata**(struct z8530_channel * *c*)
Read the data port of a Z8530 channel

Parameters

struct z8530_channel * c The Z8530 channel to read the data port from

Description

The data port provides fast access to some things. We still have all the 5uS delays to worry about.

void **write_zsreg**(struct z8530_channel * *c*, u8 *reg*, u8 *val*)
Write to a Z8530 channel register

Parameters

struct z8530_channel * c The Z8530 channel

u8 reg Register number

u8 val Value to write

Description

Write a value to an indexed register. The caller must hold the lock to honour the irritating delay rules. We know about register 0 being fast to access.

Assumes c->lock is held.

void **write_zsctrl**(struct z8530_channel * *c*, u8 *val*)
Write to a Z8530 control register

Parameters

struct z8530_channel * c The Z8530 channel

u8 val Value to write

Description

Write directly to the control register on the Z8530

void **write_zsdata**(struct z8530_channel * *c*, u8 *val*)
Write to a Z8530 control register

Parameters

struct z8530_channel * c The Z8530 channel

u8 val Value to write

Description

Write directly to the data register on the Z8530

void **z8530_flush_fifo**(struct z8530_channel * *c*)
Flush on chip RX FIFO

Parameters

struct z8530_channel * c Channel to flush

Description

Flush the receive FIFO. There is no specific option for this, we blindly read bytes and discard them. Reading when there is no data is harmless. The 8530 has a 4 byte FIFO, the 85230 has 8 bytes.

All locking is handled for the caller. On return data may still be present if it arrived during the flush.

void **z8530_rtsdtr**(struct z8530_channel * *c*, int *set*)
Control the outgoing DTS/RTS line

Parameters

struct z8530_channel * c The Z8530 channel to control;

int set 1 to set, 0 to clear

Description

Sets or clears DTR/RTS on the requested line. All locking is handled by the caller. For now we assume all boards use the actual RTS/DTR on the chip. Apparently one or two don't. We'll scream about them later.

void **z8530_rx**(struct z8530_channel * *c*)
Handle a PIO receive event

Parameters

struct z8530_channel * c Z8530 channel to process

Description

Receive handler for receiving in PIO mode. This is much like the async one but not quite the same or as complex

Note

Its intended that this handler can easily be separated from the main code to run realtime. That'll be needed for some machines (eg to ever clock 64kbits on a sparc ;)).

The RT_LOCK macros don't do anything now. Keep the code covered by them as short as possible in all circumstances - clocks cost baud. The interrupt handler is assumed to be atomic w.r.t. to other code - this is true in the RT case too.

We only cover the sync cases for this. If you want 2Mbit async do it yourself but consider medical assistance first. This non DMA synchronous mode is portable code. The DMA mode assumes PCI like ISA DMA

Called with the device lock held

void **z8530_tx**(struct z8530_channel * *c*)
Handle a PIO transmit event

Parameters

struct z8530_channel * c Z8530 channel to process

Description

Z8530 transmit interrupt handler for the PIO mode. The basic idea is to attempt to keep the FIFO fed. We fill as many bytes in as possible, its quite possible that we won't keep up with the data rate otherwise.

void **z8530_status**(struct z8530_channel * *chan*)
Handle a PIO status exception

Parameters

struct z8530_channel * chan Z8530 channel to process

Description

A status event occurred in PIO synchronous mode. There are several reasons the chip will bother us here. A transmit underrun means we failed to feed the chip fast enough and just broke a packet. A DCD change is a line up or down.

void **z8530_dma_rx**(struct z8530_channel * *chan*)
Handle a DMA RX event

Parameters

struct z8530_channel * chan Channel to handle

Description

Non bus mastering DMA interfaces for the Z8x30 devices. This is really pretty PC specific. The DMA mode means that most receive events are handled by the DMA hardware. We get a kick here only if a frame ended.

void **z8530_dma_tx**(struct z8530_channel * *chan*)
Handle a DMA TX event

Parameters

struct z8530_channel * chan The Z8530 channel to handle

Description

We have received an interrupt while doing DMA transmissions. It shouldn't happen. Scream loudly if it does.

void **z8530_dma_status**(struct z8530_channel * *chan*)
Handle a DMA status exception

Parameters

struct z8530_channel * chan Z8530 channel to process

A status event occurred on the Z8530. We receive these for two reasons when in DMA mode. Firstly if we finished a packet transfer we get one and kick the next packet out. Secondly we may see a DCD change.

void **z8530_rx_clear**(struct z8530_channel * *c*)
Handle RX events from a stopped chip

Parameters

struct z8530_channel * c Z8530 channel to shut up

Description

Receive interrupt vectors for a Z8530 that is in 'parked' mode. For machines with PCI Z85x30 cards, or level triggered interrupts (eg the MacII) we must clear the interrupt cause or die.

void **z8530_tx_clear**(struct z8530_channel * *c*)
Handle TX events from a stopped chip

Parameters

struct z8530_channel * c Z8530 channel to shut up

Description

Transmit interrupt vectors for a Z8530 that is in 'parked' mode. For machines with PCI Z85x30 cards, or level triggered interrupts (eg the MacII) we must clear the interrupt cause or die.

void **z8530_status_clear**(struct z8530_channel * *chan*)
Handle status events from a stopped chip

Parameters

struct z8530_channel * chan Z8530 channel to shut up

Description

Status interrupt vectors for a Z8530 that is in 'parked' mode. For machines with PCI Z85x30 cards, or level triggered interrupts (eg the MacII) we must clear the interrupt cause or die.

void **z8530_tx_begin**(struct z8530_channel * *c*)
Begin packet transmission

Parameters

struct z8530_channel * c The Z8530 channel to kick

Description

This is the speed sensitive side of transmission. If we are called and no buffer is being transmitted we commence the next buffer. If nothing is queued we idle the sync.

Note

We are handling this code path in the interrupt path, keep it fast or bad things will happen.

Called with the lock held.

void **z8530_tx_done**(struct z8530_channel * *c*)
TX complete callback

Parameters

struct z8530_channel * c The channel that completed a transmit.

Description

This is called when we complete a packet send. We wake the queue, start the next packet going and then free the buffer of the existing packet. This code is fairly timing sensitive.

Called with the register lock held.

void **z8530_rx_done**(struct z8530_channel * *c*)
Receive completion callback

Parameters

struct z8530_channel * c The channel that completed a receive

Description

A new packet is complete. Our goal here is to get back into receive mode as fast as possible. On the Z85230 we could change to using ESCC mode, but on the older chips we have no choice. We flip to the new buffer immediately in DMA mode so that the DMA of the next frame can occur while we are copying the previous buffer to an sk_buff

Called with the lock held

int **spans_boundary**(struct *sk_buff* * *skb*)
Check a packet can be ISA DMA'd

Parameters

struct sk_buff * skb The buffer to check

Description

Returns true if the buffer cross a DMA boundary on a PC. The poor thing can only DMA within a 64K block not across the edges of it.

CHAPTER

FIVE

MSG_ZEROCOPY

Intro

The MSG_ZEROCOPY flag enables copy avoidance for socket send calls. The feature is currently implemented for TCP sockets.

Opportunity and Caveats

Copying large buffers between user process and kernel can be expensive. Linux supports various interfaces that eschew copying, such as sendpage and splice. The MSG_ZEROCOPY flag extends the underlying copy avoidance mechanism to common socket send calls.

Copy avoidance is not a free lunch. As implemented, with page pinning, it replaces per byte copy cost with page accounting and completion notification overhead. As a result, MSG_ZEROCOPY is generally only effective at writes over around 10 KB.

Page pinning also changes system call semantics. It temporarily shares the buffer between process and network stack. Unlike with copying, the process cannot immediately overwrite the buffer after system call return without possibly modifying the data in flight. Kernel integrity is not affected, but a buggy program can possibly corrupt its own data stream.

The kernel returns a notification when it is safe to modify data. Converting an existing application to MSG_ZEROCOPY is not always as trivial as just passing the flag, then.

More Info

Much of this document was derived from a longer paper presented at netdev 2.1. For more in-depth information see that paper and talk, the excellent reporting over at LWN.net or read the original code.

paper, slides, video https://netdevconf.org/2.1/session.html?debruijn

LWN article https://lwn.net/Articles/726917/

patchset [PATCH net-next v4 0/9] socket sendmsg MSG_ZEROCOPY http://lkml.kernel.org/r/20170803202945.70750-1-willemdebruijn.kernel@gmail.com

Interface

Passing the MSG_ZEROCOPY flag is the most obvious step to enable copy avoidance, but not the only one.

Socket Setup

The kernel is permissive when applications pass undefined flags to the send system call. By default it simply ignores these. To avoid enabling copy avoidance mode for legacy processes that accidentally already pass this flag, a process must first signal intent by setting a socket option:

```
if (setsockopt(fd, SOL_SOCKET, SO_ZEROCOPY, &one, sizeof(one)))
        error(1, errno, "setsockopt zerocopy");
```

Setting the socket option only works when the socket is in its initial (TCP_CLOSED) state. Trying to set the option for a socket returned by accept(), for example, will lead to an EBUSY error. In this case, the option should be set to the listening socket and it will be inherited by the accepted sockets.

Transmission

The change to send (or sendto, sendmsg, sendmmsg) itself is trivial. Pass the new flag.

```
ret = send(fd, buf, sizeof(buf), MSG_ZEROCOPY);
```

A zerocopy failure will return -1 with errno ENOBUFS. This happens if the socket option was not set, the socket exceeds its optmem limit or the user exceeds its ulimit on locked pages.

Mixing copy avoidance and copying

Many workloads have a mixture of large and small buffers. Because copy avoidance is more expensive than copying for small packets, the feature is implemented as a flag. It is safe to mix calls with the flag with those without.

Notifications

The kernel has to notify the process when it is safe to reuse a previously passed buffer. It queues completion notifications on the socket error queue, akin to the transmit timestamping interface.

The notification itself is a simple scalar value. Each socket maintains an internal unsigned 32-bit counter. Each send call with MSG_ZEROCOPY that successfully sends data increments the counter. The counter is not incremented on failure or if called with length zero. The counter counts system call invocations, not bytes. It wraps after UINT_MAX calls.

Notification Reception

The below snippet demonstrates the API. In the simplest case, each send syscall is followed by a poll and recvmsg on the error queue.

Reading from the error queue is always a non-blocking operation. The poll call is there to block until an error is outstanding. It will set POLLERR in its output flags. That flag does not have to be set in the events field. Errors are signaled unconditionally.

```
pfd.fd = fd;
pfd.events = 0;
if (poll(&pfd, 1, -1) != 1 || pfd.revents & POLLERR == 0)
        error(1, errno, "poll");

ret = recvmsg(fd, &msg, MSG_ERRQUEUE);
if (ret == -1)
        error(1, errno, "recvmsg");

read_notification(msg);
```

The example is for demonstration purpose only. In practice, it is more efficient to not wait for notifications, but read without blocking every couple of send calls.

Notifications can be processed out of order with other operations on the socket. A socket that has an error queued would normally block other operations until the error is read. Zerocopy notifications have a zero error code, however, to not block send and recv calls.

Notification Batching

Multiple outstanding packets can be read at once using the recvmmsg call. This is often not needed. In each message the kernel returns not a single value, but a range. It coalesces consecutive notifications while one is outstanding for reception on the error queue.

When a new notification is about to be queued, it checks whether the new value extends the range of the notification at the tail of the queue. If so, it drops the new notification packet and instead increases the range upper value of the outstanding notification.

For protocols that acknowledge data in-order, like TCP, each notification can be squashed into the previous one, so that no more than one notification is outstanding at any one point.

Ordered delivery is the common case, but not guaranteed. Notifications may arrive out of order on retransmission and socket teardown.

Notification Parsing

The below snippet demonstrates how to parse the control message: the read_notification() call in the previous snippet. A notification is encoded in the standard error format, sock_extended_err.

The level and type fields in the control data are protocol family specific, IP_RECVERR or IPV6_RECVERR.

Error origin is the new type SO_EE_ORIGIN_ZEROCOPY. ee_errno is zero, as explained before, to avoid blocking read and write system calls on the socket.

The 32-bit notification range is encoded as [ee_info, ee_data]. This range is inclusive. Other fields in the struct must be treated as undefined, bar for ee_code, as discussed below.

```
struct sock_extended_err *serr;
struct cmsghdr *cm;

cm = CMSG_FIRSTHDR(msg);
if (cm->cmsg_level != SOL_IP &&
    cm->cmsg_type != IP_RECVERR)
        error(1, 0, "cmsg");

serr = (void *) CMSG_DATA(cm);
if (serr->ee_errno != 0 ||
    serr->ee_origin != SO_EE_ORIGIN_ZEROCOPY)
        error(1, 0, "serr");

printf("completed: %u..%u\n", serr->ee_info, serr->ee_data);
```

Deferred copies

Passing flag MSG_ZEROCOPY is a hint to the kernel to apply copy avoidance, and a contract that the kernel will queue a completion notification. It is not a guarantee that the copy is elided.

Copy avoidance is not always feasible. Devices that do not support scatter-gather I/O cannot send packets made up of kernel generated protocol headers plus zerocopy user data. A packet may need to be converted to a private copy of data deep in the stack, say to compute a checksum.

In all these cases, the kernel returns a completion notification when it releases its hold on the shared pages. That notification may arrive before the (copied) data is fully transmitted. A zerocopy completion notification is not a transmit completion notification, therefore.

Deferred copies can be more expensive than a copy immediately in the system call, if the data is no longer warm in the cache. The process also incurs notification processing cost for no benefit. For this reason, the kernel signals if data was completed with a copy, by setting flag SO_EE_CODE_ZEROCOPY_COPIED in field ee_code on return. A process may use this signal to stop passing flag MSG_ZEROCOPY on subsequent requests on the same socket.

Implementation

Loopback

Data sent to local sockets can be queued indefinitely if the receive process does not read its socket. Unbound notification latency is not acceptable. For this reason all packets generated with MSG_ZEROCOPY that are looped to a local socket will incur a deferred copy. This includes looping onto packet sockets (e.g., tcpdump) and tun devices.

Testing

More realistic example code can be found in the kernel source under tools/testing/selftests/net/msg_zerocopy.c.

Be cognizant of the loopback constraint. The test can be run between a pair of hosts. But if run between a local pair of processes, for instance when run with msg_zerocopy.sh between a veth pair across namespaces, the test will not show any improvement. For testing, the loopback restriction can be temporarily relaxed by making skb_orphan_frags_rx identical to skb_orphan_frags.

Symbols

A

B

C

D

E

F

G

I

K

L

N

P

R

S

U

W

X

Z