Linux Networking Documentation

Release

The kernel development community

August 18, 2018

Contents:

CHAPTER

ONE

AF_XDP

Overview

AF_XDP is an address family that is optimized for high performance packet processing.

This document assumes that the reader is familiar with BPF and XDP. If not, the Cilium project has an excellent reference guide at http://cilium.readthedocs.io/en/latest/bpf/.

Using the XDP_REDIRECT action from an XDP program, the program can redirect ingress frames to other XDP enabled netdevs, using the bpf_redirect_map() function. AF_XDP sockets enable the possibility for XDP programs to redirect frames to a memory buffer in a user-space application.

An AF_XDP socket (XSK) is created with the normal socket() syscall. Associated with each XSK are two rings: the RX ring and the TX ring. A socket can receive packets on the RX ring and it can send packets on the TX ring. These rings are registered and sized with the setsockopts XDP_RX_RING and XDP_TX_RING, respectively. It is mandatory to have at least one of these rings for each socket. An RX or TX descriptor ring points to a data buffer in a memory area called a UMEM. RX and TX can share the same UMEM so that a packet does not have to be copied between RX and TX. Moreover, if a packet needs to be kept for a while due to a possible retransmit, the descriptor that points to that packet can be changed to point to another and reused right away. This again avoids copying data.

The UMEM consists of a number of equally sized chunks. A descriptor in one of the rings references a frame by referencing its addr. The addr is simply an offset within the entire UMEM region. The user space allocates memory for this UMEM using whatever means it feels is most appropriate (malloc, mmap, huge pages, etc). This memory area is then registered with the kernel using the new setsockopt XDP_UMEM_REG. The UMEM also has two rings: the FILL ring and the COMPLETION ring. The fill ring is used by the application to send down addr for the kernel to fill in with RX packet data. References to these frames will then appear in the RX ring once each packet has been received. The completion ring, on the other hand, contains frame addr that the kernel has transmitted completely and can now be used again by user space, for either TX or RX. Thus, the frame addrs appearing in the completion ring are addrs that were previously transmitted using the TX ring. In summary, the RX and FILL rings are used for the RX path and the TX and COMPLETION rings are used for the TX path.

The socket is then finally bound with a bind() call to a device and a specific queue id on that device, and it is not until bind is completed that traffic starts to flow.

The UMEM can be shared between processes, if desired. If a process wants to do this, it simply skips the registration of the UMEM and its corresponding two rings, sets the XDP_SHARED_UMEM flag in the bind call and submits the XSK of the process it would like to share UMEM with as well as its own newly created XSK socket. The new process will then receive frame addr references in its own RX ring that point to this shared UMEM. Note that since the ring structures are single-consumer / single-producer (for performance reasons), the new process has to create its own socket with associated RX and TX rings, since it cannot share this with the other process. This is also the reason that there is only one set of FILL and COMPLETION rings per UMEM. It is the responsibility of a single process to handle the UMEM.

How is then packets distributed from an XDP program to the XSKs? There is a BPF map called XSKMAP (or BPF_MAP_TYPE_XSKMAP in full). The user-space application can place an XSK at an arbitrary place in this map. The XDP program can then redirect a packet to a specific index in this map and at this point XDP validates that the XSK in that map was indeed bound to that device and ring number. If not, the packet is

dropped. If the map is empty at that index, the packet is also dropped. This also means that it is currently mandatory to have an XDP program loaded (and one XSK in the XSKMAP) to be able to get any traffic to user space through the XSK.

AF_XDP can operate in two different modes: XDP_SKB and XDP_DRV. If the driver does not have support for XDP, or XDP_SKB is explicitly chosen when loading the XDP program, XDP_SKB mode is employed that uses SKBs together with the generic XDP support and copies out the data to user space. A fallback mode that works for any network device. On the other hand, if the driver has support for XDP, it will be used by the AF_XDP code to provide better performance, but there is still a copy of the data into user space.

Concepts

In order to use an AF_XDP socket, a number of associated objects need to be setup.

Jonathan Corbet has also written an excellent article on LWN, "Accelerating networking with AF_XDP". It can be found at https://lwn.net/Articles/750845/.

UMEM

UMEM is a region of virtual contiguous memory, divided into equal-sized frames. An UMEM is associated to a netdev and a specific queue id of that netdev. It is created and configured (chunk size, headroom, start address and size) by using the XDP_UMEM_REG setsockopt system call. A UMEM is bound to a netdev and queue id, via the bind() system call.

An AF_XDP is socket linked to a single UMEM, but one UMEM can have multiple AF_XDP sockets. To share an UMEM created via one socket A, the next socket B can do this by setting the XDP_SHARED_UMEM flag in struct sockaddr_xdp member sxdp_flags, and passing the file descriptor of A to struct sockaddr_xdp member sxdp_shared_umem_fd.

The UMEM has two single-producer/single-consumer rings, that are used to transfer ownership of UMEM frames between the kernel and the user-space application.

Rings

There are a four different kind of rings: Fill, Completion, RX and TX. All rings are single-producer/single-consumer, so the user-space application need explicit synchronization of multiple processes/threads are reading/writing to them.

The UMEM uses two rings: Fill and Completion. Each socket associated with the UMEM must have an RX queue, TX queue or both. Say, that there is a setup with four sockets (all doing TX and RX). Then there will be one Fill ring, one Completion ring, four TX rings and four RX rings.

The rings are head(producer)/tail(consumer) based rings. A producer writes the data ring at the index pointed out by struct xdp_ring producer member, and increasing the producer index. A consumer reads the data ring at the index pointed out by struct xdp_ring consumer member, and increasing the consumer index.

The rings are configured and created via the _RING setsockopt system calls and mmapped to user-space using the appropriate offset to mmap() (XDP_PGOFF_RX_RING, XDP_PGOFF_TX_RING, XDP_UMEM_PGOFF_FILL_RING and XDP_UMEM_PGOFF_COMPLETION_RING).

The size of the rings need to be of size power of two.

UMEM Fill Ring

The Fill ring is used to transfer ownership of UMEM frames from user-space to kernel-space. The UMEM addrs are passed in the ring. As an example, if the UMEM is 64k and each chunk is 4k, then the UMEM has 16 chunks and can pass addrs between 0 and 64k.

Frames passed to the kernel are used for the ingress path (RX rings).

The user application produces UMEM addrs to this ring. Note that the kernel will mask the incoming addr. E.g. for a chunk size of 2k, the log2(2048) LSB of the addr will be masked off, meaning that 2048, 2050 and 3000 refers to the same chunk.

UMEM Completetion Ring

The Completion Ring is used transfer ownership of UMEM frames from kernel-space to user-space. Just like the Fill ring, UMEM indicies are used.

Frames passed from the kernel to user-space are frames that has been sent (TX ring) and can be used by user-space again.

The user application consumes UMEM addrs from this ring.

RX Ring

The RX ring is the receiving side of a socket. Each entry in the ring is a struct xdp_desc descriptor. The descriptor contains UMEM offset (addr) and the length of the data (len).

If no frames have been passed to kernel via the Fill ring, no descriptors will (or can) appear on the RX ring.

The user application consumes struct xdp_desc descriptors from this ring.

TX Ring

The TX ring is used to send frames. The struct xdp_desc descriptor is filled (index, length and offset) and passed into the ring.

To start the transfer a sendmsg() system call is required. This might be relaxed in the future.

The user application produces struct xdp_desc descriptors to this ring.

XSKMAP / BPF_MAP_TYPE_XSKMAP

On XDP side there is a BPF map type BPF_MAP_TYPE_XSKMAP (XSKMAP) that is used in conjunction with bpf_redirect_map() to pass the ingress frame to a socket.

The user application inserts the socket into the map, via the bpf() system call.

Note that if an XDP program tries to redirect to a socket that does not match the queue configuration and netdev, the frame will be dropped. E.g. an AF_XDP socket is bound to netdev eth0 and queue 17. Only the XDP program executing for eth0 and queue 17 will successfully pass data to the socket. Please refer to the sample application (samples/bpf/) in for an example.

Usage

In order to use AF_XDP sockets there are two parts needed. The user-space application and the XDP program. For a complete setup and usage example, please refer to the sample application. The user-space side is xdpsock_user.c and the XDP side xdpsock_kern.c.

Naive ring dequeue and enqueue could look like this:

```
// struct xdp_rxtx_ring {
//  __u32 *producer;
//  __u32 *consumer;
//  struct xdp_desc *desc;
// };
```

```
// struct xdp_umem_ring {
//  __u32 *producer;
//  __u32 *consumer;
//  __u64 *desc;
// };

// typedef struct xdp_rxtx_ring RING;
// typedef struct xdp_umem_ring RING;

// typedef struct xdp_desc RING_TYPE;
// typedef __u64 RING_TYPE;

int dequeue_one(RING *ring, RING_TYPE *item)
{
    __u32 entries = *ring->producer - *ring->consumer;

    if (entries == 0)
        return -1;

    // read-barrier!

    *item = ring->desc[*ring->consumer & (RING_SIZE - 1)];
    (*ring->consumer)++;
    return 0;
}

int enqueue_one(RING *ring, const RING_TYPE *item)
{
    u32 free_entries = RING_SIZE - (*ring->producer - *ring->consumer);

    if (free_entries == 0)
        return -1;

    ring->desc[*ring->producer & (RING_SIZE - 1)] = *item;

    // write-barrier!

    (*ring->producer)++;
    return 0;
}
```

For a more optimized version, please refer to the sample application.

Sample application

There is a xdpsock benchmarking/test application included that demonstrates how to use AF_XDP sockets with both private and shared UMEMs. Say that you would like your UDP traffic from port 4242 to end up in queue 16, that we will enable AF_XDP on. Here, we use ethtool for this:

```
ethtool -N p3p2 rx-flow-hash udp4 fn
ethtool -N p3p2 flow-type udp4 src-port 4242 dst-port 4242 \
    action 16
```

Running the rxdrop benchmark in XDP_DRV mode can then be done using:

```
samples/bpf/xdpsock -i p3p2 -q 16 -r -N
```

For XDP_SKB mode, use the switch “-S” instead of “-N” and all options can be displayed with “-h”, as usual.

Credits

- Björn Töpel (AF_XDP core)
- Magnus Karlsson (AF_XDP core)
- Alexander Duyck
- Alexei Starovoitov
- Daniel Borkmann
- Jesper Dangaard Brouer
- John Fastabend
- Jonathan Corbet (LWN coverage)
- Michael S. Tsirkin
- Qi Z Zhang
- Willem de Bruijn

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

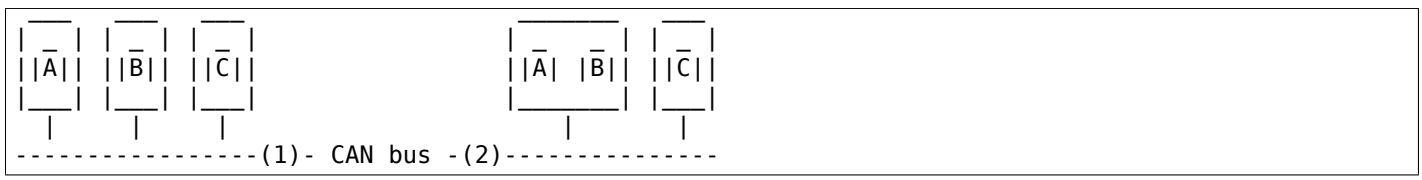
As described in *Motivation / Why Using the Socket API* the main goal of SocketCAN is to provide a socket interface to user space applications which builds upon the Linux network layer. In contrast to the commonly known TCP/IP and ethernet networking, the CAN bus is a broadcast-only(!) medium that has 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):



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 [1] 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 */
```

[1] 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* BASE DRIVER FOR THE INTEL(R) PRO/100 FAMILY OF ADAPTERS

June 1, 2018

Contents

In This Release

This file describes the Linux* Base Driver for the Intel(R) PRO/100 Family of Adapters. This driver includes support for Itanium(R)2-based systems.

For questions related to hardware requirements, refer to the documentation supplied with your Intel PRO/100 adapter.

The following features are now available in supported kernels:

- Native VLANs
- Channel Bonding (teaming)
- SNMP

Channel Bonding documentation can be found in the Linux kernel source: /Documentation/networking/bonding.txt

Identifying Your Adapter

For information on how to identify your adapter, and for the latest Intel network drivers, refer to the Intel Support website: http://www.intel.com/support

Driver Configuration Parameters

The default value for each parameter is generally the recommended setting, unless otherwise noted.

Rx Descriptors: Number of receive descriptors. A receive descriptor is a data structure that describes a receive buffer and its attributes to the network controller. The data in the descriptor is used by the controller to write data from the controller to host memory. In the 3.x.x driver the valid range for this parameter is 64-256. The default value is 256. This parameter can be changed using the command:

```
ethtool -G eth? rx n
```

Where n is the number of desired Rx descriptors.

Tx Descriptors: Number of transmit descriptors. A transmit descriptor is a data structure that describes a transmit buffer and its attributes to the network controller. The data in the descriptor is used by the controller to read data from the host memory to the controller. In the 3.x.x driver the valid range for this parameter is 64-256. The default value is 128. This parameter can be changed using the command:

```
ethtool -G eth? tx n
```

Where n is the number of desired Tx descriptors.

Speed/Duplex: The driver auto-negotiates the link speed and duplex settings by default. The ethtool utility can be used as follows to force speed/duplex.:

```
ethtool -s eth?  autoneg off speed {10|100} duplex {full|half}
```

NOTE: setting the speed/duplex to incorrect values will cause the link to fail.

Event Log Message Level: The driver uses the message level flag to log events to syslog. The message level can be set at driver load time. It can also be set using the command:

```
ethtool -s eth? msglvl n
```

Additional Configurations

Configuring the Driver on Different Distributions

Configuring a network driver to load properly when the system is started is distribution dependent. Typically, the configuration process involves adding an alias line to */etc/modprobe.d/*.conf* as well as editing other system startup scripts and/or configuration files. Many popular Linux distributions ship with tools to make these changes for you. To learn the proper way to configure a network device for your system, refer to your distribution documentation. If during this process you are asked for the driver or module name, the name for the Linux Base Driver for the Intel PRO/100 Family of Adapters is e100.

As an example, if you install the e100 driver for two PRO/100 adapters (eth0 and eth1), add the following to a configuration file in /etc/modprobe.d/:

```
alias eth0 e100
alias eth1 e100
```

Viewing Link Messages

In order to see link messages and other Intel driver information on your console, you must set the dmesg level up to six. This can be done by entering the following on the command line before loading the e100 driver:

```
dmesg -n 6
```

If you wish to see all messages issued by the driver, including debug messages, set the dmesg level to eight.

NOTE: This setting is not saved across reboots.

ethtool

The driver utilizes the ethtool interface for driver configuration and diagnostics, as well as displaying statistical information. The ethtool version 1.6 or later is required for this functionality.

The latest release of ethtool can be found from https://www.kernel.org/pub/software/network/ethtool/

Enabling Wake on LAN* (WoL)

WoL is provided through the ethtool* utility. For instructions on enabling WoL with ethtool, refer to the ethtool man page. WoL will be enabled on the system during the next shut down or reboot. For this driver version, in order to enable WoL, the e100 driver must be loaded when shutting down or rebooting the system.

NAPI

NAPI (Rx polling mode) is supported in the e100 driver.

See https://wiki.linuxfoundation.org/networking/napi for more information on NAPI.

Multiple Interfaces on Same Ethernet Broadcast Network

Due to the default ARP behavior on Linux, it is not possible to have one system on two IP networks in the same Ethernet broadcast domain (non-partitioned switch) behave as expected. All Ethernet interfaces will respond to IP traffic for any IP address assigned to the system. This results in unbalanced receive traffic.

If you have multiple interfaces in a server, either turn on ARP filtering by

1. entering:

```
   echo 1 > /proc/sys/net/ipv4/conf/all/arp_filter
```

   (this only works if your kernel's version is higher than 2.4.5), or

2. installing the interfaces in separate broadcast domains (either in different switches or in a switch partitioned to VLANs).

Support

For general information, go to the Intel support website at: http://www.intel.com/support/

or the Intel Wired Networking project hosted by Sourceforge at: http://sourceforge.net/projects/e1000 If an issue is identified with the released source code on a supported kernel with a supported adapter, email the specific information related to the issue to e1000-devel@lists.sf.net.

CHAPTER

FOUR

LINUX* BASE DRIVER FOR INTEL(R) ETHERNET NETWORK CONNECTION

Contents

Identifying Your Adapter

For more information on how to identify your adapter, go to the Adapter & Driver ID Guide at:

http://support.intel.com/support/go/network/adapter/idguide.htm

For the latest Intel network drivers for Linux, refer to the following website. In the search field, enter your adapter name or type, or use the networking link on the left to search for your adapter:

http://support.intel.com/support/go/network/adapter/home.htm

Command Line Parameters

The default value for each parameter is generally the recommended setting, unless otherwise noted.

NOTES: For more information about the AutoNeg, Duplex, and Speed parameters, see the “Speed and Duplex Configuration” section in this document.

For more information about the InterruptThrottleRate, RxIntDelay, TxIntDelay, RxAbsIntDelay, and TxAbsIntDelay parameters, see the application note at: http://www.intel.com/design/network/applnots/ap450.htm

AutoNeg

(Supported only on adapters with copper connections)

Valid Range 0x01-0x0F, 0x20-0x2F

Default Value 0x2F

This parameter is a bit-mask that specifies the speed and duplex settings advertised by the adapter. When this parameter is used, the Speed and Duplex parameters must not be specified.

NOTE: Refer to the Speed and Duplex section of this readme for more information on the AutoNeg parameter.

Duplex

(Supported only on adapters with copper connections)

Valid Range 0-2 (0=auto-negotiate, 1=half, 2=full)

Default Value 0

This defines the direction in which data is allowed to flow. Can be either one or two-directional. If both Duplex and the link partner are set to auto-negotiate, the board auto-detects the correct duplex. If the link partner is forced (either full or half), Duplex defaults to half- duplex.

FlowControl

Valid Range 0-3 (0=none, 1=Rx only, 2=Tx only, 3=Rx&Tx)

Default Value Reads flow control settings from the EEPROM

This parameter controls the automatic generation(Tx) and response(Rx) to Ethernet PAUSE frames.

InterruptThrottleRate

(not supported on Intel(R) 82542, 82543 or 82544-based adapters)

Valid Range 0,1,3,4,100-100000 (0=off, 1=dynamic, 3=dynamic conservative, 4=simplified balancing)

Default Value 3

The driver can limit the amount of interrupts per second that the adapter will generate for incoming packets. It does this by writing a value to the adapter that is based on the maximum amount of interrupts that the adapter will generate per second.

Setting InterruptThrottleRate to a value greater or equal to 100 will program the adapter to send out a maximum of that many interrupts per second, even if more packets have come in. This reduces interrupt load on the system and can lower CPU utilization under heavy load, but will increase latency as packets are not processed as quickly.

The default behaviour of the driver previously assumed a static InterruptThrottleRate value of 8000, providing a good fallback value for all traffic types,but lacking in small packet performance and latency. The hardware can handle many more small packets per second however, and for this reason an adaptive interrupt moderation algorithm was implemented.

Since 7.3.x, the driver has two adaptive modes (setting 1 or 3) in which it dynamically adjusts the InterruptThrottleRate value based on the traffic that it receives. After determining the type of incoming traffic in the last timeframe, it will adjust the InterruptThrottleRate to an appropriate value for that traffic.

The algorithm classifies the incoming traffic every interval into classes. Once the class is determined, the InterruptThrottleRate value is adjusted to suit that traffic type the best. There are three classes defined: “Bulk traffic”, for large amounts of packets of normal size; “Low latency”, for small amounts of traffic and/or a significant percentage of small packets; and “Lowest latency”, for almost completely small packets or minimal traffic.

In dynamic conservative mode, the InterruptThrottleRate value is set to 4000 for traffic that falls in class “Bulk traffic”. If traffic falls in the “Low latency” or “Lowest latency” class, the InterruptThrottleRate is increased stepwise to 20000. This default mode is suitable for most applications.

For situations where low latency is vital such as cluster or grid computing, the algorithm can reduce latency even more when InterruptThrottleRate is set to mode 1. In this mode, which operates the same as mode 3, the InterruptThrottleRate will be increased stepwise to 70000 for traffic in class “Lowest latency”.

In simplified mode the interrupt rate is based on the ratio of TX and RX traffic. If the bytes per second rate is approximately equal, the interrupt rate will drop as low as 2000 interrupts per second. If the traffic is mostly transmit or mostly receive, the interrupt rate could be as high as 8000.

Setting InterruptThrottleRate to 0 turns off any interrupt moderation and may improve small packet latency, but is generally not suitable for bulk throughput traffic.

NOTE: InterruptThrottleRate takes precedence over the TxAbsIntDelay and RxAbsIntDelay parameters. In other words, minimizing the receive and/or transmit absolute delays does not force the controller to generate more interrupts than what the Interrupt Throttle Rate allows.

CAUTION: If you are using the Intel(R) PRO/1000 CT Network Connection (controller 82547), setting InterruptThrottleRate to a value greater than 75,000, may hang (stop transmitting) adapters under certain network conditions. If this occurs a NETDEV WATCHDOG message is logged in the system event log. In addition, the controller is automatically reset, restoring the network connection. To eliminate the potential for the hang, ensure that InterruptThrottleRate is set no greater than 75,000 and is not set to 0.

NOTE: When e1000 is loaded with default settings and multiple adapters are in use simultaneously, the CPU utilization may increase non- linearly. In order to limit the CPU utilization without impacting the overall throughput, we recommend that you load the driver as follows:

```
modprobe e1000 InterruptThrottleRate=3000,3000,3000
```

This sets the InterruptThrottleRate to 3000 interrupts/sec for the first, second, and third instances of the driver. The range of 2000 to 3000 interrupts per second works on a majority of systems and is a good starting point, but the optimal value will be platform-specific. If CPU utilization is not a concern, use RX_POLLING (NAPI) and default driver settings.

RxDescriptors

Valid Range

- 48-256 for 82542 and 82543-based adapters
- 48-4096 for all other supported adapters

Default Value 256

This value specifies the number of receive buffer descriptors allocated by the driver. Increasing this value allows the driver to buffer more incoming packets, at the expense of increased system memory utilization.

Each descriptor is 16 bytes. A receive buffer is also allocated for each descriptor and can be either 2048, 4096, 8192, or 16384 bytes, depending on the MTU setting. The maximum MTU size is 16110.

NOTE: MTU designates the frame size. It only needs to be set for Jumbo Frames. Depending on the available system resources, the request for a higher number of receive descriptors may be denied. In this case, use a lower number.

RxIntDelay

Valid Range 0-65535 (0=off)

Default Value 0

This value delays the generation of receive interrupts in units of 1.024 microseconds. Receive interrupt reduction can improve CPU efficiency if properly tuned for specific network traffic. Increasing this value adds extra latency to frame reception and can end up decreasing the throughput of TCP traffic. If the

system is reporting dropped receives, this value may be set too high, causing the driver to run out of available receive descriptors.

CAUTION: When setting RxIntDelay to a value other than 0, adapters may hang (stop transmitting) under certain network conditions. If this occurs a NETDEV WATCHDOG message is logged in the system event log. In addition, the controller is automatically reset, restoring the network connection. To eliminate the potential for the hang ensure that RxIntDelay is set to 0.

RxAbsIntDelay

(This parameter is supported only on 82540, 82545 and later adapters.)

Valid Range 0-65535 (0=off)

Default Value 128

This value, in units of 1.024 microseconds, limits the delay in which a receive interrupt is generated. Useful only if RxIntDelay is non-zero, this value ensures that an interrupt is generated after the initial packet is received within the set amount of time. Proper tuning, along with RxIntDelay, may improve traffic throughput in specific network conditions.

Speed

(This parameter is supported only on adapters with copper connections.)

Valid Settings 0, 10, 100, 1000

Default Value 0 (auto-negotiate at all supported speeds)

Speed forces the line speed to the specified value in megabits per second (Mbps). If this parameter is not specified or is set to 0 and the link partner is set to auto-negotiate, the board will auto-detect the correct speed. Duplex should also be set when Speed is set to either 10 or 100.

TxDescriptors

Valid Range

- 48-256 for 82542 and 82543-based adapters
- 48-4096 for all other supported adapters

Default Value 256

This value is the number of transmit descriptors allocated by the driver. Increasing this value allows the driver to queue more transmits. Each descriptor is 16 bytes.

NOTE: Depending on the available system resources, the request for a higher number of transmit descriptors may be denied. In this case, use a lower number.

TxIntDelay

Valid Range 0-65535 (0=off)

Default Value 8

This value delays the generation of transmit interrupts in units of 1.024 microseconds. Transmit interrupt reduction can improve CPU efficiency if properly tuned for specific network traffic. If the system is reporting dropped transmits, this value may be set too high causing the driver to run out of available transmit descriptors.

TxAbsIntDelay

(This parameter is supported only on 82540, 82545 and later adapters.)

Valid Range 0-65535 (0=off)

Default Value 32

This value, in units of 1.024 microseconds, limits the delay in which a transmit interrupt is generated. Useful only if TxIntDelay is non-zero, this value ensures that an interrupt is generated after the initial packet is sent on the wire within the set amount of time. Proper tuning, along with TxIntDelay, may improve traffic throughput in specific network conditions.

XsumRX

(This parameter is NOT supported on the 82542-based adapter.)

Valid Range 0-1

Default Value 1

A value of '1' indicates that the driver should enable IP checksum offload for received packets (both UDP and TCP) to the adapter hardware.

Copybreak

Valid Range 0-xxxxxxx (0=off)

Default Value 256

Usage modprobe e1000.ko copybreak=128

Driver copies all packets below or equaling this size to a fresh RX buffer before handing it up the stack.

This parameter is different than other parameters, in that it is a single (not 1,1,1 etc.) parameter applied to all driver instances and it is also available during runtime at /sys/module/e1000/parameters/copybreak

SmartPowerDownEnable

Valid Range 0-1

Default Value 0 (disabled)

Allows PHY to turn off in lower power states. The user can turn off this parameter in supported chipsets.

Speed and Duplex Configuration

Three keywords are used to control the speed and duplex configuration. These keywords are Speed, Duplex, and AutoNeg.

If the board uses a fiber interface, these keywords are ignored, and the fiber interface board only links at 1000 Mbps full-duplex.

For copper-based boards, the keywords interact as follows:

- The default operation is auto-negotiate. The board advertises all supported speed and duplex combinations, and it links at the highest common speed and duplex mode IF the link partner is set to auto-negotiate.
- If Speed = 1000, limited auto-negotiation is enabled and only 1000 Mbps is advertised (The 1000BaseT spec requires auto-negotiation.)

- If Speed = 10 or 100, then both Speed and Duplex should be set. Auto- negotiation is disabled, and the AutoNeg parameter is ignored. Partner SHOULD also be forced.

The AutoNeg parameter is used when more control is required over the auto-negotiation process. It should be used when you wish to control which speed and duplex combinations are advertised during the auto-negotiation process.

The parameter may be specified as either a decimal or hexadecimal value as determined by the bitmap below.

Bit position	7	6	5	4	3	2	1	0
Decimal Value	128	64	32	16	8	4	2	1
Hex value	80	40	20	10	8	4	2	1
Speed (Mbps)	N/A	N/A	1000	N/A	100	100	10	10
Duplex			Full		Full	Half	Full	Half

Some examples of using AutoNeg:

```
modprobe e1000 AutoNeg=0x01 (Restricts autonegotiation to 10 Half)
modprobe e1000 AutoNeg=1 (Same as above)
modprobe e1000 AutoNeg=0x02 (Restricts autonegotiation to 10 Full)
modprobe e1000 AutoNeg=0x03 (Restricts autonegotiation to 10 Half or 10 Full)
modprobe e1000 AutoNeg=0x04 (Restricts autonegotiation to 100 Half)
modprobe e1000 AutoNeg=0x05 (Restricts autonegotiation to 10 Half or 100
Half)
modprobe e1000 AutoNeg=0x020 (Restricts autonegotiation to 1000 Full)
modprobe e1000 AutoNeg=32 (Same as above)
```

Note that when this parameter is used, Speed and Duplex must not be specified.

If the link partner is forced to a specific speed and duplex, then this parameter should not be used. Instead, use the Speed and Duplex parameters previously mentioned to force the adapter to the same speed and duplex.

Additional Configurations

Jumbo Frames

Jumbo Frames support is enabled by changing the MTU to a value larger than the default of 1500. Use the ifconfig command to increase the MTU size. For example:

```
ifconfig eth<x> mtu 9000 up
```

This setting is not saved across reboots. It can be made permanent if you add:

```
MTU=9000
```

to the file /etc/sysconfig/network-scripts/ifcfg-eth<x>. This example applies to the Red Hat distributions; other distributions may store this setting in a different location.

Notes: Degradation in throughput performance may be observed in some Jumbo frames environments. If this is observed, increasing the application's socket buffer size and/or increasing the /proc/sys/net/ipv4/tcp_*mem entry values may help. See the specific application manual and /usr/src/linux*/Documentation/ networking/ip-sysctl.txt for more details.

- The maximum MTU setting for Jumbo Frames is 16110. This value coincides with the maximum Jumbo Frames size of 16128.
- Using Jumbo frames at 10 or 100 Mbps is not supported and may result in poor performance or loss of link.
- Adapters based on the Intel(R) 82542 and 82573V/E controller do not support Jumbo Frames. These correspond to the following product names:

```
        Intel(R) PRO/1000 Gigabit Server Adapter
        Intel(R) PRO/1000 PM Network Connection
```

ethtool

The driver utilizes the ethtool interface for driver configuration and diagnostics, as well as displaying statistical information. The ethtool version 1.6 or later is required for this functionality.

The latest release of ethtool can be found from https://www.kernel.org/pub/software/network/ethtool/

Enabling Wake on LAN* (WoL)

WoL is configured through the ethtool* utility.

WoL will be enabled on the system during the next shut down or reboot. For this driver version, in order to enable WoL, the e1000 driver must be loaded when shutting down or rebooting the system.

Support

For general information, go to the Intel support website at:

http://support.intel.com

or the Intel Wired Networking project hosted by Sourceforge at:

http://sourceforge.net/projects/e1000

If an issue is identified with the released source code on the supported kernel with a supported adapter, email the specific information related to the issue to e1000-devel@lists.sf.net