SocketCAN - The official CAN API of the Linux kernel

Marc Kleine-Budde, Pengutronix

SocketCAN, the official CAN API of the Linux kernel, has been included in the kernel more than 3 years ago. Meanwhile, the official Linux repository has device drivers for all major CAN chipsets used in various architectures and bus types. SocketCAN offers the user a multiuser capable as well as hardware independent socket-based API for CAN based communication and configuration. In this paper we will at first focus on motivating the socket based approach used in SocketCAN and continue with a discussion about its supporting and opposing arguments and current limitations especially in comparison with other available Linux CAN stacks. We proceed with a close look at the structure and duties of a generic CAN driver. The usage of the most widespread userspace protocol for sending and receiving raw CAN frames (SOCK_RAW) is illustrated with a small program. The paper concludes with some remarks on the outlook of upcoming contributions and enhancements such as an isotp and a J1939 protocol implementation.

The first ideas of a socket based networking stack for CAN devices go back to 2002. There were several CAN implementations for Linux available back then, and some still are.

They take the classic character device approach, each CAN controller forming a device node, similar to serial device drivers, disregarding the Linux networking stack. These device nodes provide a simple, to some extent abstract interface to both send and receive CAN frames and configure the CAN controller.

Some of these projects are community driven while others being provided by hardware vendors. This and the nonstandard Linux CAN driver model approach has led to several drawbacks:[i]

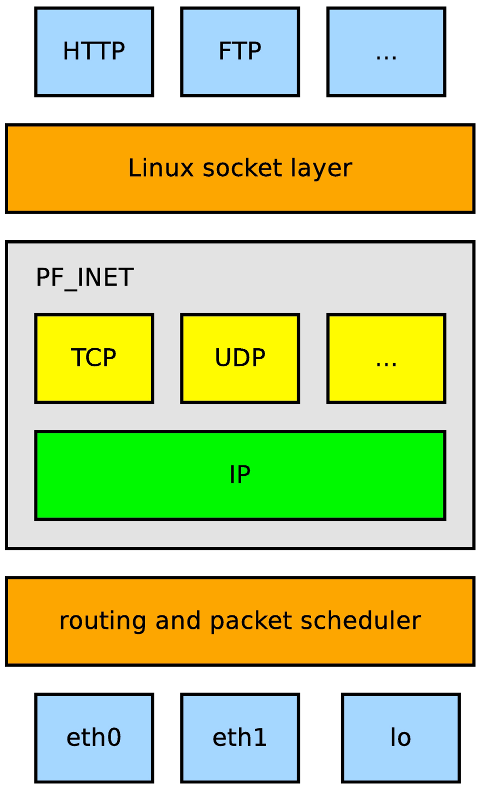
- Change of CAN hardware vendor urges the adaptation of the CAN application.
- Higher protocol levels and complex filtering have to be implemented in the userspace application.
- Only one application can use a CAN interface at a time.

Evolution of the character device drivers

LinCAN, which is part of the OCERA project, is based on “can-0.7.1” (originally created by Arnaud Westenberg) and addresses some of the already mentioned drawbacks. Especially the limitation to a single application per CAN interface has been removed by adding a “message processing” layer based on an “oriented graph of FIFOs”[ii].

The Linux networking subsystem

The Linux networking subsystem, widely known for the implementation of the TCP/IP protocol suite, is highly flexible. Next to IPv6 and IPv4, it contains several other networking protocols such as ATM, ISDN and the kernel part of the official Linux Bluetooth stack.



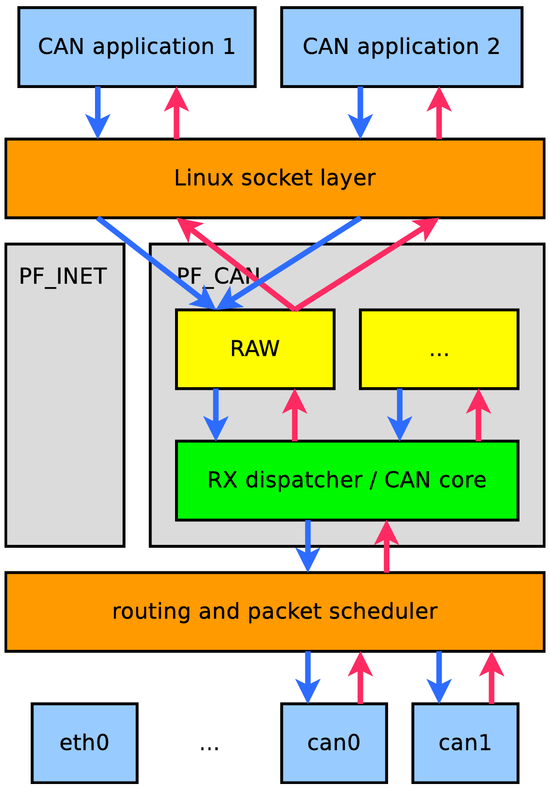
Taking a web or ftp server as an example, the above figure illustrates the different networking layers within the Linux kernel.

Starting at the application level, there is the standard POSIX socket API defining the interface to the kernel. Underneath follows the protocol layer, consisting of protocol families (here PF_INET) that implement different networking protocols. Within each family you can find several protocols (here TCP and UDP). Below this level the routing and packet scheduling layer exists followed by the last one that consists of the drivers for the networking hardware.

A socket-based approach

In order to bring CAN networking to the Linux kernel, CAN support has been added to the existing networking subsystem. This primarily consists of two parts:

1. A new protocol family **PF_CAN** including a **CAN_RAW** protocol,
2. the drivers for various CAN networking devices.



This approach brings several design advantages over the earlier mentioned character-device based solutions:

- Taking benefit of the existing and established POSIX socket API to assist the application developer.
- The new protocol family is developed against established abstractions layers, the socket layer above and the packet scheduler below.
- CAN network device drivers implement the same standardized networking driver model as Ethernet drivers.
- Communication protocols and complex filtering can be implemented inside the kernel.
- Support for multi-user and multi-application access to a single CAN interface is possible.

Multi-application support

The new protocol family **PF_CAN** and the **CAN_RAW** protocol have support for multiple CAN frame is transmitted to all applications which are interested.

But what about local originated CAN messages? Consider two embedded system connected via CAN, both systems are running a SocketCAN based application. Frames from one application reach the other application on the remote system via the CAN bus and vice versa. If these applications are concentrated on one system the CAN messages between these two applications still have to be exchanged. In order to guarantee the sequence of frames the local originated message is put into the RX queue, when the transmission on the wire is complete. This is usually done in the transmission complete interrupt handler of the sending CAN controller. Echo capable drivers set the “**IFF_ECHO**” flag in the netdevice's **flags** field, otherwise the framework will try to emulate echoing but the order of CAN frames cannot be guaranteed anymore.

Drawbacks and limitations

Using a networking subsystem which has been designed for much larger packages (64 bytes for a minimal Ethernet frame, compared to the maximal 8 data bytes in a can frame) brings more memory overhead than simpler character device solutions.

Moreover as the above figure indicates the packet scheduler is a shared resource among all networking devices (both Ethernet and CAN). Heavy traffic on the Ethernet finally leads to delays in CAN traffic. See paper "Timing Analysis of Linux CAN Drivers"[iii] for a more detailed analysis.

SocketCAN does not support hardware filtering of incoming CAN frames. Currently all CAN frames are received and passed to the CAN networking layer core, which processes the application specific filter lists. Activation of hardware filters would lead to an overall reduction of the received CAN traffic, but are – in contrast to the per application filters – a global setting. In a multi-user, multi-application scenario hardware filters are not feasible until the overall CAN design has been finalized and it is well known which CAN data is needed on the system.

CAN networking device drivers

With a high level of abstraction the functionality of a CAN networking device driver can be described as follows:

- The kernel driver initializes and configures the hardware,
- incoming CAN frames are received and pushed to the upper layer,
- outgoing frames are obtained from the upper layer and transmitted to the wire

The above listed requirements are – from this point of view – almost identical to Ethernet drivers, apart from the fact that instead of Ethernet frames, CAN frames are handled. The design decision is: Make use of the already established standard abstraction of an Ethernet driver in the Linux kernel for CAN networking drivers, as well.

Hardware initialization

The initialization and configuration of the hardware is usually a two-step process:

1. Once only via the "**probe()**" function.
2. Every time an interface is opened ("**ifconfig can0 up**") via the "**open()**" callback.

When a driver is loaded, the kernel calls the driver's "**probe()**" function once per suitable device. This function performs the basic initialization: all resources like address ranges, IRQ numbers, clocks and memory are requested.

```
static int flexcan_probe(
        structplatform_device *pdev)
{
    struct net_device *dev;
    struct flexcan_priv *priv;
[...]
    dev = alloc_candev(
        sizeof(struct flexcan_priv), 0);
    dev->netdev_ops =
        &flexcan_netdev_ops;
[...]
    priv = netdev_priv(dev);
    priv->can.bittiming_const =
        &flexcan_bittiming_const;
[...]
    return register_candev(dev);
}
```

As shown in the listing, the driver allocates the structure describing a networking device within the Linux networking stack as a "**struct net_device**". Several variables are assigned, most importantly "**netdev_ops**", which contains pointers to the management functions of the interface. CAN devices usually implement just three:

- The "**open**" and "**close**" callbacks point to the second part of the hardware configuration as already mentioned above.
- "**start_xmit**" - short for: start to transmit, is called by the packet scheduler when a CAN frame comes from an application and should be transmitted by the driver.

The next significant structure is "**struct bittiming_const**". It describes the bit timing limits (tseg1, tseg2, …) of the hardware in multiple of the Time Quantum, i.e. in a clock rate independent way.

The CAN framework contains an algorithm to calculate the actual bit timing parameter based on the requested bit rate, the current can clock and the bit timing parameters.
Finally with "**register_candev()**" the CAN device is registered at the networking subsystem. The device will now show up as a CAN interface ("**can0**") in the list of available interfaces ("**ifconfig -a**"), but remains inactive.

The second part of the hardware initialization is accomplished upon activation of the CAN interface by the user ("**ifconfig can0 up**"). The network layer calls the previously registered "**open()**" function. In this case the driver must finish the configuration and should be ready to send and receive CAN frames. This usually includes programming of the desired bitrate into the hardware, requesting an interrupt and activating the interrupt sources.

CAN frame reception

Looking at the receive path we can distinguish between CAN controllers with a single frame receive buffer (or mailbox) and controllers with multiple ones. As SocketCAN doesn't support receive filters (yet), all buffers and mailboxes must be configured to accept all CAN frames. If the hardware features permit, multiple buffers can be linked to a FIFO or circular buffer. Drivers must take care to preserve the order of incoming CAN frames, as higher level protocols and applications rely on the correct frame sequence.

With the reception of a CAN frame the controller issues an interrupt and Linux will execute the registered interrupt handler. There are two possibilities to handle incoming packets:

1. read the frames immediately in the IRQ handler or
2. schedule a routine to read multiple frames later in a software IRQ context. This technique is called "NAPI".

The immediate frame processing is used on controllers with a single receive buffer or low end hardware with a small FIFO depth.

Delayed processing depends on hardware with a large RX FIFO, capable of buffering incoming CAN frames until the NAPI handler starts. This procedure leads to less time spent inside the interrupt handler, which increases the system's reactivity and helps to decrease the IRQ load, as multiple CAN frames can be processed within a single scheduled NAPI request.

The following listing illustrates the steps to read a single frame and pass it to the upper layer – the packet scheduler.

```
static int flexcan_read_frame(
   struct net_device *dev)
{
    struct can_frame *cf;
    struct sk_buff *skb;

    skb = alloc_can_skb(dev, &cf);
    flexcan_read_fifo(dev, cf);
    netif_receive_skb(skb);

    return 1;
}
```

Here, two important data types are utilized:

- "**struct sk_buff**" The generic data type in the Linux kernel to describe a socket buffer in the networking stack. It includes meta data and the payload.
- "**struct can_frame**" The SocketCAN abstraction of a CAN frame, consists of the CAN id, the data length code (dlc) and the eight data bytes.

First, the driver allocates a generic buffer with the "**alloc_can_skb()**" function, which marks the buffer as a CAN frame and lets "**cf**" point to the "**struct can_frame**" within the buffer's payload. In this example a hardware dependent helper function "**flexcan_read_fifo()**" is called to read the CAN frame from the controller's FIFO and store it in the "**can_frame**" and thus into the generic buffer. In a NAPI capable driver like this the "**netif_receive_skb()**" function is used to push the CAN frame into the packet scheduler. From an IRQ handler the buffer is passed via the "**netif_rx()**" function instead.

CAN frame transmission

The transmission of a CAN frame is originated in the local system, usually in the userspace. For example: an application wants to send raw CAN frames (abstracted by the "**struct can_frame**"), a **CAN_RAW** socket is opened and a "**write()**" or "**send()**" system call is issued. The **CAN_RAW** protocol copies the CAN frame into the kernel space and passes it to the packet scheduler. During the run of the packet scheduler in a soft IRQ context the driver's "**start_xmit()**" function will be called to activate the transmission of the CAN frame.

The above outlined sequence is missing the flow control. An application might generate CAN frames faster than the CAN hardware is able to send. The CAN networking subsystem implements flow control in the layers above the packet scheduler. The driver has a simple, standard interface to control the flow of CAN frames coming from the packet scheduler.

Each interface has a transmission queue in the packet scheduler. During activation of the network interface (see "**open()**" callback above) the transmission queue is started. The packet scheduler may now call the driver's "**start_xmit()**" function to trigger a packet transmission. The driver must stop the queue if there are no more transmission buffers left in the hardware.

An "**xmit()**" function of a driver using only one transmit buffer looks as follows:

```
static int flexcan_start_xmit(
    struct sk_buff *skb,
    struct net_device *dev)
{
    struct can_frame *cf =
        (struct can_frame *)skb->data;

    netif_stop_queue(dev);
    can_put_echo_skb(skb, dev, 0);
    flexcan_write_frame(dev, cfe);

    return NETDEV_TX_OK;
}
```

The first argument of the "**xmit()**" function is a pointer to the generic socket buffer that should be transmitted. The buffer's payload ("**skb->data**") contains the standard SocketCAN frame. As the driver makes only use of one TX buffer, the queue is now stopped with the "**netif_stop_queue()**" function. Then the buffer is queued for later echoing "**can_put_echo_skb()**" (see multi application support above). Finally the CAN frame is transmitted by the hardware dependent function "**flexcan_write_frame()**". Returning **NETDEV_TX_OK** indicates the successful start of transmission to the packet scheduler
The "transmission complete" part of the interrupt handler is shown below:

```
static irqreturn_t flexcan_irq(
    int irq, void *dev_id)
{
    struct net_device *dev = dev_id;
    u32 reg_iflag1;

[...]

    /* transmission complete IRQ */
    if (reg_iflag1 & FLEXCAN_TX_IRQ) {
        flexcan_ack_tx_irq(dev);
        can_get_echo_skb(dev, 0);
        netif_wake_queue(dev);
    }

    return IRQ_HANDLED;
}
```

If a transmission complete interrupt is detected, it is first acknowledged, then the previously queued socket buffer is echoed back with "**can_get_echo_skb()**". As the only hardware TX buffer is now free, the queue has to be woken up with "**netif_wake_queue()**".

If the hardware implements a TX FIFO, the driver can make use of it, too. The TX queue stays active until all buffers in the hardware FIFO are occupied and is reactivated if there is free space in the FIFO again. If using more than one TX buffer at a time, the driver has – analogous to the RX path – to take care of preserving the order of CAN frames.

Applications and the CAN_RAW protocol

The simplest of methods to access the CAN bus is to send and receive raw CAN frames. This programming interface, which is similar to the character device drivers, is offered by the **CAN_RAW** protocol. The application developer creates a networking socket and uses the standard system calls to read and write CAN frames, represented by the "**struct can_frame**". A CAN frame is defined as following:

```
/* special address description flags
   for the CAN_ID */

/* EFF/SFF is set in the MSB */
#define CAN_EFF_FLAG 0x80000000U

/* remote transmission request */
#define CAN_RTR_FLAG 0x40000000U

/* error frame */
#define CAN_ERR_FLAG 0x20000000U

struct can_frame {
    /* 32 bit CAN_ID +
       EFF/RTR/ERR flags */
    canid_t can_id;

    /* data length code: 0 .. 8 */
    __u8    can_dlc;

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

"**can_id**" is 32 bit wide, holding the CAN id in the lower 11 bit. An extended CAN id is indicated by a set "**CAN_EFF_FLAG**" flag, the CAN id then covers the lower 29 bits. An RTR frame is signaled by the "**CAN_RTR_FLAG**" flag. "**can_dlc**" defines the number of used data bytes in the CAN frame, The payload of 8 bytes is located in the "**data**" array.

The following listing shows an example userspace program. First a socket is opened, the parameters of the "**socket()**" function request a "**CAN_RAW**" socket. Then the socket is "**bind()**" to the first CAN interface. A CAN frame is filled with data and then transmitted with the "**write()**" call. To receive CAN frames, "**read()**" is used in an analog way.

```
/* omitted vital #includes and error checking */

int main(int argc, char **argv)
{
    struct ifreq ifr;
    struct sockaddr_can addr;
    struct can_frame frame;
    int s;

    memset(&ifr, 0x0, sizeof(ifr));
    memset(&addr, 0x0, sizeof(addr));
    memset(&frame, 0x0, sizeof(frame));

    /* open CAN_RAW socket */
    s = socket(PF_CAN, SOCK_RAW, CAN_RAW);

    /* convert interface sting "can0" into interface index */
    strcpy(ifr.ifr_name, "can0");
    ioctl(s, SIOCGIFINDEX, &ifr);

    /* setup address for bind */
    addr.can_ifindex = ifr.ifr_ifindex;
    addr.can_family     = PF_CAN;

    /* bind socket to the can0 interface */
    bind(s, (struct sockaddr *)&addr, sizeof(addr));

    /* first fill, then send the CAN frame */
    frame.can_id = 0x23;
    strcpy((char *)frame.data, "hello");
    frame.can_dlc = 5;
    write(s, &frame, sizeof(frame));

    /* first fill, then send the CAN frame */
    frame.can_id = 0x23;
    strcpy((char *)frame.data, "iCC2012");
    frame.can_dlc = 7;
    write(s, &frame, sizeof(frame));
    close(s);

    return 0;
```

Outlook and Conclusion

Although the kernel internal interfaces are stable, there is a constant evolution in the kernel going on. Reevaluation of existing concepts, improvement, new features and consolidation doesn't stop before the SocketCAN core or it's drivers. For example at the time of writing the error handling in the CAN drivers are consolidated and unified.

Another interesting topic are CAN protocols (next to **CAN_RAW**) that have not been mentioned in this paper. There is **CAN_BCM**, which stands for broadcast manager. It's mainly used in the automotive domain where cyclic sending of messages is needed. The upcoming Kernel version v3.2 will be support **CAN_GW** a Kernel based gateway/router that routes CAN messages between CAN
interfaces and optionally modifies them. In development and not part of the official Kernel is support for **CAN_ISOTP which** implements the ISO 15765-2 CAN transport protocol. It allows a reliable point-to-point communication over a CAN infrastructure.
Development for automotive protocol SAE J1939 has just been started.

The SocketCAN framework presents the developer a multi application capable, standard POSIX socket based API to send and receive raw CAN frames independent from the used
CAN controller.
It further offers the driver developer a standard network driver model known from Ethernet drivers. The **PF_CAN** protocol layer provides a Kernel internal infrastructure for CAN frame sending/reception and filtering to implement more complex protocols inside the Kernel.

Marc Kleine-Budde
Pengutronix e.K.
Peiner Straße 6-8
31137 Hildesheim
Phone: +49 51 21 / 20 69 17 - 0
Fax: +49 51 21 / 20 69 17 - 55 55
mkl@pengutronix.de
www.pengutronix.de/