The CAN networking subsystem of the Linux kernel

Dr. Oliver Hartkopp, Volkswagen AG

Since Linux 2.6.25 (issued 2008-04-17) the Linux mainline kernel supports the network protocol family PF_CAN providing standardized programming interfaces for CAN users and CAN driver developers. This paper provides an overview of the implemented technologies and challenges to integrate the Controller Area Network into a non-real-time multiuser/multitasking operating system. Due to the standardized network driver model for CAN hardware a wide range of different CAN controllers and System-on-Chip CAN IP cores are supported by Linux out-of-the-box. In opposite to usual embedded CAN ECUs the Linux networking system is designed to handle multiple CAN applications using multiple CAN busses at the same time. The integration of the CAN infrastructure into the networking stack allows to implement CAN-specific transport protocols like ISO 15765-2 or high-performance CAN frame gateways inside the operating system context. Finally the paper discusses solutions for expected prioritization issues when executing multiple CAN applications and summarizes requirements for Linux-preferred CAN controller concepts.

Today the CAN bus is used in a wide range of control applications. Usually the content based addressing is practiced to transfer information by sending single CAN frames. In the automotive context cyclic transmissions are used to detect the absence of the data source within a reasonable time. E.g. when the cyclic status information of a CAN controlled media player is not received every 200ms the media player can be disabled in the HMI of the vehicles Infotainment system.
Additionally so-called CAN transport protocols are used in the vehicles on-board diagnostics (OBD) to provide a virtual point-to-point connection between the diagnosis tester and the CAN ECU. With help of these CAN transport protocols (like ISO 15765-2 [1]) PDU length up to 4095 bytes can be transferred via CAN which is used e.g. for firmware updates.

Character device drivers for CAN

Having these two use cases in mind the known CAN drivers for Linux were not able to fulfill any of them appropriately in 2002. Due to the fact that known CAN drivers based on the character device driver model the interface to the CAN controller was simple and direct – as known from character based serial drivers. The various CAN character device drivers presented a more or less abstract programming interface that is specific to CAN controller capabilities or vendor requirements. Especially the functionality provided by the CAN driver is reduced to apply CAN identifier filters.

To implement content filters for the payload of cyclic CAN messages these messages had to be processed in user-space context. E.g. if only a state change in the payload was relevant for the application each received CAN message had to be transferred from kernel-space into user-space for a comparison with the former one.

The implementation of a CAN transport protocol has to fulfill restrictive timing requirements down to a few milliseconds. This cannot be ensured in the user-space context where several processes in a multitasking system share the same CPU. Depending on the system load the processes get their CPU time regulated by the system scheduler and the resolution of the system timer. The common alternative to implement CAN transport protocols for multitasking operating systems is to have a separated embedded CAN CPU or to use a real-time variant of the selected multitasking operating system (OS).

CAN networking in Linux

The point-to-point network communication in multitasking operating systems is well known from the Internet protocol (IP) communication. Different connection

oriented and connectionless protocols based on the IP like the TCP and UDP are state of the art in common operating systems. Adapting this internet networking technology to realize the similar requirements of CAN transport protocols leads to an implementation of the CAN access inside the OS network stack.

This approach has several advantages over the formerly described character device driver model:

- Standard programming interfaces for system calls ('network sockets')
- Standard network driver model known from Ethernet drivers
- Established abstraction layers in the network stack
- Communication protocols implemented inside the operating system context
- Multiuser access to the network

The definition of a network device driver interface for the Controller Area Network is the first step to unify the CAN bus access. As known from Ethernet networking the driver abstraction allows the replacement of networking hardware without modifying the user applications like e-mail clients or web browsers. On the other hand the CAN network device definition allows the hardware vendor to focus on the driver development without being in charge to specify and implement user-space programming interfaces and tools.

The direct access to network devices can be performed with a privileged access to the PF_PACKET socket interface. But using the PF_PACKET protocol family sockets for CAN network devices has some vital drawbacks:

1. Privileged access rights are needed (Administrator only)
2. No full network transparency for local CAN applications (See [2])
3. No efficient traffic filtering based on CAN identifiers
4. No integration of CAN specific (transport) protocols

Especially the missing network transparency (point 2) turns out as a knock-out criterion in a multiuser environment, as different CAN applications would have a different view of the existing CAN bus traffic [2].

Network protocol family PF_CAN

To overcome with the limitations caused by accessing CAN network devices with PF_PACKET sockets a new protocol family for the Controller Area Network has been created. The integration of this PF_CAN protocol family into the Linux network stack can be derived from existing network protocol families like DECnet, Appletalk or AX.25 that also use the Linux networking infrastructure with different networking hardware and protocols.

To establish a separate data path for CAN messages and CAN devices the data buffers and network devices are identified with new type definitions:

- ETH_P_CAN identifies network data buffers that contain CAN messages
- ARPHRD_CAN identifies CAN network devices that handles network data buffers marked as ETH_P_CAN type

Based in these definitions a new PF_CAN protocol family implementation can register itself to be responsible for ETH_P_CAN marked buffers containing CAN messages.

Together with the PF_CAN/AF_CAN value definitions and a new CAN socket address structure the missing link for the socket programming interface for user-space applications is established. From this point the protocol family PF_CAN provides a framework that manages the CAN data flow from the user to the CAN network driver and vice versa.

Inside the operating system PF_CAN offers programming interfaces for various CAN based protocols. Comparable to UDP and TCP being different protocols of the protocol family PF_INET for internet protocol networking new CAN relevant functionalities can be hosted inside the PF_CAN framework. The different CAN protocols can be accessed from user-space applications with different protocol numbers that are functional available

when a CAN protocol implementation registers itself at the PF_CAN framework.

To provide a multiuser access to CAN traffic the PF_CAN core offers internal functions to CAN protocol implementations to send and receive CAN frames [3]:

can_send()
Send a data buffer containing a CAN frame to a CAN device. Performs sanity checks for CAN frames (e.g. dlc) and ensures the local echo functionality.

can_rx_register()
Register a callback function that is executed on the reception of a CAN identifier that matches the given mask/value filter.

can_rx_unregister()
Remove a formerly registered subscription.

Depending on the given mask/value filter different filtersets are used to reduce the CPU consumption when checking the registered subscriptions at runtime. The per-interface created filtersets are checked in a software-interrupt on every CAN frame reception. Due to the described multiuser requirement the CAN drivers intentionally do not support any system wide CAN identifier filters. The performance of the software-interrupt based CAN filters has been evaluated in [4].

CAN Protocols in PF_CAN

The protocol CAN_RAW offers a similar programming interface as known from the CAN character device drivers. Analogue to opening a character device file in Linux (e.g. /dev/can0) the CAN application programmer creates a network socket and reads/writes specific data structures representing CAN frames.

Additional to the fact that multiple instances of CAN_RAW sockets can be created at the same time several different CAN identifier filters can be applied to each created socket separately. Due to the different filters for each CAN_RAW socket a specific view to the selected CAN bus traffic can be archived and separately handled by different applications and users on the system.

The CAN_RAW sockets identifier filters reference directly to the per-interface filtersets provided by the PF_CAN framework. The simple callback registration delivers the subscribed CAN frames directly to the requesting socket instance to be read by the user-space CAN application.

In automotive networks the CAN messages are not only used to transfer signal values in the CAN frames payload. Sending CAN frames periodically e.g. every 200ms allows to detect failures of the originating CAN node. This mode of operation has two drawbacks regarding the bus load and the message processing. Even when the reception of CAN traffic is reduced by filters and/or message buffers the CAN frame payload needs to be checked whether the relevant signal has changed.

To reduce the effort for payload content filtering the CAN broadcast manager protocol (CAN_BCM) has been integrated into PF_CAN. The CAN_BCM is a programmable content filter and timeout handler for incoming CAN frames and can also manage the sending of cyclic messages in operating system context.
Due to the Linux high resolution timers the precise sending of messages is independent from process scheduling. The content filter that checks changes in the 64 bit payload is processed in the software interrupt at CAN frame reception time to ensure that only relevant changes are passed to the CAN application in user-space.

CAN_BCM transmission path functions:

- Cyclic transmission of a CAN message using a given interval time
- Redefinition of CAN message content and interval timers at runtime
- Counting of performed intervals with automatic switching to a second interval time

- Immediate transmission of changed CAN message content with/without starting the timer interval
- Single transmission of CAN messages

BCM receive path functions:

- Receive filters to check changes in relevant CAN message elements (payload, data length code)
- Receive filters for single CAN identifiers (no content matching)
- Receive filters for multiplex CAN messages (e.g. with indices inside the CAN message payload)
- Receive filters for monitoring the data length code of the CAN message
- Respond to receive transmission request (RTR) CAN messages
- Time-out-monitoring of CAN messages
- Reduction of the update rate for content filter changes messages (throttling)

Depending on the use case payload content filtering and throttling can provide a significant reduction of the system load (see [5] p. 102-103). Additionally the comfortable CAN_BCM socket programming interface reduces the complexity of the CAN application and supersedes a complex and imprecise timer handling in user-space.

CAN transport protocols in OS context

Known solutions that implement CAN transport protocols use an embedded CAN processor in the CAN hardware interface to ensure the ambitious protocol timing requirements. The availability of Linux high resolution timers for the precise sending of CAN messages lead to the question if these timing requirements of CAN protocols can be handled inside the PF_CAN framework. A public available implementation [6] of the ISO 15765-2 CAN transport protocol [1] has been evaluated and compared to a commercial diagnosis tool [7]. It turned out that minimum response time of the Linux implementation was at least twice the time and in the worst case ten times longer than with the embedded CAN CPU solution. As the 4ms delay of the worst case scenario was still covered by the specified protocol timeout of 1000ms the PF_CAN based implementation is conform to ISO 15765-2. Besides the measured overall timing ranges the open source implementation processed the CAN protocol messages two times faster than the embedded solution (average values).

CAN frame routing using PF_CAN

In internet protocol networking the routing and forwarding of IP traffic through different network devices is common practice. The routing and modifying operations for IP packets are based on IP addressing schemata. As the qualified CAN addressing is based on the CAN identifier and the CAN bus interface the existing routing implementations inside the operating system context are not suitable for the Controller Area Network.

Attempts to route CAN frames efficiently with user-space applications like candump [8] failed through the existing process scheduling which lead to drops and non-deterministic delays depending in the size of the per-socket receive buffer size.

To integrate an efficient CAN message routing in Linux the created CAN gateway (CAN_GW) makes use of the PF_CAN filter subscription infrastructure. The registered filters trigger callback functions in the CAN gateway where the received CAN frames can optionally be modified before they are sent to the outgoing CAN interface. The one-hop CAN message forwarding supports the following on-the-fly modifications of the CAN frame elements CAN-ID, CAN-DLC, CAN-DATA:

1. AND: logical ‘and’ element with value
2. OR: logical ‘or’ element with value
3. XOR: logical ‘xor’ element with value
4. SET: set element to new value

After performing one or more of these optional CAN frame modifications in the described order a potentially corrupted payload data checksum can be built on demand. Additional to a simple XOR checksum calculation for a given data set, three CRC8 profiles can be selected that can calculate a CRC8 value based on an

individual 256 byte CRC table. With this functionality the CAN gateway is able to create AUTOSAR End-to-End-Library [9] compliant CAN messages on-the-fly even after modification.

The implementation of the CAN gateway is optimized towards performance as the optional modification and checksum calculating operations are executed in the software interrupt context. Detailed performance measurements of CAN_GW have been made [4]. The CAN gateway implements the Linux routing capability for the protocol family PF_CAN and supports the netlink configuration interface [10] that is also used for the routing configuration of internet protocol data.

CAN frame traffic shaping

Due to the new multiuser capabilities of the CAN networking implementation in Linux a new challenge to manage the host access to a single CAN bus medium arises. For example a CAN application A sends a status information on CAN-ID 0x123 every 200ms. Application B sends a 4095 byte PDU via ISO 15765-2 without a tx delay requested from the communication partner. This leads to an expected transmission sequence like:

```
deltaT ID      data
0.200  0x123  00 00 00 20
0.200  0x123  00 00 00 21
0.200  0x123  00 00 00 22
0.100  0x730  02 03 04 05 06 07 08 09
0.001  0x730  02 03 04 05 06 07 08 09
0.001  0x730  02 03 04 05 06 07 08 09
0.001  0x730  02 03 04 05 06 07 08 09
0.001  0x730  02 03 04 05 06 07 08 09
(… 500 CAN frames later …)
0.001  0x730  02 03 04 05 06 07 08 09
0.001  0x730  02 03 04 05 06 07 08 09
0.001  0x123  00 00 00 23
0.001  0x123  00 00 00 24
0.001  0x123  00 00 00 25
0.001  0x123  00 00 00 26
0.200  0x123  00 00 00 27
0.200  0x123  00 00 00 28
```

Due to the fact that the ISO 16765-2 protocol pushed more than 500 CAN frames en-bloc into the CAN network device queue the relevant status information is not send on the CAN bus for the ISO-TP frames transmission time. Depending on the used bitrate the block transfer time the cyclic status information can be inhibited which leads to inconsistencies and timeout errors in the data sink.

The described data flow effect is a known problem in internet protocol networking too. To be able to browse the web in a smooth way while having peer-to-peer data communication the traffic can be controlled depending on its content. This so-called ‘traffic shaping’ for peer-to-peer connections is part of a traffic control framework inside the Linux networking. The various traffic control capabilities to prioritize, throttle or drop packets are designed to handle internet protocol traffic and are not aware of Controller Area Network identifiers. Therefore the classifiers to sort and separate packets into network queues have been extended with a new CAN classifier implementation.

This CAN classifier support enables the usage of the entire traffic control (TC) capabilities of the Linux networking subsystem [11] although not every queueing discipline is suitable for CAN use cases. The traffic control configuration requires administrative access rights which concentrate the host configuration to a single point. This allows setting up the local CAN node fulfilling the requirements of the different CAN applications and the other CAN bus participants.

This setup can be used to solve the initial example with the modified ‘tc’ tool [12]:

```
# create a can0 prio traffic control handle
$ tc qdisc add dev can0 root handle 1: prio

# sort CAN-ID 0x123 to a separate queue
$ tc filter add dev can0 parent 1:0 prio 1 \
      can sffid 0x123 flowid 1:1

# sort CAN-ID 0x124 to a separate queue
$ tc filter add dev can0 parent 1:0 prio 2 \
      can sffid 0x124 flowid 1:2

# catch the rest into the default class
$ tc filter add dev can0 parent 1:0 prio 3 \
      can sffid 0x0:0x0 flowid 1:3
```

When the CAN identifier 0x123 is to be sent, the CAN frame is sorted into the prio 1 queue. The frames in the prio 3 queue are sent to the CAN bus when the higher priority queues are empty.

An alternative utilization for traffic control can be the throttling of outgoing traffic with a token bucket filter. This can be used to slow down the traffic on a local virtual CAN interface to have a realistic throughput like 125 kbit/s even without real CAN hardware. Usually the runtime created virtual CAN network devices are not limited in bandwidth like the loopback device for local internet protocol traffic.

Multiuser requirements obviously need to be solved on the host differently than in known embedded systems which has an impact on the CAN network driver implementation. Most CAN controllers provide a set of several transmit objects that can be accessed via different memory registers. Based e.g. on the CAN identifiers stored in the transmit objects an 'intelligent' CAN controller is able to decide which object to transmit first. To be able to specify the sequence of sent CAN messages with queueing disciplines the CAN controller has be configured in a simple FIFO mode. Therefore only one transmit object is used in the Linux CAN network drivers even when the CAN controller itself provides more than one transmit object. To ensure the maximum CAN throughput so-called shadow registers are used if available. Shadow registers allow writing the next CAN message into the CAN controller while the current transmission is in progress.

The CAN classifier for Linux queueing disciplines is currently evaluated with a prototypic implementation [13]. Analogue to the CAN gateway this traffic control approach has the potential to become part of the Linux mainline kernel.

Summary

The Linux subsystem for the Controller Area Network standardized the programming interfaces for CAN driver programmers and CAN application programmers. In comparison to a programming interface with a character device driver model, the chosen approach allows to implement various CAN specific communication protocols inside the operating system as well as the reuse of established and powerful networking techniques like network traffic control. The clear and simple network device driver interface known from Ethernet drivers leads to an easy exchangeability of CAN hardware without changing the existing applications. The Open Source development process in the Linux kernel enables the contribution of new CAN drivers and CAN based protocols for everyone in an open community and is independent from single vendors.

For novice CAN application developers there is no break in the programming philosophy known from other networking technologies and the stable socket programming interface promises a protection of investment. Especially CAN drivers that found their way into the Linux mainline kernel are continuously maintained and fixed in a community process so that they stay operational for all upcoming Linux kernel versions.

Linux targets a wide range of embedded systems and control applications. With the introduced network protocol family PF_CAN the Controller Area Network is supported in the Linux operating system like any other communication network and embedded serial bus system out-of-the-box.

References

[1] "ISO 15765-2: Road vehicles – Diagnostics on Controller Area Networks (CAN) - Part 2: Network layer services," Geneva, Switzerland, 2004.

[2] "CAN documentation - Linux Cross Reference," 2011. [Online]. Available: http://lxr.linux.no/#linux+v3.0/Documentation/networking/can.txt#L175. [Accessed 12 12 2011].

[3] "CAN core.h - Linux Cross Reference," [Online]. Available: http://lxr.linux.no/#linux+v3.0/include/linux/can/core.h#L44. [Accessed 12 12

2011].

[4] M. Sojka, P. Pisa, S. Ondrej, O. Hartkopp and Z. Hanzalek, "Timing Analysis of a Linux-Based CAN-to-CAN Gateway," Proceedings of the 13th Real Time Linux Workshop, 20 10 2011. [Online]. Available: https://lwn.net/images/conf/rtlws-2011/paper.22.html. [Accessed 12 12 2011].

[5] O. Hartkopp, "Programmierschnittstellen für eingebettete Netzwerke in Mehrbenutzerbetriebssystemen am Beispiel des Controller Area Network (Dissertation)," [Online]. Available: http://edoc.bibliothek.uni-halle.de/servlets/DocumentServlet?id=9738. [Accessed 12 12 2011].

[6] "ISO 15765-2 implementation on SocketCAN project repository," [Online]. Available: https://gitorious.org/linux-can/can-modules/blobs/master/net/can/isotp.c. [Accessed 12 12 2011].

[7] "VAS5163 Workshop Equipment," [Online]. Available: http://www.dne-gmbh.de/elektronik/de/gast5163/index.htm. [Accessed 12 12 2011].

[8] "candump implementation on SocketCAN project repository," [Online]. Available: https://gitorious.org/linux-can/can-utils/blobs/master/candump.c. [Accessed 12 12 2011].

[9] AUTOSAR, "Specification of SW-C End-to-End Communication Protection Library," [Online]. Available: http://www.autosar.org/download/R4.0/AUTOSAR_SWS_E2ELibrary.pdf. [Accessed 12 12 2011].

[10] "netlink Kernel API - The Linux Foundation," [Online]. Available: http://www.linuxfoundation.org/collaborate/workgroups/networking/netlink. [Accessed 12 12 2011].

[11] W. Almesberger, "Linux network traffic control - implementation overview," Proceedings of 5th Annual Linux Expo, Raleigh, NC, May 1999, pp. 153-164, [Online]. Available: http://www.almesberger.net/cv/papers/tcio8.pdf. [Accessed 12 12 2011].

[12] "CAN modified TC tool for Linux traffic control," [Online]. Available: https://rtime.felk.cvut.cz/gitweb/lisovros/iproute2_canprio.git/blob/HEAD:/tc/f_can.c. [Accessed 12 12 2011].

[13] "CAN classifier for Linux traffic control," [Online]. Available: https://rtime.felk.cvut.cz/gitweb/lisovros/linux_canprio.git/blob/canprio:/net/sched/cls_can.c. [Accessed 12 12 2011].

Dr. Oliver Hartkopp
Volkswagen AG
Brieffach 1777
38436 Wolfsburg, Germany
+49 5361 9 36244
oliver.hartkopp@volkswagen.de
http://www.volkswagenag.com