Linux CAN XL support and programming

Dr. Oliver Hartkopp (Volkswagen)

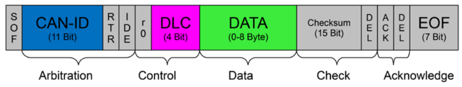
NMEA 2000 is a plug-and-play communications CAN-based standard used for connecting marine sensors and display units within ships and boats. It sits amongst other NMEA marine communications protocols from NMEA 0183 at the lower-end through to the Ethernet-based NMEA ONENET standard. NMEA 2000 itself uses many of the features that are in common with SAEJ1939 and ISO11783. The standard has enabled the easy integration of electronic devices into a vessel.
However, as with all CAN-based protocols, several vulnerabilities to cyberattacks have been identified. Many are at the CAN level, whilst others are in common with those protocols from the SAEJ1939 family of protocols.
This paper will discuss the known vulnerabilities that have been identified with the NMEA 2000 protocol. These include weaknesses with the address claim and transport protocols, and covert communication channels using methods based on steganography. Activities with the aim of making NMEA 2000 robust to cyberattacks are described.

Twenty years ago the SocketCAN concept was used to carry out CAN communication in two vehicle demonstators. Since then the Linux socket application programming interface (API) and the application binary interface (ABI) remains stable in a way that a todays Linux kernel can still execute those 20 year old binaries – given the same processor architecture.

When the SocketCAN project was applied to the Linux kernel 2.6.25 in January 2008 the API was „carved in stone“. Since then an infrastructure e.g. to configure bitrate settings for CAN network interfaces and a number of additional network layer functionalities (e.g. ISO15765-2, J1939) was added. A huge extension of the APIs to handle CAN related content and the CAN interface configuration became necessary when CAN FD was announced at the iCC 2012.

The evolution of CAN frame data structures

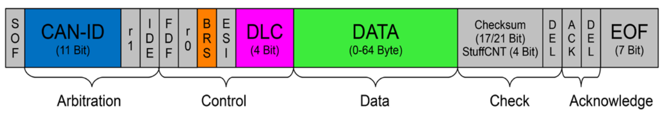
The common CAN CC (Classical CAN) data structure is used whenever CAN frames need to be exchanged between the application (user space) and the Linux OS kernel (kernel space):
The struct can_frame has a fixed size of 16 byte and contains the 11/29 bit CAN




```
struct can_frame {
        canid_t can_id;  /* 32 bit CAN_ID + EFF/RTR/ERR flags */
        __u8    len;     /* frame payload length in byte (0 .. 8) */
        __u8    __pad;   /* padding */
        __u8    __res0;  /* reserved / padding */
        __u8    len8_dlc; /* optional DLC for 8 byte payload length */
        __u8    data[8] __attribute__((aligned(8)));
};
```

identifier, a data length information, some flags (e.g. RTR) and a comparably new len8_dlc element to be able to send and receive CC DLC values from 9 .. 15 when the data length is 8 byte.

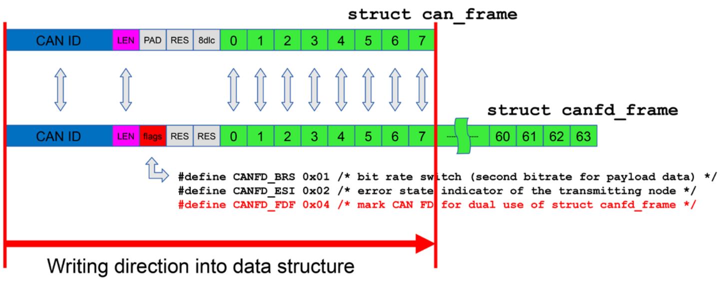
With CAN FD the number of data bytes was extended to 64 byte which increases the size of the struct canfd_frame to 72 byte. A flags element has been added to carry new CAN FD specific bits like the Error State Indicator (ESI) and the Bit Rate Setting (BRS) to be able to switch to the second bit rate in the data section on a per frame basis.




```
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)));
};
```

When the Linux network socket has been enabled to receive CAN FD frames with a so-called setsockopt() system call, a reading of CAN frames can pass either a CAN CC frame or a CAN FD frame. Therefore the data structure used to store the received CAN frame has to able to contain the maximum frame size (72 byte). The number of received bytes which is the return value of the read() syscall will then be either 16 byte (CAN CC) or 72 byte (CAN FD).

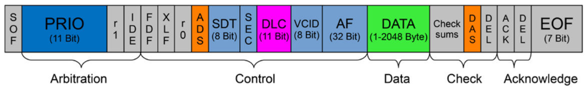
As the two data structures share the same layout it is possible to store CAN CC frame content into a struct canfd_frame:



When using the struct canfd_frame in this kind of dual-use mode, the CANFD_FDF bit can be used to identify the CAN frame type. Since Linux v6.1 the CANFD_FDF bit is automatically set in the flags element when reading CAN FD frames from the CAN_RAW socket.

With the introduction of CAN XL not only the support for a third data bit rate has to be implemented in the CAN network driver infrastructure but the former concept of copying the fixed maximum size of the CAN XL frame data structure has to be questioned. In the case of a CAN XL data length of less than 2048 only the required number of data bytes should be copied between the user space and the Linux kernel space and vice versa. When the setsockopt() to receive CAN XL frames is enabled the read() system call now might return length values of 16 (CAN CC), 72 (CAN FD) or 13 .. 2060 byte for CAN XL, where the CAN XL header content has a size of 12 byte followed by 1 .. 2048 byte of data.

As the possible length values from CAN XL read() operations now cover the existing length values of CAN CC and CAN FD



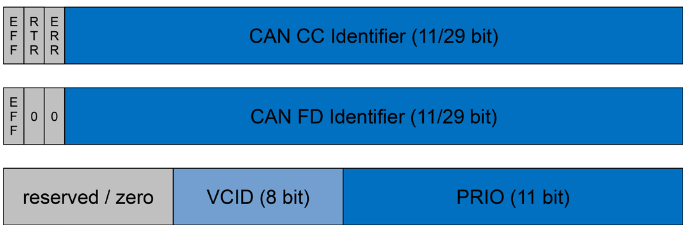

```
struct canxl_frame {
        canid_  prio;   /* 11 bit priority for arbitration / 8 bit VCID */
        __u8    flags;  /* additional flags for CAN XL */
        __u8    sdt;    /* SDU (service data unit) type */
        __u16   len;    /* frame payload length in byte (1 .. 2048) */
        __u32   af;     /* acceptance field */
        __u8    data[2048];
};
```

the CAN XL frame needs an alternative indicator. Therefore the newly introduced CANXL_XLF bit with the value 0x80 (dec 128) is always set in the canxl_frame.flags element. As the canxl_frame.flags element shares the position of the can_frame.len and canfd_frame.len elements, the value of the CANXL_XLF bit can be clearly identified because the length information of CC/FD frames can only have values up to 64.

When e.g. using a C-union which contains a struct can_frame, a struct canfd_frame and a struct canxl_frame the flow of checks after the read() system call could look like this:

1. read bytes < 0 ➔ standard Linux read() failure: error handling for read() failure
2. read bytes < 13 ➔ not even a CAN XL header: error handling for read() failure
3. canxl_frame.flags & CANXL_XLF == CANXL_XLF ➔ CAN XL frame
4. read bytes == 72 ➔ CAN FD frame
5 read bytes == 16 ➔ CAN CC frame
6. error handling for read() failure

The 11 bit CAN XL priority filtering is processed with the established 32 bit mask/value filters of the CAN_RAW socket analogue to the CAN identifier filtering for 11 bit CAN IDs of CAN CC/FD frames. The 8 bit CAN XL Virtual CAN Network Identifier (VCID) is placed in the bits 16 .. 23 of the CAN XL priority element. The bits 24 .. 31 have to be set to zero.



A VCID value of zero represents an „untagged" CAN XL frame. Since Linux v6.9 the

CAN_RAW socket supports the filtering and the generation of VCID values via setsockopt() interface to be able to read and write VCID content. This dedicated VCID handling for CAN XL frames can be cascaded with the common 32 bit CAN_RAW receive filter.

Summarizing the evolution of the CAN frame data structures and the socket options that enable the CAN FD and CAN XL traffic, the initial paradigm to access CAN content over Linux network sockets could be maintained over time. This helps application programmers to get used to work with CAN XL based on their existing knowledge about how to work with CAN CC and CAN FD setups.

Virtual CAN interfaces

The virtual CAN interfaces in Linux provide a local echo of CAN content, so that multi user applications on the same host can interact with each other via CAN. In addition to the virtual CAN driver vcan, the vxcan can establish local CAN traffic between different network namespaces which is e.g. needed for containerization (LXC/Docker/etc). These two virtual CAN drivers have been upgraded together with the initial network layer support for CAN XL in Linux v.6.1 in December 2022.

The different supported CAN proto cols (CC/FD/XL) for virtual CAN interfaces are configured by the maximum transfer unit (MTU) value of the virtual CAN device. Like on real CAN interfaces it is possible to configure the virtual CAN bus for CAN CC, CAN FD or CAN XL – where CAN XL covers CAN FD/ CC content and CAN FD covers CAN CC content. To limit the CAN XL data length on a CAN XL bus segment e.g. to meet real-time requirements it is possible to reduce the CAN XL MTU which then enforces the maximum CAN XL data length on that CAN XL interface.

- MTU = 16 ➔ CAN CC interface, data length 0 .. 8
- MTU = 72 ➔ CAN FD interface, data length 0 .. 64 (default MTU since Linux v4.12)
- MTU = 76 .. 2060 ➔ CAN XL interface, data length 1 .. 2048

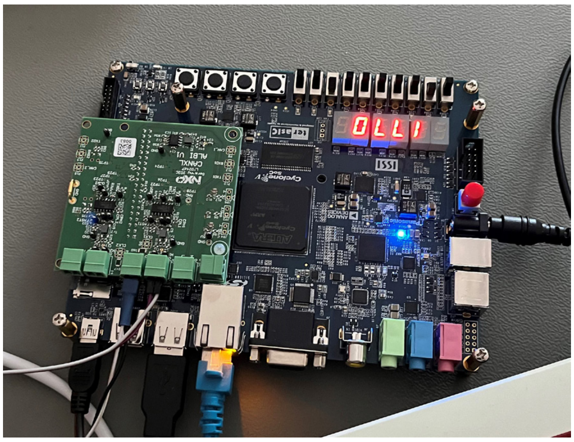
To create a virtual CAN XL interface vcanxl0 the ip tool from the iproute2 package is needed:

- ip link add name vcanxl0 type vcan
- ip link set vcanxl0 mtu 2060
- ip link set vcanxl0 up

At this point vcanxl0 can be used as virtual CAN interface for Linux CAN XL applications.

CAN XL hardware driver support

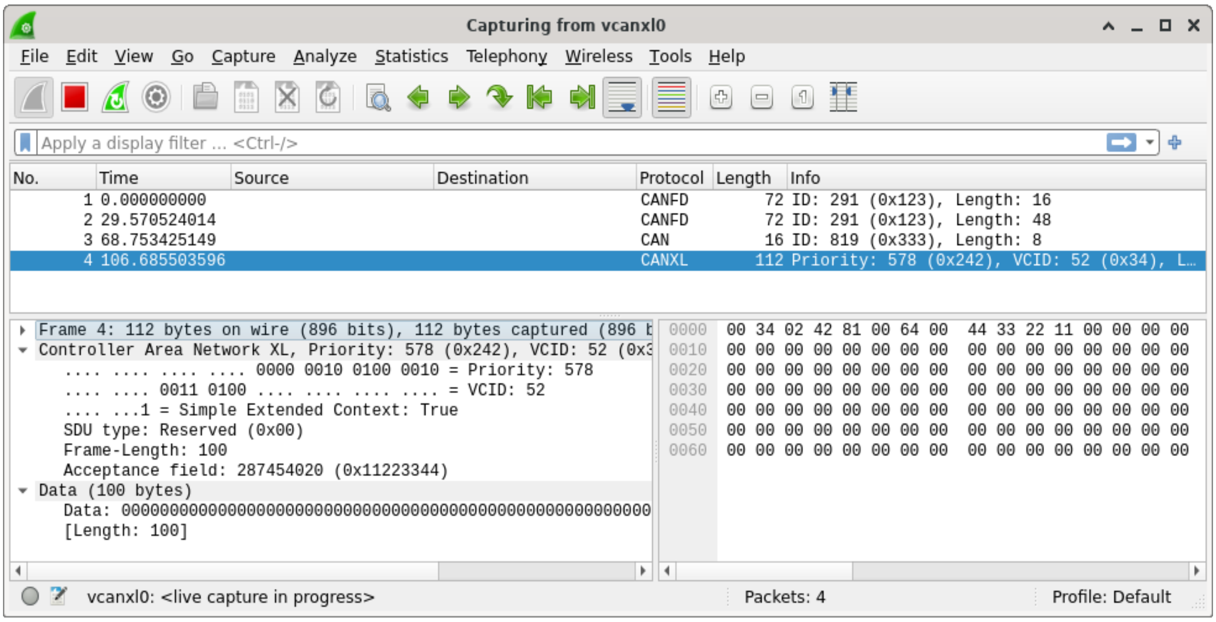
The current setup to develop and test CAN XL hardware drivers for Linux is a Terasic DE1-SoC FPGA with Ubuntu 20.04 equipped with a 3-channel CAN transceiver board (XL/XL/FD) from NXP and a 3-channel XCANB CAN XL controller FPGA IP core from BOSCH.



Based on the XCANB API code example from BOSCH three CAN XL network devices have been implemented in a recent Linux kernel with CAN XL support (v6.1+). The setup is to be used at the CiA CAN XL plugfest, to validate CiA CAN XL segmentation and encapsulation protocols. Based on this hardware setup the CAN driver infrastructure will be extended to support the new CAN XL bitrates and other CAN XL specific settings.

Linux CAN XL applications

At time of writing the CAN XL VCID support was integrated into the Linux kernel. Due to this extension the official can-utils package with candump and canplayer still do not support CAN XL. Some tools to generate and display CAN XL frames on the command line have been implemented by the author to validate and test the CiA CAN XL protocol PoC implementations (see below). During the discussion about the VCID integration the CAN XL support has been integrated into the Wireshark project, that can now correctly display CAN CC, CAN FD and CAN XL content since Wireshark v4.2.3:



Various CAN CiA protocols have been implemented to verify the ongoing standardization efforts. The PoCs make use of virtual CAN XL interfaces and need a Linux kernel v6.1+ with CAN XL support. To set up and execute the PoC implementations follow the README file.

CAN CiA 611-1 (Service Data Unit Types)

- SDTs
  - 0x03 : CAN CC / CAN FD
  - 0x06 : CAN CC
  - 0x07 : CAN FD
- URL: https://github.com/hartkopp/can-cia-611-1-poc

CAN CiA 611-2 (Multi-PDU)

- SDTs
  - 0x08 : CiA 611-2 (Multi-PDU)
- URL: https://github.com/hartkopp/can-cia-611-2-poc

CAN CiA 613-3 (Fragmentation)

- URL: https://github.com/hartkopp/can-cia-613-3-poc

Dr. Oliver Hartkopp
Volkswagen
PO Box 1777
DE-38436 Wolfsburg
www.volkswagen.de