Linux and ISO 15765-2 with CAN FD

Dr. Oliver Hartkopp, Volkswagen AG

Only two weeks after disclosure of the CAN FD main features at 13th iCC [1] the Linux CAN community started to discuss about a seamless integration of CAN FD into the CAN subsystem of the Linux operating system. This paper gives a comprehensive survey about the integration, configuration and usability of CAN FD in the Linux operating system as well as an introduction into the new ISO15765-2:2015 with CAN FD support.

With the integration of the socket-based CAN support in Linux 2.6.25 [2] in April 2008 the data structures and programming interfaces were defined and therefore fixed in an application binary interface (ABI). This fixed ABI implies that Linux CAN (aka SocketCAN) applications that were compiled and linked statically for Linux in 2008 are able to run on the latest Linux system with a recent Linux 4.x kernel.
Analogue to this guaranteed binary backward compatibility for applications the introduction of the common CAN driver interface in Linux 2.6.31 in September 2009 fixed the way how CAN network interfaces are configured in terms of bitrate and other CAN controller specific settings.

From the perspective of CAN application programmers the formerly settled properties of up to eight bytes of payload and a single bitrate to be set into the CAN controller became uncertain. With CAN FD the known CAN bitrate configuration is doubled and the data structures to hold CAN frame contents are increased in size which can lead to buffer overflows when the former CAN frame data structure is accidently used. Preserving the simple and established SocketCAN programming interface under the new conditions with CAN FD is an ambitious task which has been accomplished by the Linux CAN community instantaneously after 13th iCC.

CAN FD data structures

As the associated CAN interface and the timestamp of the CAN frame are provided by existing Linux programming interfaces the data structure which holds the CAN frame content is the elementary data definition for SocketCAN. The original classic CAN frame data structure is defined as:

```
struct can_frame {
        canid_t can_id;
        __u8    can_dlc;
        __u8    __pad;
        __u8    __res0;
        __u8    __res1;
        __u8    data[8]; /*aligned*/
};
```

The can_id contains the CAN Identifier with additional bit values e.g. to point out a 29 bit identifier or RTR frames. The can_dlc contains the number of used bytes in the data[ ] byte array. Remark: The padding and reserved bytes have been added recently to be in line with the CAN FD definitions. These extensions do not have an impact on the application binary interface as the data[ ] was always 64 bit aligned (see linux/can.h [3] for details).

For the CAN FD frame a separate data structure has been defined:

```
struct canfd_frame {
        canid_t can_id;
        __u8    len;
        __u8    flags;
        __u8    __res0;
        __u8    __res1;
        __u8    data[64]; /*aligned*/
};
```

The major differences are the introduced flags element which holds CAN FD frame specific flags like CANFD_BRS and CANFD_ESI and the len element. The len element shares the position with the can_dlc element of the classic CAN frame and (still) contains the number of used bytes in the data[ ] byte array. In classic CAN applications the can_dlc value was usually used as plain numeric length information as there was a 1:1 mapping from the 'data length code' and the data length. Using CAN FD frames the data length code mapping is performed on the CAN driver level which makes the software adaption for CAN FD pretty easy.

Processing length information to print CAN payload data (before CAN FD support):

```
struct can_frame cframe;

for (i=0; i < cframe.can_dlc; i++)
    printf(„%02X „, cframe.data[i]);
```

Processing length information to print CAN payload data (with CAN FD support):

```
struct canfd_frame cframe;

for (i=0; i < cframe.len; i++)
    printf(„%02X „, cframe.data[i]);
```

This example points out the main change for application programmers when moving their code to (additionally) support CAN FD. Several code references how to move from classic CAN to CAN FD can be retrieved from the code changes [6] in the Linux can-utils package which has been adapted when Linux 3.6 was released in 2012. The can-utils user space tools to send, receive, store and replay CAN traffic can be found as source code on GitHub [4] and as pre-compiled package 'can-utils' in your preferred Ubuntu/Debian based Linux distribution [5].

CAN FD network infrastructure

Both the classic CAN frames and the CAN FD frames are processed inside the Linux network infrastructure in so called socket buffers. With the introduction of CAN FD a second type of CAN related socket buffers was created to hold the canfd_frame data structures.

As legacy CAN applications only can cope with classic CAN frames a new socket option CAN_RAW_FD_FRAMES is defined for CAN_RAW sockets to enable the reception and transmission of CAN FD frames.

When CAN FD is enabled for the socket e.g. the read() system call can return with two different length information:

- 16 bytes for classic CAN frames
- 72 bytes for CAN FD frames

Therefore the buffer which is assigned to be utilized by the read() system call has to be a of the size of a struct canfd_frame when CAN FD is enabled. As the CAN FD controller still might receive classic CAN frames in this FD enabled mode the struct canfd_frame might be filled with the shorter struct can_frame content. Due to the identical layout - e.g. with the can_dlc and len element – a classic CAN frame can be stored inside the CAN FD frame structure. To distinguish the frame type only the length information has to be evaluated with is returned by the read() system call:

- 16 bytes → classic CAN frame
- 72 bytes → CAN FD frame

For convenience reasons these values are defined as the 'maximum transfer unit' (MTU) in the linux/can.h [3] include file as CAN(FD)_MTU values:

```
#define CAN_MTU   (sizeof(struct can_frame))
#define CANFD_MTU (sizeof(struct canfd_frame))
```

CAN FD driver infrastructure

With Linux 3.6 the CAN data structures and the network infrastructure have been extended to support CAN FD. Along with these changes the virtual CAN driver (vcan) has been updated in a way that it could be switched to be a classic CAN or CAN FD interface. By setting the vcan's MTU value to CANFD_MTU (72) with the existing ip tool from the iproute2 package the virtual CAN interface presents itself as a CAN FD interface.

While this virtual CAN driver did a good job when testing and enhancing the new CAN FD infrastructure and the user tools it should last more than a year until the first CAN FD hardware became available. The first CAN FD driver that emerged in the Linux mainline kernel was for the Bosch M_CAN

IP core version 3.0.1 (non-ISO). The driver was included in Linux 3.18 in December 2014 and tagged as a 'fixed non-ISO' CAN FD controller later. In April 2015 the PEAK System PCAN USB (pro) FD driver was released with Linux 4.0. These USB adapters can be switched to be ISO/non-ISO at controller configuration time.

With classic CAN the configuration was done with the ip tool from the iproute2 [7] package [8] in order to specify the bitrate and additional controller specific settings like the sampling-point, synchronization jump width, listen-only mode, triple sampling, one-shot mode, etc.

The bitrate can be specified with either the time quanta (tq), propagation segment (prop_seg) and phase buffer segments (phase_seg1 phase_seg2) or by providing a numeric bitrate value which is then processed by the bitrate calculation algorithm inside the Linux kernel. The latter needs a set of controller specific bit timing constants that define e.g. the allowed minimum and maximum values for the time segments, bitrate prescaler, etc.

For CAN FD these bitrate specific settings have to be doubled to specify a second bitrate: The data bitrate when BRS is set.

This summarizes to these extensions:

- Second bitrate infrastructure
- Enable/Disable CAN FD mode
- Configure ISO/non-ISO mode

When the CAN FD mode is to be enabled the data bitrate has to be specified and it has to be greater or equal to the arbitration bitrate which is placed in the first bitrate infrastructure known from classic CAN. The CAN FD mode setting changes the CAN interface MTU to CAN_MTU or CANFD_MTU accordingly.

Depending on the CAN FD controller capabilities the ISO/non-ISO mode can be specified by the ip tool or it is fixed with the controller. E.g. the M_CAN IP version 3.0.1 is fixed to non-ISO, which cannot be changed at configuration time. On the other hand the PEAK USB FD adapters can switch between ISO and non-ISO at configuration time. The attempt to modify a fixed ISO/non-ISO flag leads to an invalid operation return code.

Finally the configuration of CAN FD controllers became very similar to the classic CAN controllers by just adding a second bitrate set for the data bitrate and two CAN FD specific configuration flags. The ip tool from the iproute2 package was updated for the release of Linux 3.15 to support the second bitrate and the CAN FD mode switching. The ISO/non-ISO configuration was integrated in Linux 3.19 but backported to Linux 3.18 to be able to tag the existing M_CAN driver properly.

ISO 15765-2:2015 with CAN FD

The ISO 15765-2 CAN transport protocol (TP) usually creates a point-to-point data connection using two defined CAN identifiers – one for each communication endpoint (e.g. diagnosis equipment and engine control unit). To be able to send data PDUs that do not fit into a single CAN frame the ISO PDUs are segmented using a bi-directional segmentation protocol. This protocol is implemented using (at least) the first byte of the CAN frame payload – the so called 'protocol control identifier' PCI.

The PCI byte is defined as:

Table 1: ISO 15675-2 PCI

PCI	function	nibble	bit value
SF	Single Frame	0	0000xxxx
FF	First Frame	1	0001xxxx
CF	Consecutive Frame	2	0010xxxx
FC	Flow Control	3	0011xxxx

While SF, FF and CF are sending PDU data from node A to node B the FC is a communication entity that is sent from node B to node A in order to throttle the communication flow according to the recipients (node B) needs.

When the content of the PDU fits into a single frame the SF frame is generated. Simplified the PDU content has to be 7 or less bytes on classic CAN as one byte is always consumed by the PCI byte.

When the content of the PDU does not fit into a single CAN frame a FF frame is generated which contains the PCI byte, length information and some first data bytes of the PDU. When node B is able to receive the advertised number of bytes it answers with a FC frame to get more segmented data in the form of CF frames.
Due to the mandatory PCI byte which consumes at least one byte from each CAN frame payload the protocol overhead is equal or greater than 12.5% in classic CAN setups with 8 bytes per frame.

With CAN FD up to 64 bytes of payload can be transmitted inside a CAN frame. This moves the lower limit of overhead for ISO TP to 1/64 = 0,015625 ~ 1.6%. Even if we always need to add the standard overhead of the CAN Identifier, control fields and CRC in both cases this is a huge improvement which can be even extended when using a higher bitrate in the data section (BRS enabled).

With the knowledge from his own ISO15765-2 [10] implementation for Linux and the CAN FD changes in Linux the author initiated the adaption of ISO TP for CAN FD at DIN/ISO committee in early 2013. As the Linux kernel was already supporting CAN FD at that time the changes of the existing classic CAN implementation assisted the conceptual work. Whenever a concept was discussed the public available implementation [9] gave an indication of the expectable complexity of that approach.

Table 2: ISO 15675-2 PCI for classic CAN

PCI B[0]	B[1]	B[2]	B[3]	B[4]
SF 0000 LLLL	data	data	data	data
FF 0001 LLLL	LLLLLLLL	data	data	data
CF 0010 NNNN	data	data	data	data
FC 0011 FFFF	Blocksize	STm	n.a.	n.a.

(Formatting: All tables are cut after byte 4)

- LLLL : PDU length information
- NNNN : sequence number
- FFFF : flow status information
- Blocksize : 0 .. 15 (0 = disabled)
- STm : Separation Time minimum
- data : PDU payload data
- n.a. : not assigned
- B[x] : byte x in CAN frame payload

While CF and FC frames are not really affected by the increased CAN frame length, the possible PDU length of up to 63 bytes cannot be described in the four length bits available in the SF PCI byte.

To be able to discuss different CAN frame payload sizes the 'link layer data length' (LL_DL) has been introduced into the ISO document. As long as the LL_DL is 8 bytes – as known from classic CAN – the new ISO TP PDU segmentation concept behaves exactly like the former specification of ISO TP.

When the LL_DL is defined to be greater than 8 bytes (12, 16, 20, .., 64) the length information in the SF frame PCI is set to zero and the length information is stored in the following byte (Byte 1). Setting the length information in the SF PCI byte to zero is a protocol violation in the former ISO 15765-2 specification which makes older implementations ignoring these SF frames. On the other side this concept reduces the maximum possible SF PDU size to LL_DL – 2 bytes (e.g. 62 bytes for CAN FD frames with 64 bytes).

As the configured LL_DL value is unknown on the receiver side, the receiver automatically adapts to the sender LL_DL depending on the frame length of the FF frame when starting a segmented communication.

Another enhancement of the FF definition is basically not CAN FD dependent. The 12 bits for the FF length information allows PDU sizes of up to 4095 bytes. To be able to transfer larger PDUs (e.g. for measurement data, configuration data, bootloader update, etc.) a similar concept as known from the SF length was developed: By setting the former FF length information to zero, the sender indicates that the length information is available in the following 4 bytes. This allows PDU sizes up to 2^{32}-1 bytes (~4GB). When the new receiving implementation detects this former protocol violation it takes the next four bytes and can receive the 'jumbo' PDU with more than 4095 bytes – even in classic CAN setups.

Table 3 and Table 4 depict the PCI changes for the extended length information in SF and FF frames.

Table 3: SF PCI for LL_DL > 8

PCI B[0]	B[1]	B[2]	B[3]	B[4]
SF 0000 0000	LLLLLLLL	data	data	data

Table 4: FF PCI for PDU length > 4095

PCI B[0]	B[1]	B[2]	B[3]	B[4]	B[5]
FF 0001 0000	0	Len	Len	Len	Len

The length information is presented in high-byte first order as known from the former FF length information.

A useful aspect of CAN FD enabled ISO TP communication is the fact that classic CAN frames and CAN FD do not interfere in a CAN FD enabled setup. This means that the CAN architect may assign two CAN identifiers for the communication with classic CAN frames – and he may assign the identical(!) CAN identifiers for a CAN FD enabled communication. As classic CAN and CAN FD frames distinguish on the wire two independent ISO TP communications can be performed on the CAN bus in this way.

Finally the introduction of CAN FD frames in ISO 15765-2 leads to a mandatory padding in the case that the PDU payload doesn't fit exactly into the CAN FD frame payload. In such cases the rest of the CAN FD frame shall be filled with 0xCC byte values as recommended by Bosch. The 0xCC data content allows the minimum of alternating bus level changes (EMI friendly) without the need to insert stuff bits.

ISO 15765-2:2015 with Linux

While sending ISO TP PDUs in Linux is just about opening a socket and read/write PDU data to the given file handle the configuration of ISO TP communication is done by so called socket options.
These socket options are passed to the socket at creation time to specify values like block size (BS), STmin, extended addressing parameters or padding configurations. To be able to take advantage of the CAN FD implementation a single new socket option CAN_ISOTP_LL_OPTS has been introduced to configure the link layer. The data structure to configure the link layer options is defined in [11] as

```
struct can_isotp_ll_options {
        __u8 mtu;
        __u8 tx_dl;
        __u8 tx_flags;
};
```

The element mtu specifies the generated and accepted CAN frame type. As described above the mtu can take values of either CAN_MTU (16) to handle classic CAN frames or CANFD_MTU (72) to work with CAN FD frames only.
The tx_dl element specifies the LL_DL value for generated CAN (FD) frames as the protocol stack adapts to in incoming LL_DL (rx_dl) automatically. The valid values for tx_dl are specified by valid CAN FD data lengths beginning with eight:
8, 12, 16, 20, 24, 32, 48, 64
N.B. when the mtu is set to CAN_MTU only a tx_dl value of eight is allowed.
Finally the tx_flags element content is set into the flags element of the canfd_frame structure at frame creation time to configure the CANFD_BRS setting for this socket.

ISO 15765-2:2015 CAN FD performance

While the ISO TP implementation for CAN FD hypothesized an increased performance in calculations and on the virtual CAN interfaces the tests on real CAN FD hardware were awaited eagerly. With Linux 4.0 the driver for the PEAK USB FD was available in a stable operating system environment where it made sense to take measurements with the latest ISO 15765-2:2015 implementation.

With a set of shell scripts the existing ISO TP command line tools have been arranged in a way that classic CAN and different CAN FD based communication setups can be brought into meaningful relation.

The setup consists of two Linux PCs each with an USB FD adapter connected to each other with a terminated twisted pair CAN line.

The timestamps are taken from the receiving node to make sure the entire PDU hit the CAN bus. The test applications transferred and received a PDU of 30.000 bytes with an arbitration bitrate of 500 kbit/s and different values for data bitrates (2/4/8 Mbit/s). The separation time minimum (STmin) was set to either 500μs or 100μs. As the test values for 500μs did not differ substantially for different data bitrates only a single table for the 500μs measurement is depicted below. To have a realistic and safe transport the block size was set to its maximum of 15.

Table 5: Test 500μs STmin 0.5/2 Mbit/s

CAN	LL_DL	BRS	secs	Bytes/s
classic	8	-	2,926	10.256
FD	8	no	2,933	10.228
FD	8	yes	2,915	10.295
FD	16	no	1,380	21.754
FD	16	yes	1,351	22.205
FD	32	no	0,791	37.926
FD	32	yes	0,662	45.385
FD	64	no	0,625	48.000
FD	64	yes	0,329	91.463

As the separation time was 500μs there could be seen no effect when increasing the data bitrate. At higher data bitrates the CAN bus had to handle fewer loads but bus load was not the value we wanted to pay attention at in this setup.

Table 6: Test 100μs STmin 0.5/2 Mbit/s

CAN	LL_DL	BRS	secs	Bytes/s
classic	8	-	1,460	20.562
FD	8	no	1,750	17.152
FD	8	yes	1,172	25.597
FD	16	no	1,085	27.649
FD	16	yes	0,548	54.744
FD	32	no	0,793	37.878
FD	32	yes	0,330	91.185
FD	64	no	0,614	48.859
FD	64	yes	0,225	133.333

Table 7: Test 100μs STmin 0.5/4 Mbit/s

CAN	LL_DL	BRS	secs	Bytes/s
classic	8	-	1,460	20.547
FD	8	no	1,749	17.162
FD	8	yes	1,166	25.751
FD	16	no	1,086	27.649
FD	16	yes	0,545	55.045
FD	32	no	0,792	37.878
FD	32	yes	0,265	113.207
FD	64	no	0,614	48.859
FD	64	yes	0,163	185.185

Table 8: Test 100μs STmin 0.5/8 Mbit/s

CAN	LL_DL	BRS	secs	Bytes/s
classic	8	-	1,462	20.533
FD	8	no	1,752	17.133
FD	8	yes	1,150	26.109
FD	16	no	1,085	27.649
FD	16	yes	0,545	55.147
FD	32	no	0,792	37.878
FD	32	yes	0,266	113.207
FD	64	no	0,614	48.939
FD	64	yes	0,131	230.769

With the relatively short STmin of 100μs the PDU data throughput can be increased by factor 11 (230.769 / 20.533) – even with a configured block size of 15 which requires the receiving node to acknowledge every 15th CF frame. Without bitrate setting (BRS) the benefit of 64 byte CAN frames reduces to factor 2.5 due to the better overhead ratio. Finally the measurements points out that using CAN FD without BRS and with LL_DL of 8 preforms worse than classic CAN. As CAN FD introduces additional control bits, an increased CRC field size and a stuff bit counter in the latest ISO implementation this performance reduction was expected.

Summary

The new CAN FD protocol doesn't only break the compatibility to classic CAN on the wire – it also breaks programming interfaces and extends configuration options by introducing new bitrates and payload lengths. This paper gives an insight how programming interfaces have been altered in Linux in an evolutionary way without putting the existing application programming concept into question. Some of the presented ideas

may be reused in other embedded setups – some may be too Linux specific to do so.
By today CAN FD is fully supported by Linux and by the provided tools to handle and configure CAN FD specific content and functionalities. Together with the free ISO15765-2:2015 implementation Linux is recommended as a stable and sustainable testing and product platform for future CAN FD applications.

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

References

[1] 13th iCC 2012 – Paper Hartwich (Bosch) http://www.can-cia.org/fileadmin/resources/documents/proceedings/2012_hartwich.pdf
[2] http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git
[3] http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/include/uapi/linux/can.h
[4] https://github.com/linux-can/can-utils
[5] https://packages.debian.org/stable/can-utils
[6] https://github.com/linux-can/can-utils/commit/e7631bd7f94804962e48cde2e7de37370c31a8b8
[7] http://git.kernel.org/cgit/linux/kernel/git/shemminger/iproute2.git/
[8] https://packages.debian.org/stable/iproute2
[9] https://github.com/hartkopp/can-isotp-modules
[10] ISO 15765-2:2011 document http://www.iso.org/iso/catalogue_detail.htm?csnumber=54499
[11] https://github.com/hartkopp/can-isotp-modules/blob/master/include/socketcan/can/isotp.h