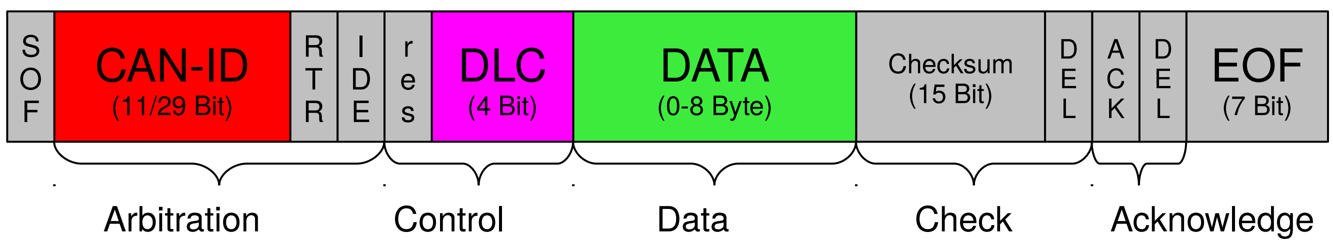
# The CAN Subsystem of the Linux Kernel

A ~~Linux CAN driver~~ swiss army knife for automotive use-cases

Presentation for Automotive Grade Linux F2F, 2017-04-04, Microchip (Karlsruhe)

# Controller Area Network simplified for nerds

- Media access by CSMA/CR
- Structure of a CAN frame:




- Simplified: [CAN Identifier] [Data length] [Data 0..8]
- Content addressing (by CAN Identifier & CAN Bus)
- No MAC / Node addresses / ARP / Routing – just plain OSI Layer 2
- Incompatible Upgrade CAN FD (ISO 11898-1:2015), explained later

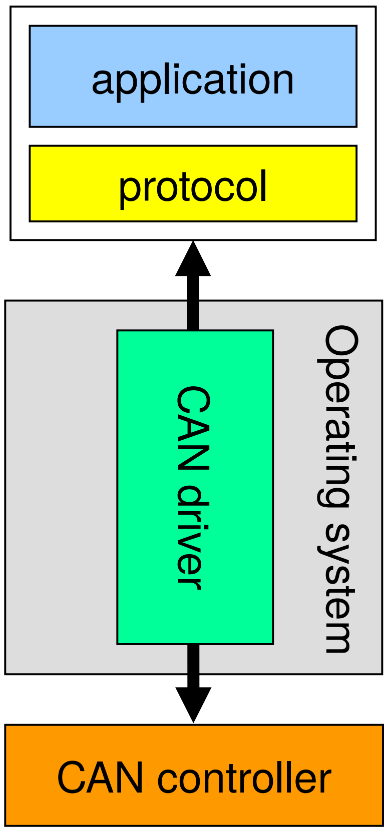
## Application areas for the Controller Area Network

- Industrial control applications (e.g. using the CANopen protocol)
- Food processing (e.g. on fish trawlers)
- Vehicles (Passenger Cars, Trucks, Fork lifters, etc.)
- Research (e.g. Nuclear physics)
- Spacecrafts, Marina
- Oil platforms
- Wind energy plants
- Measurement / Sensors
- Special Effects

## Usage of the CAN bus in a vehicle

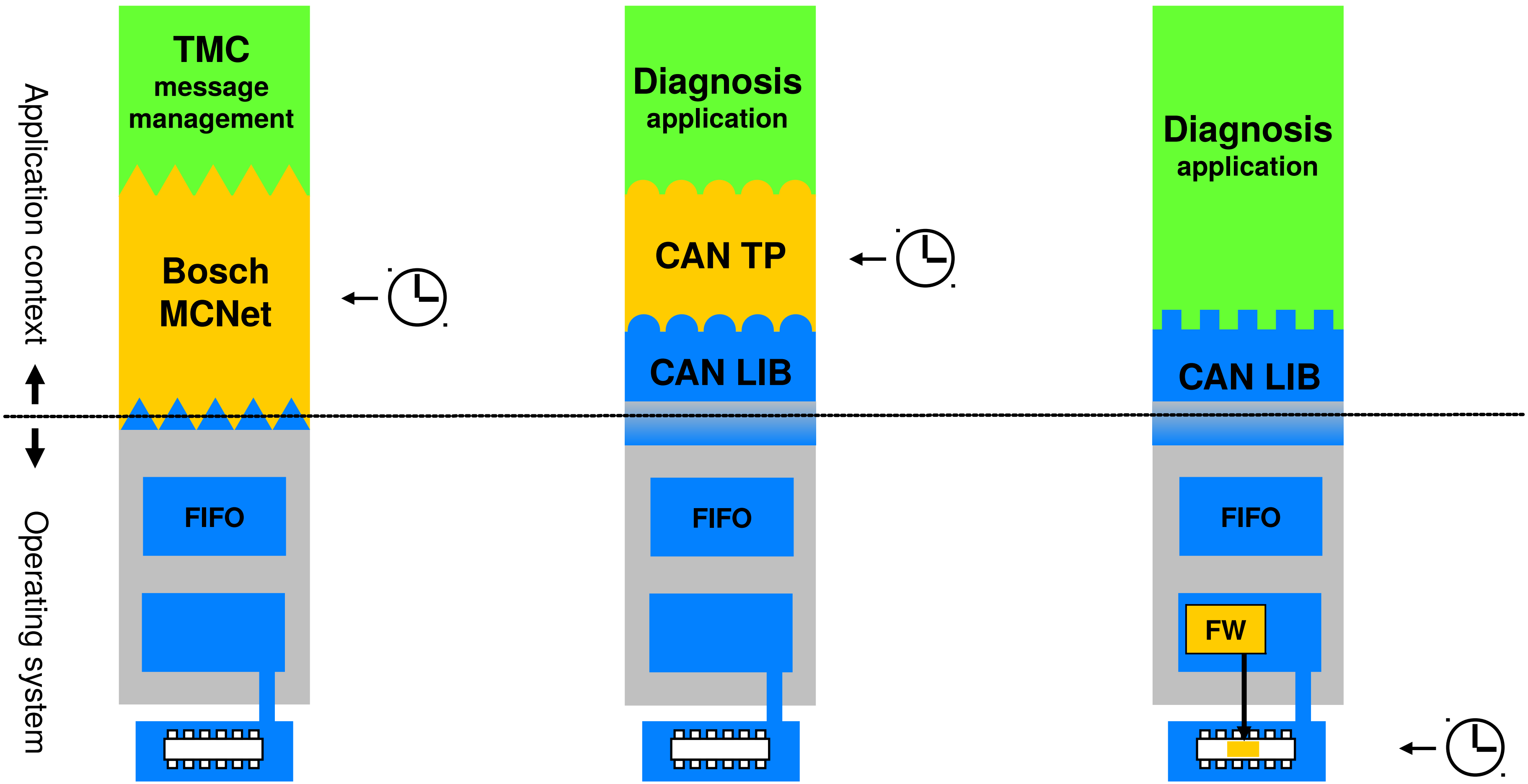
- Simple CAN broadcast messages
- Cyclic sent CAN messages
  (for failure detection)
- Multiplex CAN messages
  (containing an index for different data payload)
- Transport protocols
  (virtual point-to-point connection via CAN, e.g. ISO-TP: ISO 15765-2)

# The former concepts for CAN access

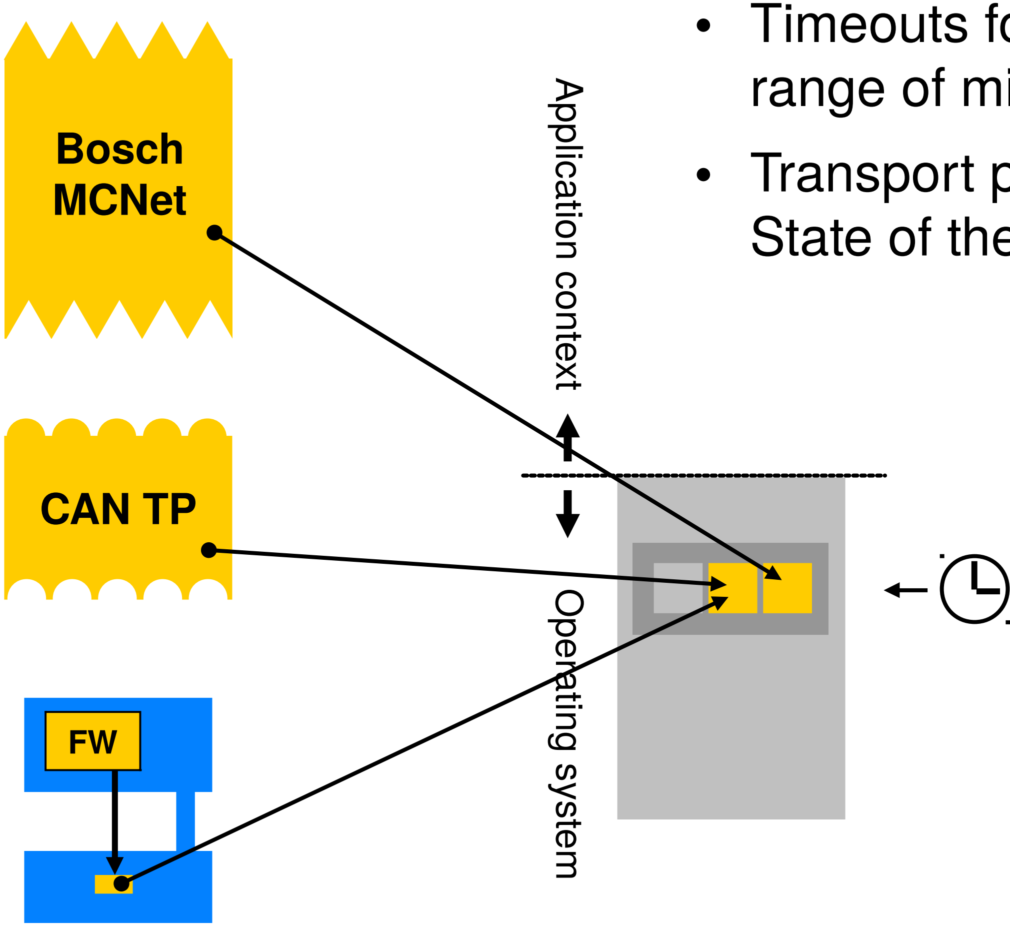



- Only one application can use the CAN bus at a time
- There was no standard Linux CAN driver model
  - Every CAN hardware vendor sells his own driver bundled to his CAN hardware
  - The change to a different CAN hardware vendor urges the adaptation of the CAN application(!) => Vendor Lock-In
- CAN application protocols and intelligent content filters need to be implemented in userspace

# Former automotive CAN transport protocol implementations



# Idea: Meet timing restictions in the operating system context

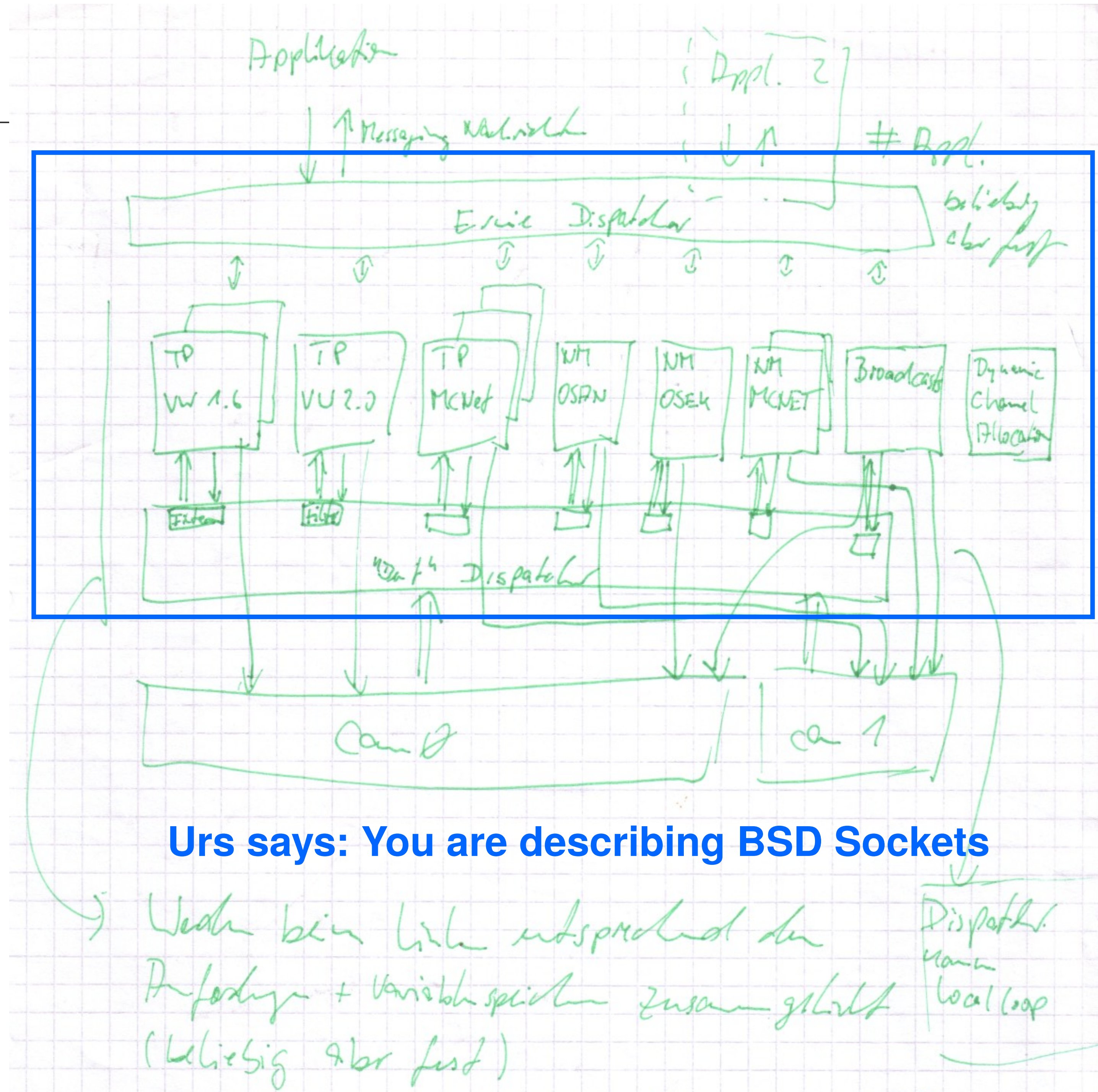



- Timeouts for CAN transport protocols in the range of milliseconds can be realized
- Transport protocols in operation systems are State of the art (eg. TCP known from TCP/IP)

**Idea:**

Implement CAN transport protocols inside the operating system context

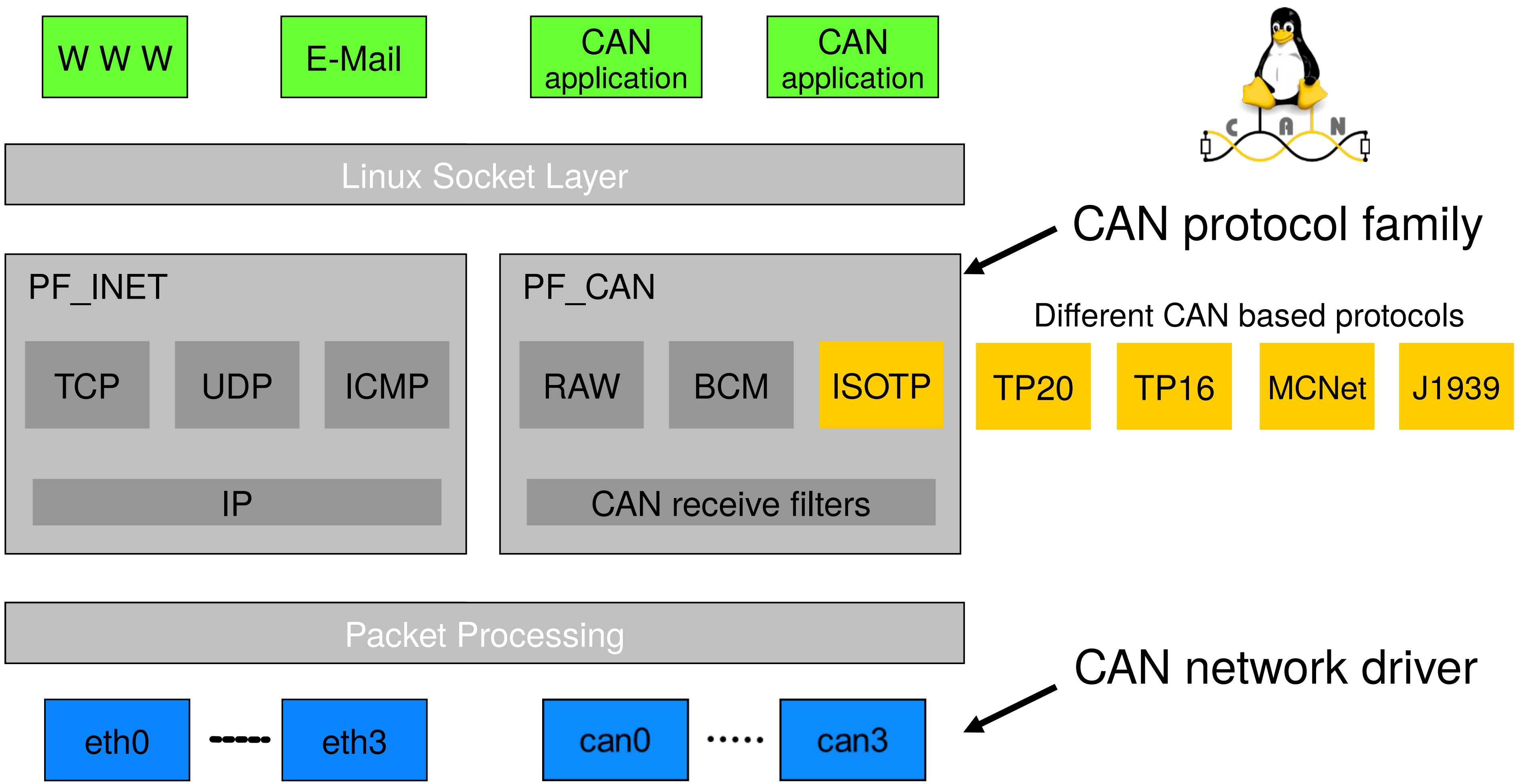
# Concept idea from 2001




**Urs says: You are describing BSD Sockets**

Oliver Hartkopp

# New protocol family for the Controller Area Network (PF_CAN)



## Implications of using network sockets

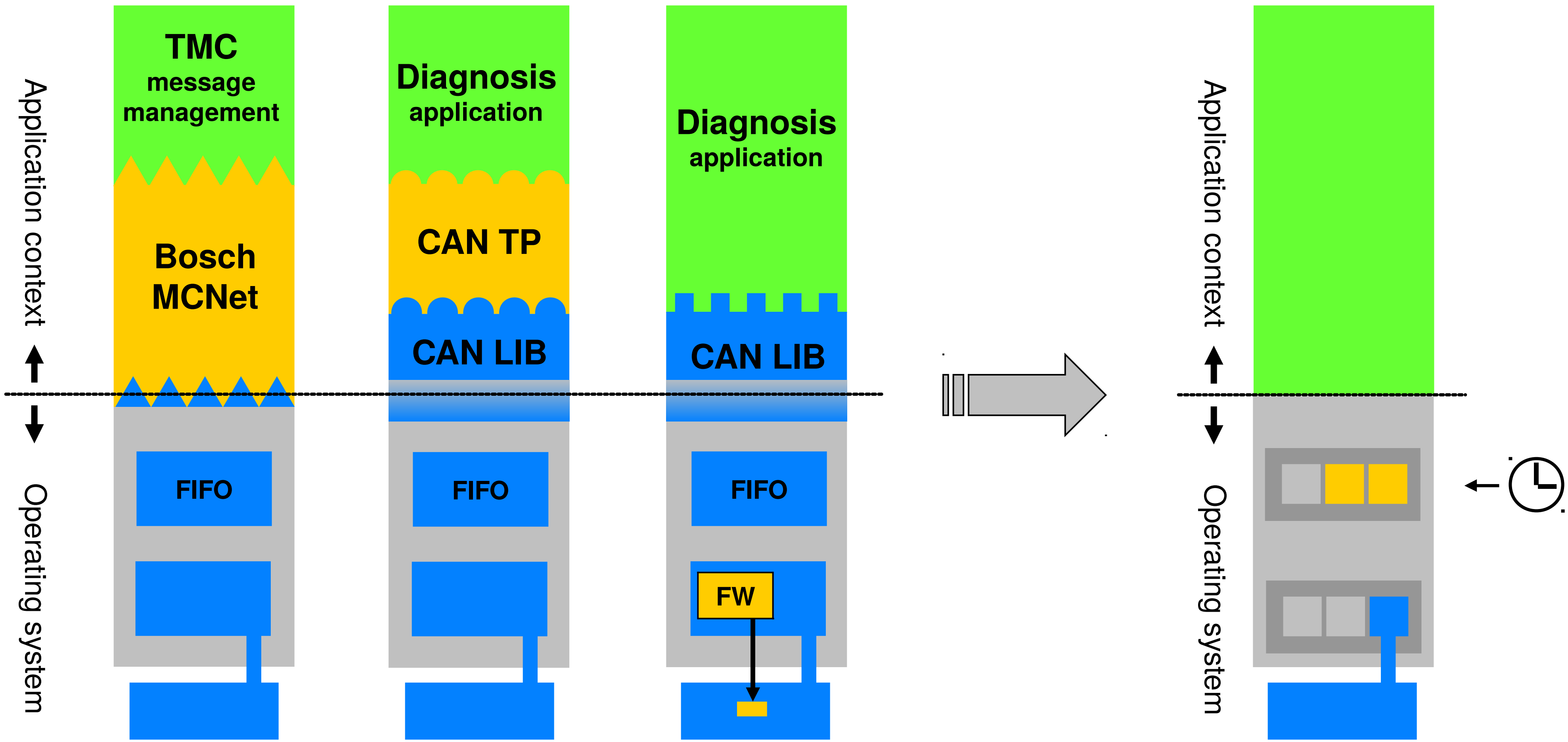
- Established socket programming interface to the operating system
- Network driver model for networking hardware (e.g. Ethernet cards)
- Protocols and routing inside the operating system (e.g. for TCP/IP)
- Random number of instances of network protocols
- Existing infrastructure for example for
    - efficient message queues
    - the integration of network hardware drivers

# The socket programming interface

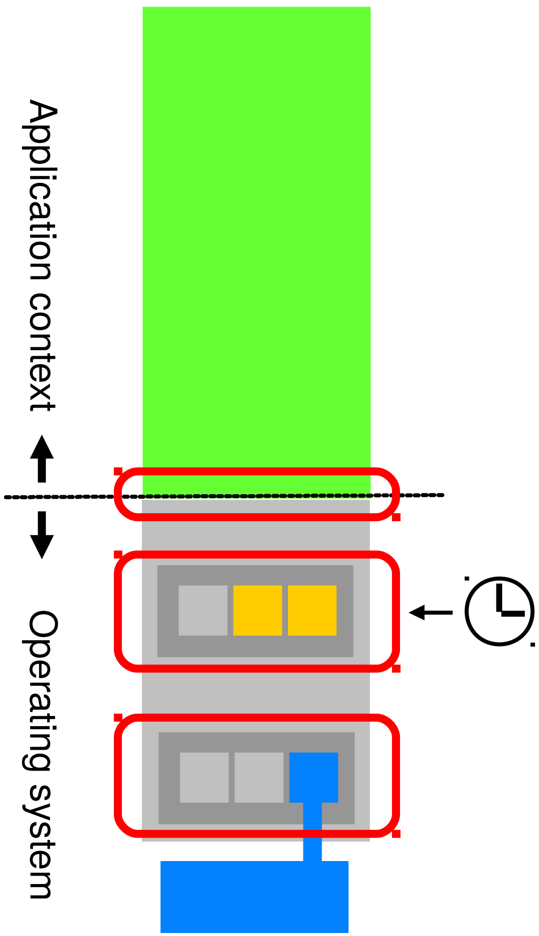
## example: CAN-over-WLAN Bridge

```
(..) /* some source code - don't worry */

int can;                  /* socket handle */
int wlan;
struct can_frame mymsg;   /* data structure for CAN frames */


can = socket(PF_CAN, SOCK_DGRAM, CAN_RAW);   /* CAN RAW Socket */
wlan = socket(PF_INET, SOCK_DGRAM, 0);       /* UDP/IP Socket  */

(..) /* set addresses and CAN filters */

bind(can, (struct sockaddr *)&can_addr, sizeof(can_addr));
connect(wlan, (struct sockaddr *)&in_addr, sizeof(in_addr));

while (1) {
           read(can,  &mymsg, sizeof(struct can_frame));
           write(wlan, &mymsg, sizeof(struct can_frame));
}
```

# Technical improvement with SocketCAN



# The standardized CAN programming interface

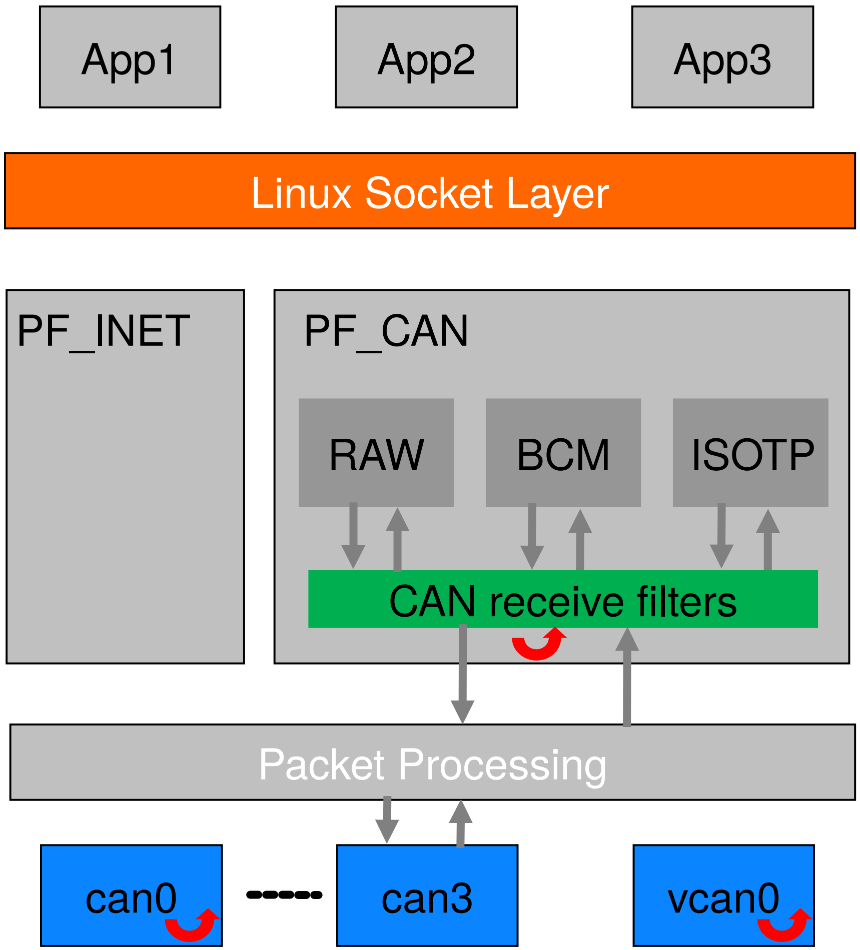



- Definition of CAN specific
  - data structures (eg. struct can_frame)
  - protocols incorporated in the protocol family PF_CAN
  - characteristics of CAN network devices (e.g. bitrates)

- Realizing the requirements from CAN users
  - CAN access without transport protocols ('raw')
  - Filtering of CAN messages
  - Performance
  - Transparency and multi-user capabilities

- Generic interface definition for the use in other operating systems (like QNX, BSD Unix, Windows)

# Highlights of the protocol family PF_CAN




Standard BSD network socket programming interface

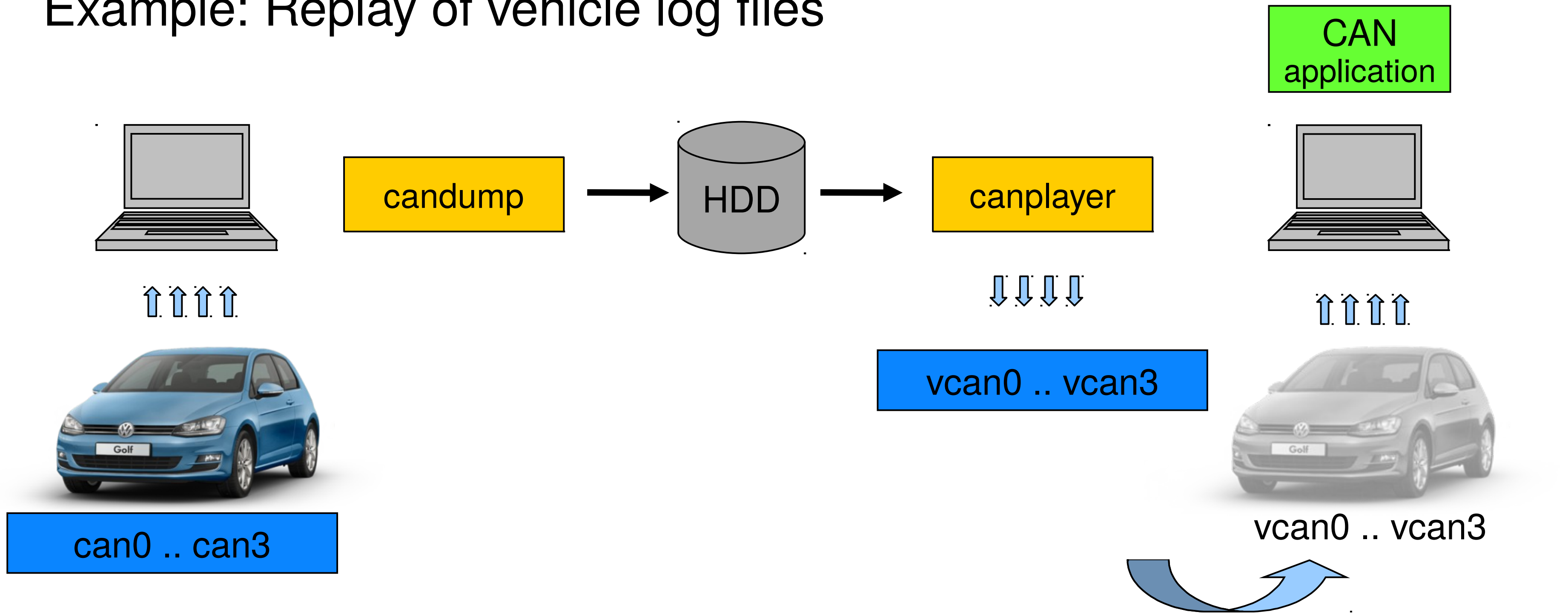
Receive filter lists handled inside a software interrupt (Linux NET_RX softirq)

network device driver model

Network transparency:
local echo of **sent** CAN frames on successful transmission

# Virtual CAN network device driver (vcan)

- No need for real CAN hardware (available since Linux 2.6.25)
- Local echo of sent CAN frames 'loopback device'
- vcan instances can be created at run-time
- Example: Replay of vehicle log files



## How to create a virtual CAN network device

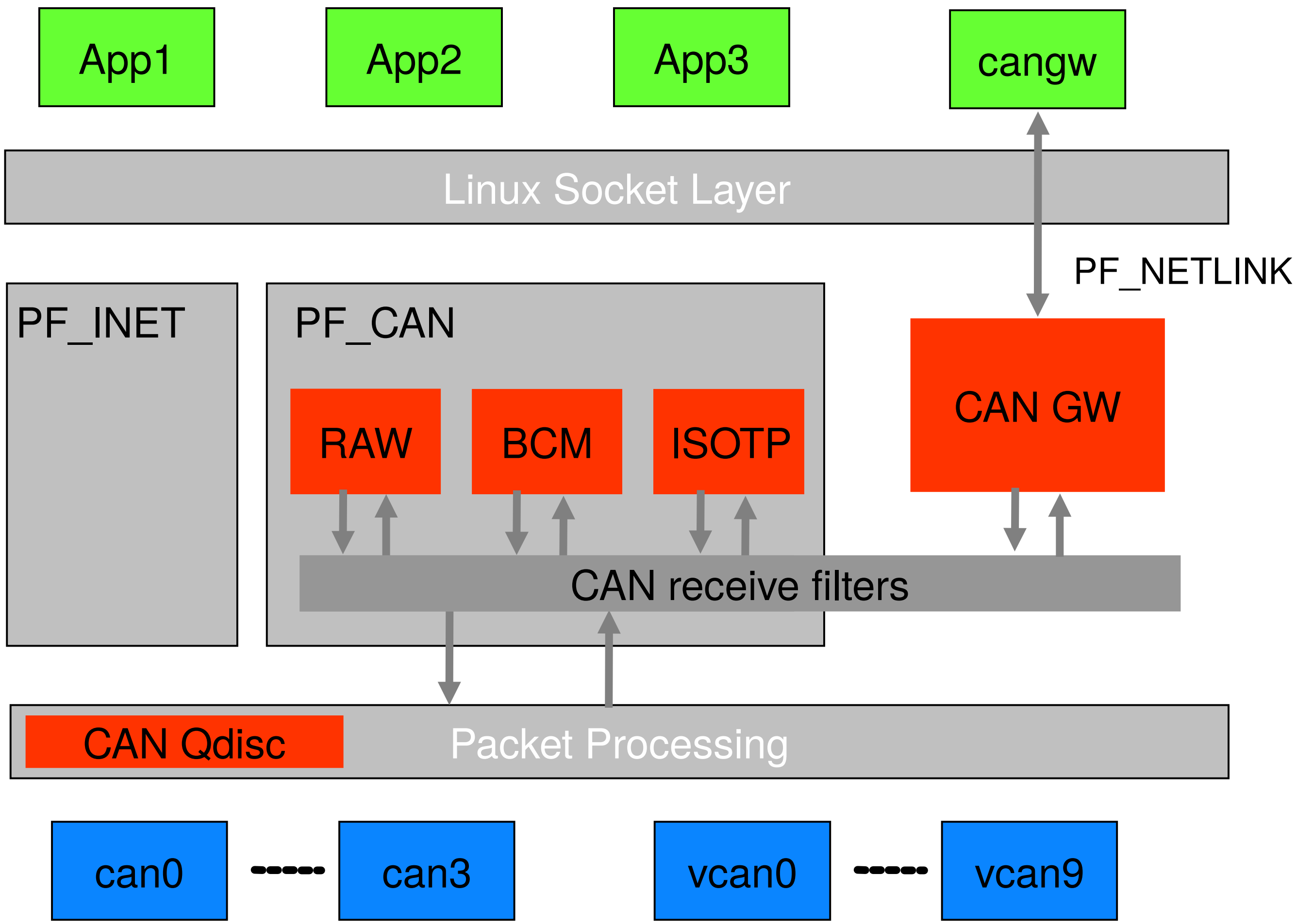
- Loading the virtual CAN driver into the Linux kernel

```
sudo modprobe vcan
```

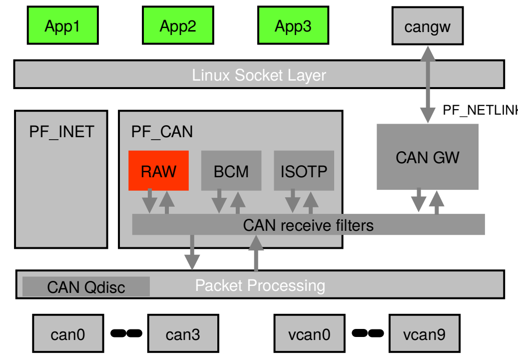
- Create a virtual CAN interface

```
sudo ip link add type vcan
sudo ip link add dev helga type vcan
sudo ip link set vcan0 up
sudo ip link set helga up
```

# CAN network layer protocols and CAN frame processing



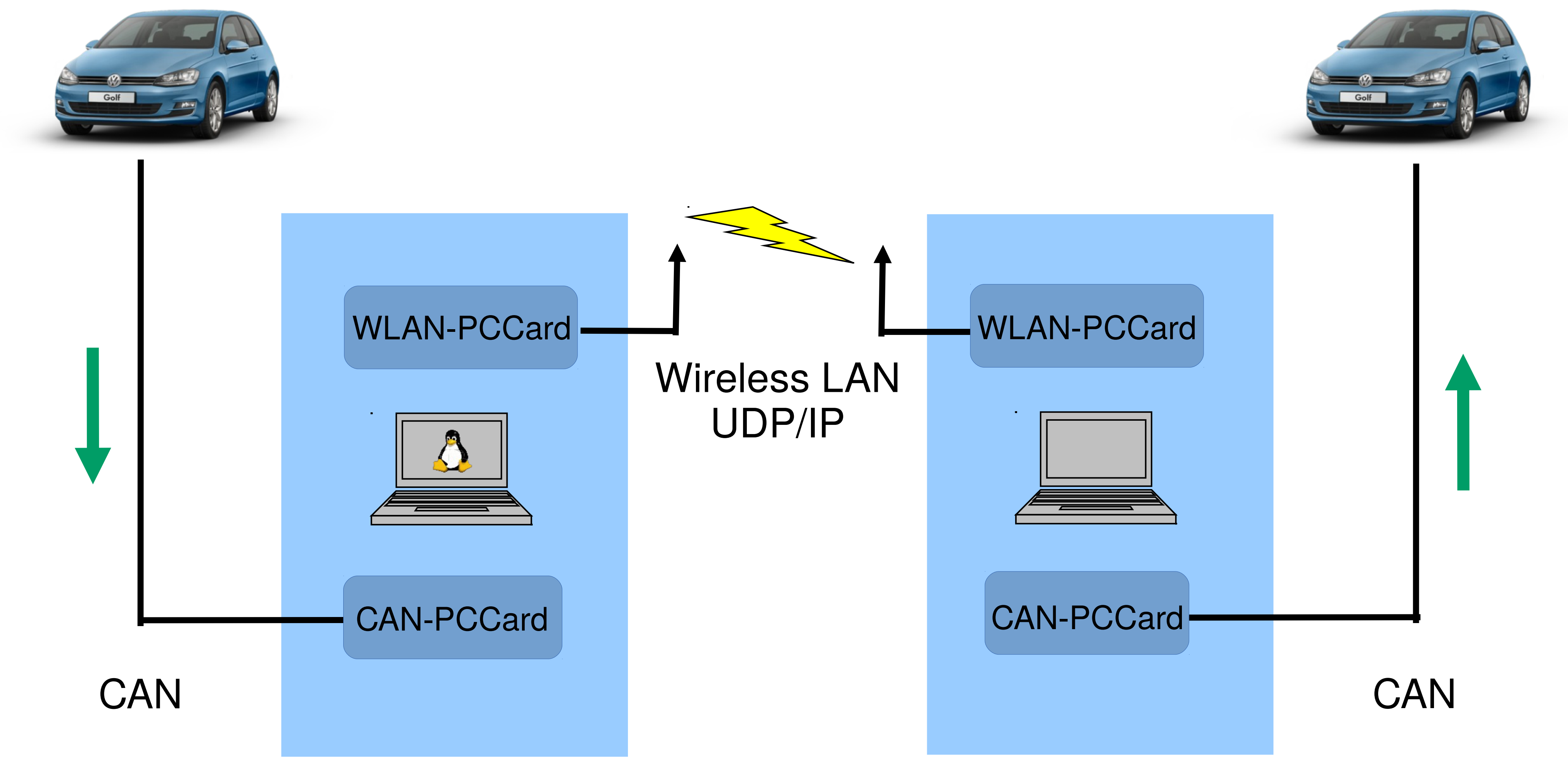
# CAN_RAW – Reading and writing of raw CAN frames




- Similar to known programming interfaces
  - A socket feels like a private CAN interface
  - per-socket CAN identifier receive filtersets
  - Linux timestamps in nano second resolution
  - Easy migration of existing CAN software

- Multiple applications can run independently
  - Network transparency through local echo of sent frames
  - Functions can be split into different processes

# CAN_RAW – Example CAN-over-WLAN Bridge



# CAN_RAW – Example CAN-over-WLAN Bridge

```
(..) /* some source code - don't worry */

int can;                    /* socket handle */
int wlan;
struct can_frame mymsg;     /* data structure for CAN frames */


can = socket(PF_CAN, SOCK_DGRAM, CAN_RAW);   /* CAN RAW Socket */
wlan = socket(PF_INET, SOCK_DGRAM, 0);       /* UDP/IP Socket  */

(..) /* set addresses and CAN filters */

bind(can, (struct sockaddr *)&can_addr, sizeof(can_addr));
connect(wlan, (struct sockaddr *)&in_addr, sizeof(in_addr));

while (1) {
           read(can,  &mymsg, sizeof(struct can_frame));
           write(wlan, &mymsg, sizeof(struct can_frame));
}
```

# CAN_RAW socket options

```
/* for socket options affecting the socket (not the global system) */

enum {
        CAN_RAW_FILTER = 1,     /* set 0 .. n can_filter(s)          */
        CAN_RAW_ERR_FILTER,     /* set filter for error frames       */
        CAN_RAW_LOOPBACK,       /* local loopback (default:on)       */
        CAN_RAW_RECV_OWN_MSGS,  /* receive my own msgs (default:off) */
        CAN_RAW_FD_FRAMES,      /* allow CAN FD frames (default:off) */
        CAN_RAW_JOIN_FILTERS,   /* all filters must match to trigger */
};

/**
 * A filter matches, when
 *
 *          <received_can_id> & mask == can_id & mask
 */
struct can_filter {
        canid_t can_id;
        canid_t can_mask;
};
```

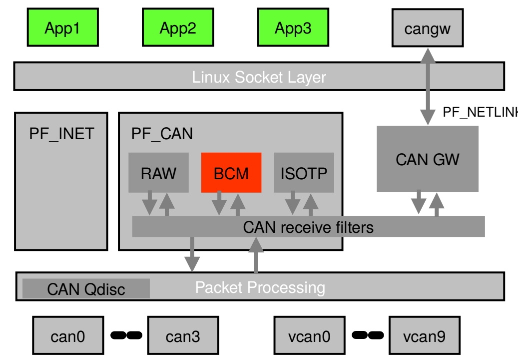
https://www.kernel.org/doc/Documentation/networking/can.txt

4.1.1.1 CAN filter usage optimisation

## CAN_RAW related can-utils

- **`candump`** – display, filter and log CAN data to files
- **`cansend`** – send a single frame
- **`cangen`** – generate (random) CAN traffic
- **`canplayer`** – replay CAN logfiles
- **`canbusload`** – calculate and display the CAN busload

# CAN_BCM – timer support and filters for cyclic messages

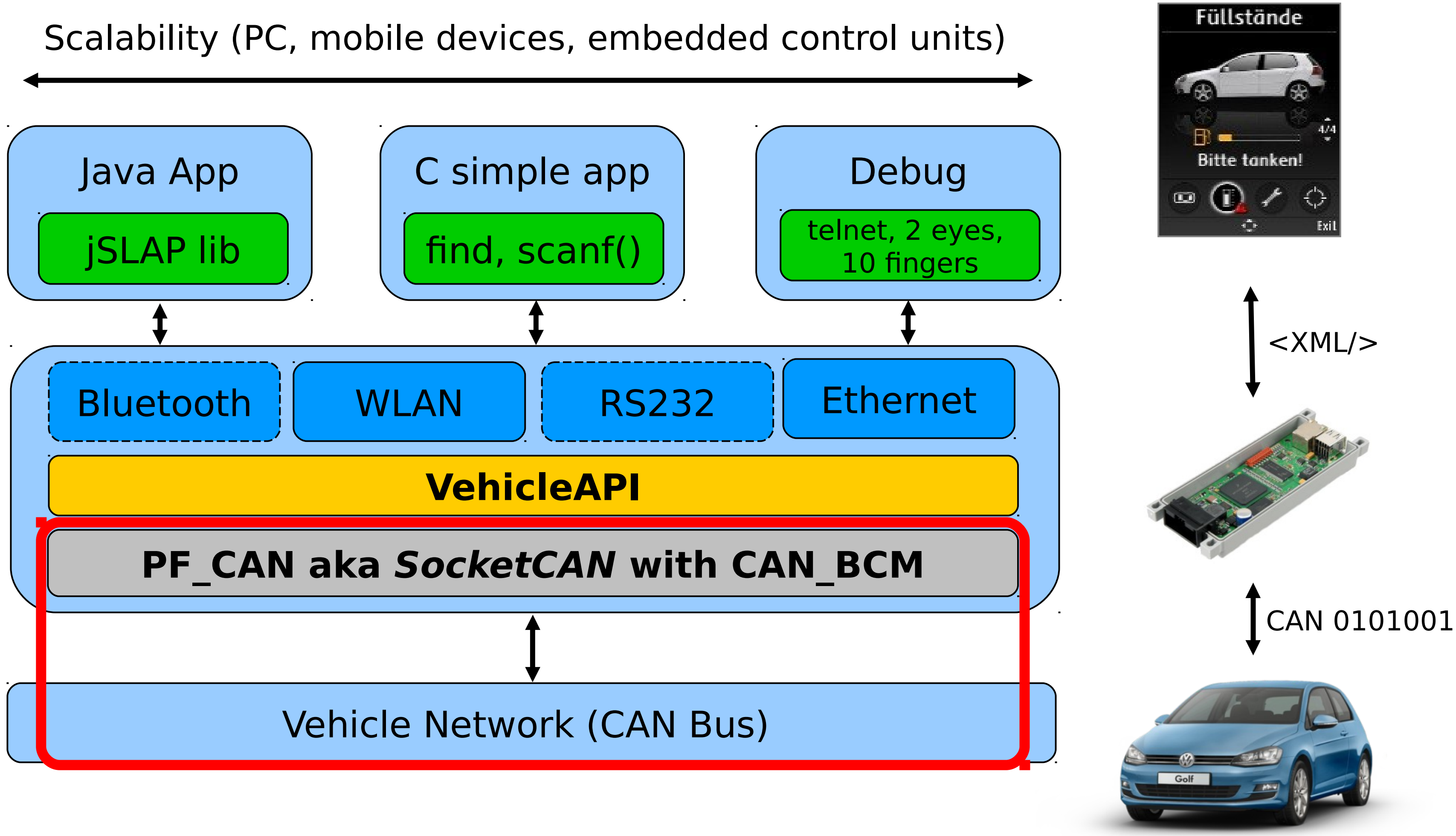



- Executes in operating system context
- Programmable by BCM socket commands

- CAN receive path functions
  - Filter bit-wise content in CAN frame payload
  - Throttle update rate for changed received data
  - Detect timeouts of cyclic messages (deadline monitoring)

- CAN transmit path functions
  - Autonomous timer based sending of CAN frames
  - Multiplex CAN messages and instant data updates

# CAN_BCM – Vehicle data access prototyping technology



# CAN_BCM programming interface opcodes

```
enum {
        TX_SETUP = 1,   /* create (cyclic) transmission task */
        TX_DELETE,      /* remove (cyclic) transmission task */
        TX_READ,        /* read properties of (cyclic) transmission task */
        TX_SEND,        /* send one CAN frame */
        RX_SETUP,       /* create RX content filter subscription */
        RX_DELETE,      /* remove RX content filter subscription */
        RX_READ,        /* read properties of RX content filter subscription */
        TX_STATUS,      /* reply to TX_READ request */
        TX_EXPIRED,     /* notification on performed transmissions (count=0) */
        RX_STATUS,      /* reply to RX_READ request */
        RX_TIMEOUT,     /* cyclic message is absent */
        RX_CHANGED      /* updated CAN frame (detected content change) */
};
```

# CAN_BCM programming interface msg structure & flags

```
struct bcm_msg_head {
        __u32 opcode;
        __u32 flags;
        __u32 count;
        struct bcm_timeval ival1, ival2;
        canid_t can_id;
        __u32 nframes;
        struct can_frame frames[0];
};
```

```
#define SETTIMER            0x0001
#define STARTTIMER          0x0002
#define TX_COUNTEVT         0x0004
#define TX_ANNOUNCE         0x0008
#define TX_CP_CAN_ID        0x0010
#define RX_FILTER_ID        0x0020
#define RX_CHECK_DLC        0x0040
#define RX_NO_AUTOTIMER     0x0080
#define RX_ANNOUNCE_RESUME  0x0100
#define TX_RESET_MULTI_IDX  0x0200
#define RX_RTR_FRAME        0x0400
#define CAN_FD_FRAME        0x0800
```

# CAN_BCM programming interface example

```
if ((s = socket(PF_CAN, SOCK_DGRAM, CAN_BCM)) < 0) {
        perror("socket");
        return 1;
}

addr.can_family = PF_CAN;
strcpy(ifr.ifr_name, "vcan2");
ioctl(s, SIOCGIFINDEX, &ifr);
addr.can_ifindex = ifr.ifr_ifindex;

if (connect(s, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
        perror("connect");
        return 1;
}

txmsg.msg_head.opcode  = RX_SETUP;
txmsg.msg_head.can_id  = 0x042;
txmsg.msg_head.flags   = SETTIMER|RX_FILTER_ID;
txmsg.msg_head.ival1.tv_sec = 4;
txmsg.msg_head.ival1.tv_usec = 0;
txmsg.msg_head.ival2.tv_sec = 2;
txmsg.msg_head.ival2.tv_usec = 0;
txmsg.msg_head.nframes = 0;
```

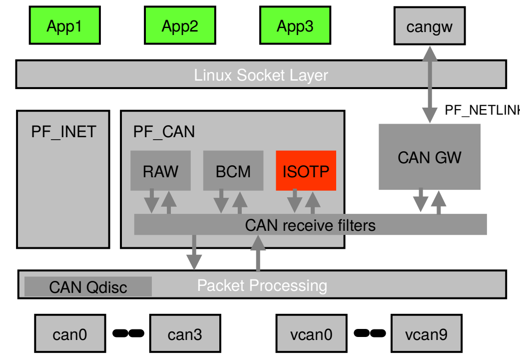
Multiple **`RX_SETUP`**‘s on different CAN interfaces via **`sendto()`** syscall

# CAN_BCM related can-utils

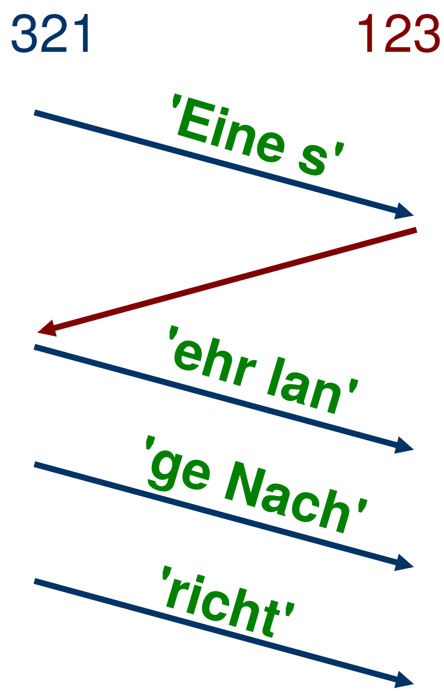
- **`cansniffer`** – display differences (in short)
- **`bcmserver`** – interactive BCM configuration (remote/local)
- **`socketcand`** – use CAN_BCM sockets via TCP/IP sockets

https://github.com/linux-can/can-utils
https://github.com/dschanoeh/socketcand

# CAN_ISOTP – CAN transport protocol ISO 15765-2:2016




- Segmented transfer of application content
- Transfer up to 4095 (*) bytes per ISO-TP PDU
- Bidirectional communication on two CAN IDs




First Frame

Flow Control (stmin = 1 sec)

Consecutive Frame

Consecutive Frame

Consecutive Frame



(*) = 15765-2:2016: 32 bit = 4GB

# CAN_ISOTP – Code example (UDS is just one step ahead!)

Creation of a point-to-point ISO 15765-2 transport channel:

```
struct ifreq ifr;
struct sockaddr_can addr;
char data[] = "Eine sehr lange Nachricht";          /* "a very long message" */

s = socket(PF_CAN, SOCK_DGRAM, CAN_ISOTP);          /* create socket instance */

addr.can_family = AF_CAN;                           /* address family AF_CAN */
addr.can_ifindex = ifr.ifr_ifindex;                 /* CAN interface index e.g. for can0 */
addr.can_addr.tp.tx_id = 0x321;                     /* transmit on this CAN ID */
addr.can_addr.tp.rx_id = 0x123;                     /* receive on this CAN ID */

bind(s, (struct sockaddr *)&addr, sizeof(addr));    /* establish datagramm communication */

write(s, data, strlen(data));                       /* sending of messages */
read(s, data, strlen(data));                        /* reception of messages */

close(s);                                           /* close socket instance */
```

’Normal’ application programmers can easily write applications for the vehicle using established techniques from the standard-IT!

# Open Source tools for ISO-TP

Sending of “Eine sehr lange Nachricht” via ISO-TP

```
oliver@linuxbox:~$ echo "45 69 6e 65 20 73 65 68 72 20 6c 61 6e 67 65 20
4e 61 63 68 72 69 63 68 74" | isotpsend -s 321 -d 123 can0
```

```
oliver@linuxbox:~$ isotpdump -c -a -s 321 -d 123 can0
 can0  321  [8]  [FF] ln: 25   data: 45 69 6E 65 20 73     -  'Eine s'
 can0  123  [3]  [FC] FC: 0 = CTS # BS: 0 = off # STmin: 0x00 = 0 ms
 can0  321  [8]  [CF] sn: 1    data: 65 68 72 20 6C 61 6E  -  'ehr lan'
 can0  321  [8]  [CF] sn: 2    data: 67 65 20 4E 61 63 68  -  'ge Nach'
 can0  321  [6]  [CF] sn: 3    data: 72 69 63 68 74        -  'richt'
```

```
oliver@linuxbox:~$ candump -a can0
  can0  321  [8] 10 19 45 69 6E 65 20 73   '..Eine s'
  can0  123  [3] 30 00 00                  '0..'
  can0  321  [8] 21 65 68 72 20 6C 61 6E   '!ehr lan'
  can0  321  [8] 22 67 65 20 4E 61 63 68   '"ge Nach'
  can0  321  [6] 23 72 69 63 68 74         '#richt'
```

(colored by hand)

## CAN_ISOTP related can-utils

- **`isotpsend`** – send a single ISO-TP PDU
- **`isotprecv`** – receive ISO-TP PDU(s)
- **`isotpsniffer`** – 'wiretap' ISO-TP PDU(s)
- **`isotpdump`** – 'wiretap' and interpret CAN messages (CAN_RAW)
- **`isotptun`** – create a bi-directional IP tunnel on CAN via ISO-TP
- **`socketcand`** – use CAN_ISOTP sockets via TCP/IP sockets

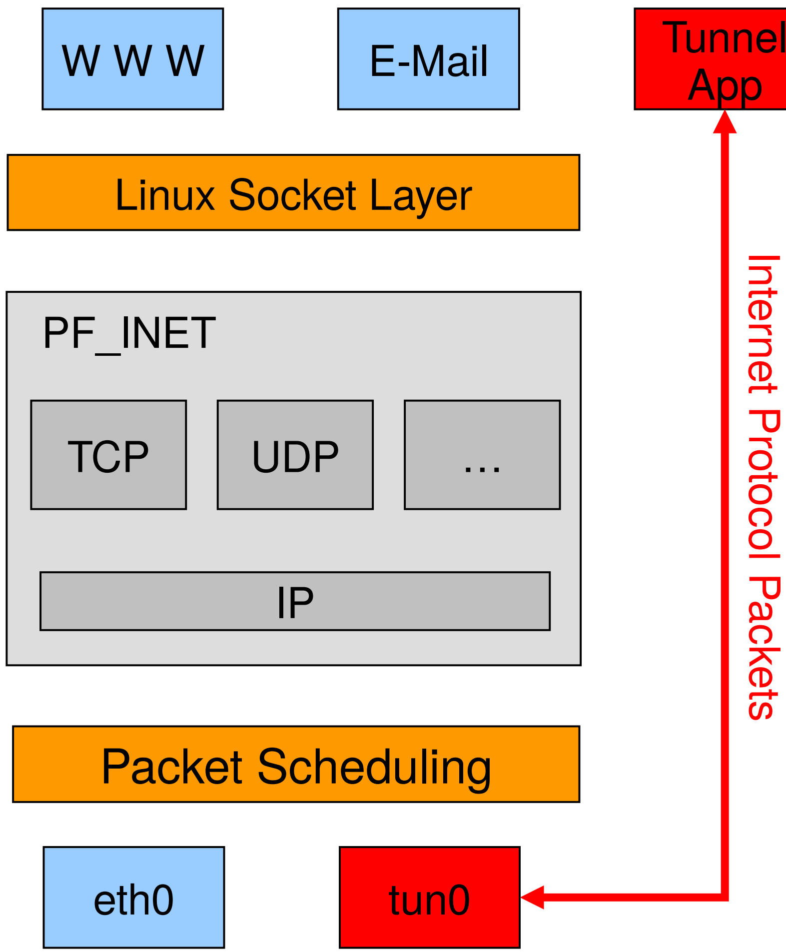
https://github.com/linux-can/can-utils
https://github.com/dschanoeh/socketcand

# CAN_ISOTP options of isotpsend

```
Usage: isotpsend [options] <CAN interface>
Options: -s <can_id>  (source can_id. Use 8 digits for extended IDs)
         -d <can_id>  (destination can_id. Use 8 digits for extended IDs)
         -x <addr>[:<rxaddr>] (extended addressing / opt. separate rxaddr)
         -p [tx]:[rx] (set and enable tx/rx padding bytes)
         -P <mode>    (check rx padding for (l)ength (c)ontent (a)ll)
         -t <time ns> (frame transmit time (N_As) in nanosecs)
         -f <time ns> (ignore FC and force local tx stmin value in nanosecs)
         -D <len>     (send a fixed PDU with len bytes - no STDIN data)
         -L <mtu>:<tx_dl>:<tx_flags> (link layer options for CAN FD)

CAN IDs and addresses are given and expected in hexadecimal values.
The pdu data is expected on STDIN in space separated ASCII hex values.
```

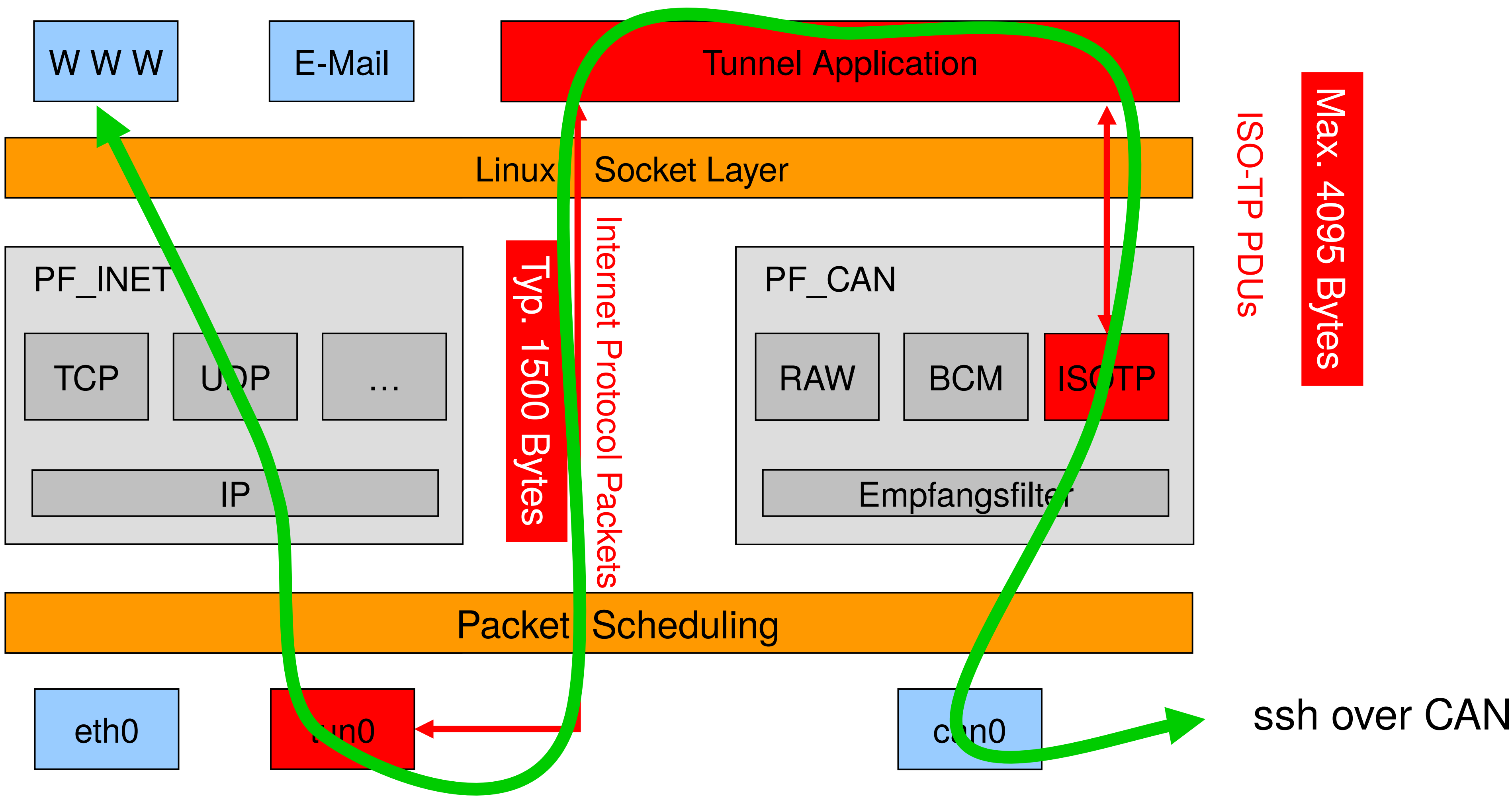
https://github.com/linux-can/can-utils
https://github.com/hartkopp/can-isotp-modules

# PPPoC: How to build an Internet Protocol Tunnel?

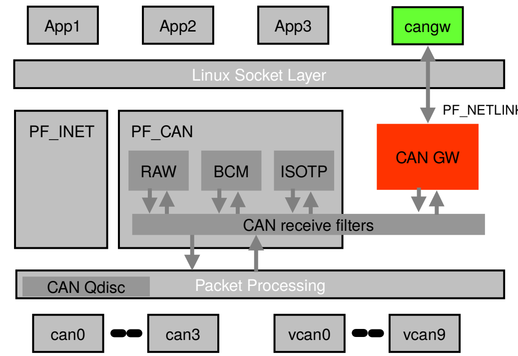



```
int t;
struct ifreq ifr;

t = open("/dev/net/tun", O_RDWR);

memset(&ifr, 0, sizeof(ifr));
ifr.ifr_flags = IFF_TUN | IFF_NO_PI;

strncpy(ifr.ifr_name, "tun%d", IFNAMSIZ);
ioctl(t, TUNSETIFF, (void *) &ifr);

/* now we have a tun0 (or tun1 or ...)   */
/* netdevice connected to filedescriptor t */
```

# PPPoC: Internet Protokoll Tunnel over ISO 15765-2

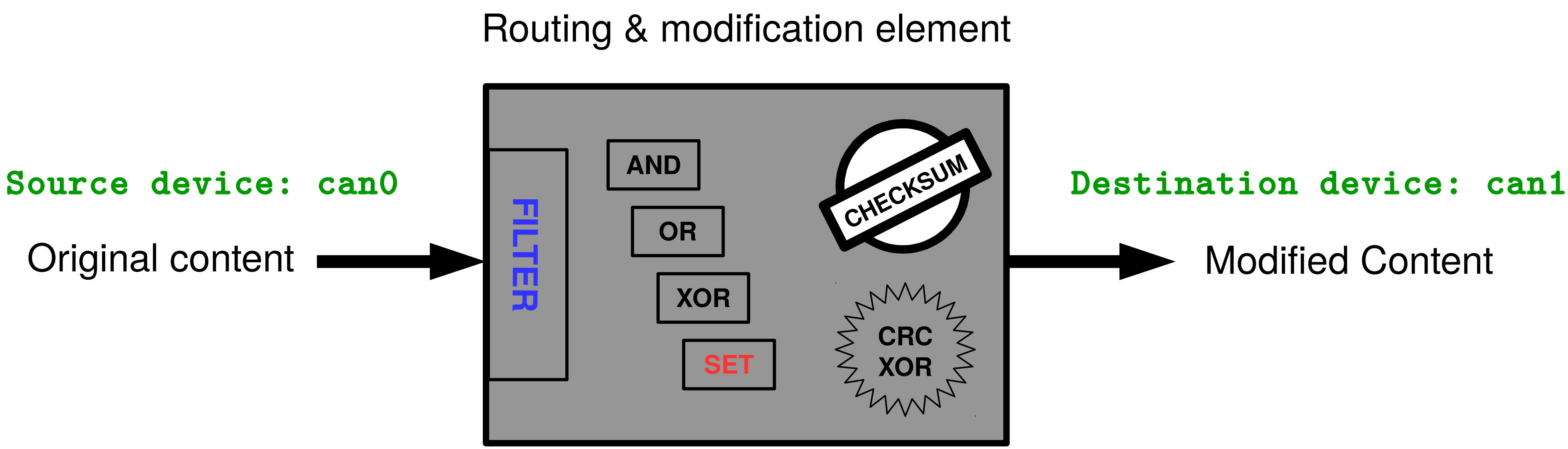


# CAN_GW – Linux kernel based CAN frame routing



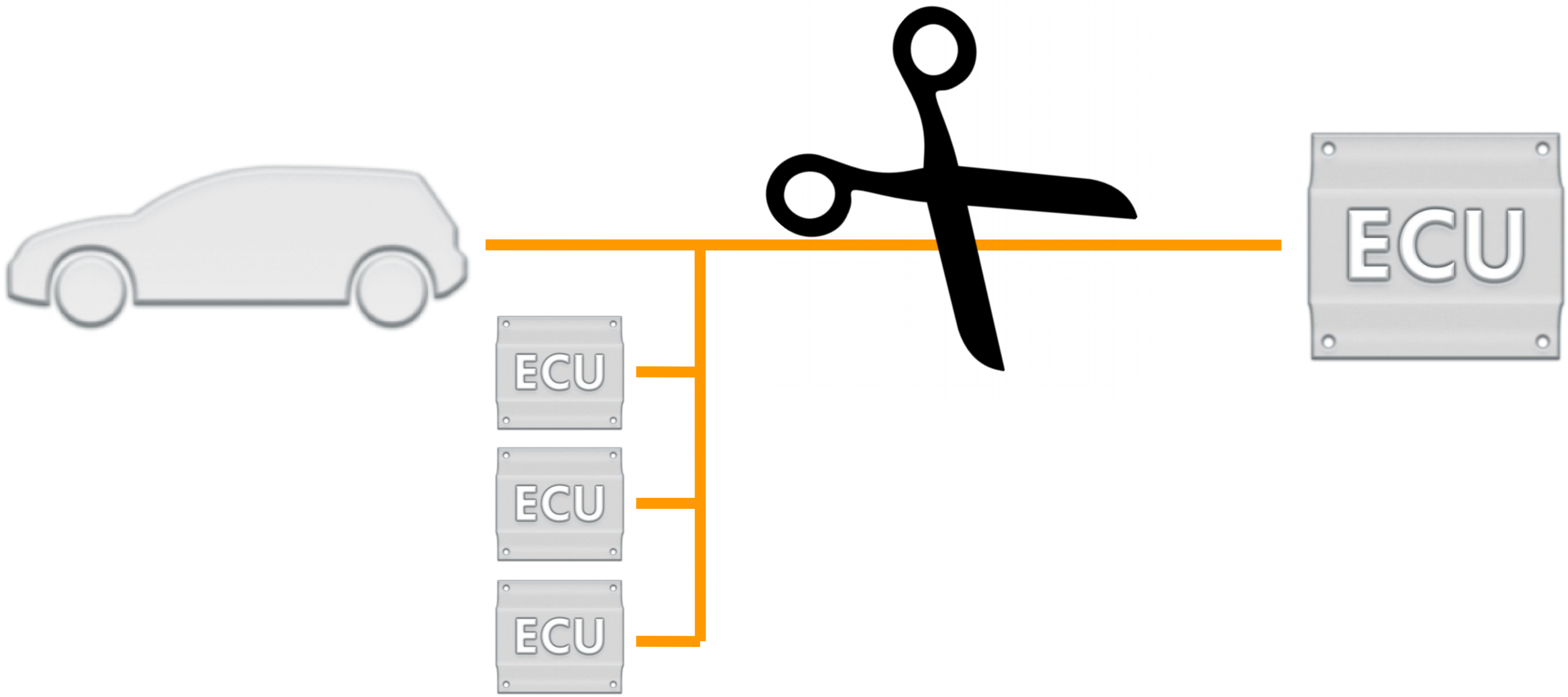

- Efficient CAN frame routing in OS context
- Re-use of Linux networking technology
  - PF_CAN receive filter capabilities
  - Linux packet processing NET_RX softirq
  - PF_NETLINK based configuration interface (known from Linux network routing configuration like 'iptables')
- Optional CAN frame modifications on the fly
  - Modify CAN identifier, data length code, payload data with AND/OR/XOR/SET operations
  - Calculate XOR and CRC8 checksums after modification
  - Support of different CRC8 profiles (1U8, 16U8, SFFID_XOR)

# CAN_GW – Routing & modification configuration entity

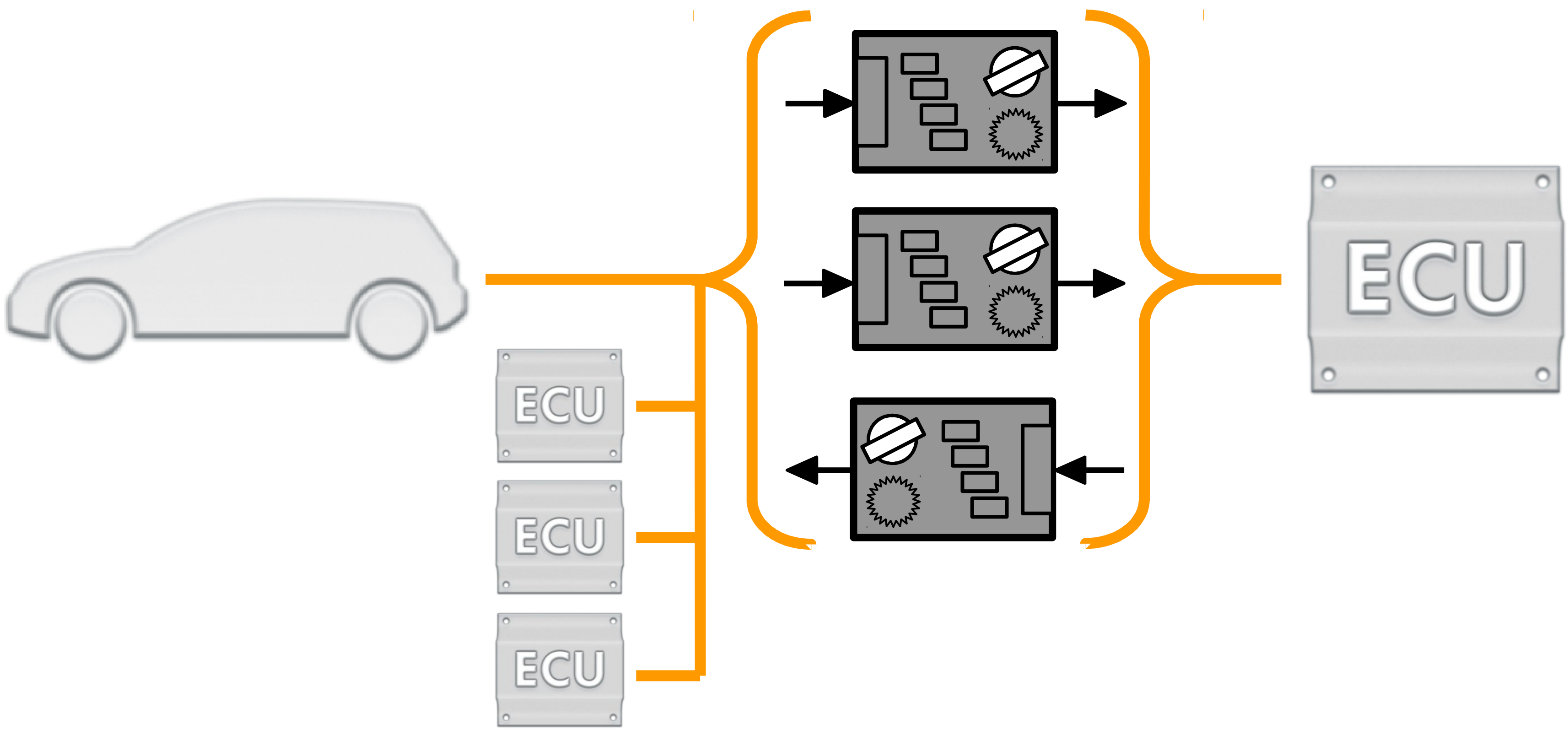



```
cangw -A -s can0 -d can1 -e -f 123:C00007FF -m SET:IL:333.4.1122334455667788
```

# CAN_GW – Routing & modification ‘preparation’



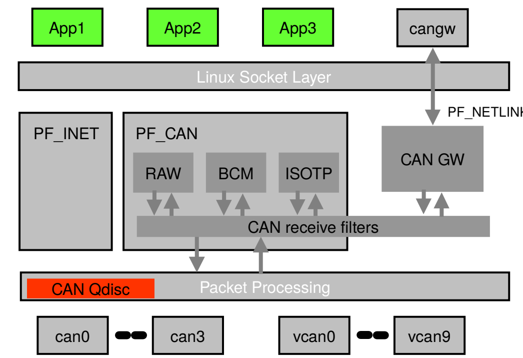
# CAN_GW – Routing & modification setup example



# CAN Gateway userspace tool

```
Usage: cangw [options]

Commands:  -A (add a new rule)
           -D (delete a rule)
           -F (flush / delete all rules)
           -L (list all rules)

Mandatory: -s <src_dev>  (source netdevice)
           -d <dst_dev>  (destination netdevice)
Options:   -t (preserve src_dev rx timestamp)
           -e (echo sent frames - recommended on vcanx)
           -i (allow to route to incoming interface)
           -u <uid> (user defined modification identifier)
           -l <hops> (limit the number of frame hops / routings)
           -f <filter> (set CAN filter)
           -m <mod> (set frame modifications)
           -x <from_idx>:<to_idx>:<result_idx>:<init_xor_val> (XOR checksum)
           -c <from>:<to>:<result>:<init_val>:<xor_val>:<crctab[256]> (CRC8 cs)
           -p <profile>:[<profile_data>] (CRC8 checksum profile & parameters)

Values are given and expected in hexadecimal values. Leading 0s can be omitted.

<filter> is a <value>:<mask> CAN identifier filter

<mod> is a CAN frame modification instruction consisting of
<instruction>:<can_frame-elements>:<can_id>.<can_dlc>.<can_data>
 - <instruction> is one of 'AND' 'OR' 'XOR' 'SET'
 - <can_frame-elements> is _one_ or _more_ of 'I'dentifier 'L'ength 'D'ata
 - <can_id> is an u32 value containing the CAN Identifier
 - <can_dlc> is an u8 value containing the data length code (0 .. 8)
 - <can_data> is always eight(!) u8 values containing the CAN frames data
The max. four modifications are performed in the order AND -> OR -> XOR -> SET

Example:
cangw -A -s can0 -d vcan3 -e -f 123:C00007FF -m SET:IL:333.4.1122334455667788

Supported CRC 8 profiles:
Profile '1' (1U8)       - add one additional u8 value
Profile '2' (16U8)      - add u8 value from table[16] indexed by (data[1] & 0xF)
Profile '3' (SFFID_XOR) - add u8 value (can_id & 0xFF) ^ (can_id >> 8 & 0xFF)
```

# Traffic shaping for CAN frames




- Multiple applications can share one CAN bus
- Different per-application requirements
  - **Timing requirements** for cyclic messages or transport protocol timeouts
  - **Bandwidth requirements**
- How to ensure priority handling for outgoing CAN frames ?? (CAN network interfaces just implement a short FIFO queue)
- Similar requirements are known from Internet Protocol traffic (e.g. to reduce bandwidth for peer-to-peer networking)

# Traffic shaping for CAN frames – Why you need it ...

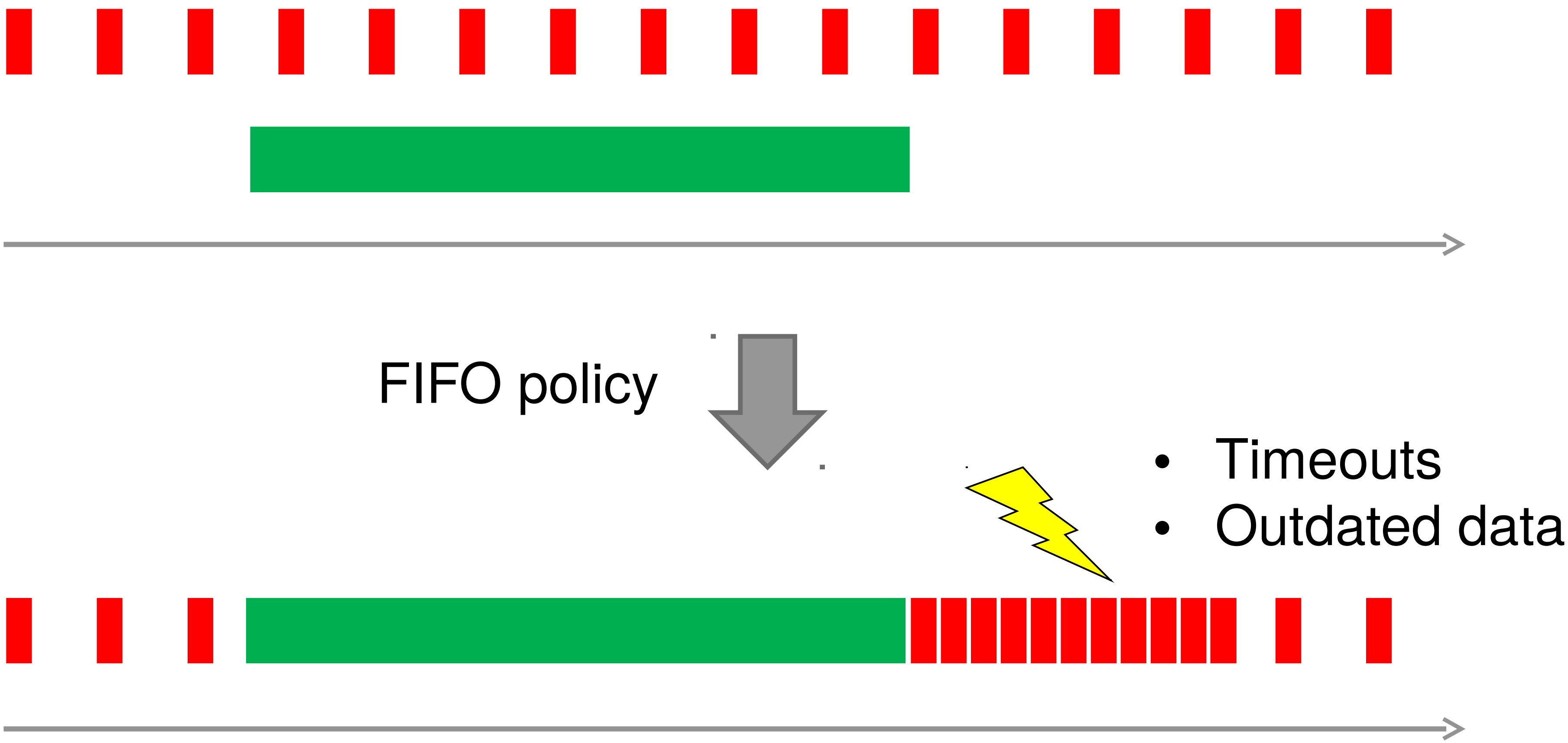
App1

Send status information every 200ms

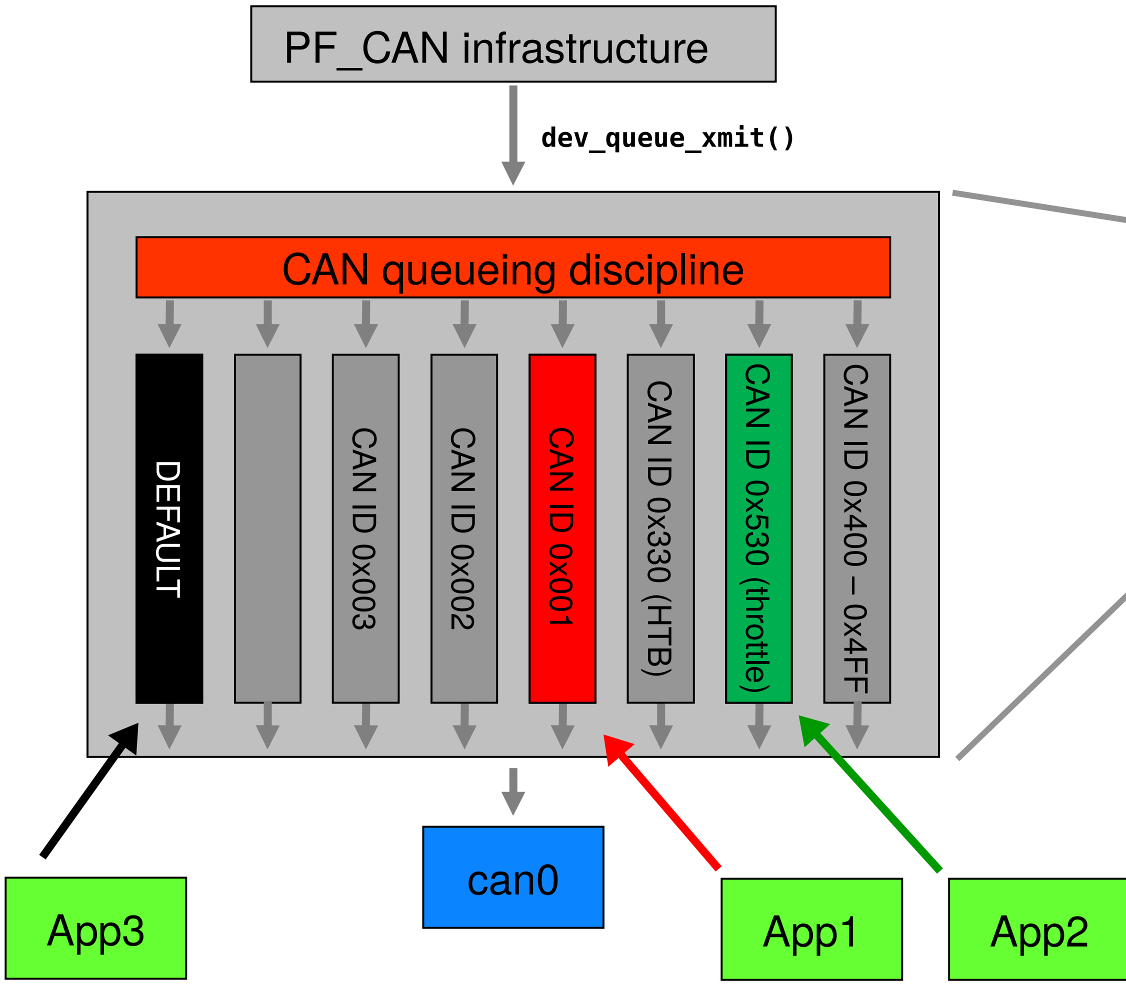
App2

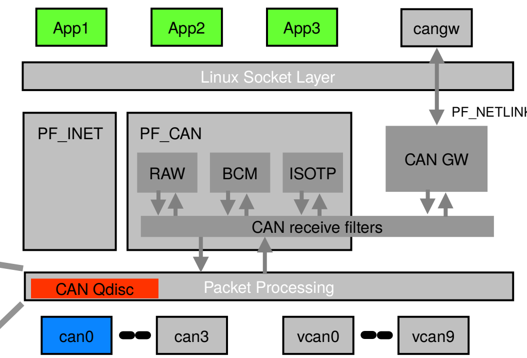
Send bulk data, like a ISO TP PDU with stmin = 0 (no delay)

# Traffic shaping for CAN frames – 'FIFO only' does not fit



# Linux Queueing Disciplines for CAN Frames

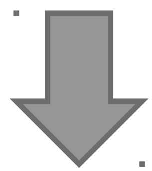


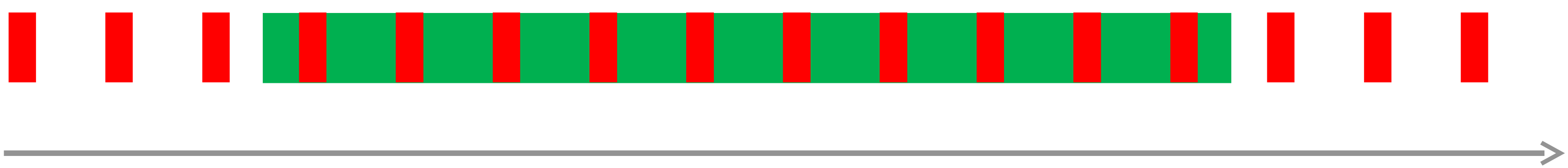




- Implement a new **ematch** rule to handle CAN IDs (`net/sched/em_canid.c`)
- Extend the **traffic control tool** `tc` to support CAN IDs

# Traffic shaping for CAN frames – Example

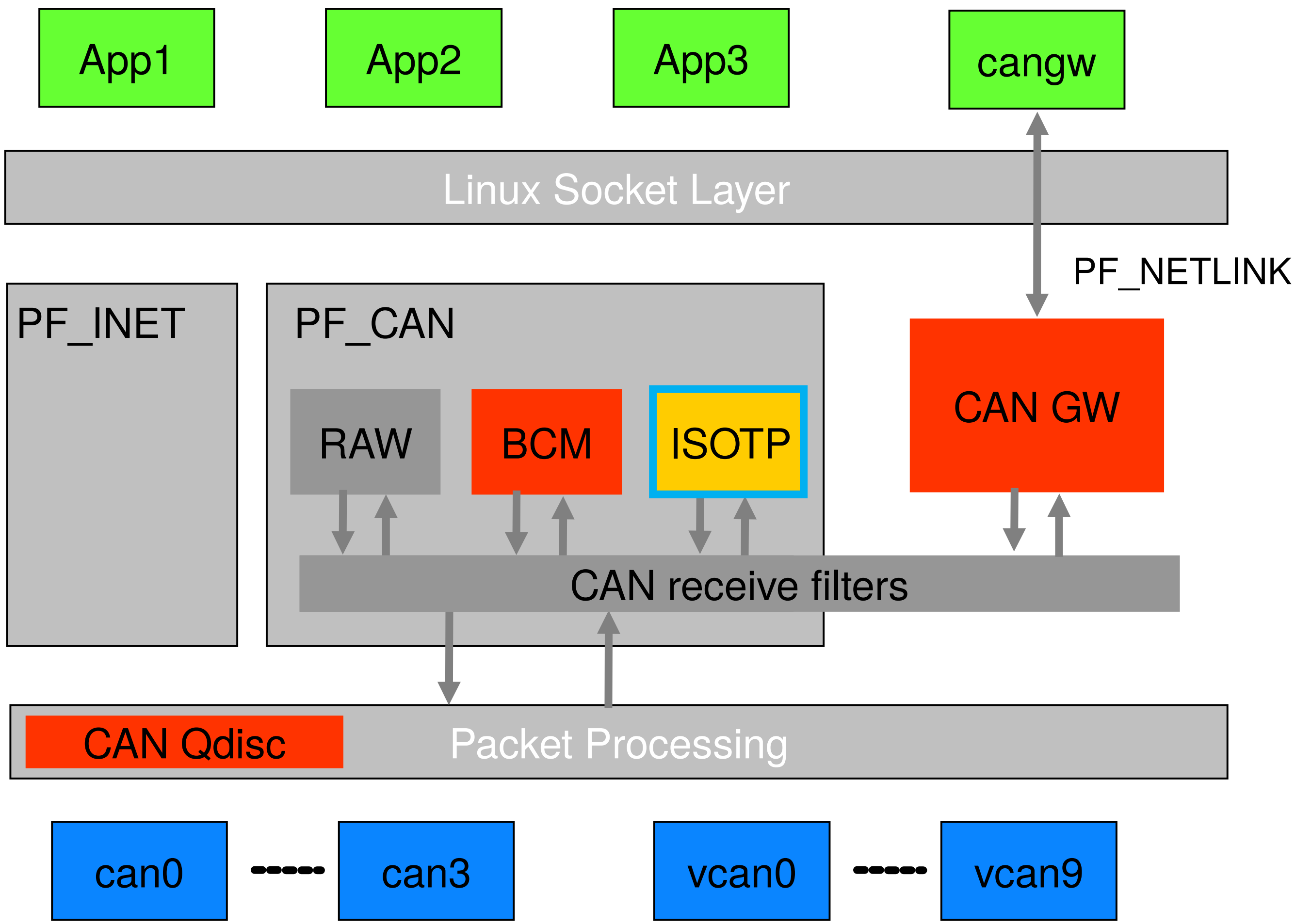
Application specific
Qdisc configuration
(by host admin / root)

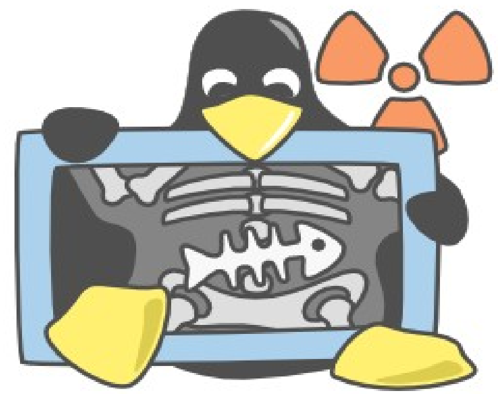




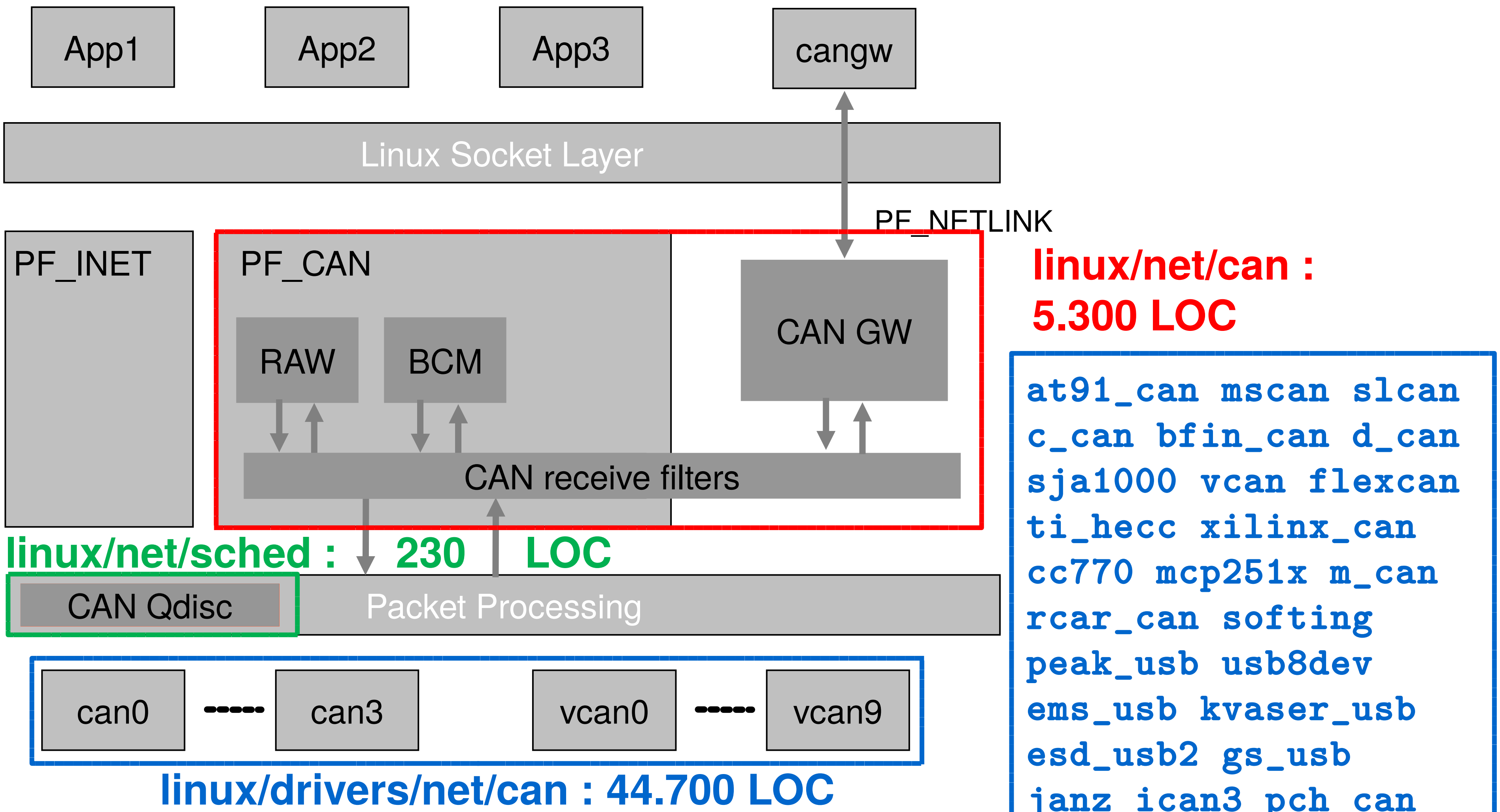
http://rtime.felk.cvut.cz/can/socketcan-qdisc-final.pdf

# Summary


App1
App2
App3
cangw
Linux Socket Layer
PF_NETLINK
PF_INET
PF_CAN
RAW
BCM
ISOTP
CAN GW
CAN receive filters
CAN Qdisc
Packet Processing
can0
can3
vcan0
vcan9

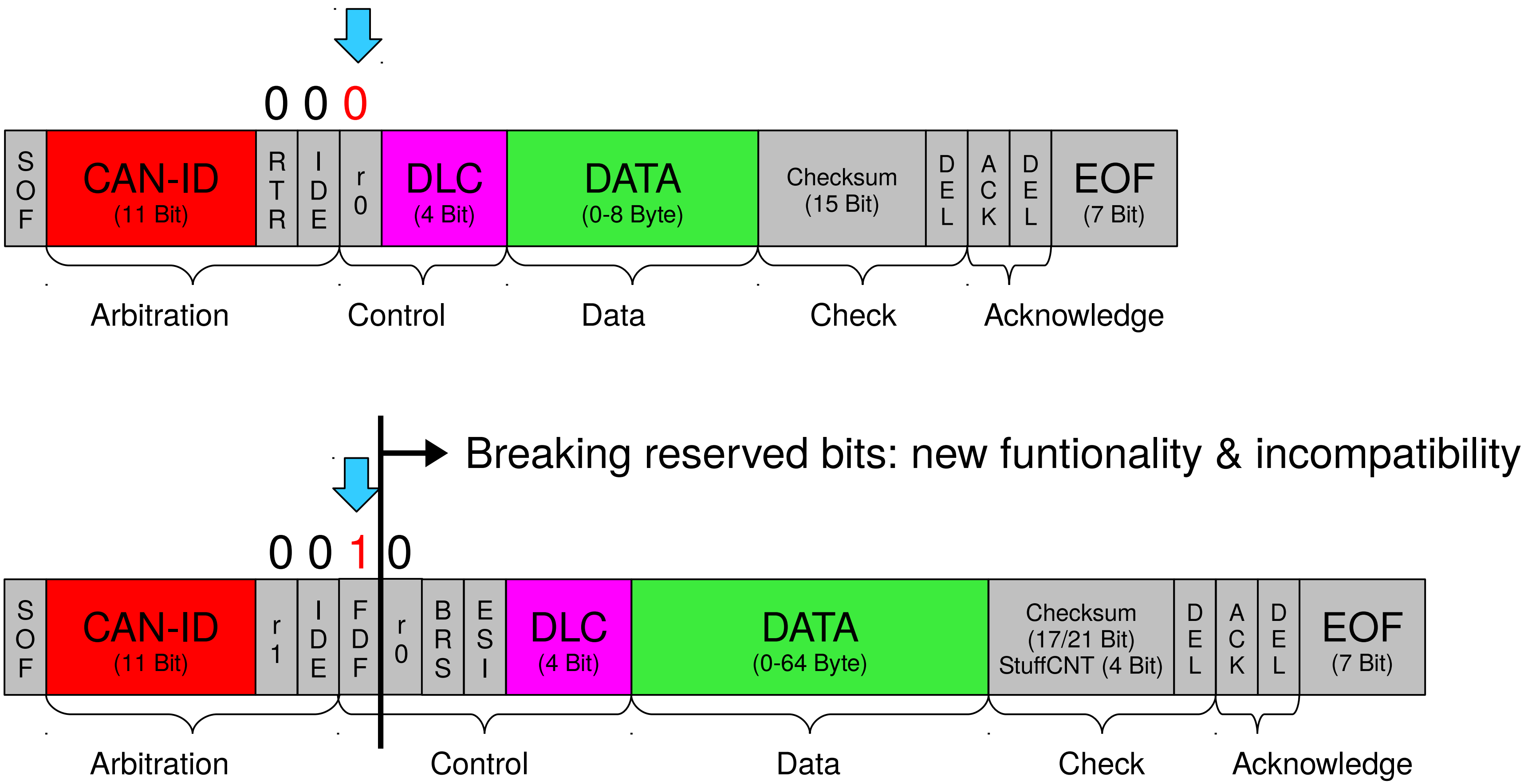


# CAN related Lines of Code summary (Linux 4.11–rc4)

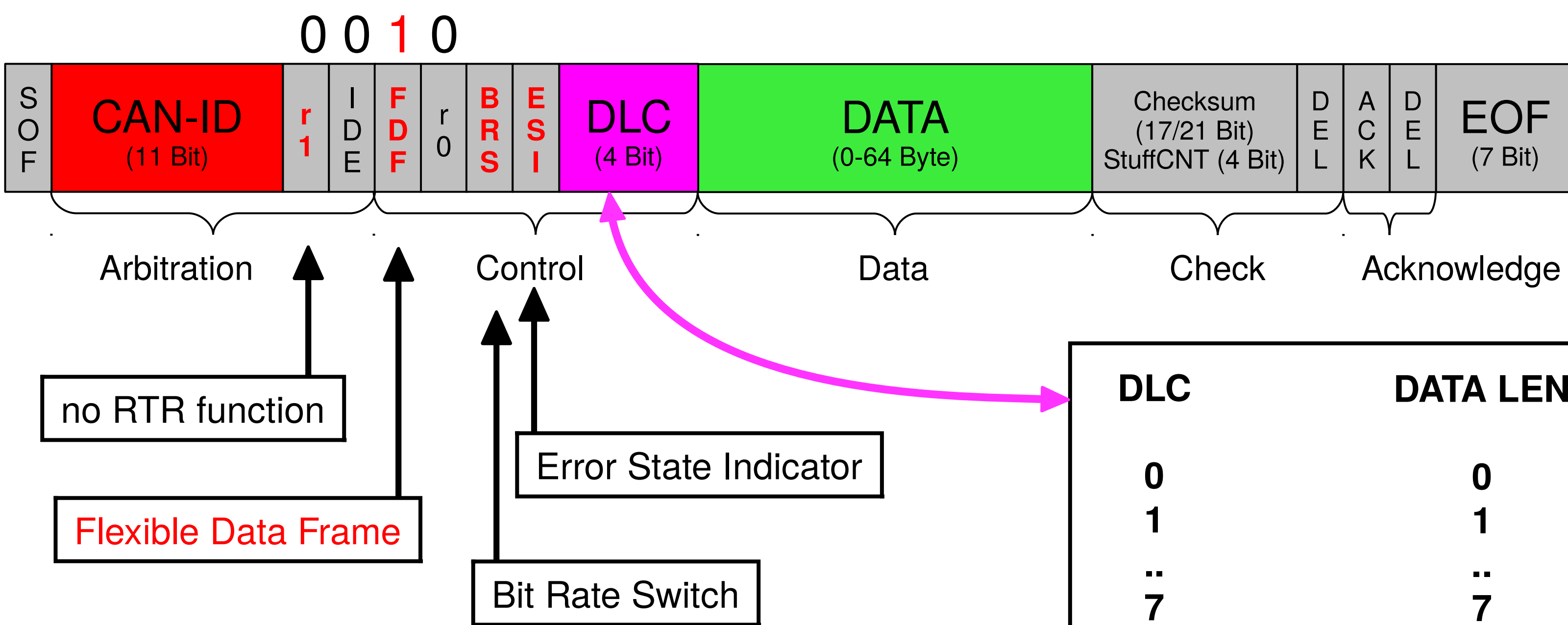


# CAN FD Integration in Linux
# Adopting CAN with Flexible Data rate

# Switching from CAN 2.0B to CAN FD by using the reserved bit



# CAN FD – new bits and definitions in detail

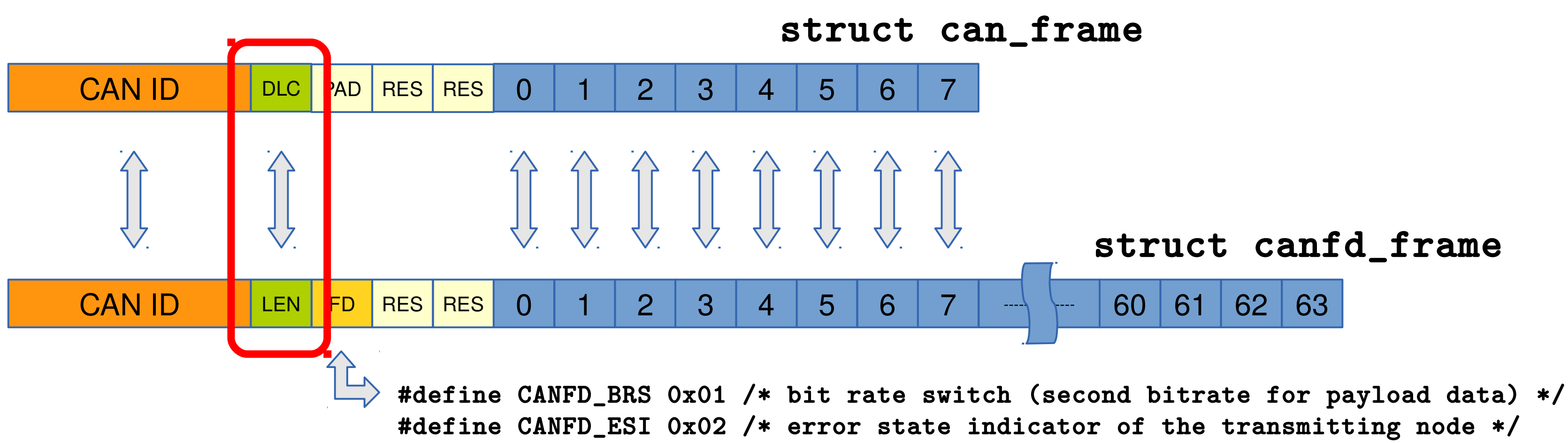



| DLC | DATA LEN |
|---|---|
| 0 | 0 |
| 1 | 1 |
| .. | .. |
| 7 | 7 |
| 8 | 8 |
| 9 | 12 |
| A | 16 |
| B | 20 |
| C | 24 |
| D | 32 |
| E | 48 |
| F | 64 |

***non-linear(!!)*** **mapping : DLC ⇔ payload length**

# Linux CAN FD length information and data structure

- DLC mostly has been used as plain payload length information (1:1 mapping)
- But CAN FD implements a **non-linear length** definition
- Introduce a structure element 'len' for CAN FD to preserve common usage
- The mapping of DLC ⇔ LEN and vice versa is done *invisible* in the CAN driver



# Compatible data structure layout for CAN2.0B and CAN FD

- CAN2.0B data structure

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

- CAN FD data structure

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

# Preserve common processing of length information

- Processing length information with CAN data structure

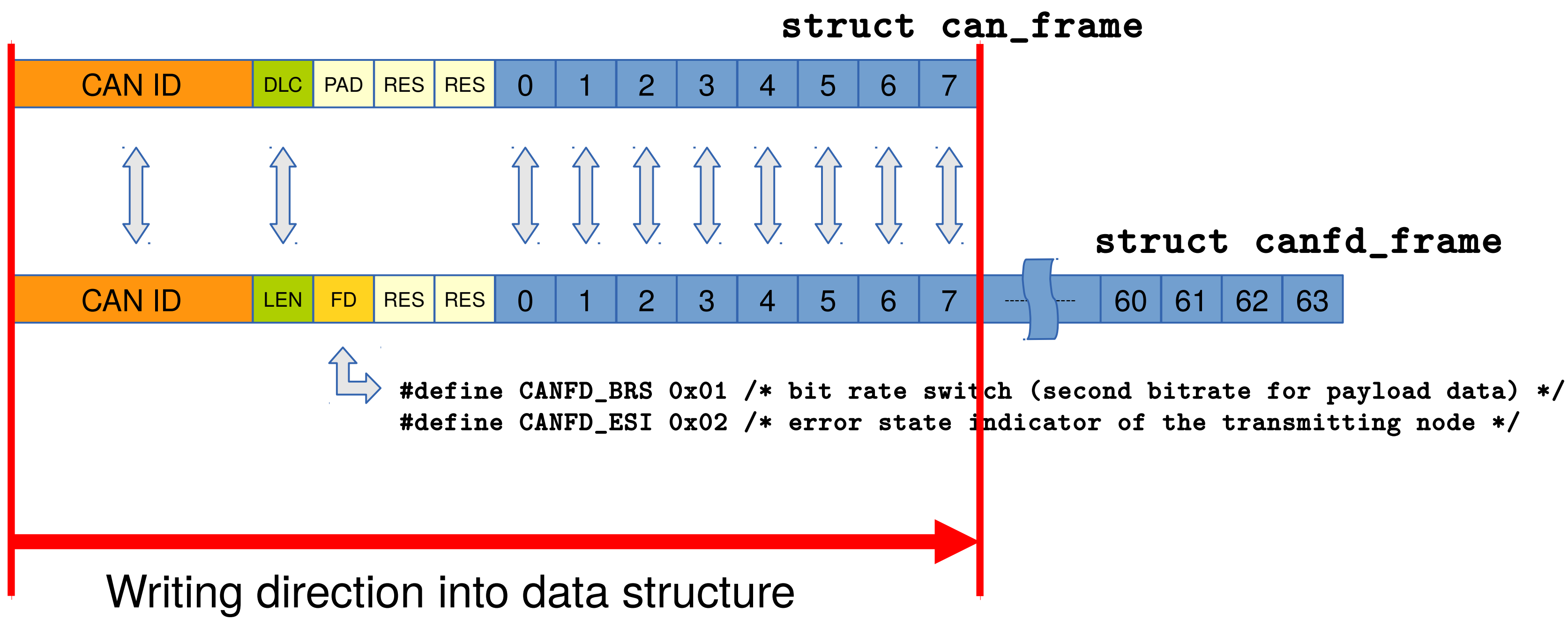
```
struct can_frame cframe;

for (i=0; i < cframe.can_dlc; i++)
            printf("%02X ", cframe.data[i]); /* print payload */
```

- Processing length information with CAN FD data structure

```
struct canfd_frame cframe;

for (i=0; i < cframe.len; i++)
            printf("%02X ", cframe.data[i]); /* print payload */

/* cframe.len = plain data length from 0 to 64 byte */
```

# CAN FD data structure – dual use with Classic CAN layout

Writing CAN 2.0B data into a CAN FD data structure creates valid content.



# How to activate CAN FD on a CAN_RAW socket

- Reading and writing CAN data structures

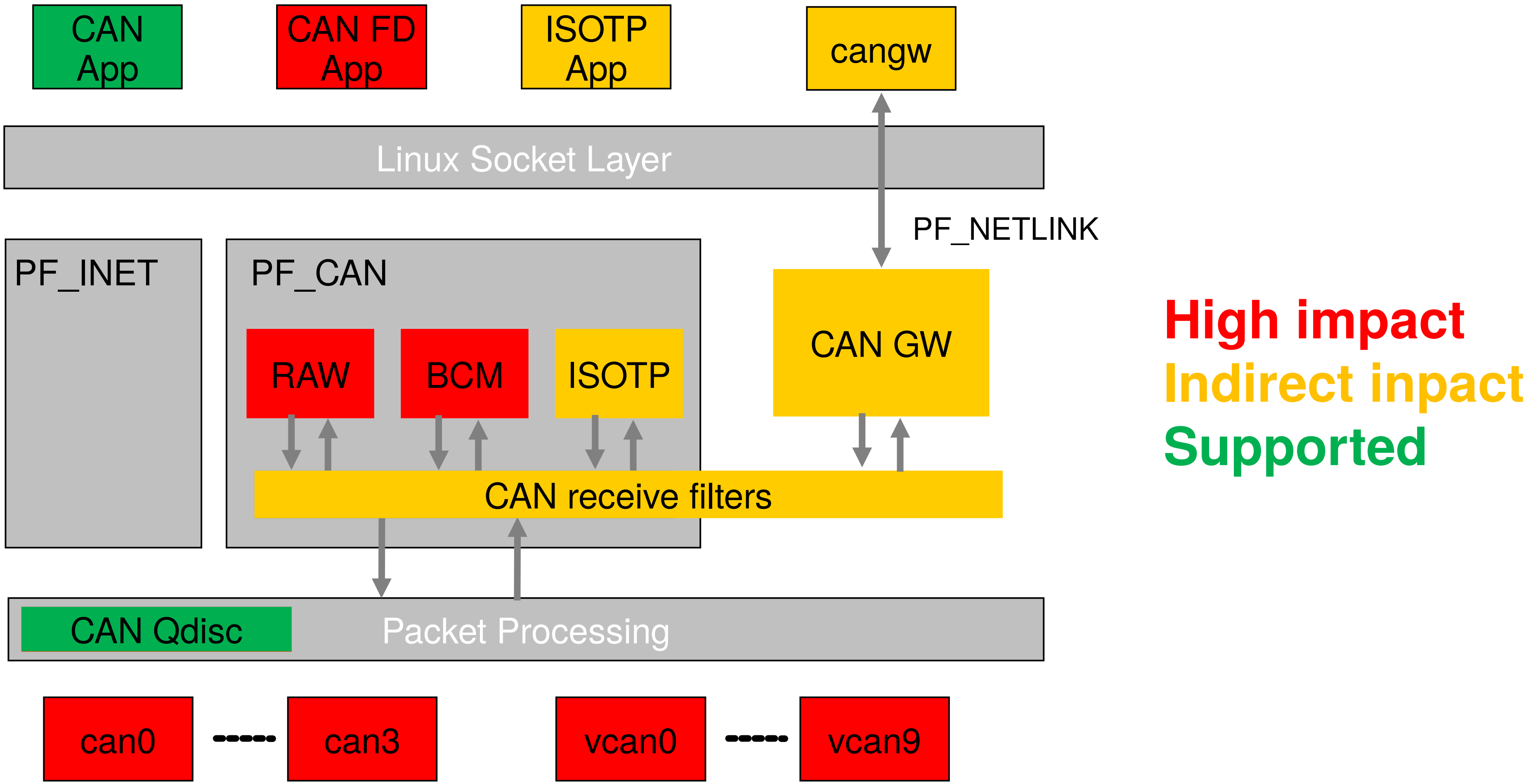
```
struct can_frame cframe;
int s = socket(PF_CAN, SOCK_DGRAM, CAN_RAW);
(..)
nbytes = read(s, &cframe, sizeof(struct can_frame));
```

- Switch the socket into CAN FD mode with `setsockopt()` syscall

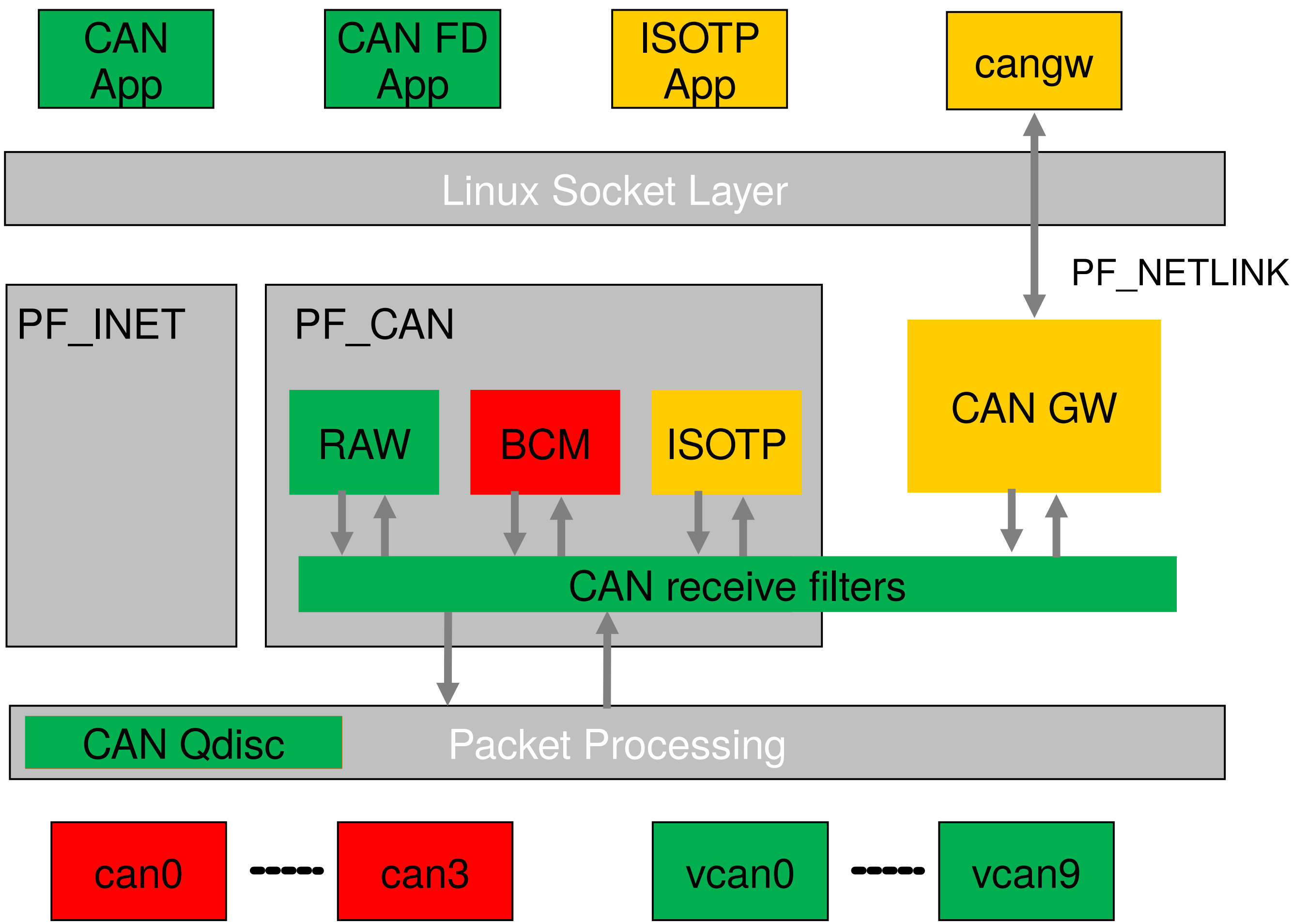
```
struct canfd_frame cframe;
int s = socket(PF_CAN, SOCK_DGRAM, CAN_RAW);
setsockopt(s, SOL_CAN_RAW, CAN_RAW_FD_FRAMES, ...);
(..)
nbytes = read(s, &cframe, sizeof(struct canfd_frame));
```

# Impact of CAN FD to Linux and CAN applications

## Depending on `struct can_frame`

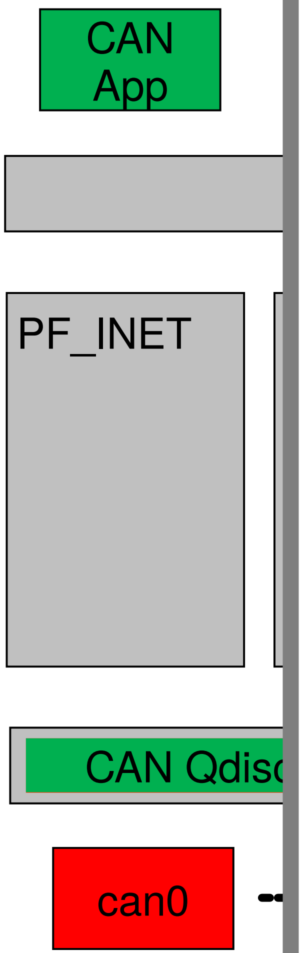


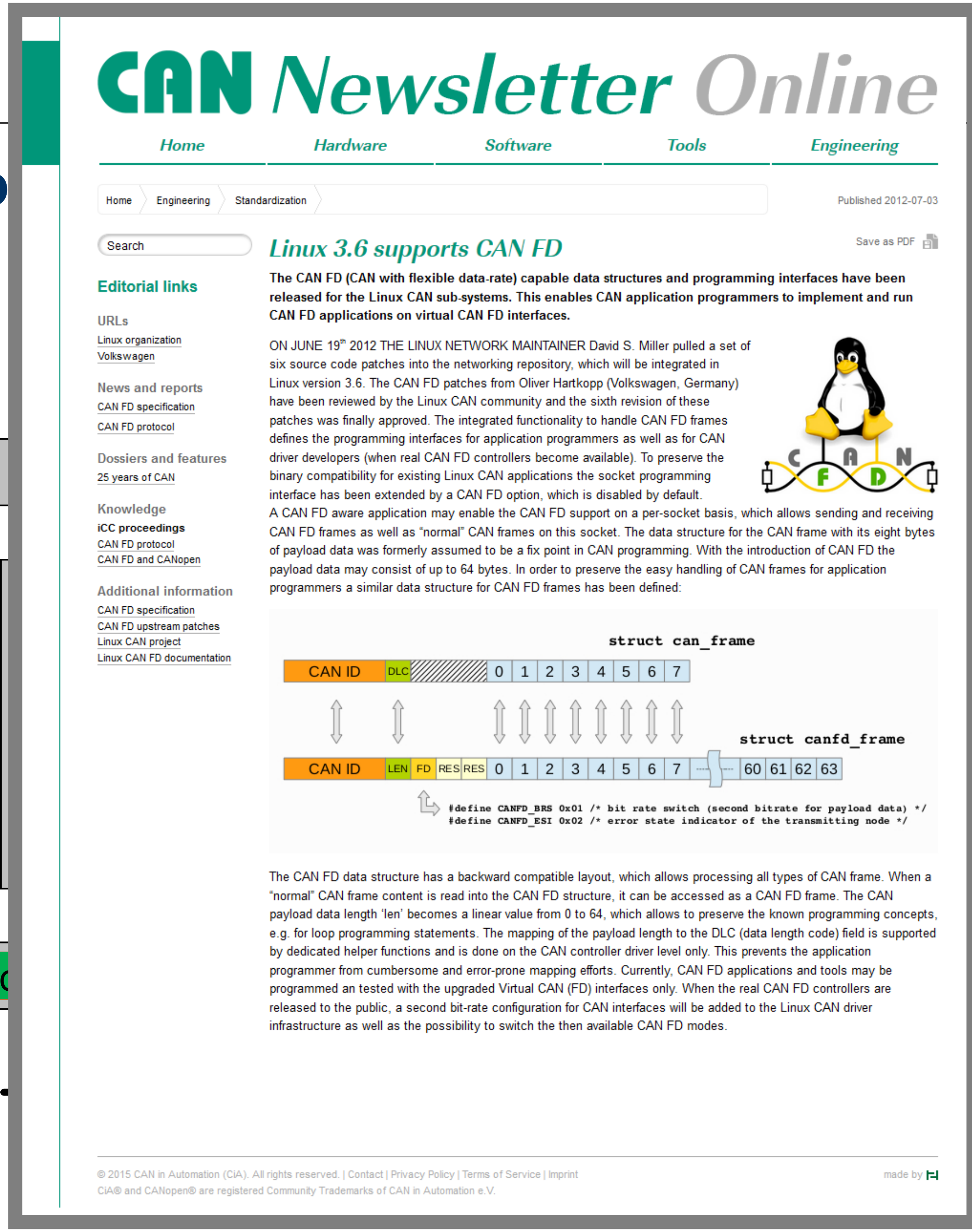
# CAN FD support since Linux 3.6 (Summer 2012)




**High impact**
**Indirect inpact**
**Supported**

# CAN FD sup



CAN Newsletter Online

Home | Hardware | Software | Tools | Engineering

Home > Engineering > Standardization

Published 2012-07-03

Search

**Editorial links**

URLs
Linux organization
Volkswagen

News and reports
CAN FD specification
CAN FD protocol

Dossiers and features
25 years of CAN

Knowledge
**iCC proceedings**
CAN FD protocol
CAN FD and CANopen

Additional information
CAN FD specification
CAN FD upstream patches
Linux CAN project
Linux CAN FD documentation

## *Linux 3.6 supports CAN FD*

Save as PDF

**The CAN FD (CAN with flexible data-rate) capable data structures and programming interfaces have been released for the Linux CAN sub-systems. This enables CAN application programmers to implement and run CAN FD applications on virtual CAN FD interfaces.**

ON JUNE 19th 2012 THE LINUX NETWORK MAINTAINER David S. Miller pulled a set of six source code patches into the networking repository, which will be integrated in Linux version 3.6. The CAN FD patches from Oliver Hartkopp (Volkswagen, Germany) have been reviewed by the Linux CAN community and the sixth revision of these patches was finally approved. The integrated functionality to handle CAN FD frames defines the programming interfaces for application programmers as well as for CAN driver developers (when real CAN FD controllers become available). To preserve the binary compatibility for existing Linux CAN applications the socket programming interface has been extended by a CAN FD option, which is disabled by default.
A CAN FD aware application may enable the CAN FD support on a per-socket basis, which allows sending and receiving CAN FD frames as well as "normal" CAN frames on this socket. The data structure for the CAN frame with its eight bytes of payload data was formerly assumed to be a fix point in CAN programming. With the introduction of CAN FD the payload data may consist of up to 64 bytes. In order to preserve the easy handling of CAN frames for application programmers a similar data structure for CAN FD frames has been defined:

struct can_frame

CAN ID | DLC | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

struct canfd_frame

CAN ID | LEN | FD | RES | RES | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 60 | 61 | 62 | 63

```
#define CANFD_BRS 0x01 /* bit rate switch (second bitrate for payload data) */
#define CANFD_ESI 0x02 /* error state indicator of the transmitting node */
```

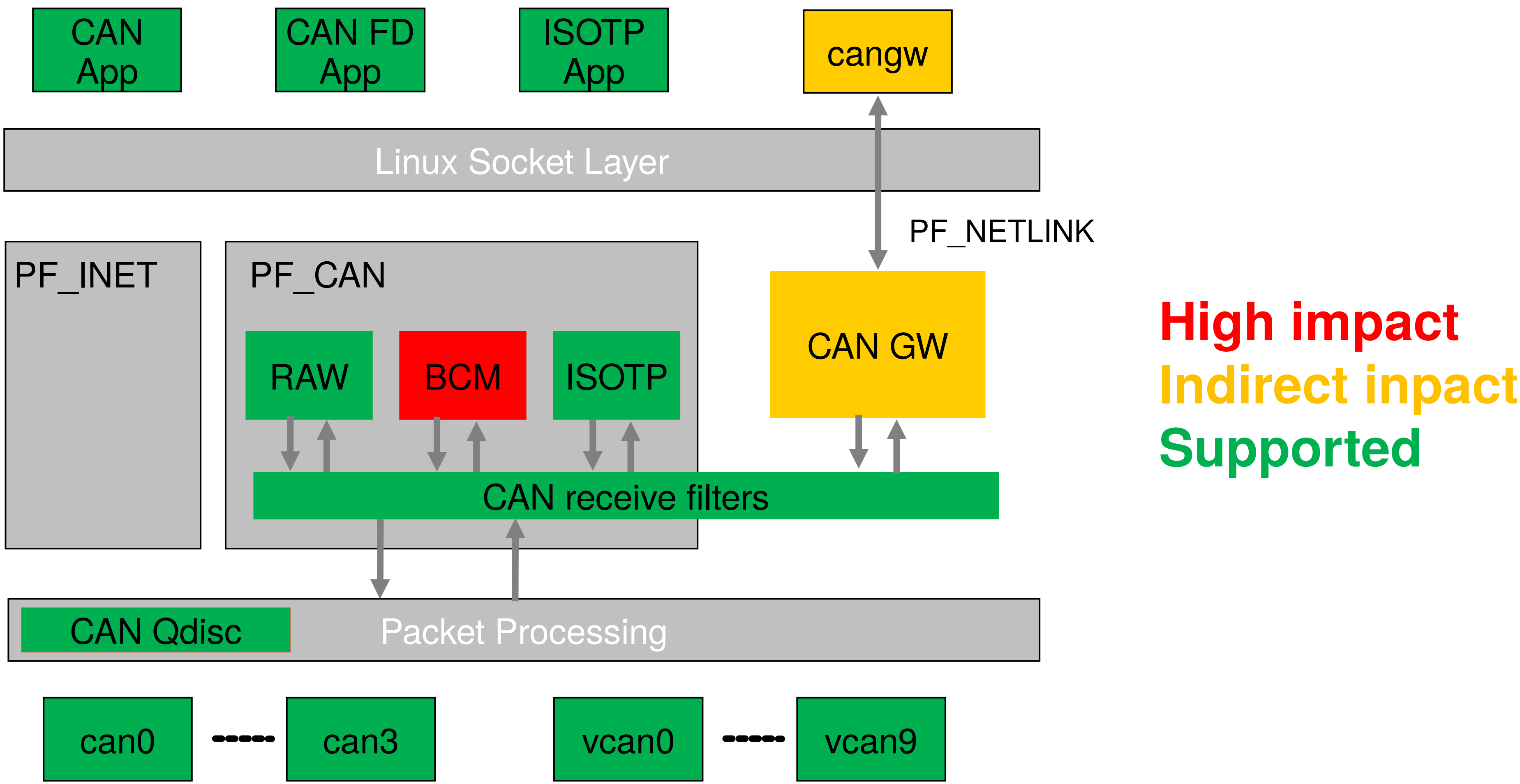
The CAN FD data structure has a backward compatible layout, which allows processing all types of CAN frame. When a "normal" CAN frame content is read into the CAN FD structure, it can be accessed as a CAN FD frame. The CAN payload data length 'len' becomes a linear value from 0 to 64, which allows to preserve the known programming concepts, e.g. for loop programming statements. The mapping of the payload length to the DLC (data length code) field is supported by dedicated helper functions and is done on the CAN controller driver level only. This prevents the application programmer from cumbersome and error-prone mapping efforts. Currently, CAN FD applications and tools may be programmed an tested with the upgraded Virtual CAN (FD) interfaces only. When the real CAN FD controllers are released to the public, a second bit-rate configuration for CAN interfaces will be added to the Linux CAN driver infrastructure as well as the possibility to switch the then available CAN FD modes.



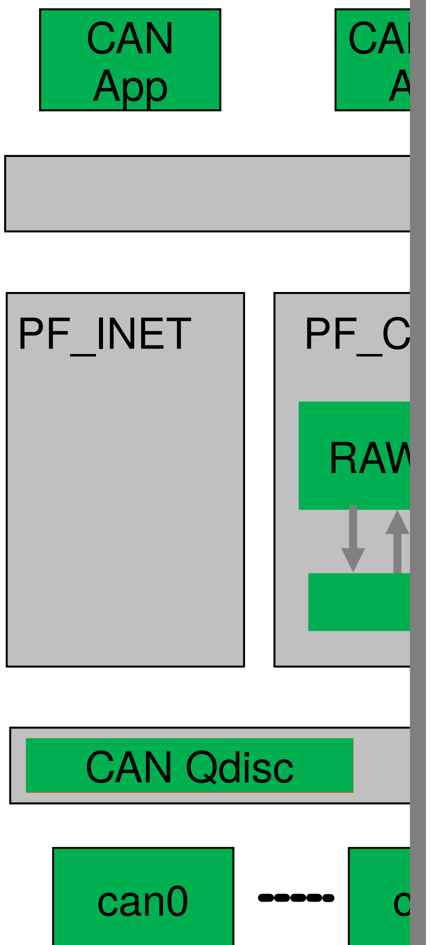
made by

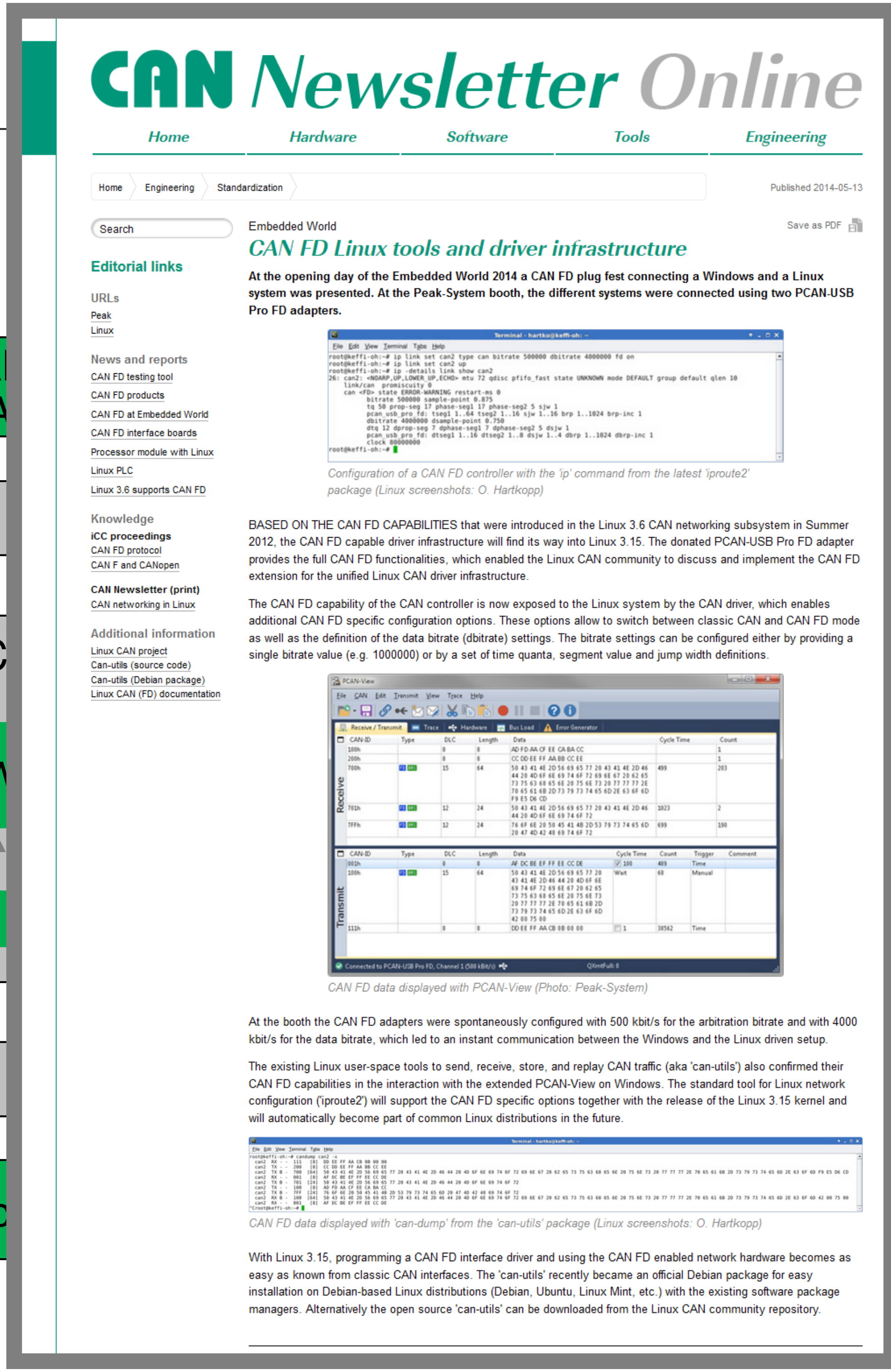
High impact
Indirect inpact
Supported

# Current CAN FD support since Linux 3.15 (Embedded W 2014 )



# Current CAN FD … bedded W 2014 )



CAN Newsletter Online

Home | Hardware | Software | Tools | Engineering

Home › Engineering › Standardization

Published 2014-05-13

Search

**Editorial links**

URLs
Peak
Linux

News and reports
CAN FD testing tool
CAN FD products
CAN FD at Embedded World
CAN FD interface boards
Processor module with Linux
Linux PLC
Linux 3.6 supports CAN FD

Knowledge
**iCC proceedings**
CAN FD protocol
CAN F and CANopen

**CAN Newsletter (print)**
CAN networking in Linux

Additional information
Linux CAN project
Can-utils (source code)
Can-utils (Debian package)
Linux CAN (FD) documentation

Embedded World

Save as PDF

## *CAN FD Linux tools and driver infrastructure*

**At the opening day of the Embedded World 2014 a CAN FD plug fest connecting a Windows and a Linux system was presented. At the Peak-System booth, the different systems were connected using two PCAN-USB Pro FD adapters.**

*Configuration of a CAN FD controller with the 'ip' command from the latest 'iproute2' package (Linux screenshots: O. Hartkopp)*

BASED ON THE CAN FD CAPABILITIES that were introduced in the Linux 3.6 CAN networking subsystem in Summer 2012, the CAN FD capable driver infrastructure will find its way into Linux 3.15. The donated PCAN-USB Pro FD adapter provides the full CAN FD functionalities, which enabled the Linux CAN community to discuss and implement the CAN FD extension for the unified Linux CAN driver infrastructure.

The CAN FD capability of the CAN controller is now exposed to the Linux system by the CAN driver, which enables additional CAN FD specific configuration options. These options allow to switch between classic CAN and CAN FD mode as well as the definition of the data bitrate (dbitrate) settings. The bitrate settings can be configured either by providing a single bitrate value (e.g. 1000000) or by a set of time quanta, segment value and jump width definitions.

*CAN FD data displayed with PCAN-View (Photo: Peak-System)*

At the booth the CAN FD adapters were spontaneously configured with 500 kbit/s for the arbitration bitrate and with 4000 kbit/s for the data bitrate, which led to an instant communication between the Windows and the Linux driven setup.

The existing Linux user-space tools to send, receive, store, and replay CAN traffic (aka 'can-utils') also confirmed their CAN FD capabilities in the interaction with the extended PCAN-View on Windows. The standard tool for Linux network configuration ('iproute2') will support the CAN FD specific options together with the release of the Linux 3.15 kernel and will automatically become part of common Linux distributions in the future.

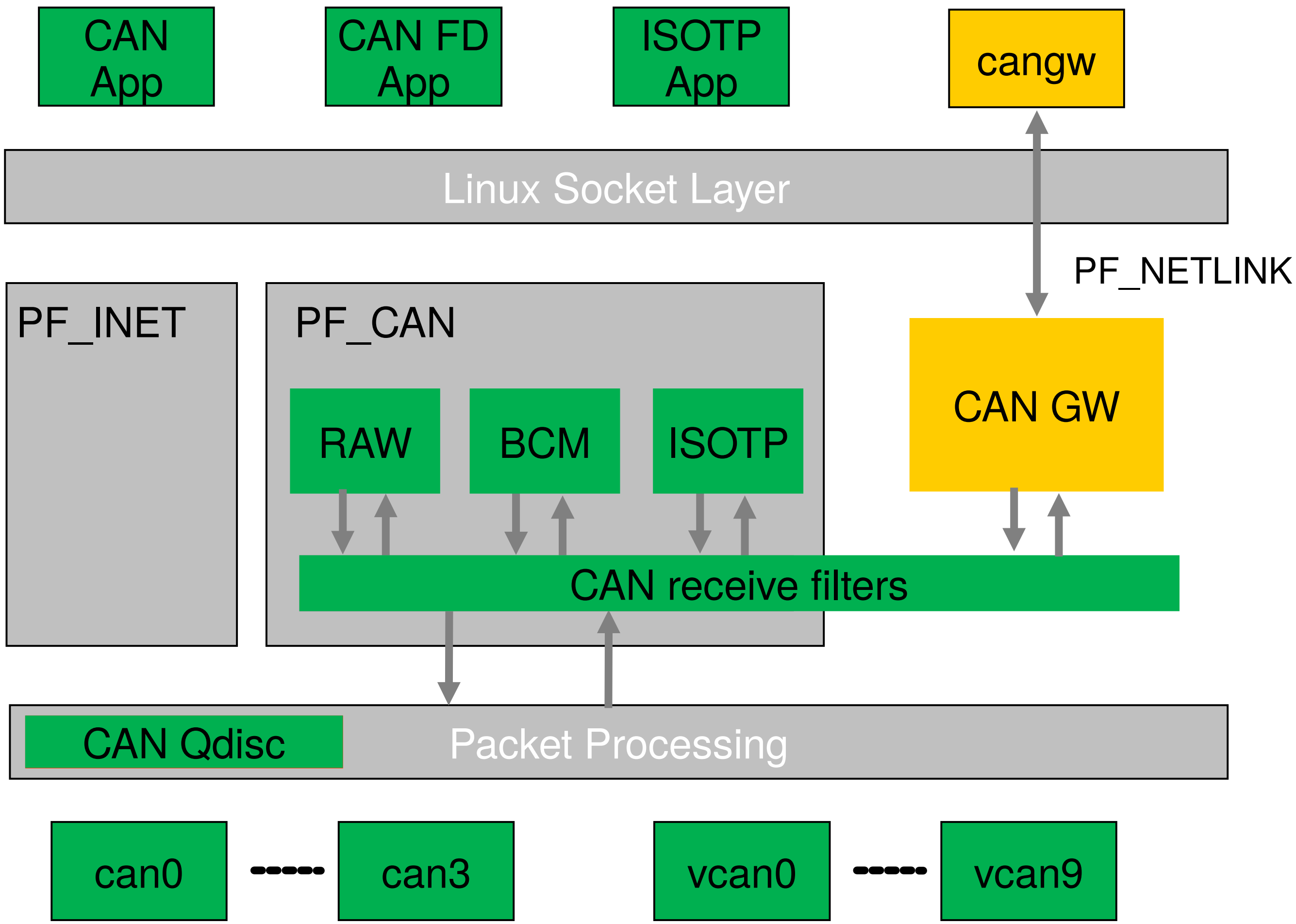
*CAN FD data displayed with 'can-dump' from the 'can-utils' package (Linux screenshots: O. Hartkopp)*

With Linux 3.15, programming a CAN FD interface driver and using the CAN FD enabled network hardware becomes as easy as known from classic CAN interfaces. The 'can-utils' recently became an official Debian package for easy installation on Debian-based Linux distributions (Debian, Ubuntu, Linux Mint, etc.) with the existing software package managers. Alternatively the open source 'can-utils' can be downloaded from the Linux CAN community repository.

**High impact**
**Indirect inpact**
**Supported**

Oliver Hartkopp

# Current CAN FD support since Linux 4.8 (October 2016)




**High impact**
**Indirect inpact**
**Supported**

## Outlook & new fancy stuff

- New drivers: M_CAN for IP cores v3.1+, PEAK PCI FD, Microchip CAN Bus Analyzer with fixed bitrate settings & termination
- Mainlining of ISO 15765-2:2016 and J1939 implementations
  https://github.com/hartkopp/can-isotp-modules https://github.com/kurt-vd/test-can-j1939
- CAN FD support for CAN_GW (any use-cases out there?)
- Network Namespaces Support for cgroups, LXC, Docker
  - RFC Patch [v2] from Mario Kicherer 2017-02-21
    http://marc.info/?l=linux-can&m=148767639224547&w=2
  - Tested with virtual & real CAN interfaces
    http://marc.info/?l=linux-can&m=149046502301622&w=2
  - But CAN_BCM / CAN_ISOTP support currently missing
  - CAN_GW suggested for inter namespace communication
    http://marc.info/?l=linux-can&m=149054987117099&w=2

# Many thanks!

```
$> cat linux/MAINTAINERS | grep -B 2 -A 14 Hartkopp

CAN NETWORK LAYER
M:      Oliver Hartkopp <socketcan@hartkopp.net>
M:      Marc Kleine-Budde <mkl@pengutronix.de>
L:      linux-can@vger.kernel.org
W:      https://github.com/linux-can
T:      git git://git.kernel.org/pub/scm/linux/kernel/gut/mkl/linux-can.git
T:      git git://git.kernel.org/pub/scm/linux/kernel/gut/mkl/linux-can-next.git
S:      Maintained
F:      Documentation/networking/can.txt
F:      net/can/
F:      include/linux/can/core.h
F:      include/uapi/linux/can.h
F:      include/uapi/linux/can/bcm.h
F:      include/uapi/linux/can/raw.h
F:      include/uapi/linux/can/gw.h

$> _
```