SocketCan and J1939

Oleksij Rempel – ore@pengutronix.de
Marc Kleine-Budde – mkl@pengutronix.de

My assumption about you

- What is J1939 and do I actually need it?
- I use this protocol as user space stack in some product and it works for me. Why should I care about kernel stack?
- Just skip 1. and 2., and tell me how can I use the kernel stack?!

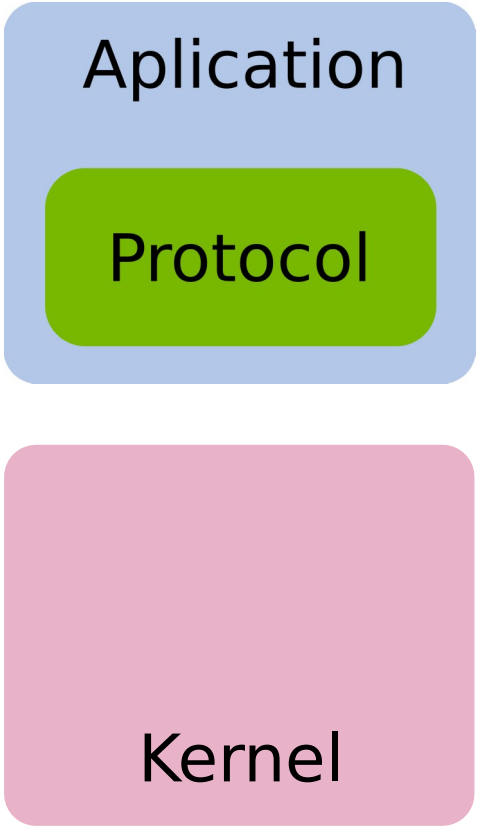
... spend some words on CAN

- something different than Ethernet
- 2 wire cable
- speed up to 1 Mbit (only)
- 8 bytes per frame

CAN: every thing is a broadcast

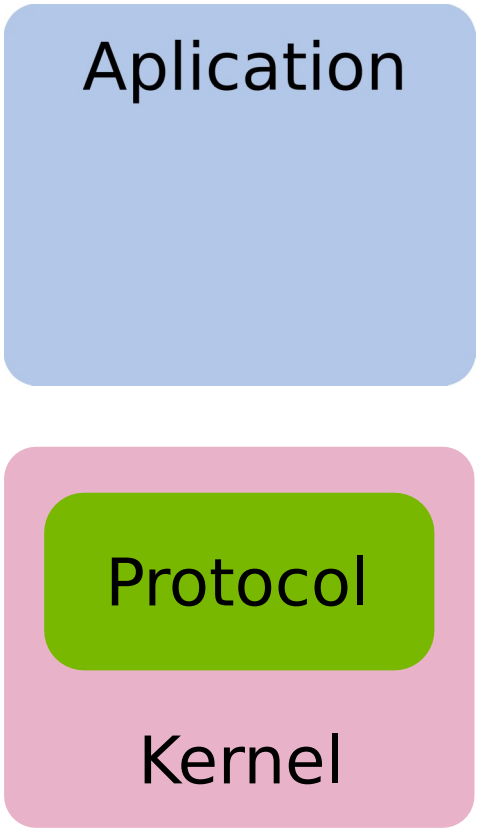
- 11 bit or 29 bit address (CAN-ID)
- priorisation of CAN frames by CAN-ID
- CSMCA (Carrier Sense Multiple Collision Avoidance)
- CAN frames are broadcasted

State of CAN infrastructure before 2013



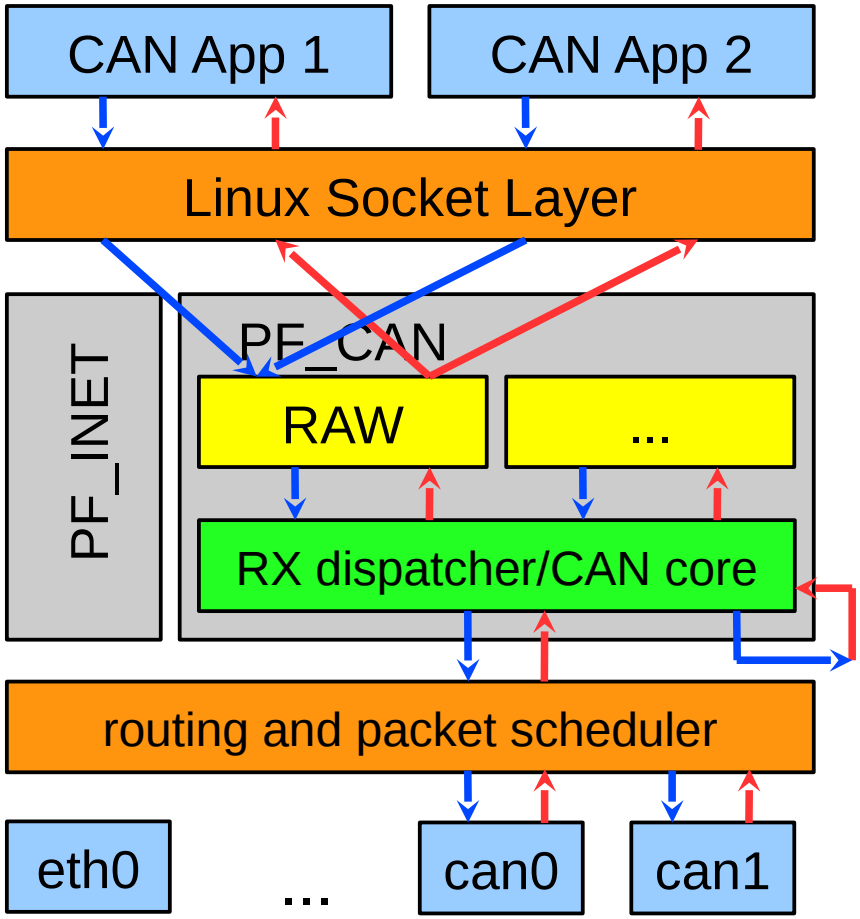
- Different kernel and user-space drivers
- No compatibility
- No unified tooling
- Bad testing coverage

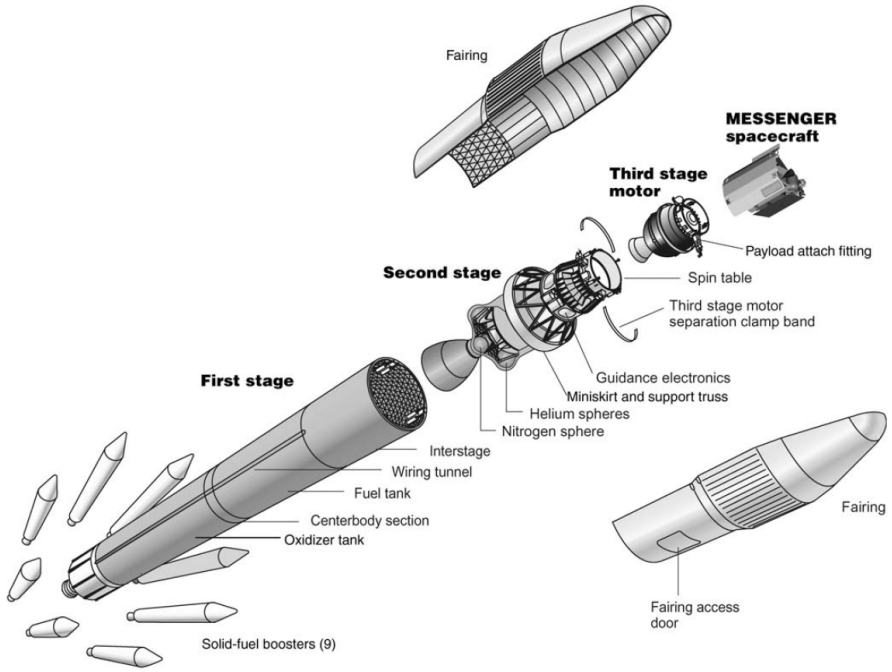
SocketCAN now



- Hardware abstraction layer
- One socket interface for all applications
- SoC vendors do mainline linux CAN drivers

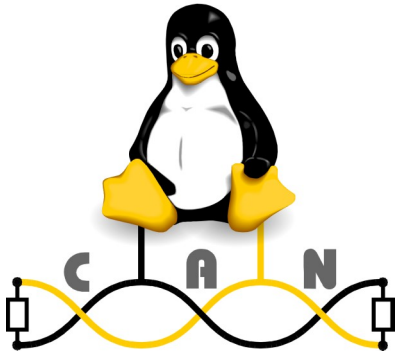
SocketCAN isn‘t Rocket Science!





Delta II rocket with MESSENGER

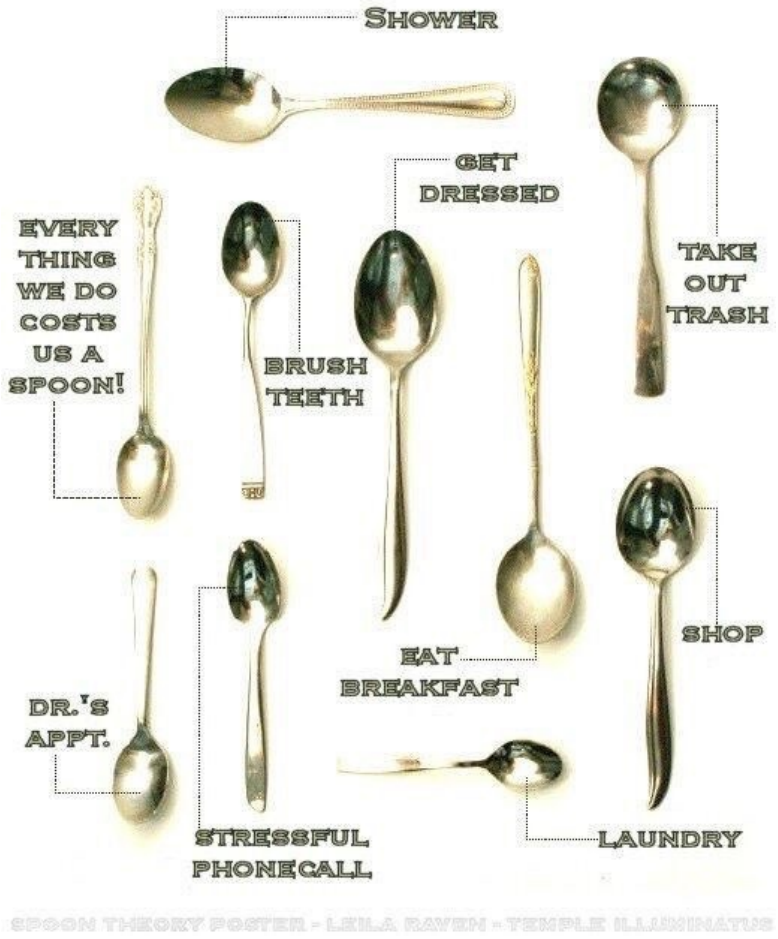
SocketCAN: infrastructure



- CAN-utils
- CAN-tests
- Wireshark
- ...?

CAN: 0...8 bytes per CAN frame (only)

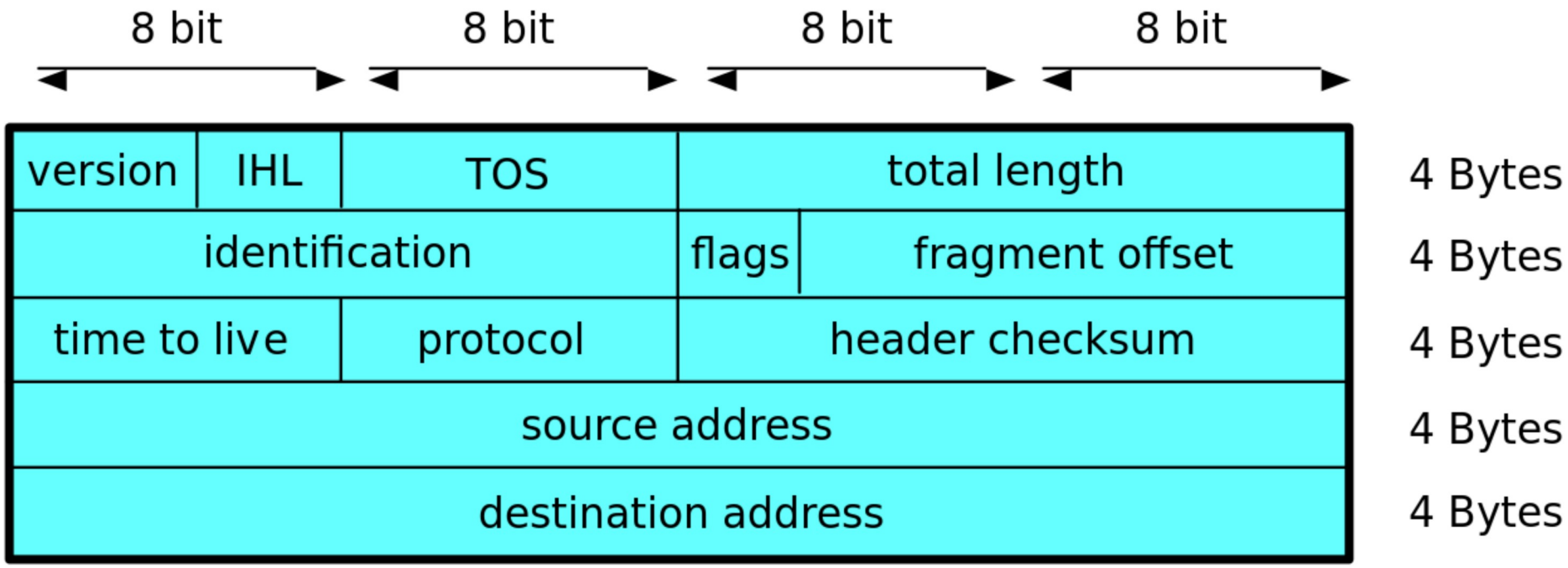
A spoon of bytes!!!



Main motivation for J1939

- CAN bus is slow and packages are very small.

For example IP Header is 20 Bytes...



For example IP Header is 20 Bytes...

What is SAE J1939?

Recommendation for:

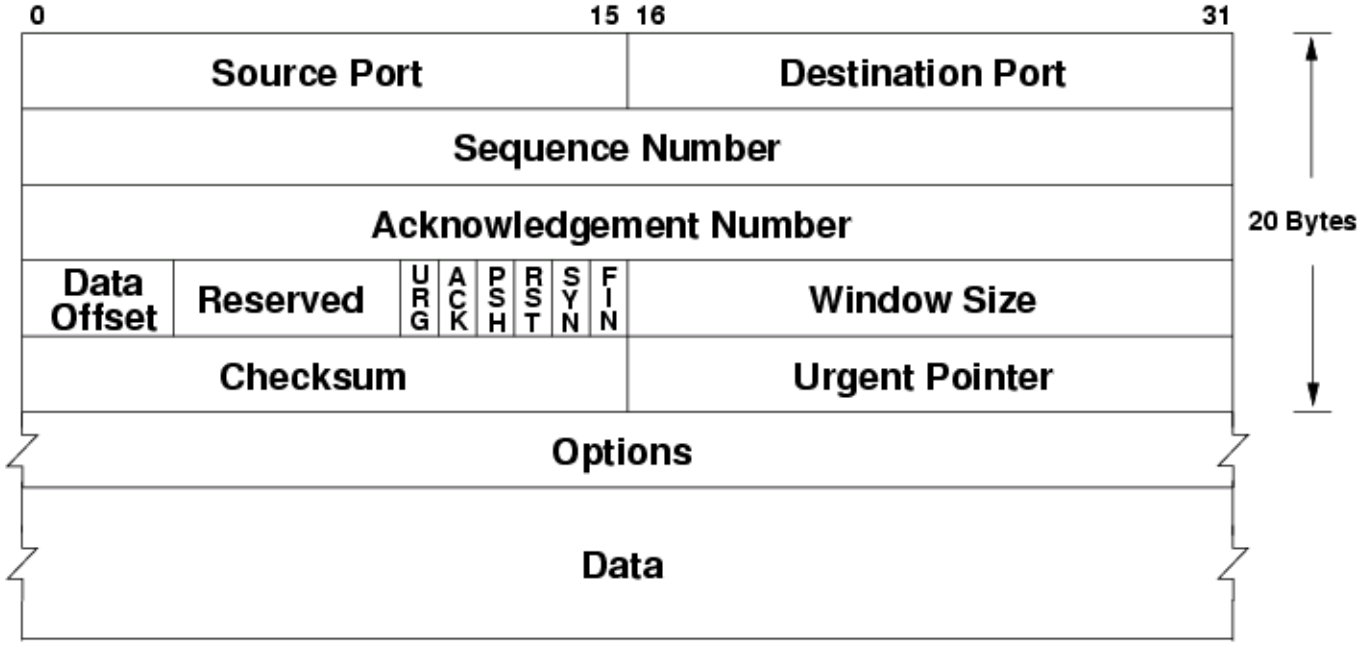
- Physical Layer
- Defines PGNs (Parameter Group Number)
- PGN identifies a message's function and meaning of associated data

What is SAE J1939?

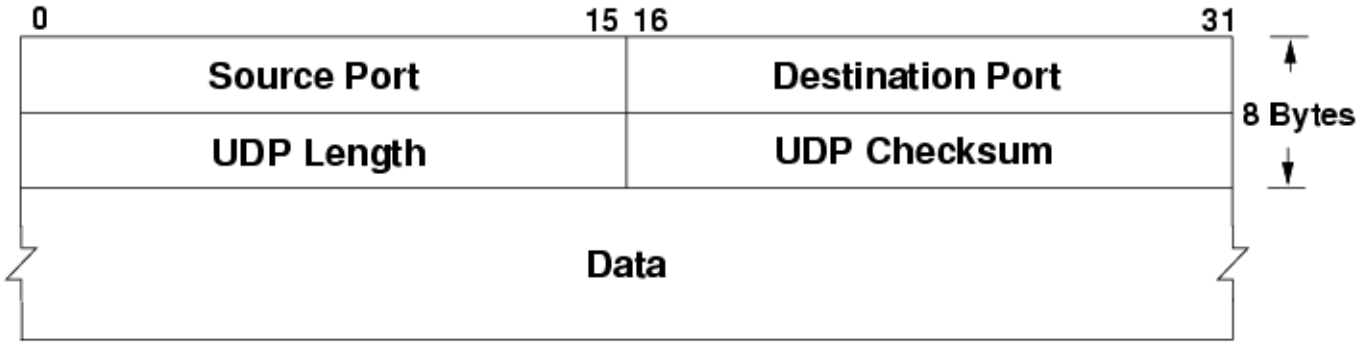
One spoon it is. .
@officialdoyoueven 👈

- Transport Protocol / Extended Transport Protocol
- Reliable send/receive large amounts of data
- Transport Protocol = 1792 bytes, Extended Transport Protocol ~ 112 MiB

J1939 TP is like TCP (20 Byte header !!!)



How about UDP?! (8 byte header)



SocketCAN with J1939 stack

- Same situation as with Linux CAN

  before SocketCAN

- Different user space and kernel implementations

Why kernel stack: CPU load and timings

- busy CAN bus about 2000 pps (or more?)
- (Spoons) per second * socket
- relative relaxed timing requirements in general
- ...but not on a loaded single core 400 MHz ARMv5 (imx28)

Different user space implementations

- 1. Multiple processes with userspace stack (J1939 daemon)
- 2. One library used by different applications
- 3. All in one. One application with J1939 stack and many threads.
- 4. Different J1939 stack variants per developer

1. Multiple processes with userspace stack

- one J1939 process running to parse J1939 traffic and communicate with multiple applications
- long round trip times:
- [Kernel - CAN_RAW socket] → J1939 stack → pipes/unix domain sockets/tcp → application

2. One library used by different applications

- the load on the CAN bus will be increased as well. For example: more Address Claiming requests.
- Increased memory usage. For example: same TP or ETP should be reconstructed separately multiple times on same system.

3. All in one

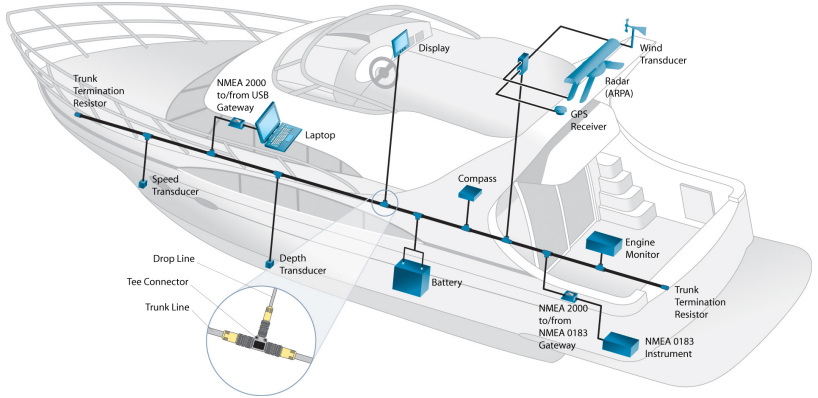
- no isolation of processes
- malfunction/security problem in one thread will affect other applications/threads

4. Different J1939 variants per developer

- Many end devices are made by chain of different suppliers.
- Each chain part is using own software and great, special version of J1939 stack.

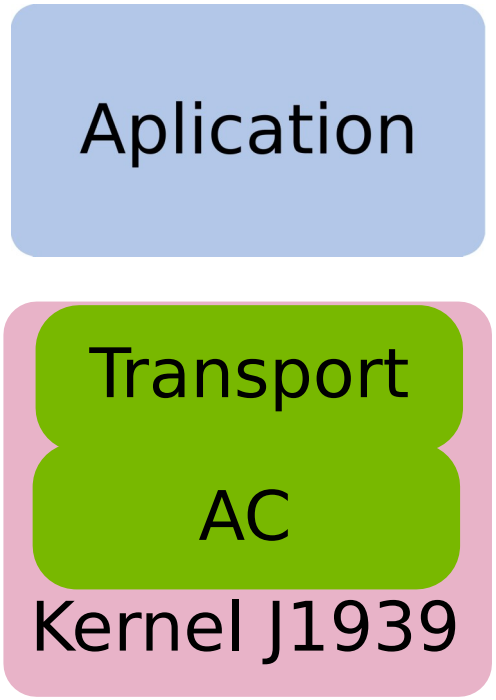
SAE J1939 Linux Kernel Implementation

- Should be able to cover:
  - SAE J1939
  - IsoBUS
  - NMEA2000
  - MilCAN A



SAE J1939 Linux Kernel Implementation

- Simple programming model
  - Well known socket interface.
- Better performance
- Kernel don't cares about data or PGN except of: AC and (E)TP



How to use kernel SAE J1939 stack?

- Jacd and jcat: https://github.com/linux-can/can-utils
- Kernel: Documentation/networking/j1939.rst

Challanges

- MTU: ~112 MiB (solved)
- Proper way to export address claiming cache to the userspace
- Quirky buses.
- Test automation (follow osmocom testing experience?)

Thank you!

Questions?