Spin42

Reverse-engineering CAN and building ECUs using Elixir

Thibault Poncelet

Background

- After 7 years building an Open Banking aggregator in Elixir, We wanted to learn something new and test if Elixir could be a good fit for an automotive use case.
- In 2024, we have built an "Open Vehicle Control System": An Elixir based devkit, allowing to upgrade any vehicle.
- "Upgrade" meaning:
  - Engine swap
  - EV retrofit
  - Infotainment system
  - Autonomous driving

EV retrofit

2007 VW Polo Diesel

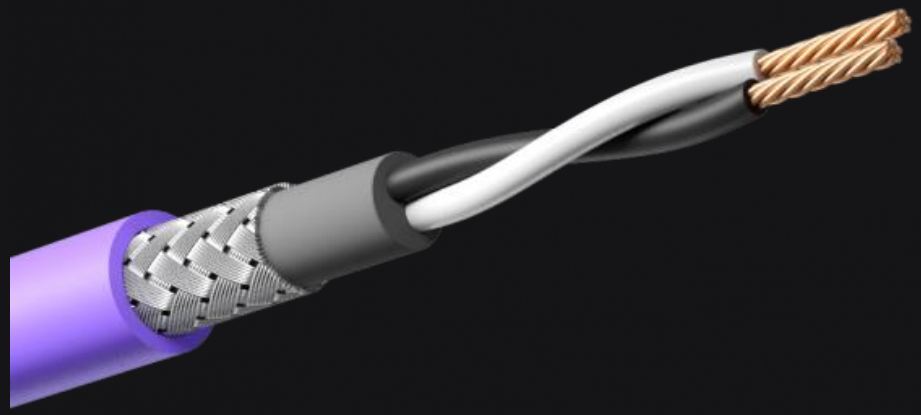
+

2013 Nissan Leaf EM57 motor

Today's focus:

How to display the Nissan motor RPMs on the Polo's dashboard?

CAN communication bus



CAN bus (Controller Area Network) is where all car components talk together

Standard protocol in automotive, aeronautics, industry, ...

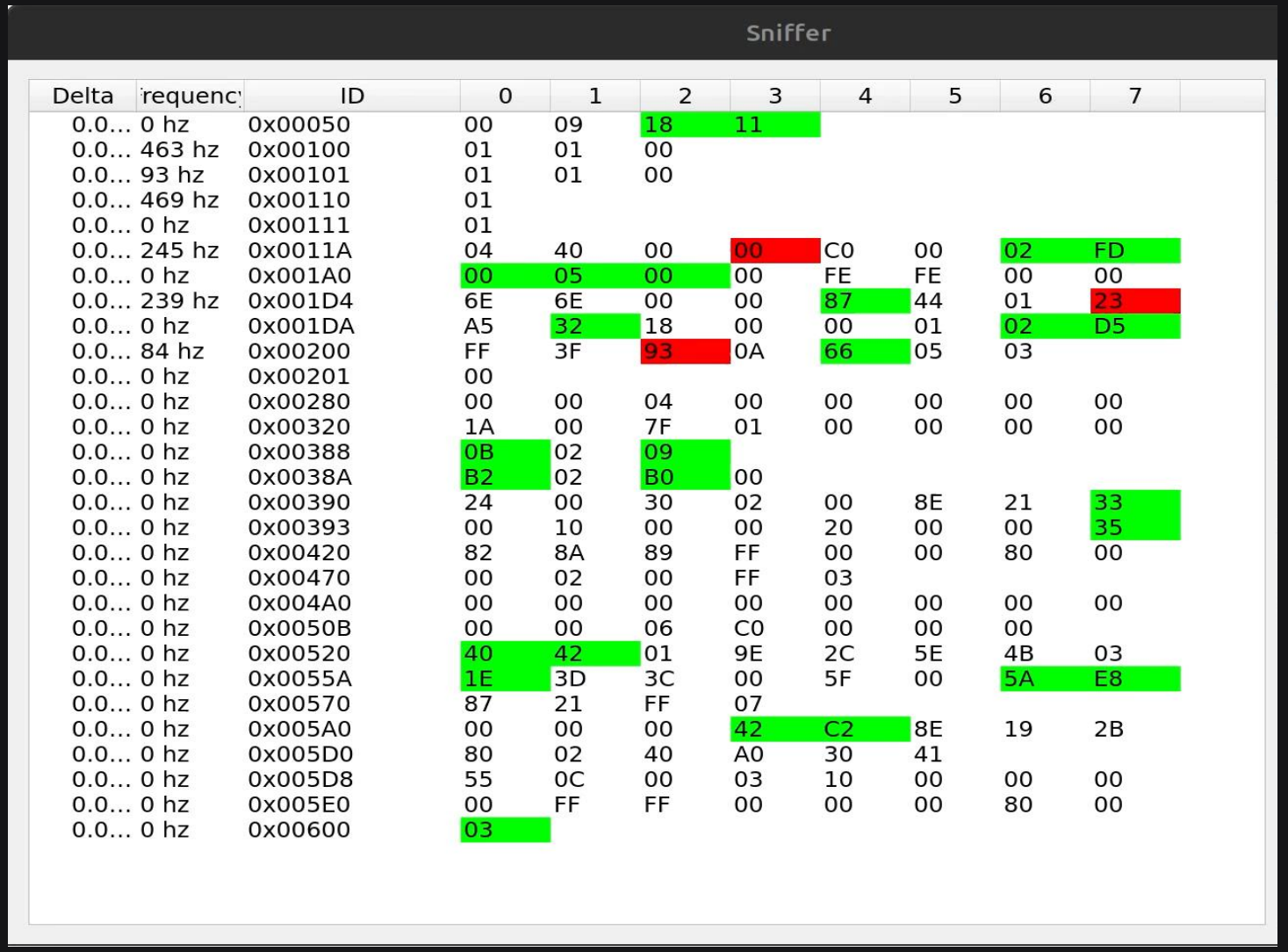
Built-in support in the Linux kernel: libsocketcan

Although CAN is standard, the messages you transfer through it are not

CAN communication bus

Data exchanged on CAN is represented as a series of bytes

A frame with a specific ID is published periodically on the bus

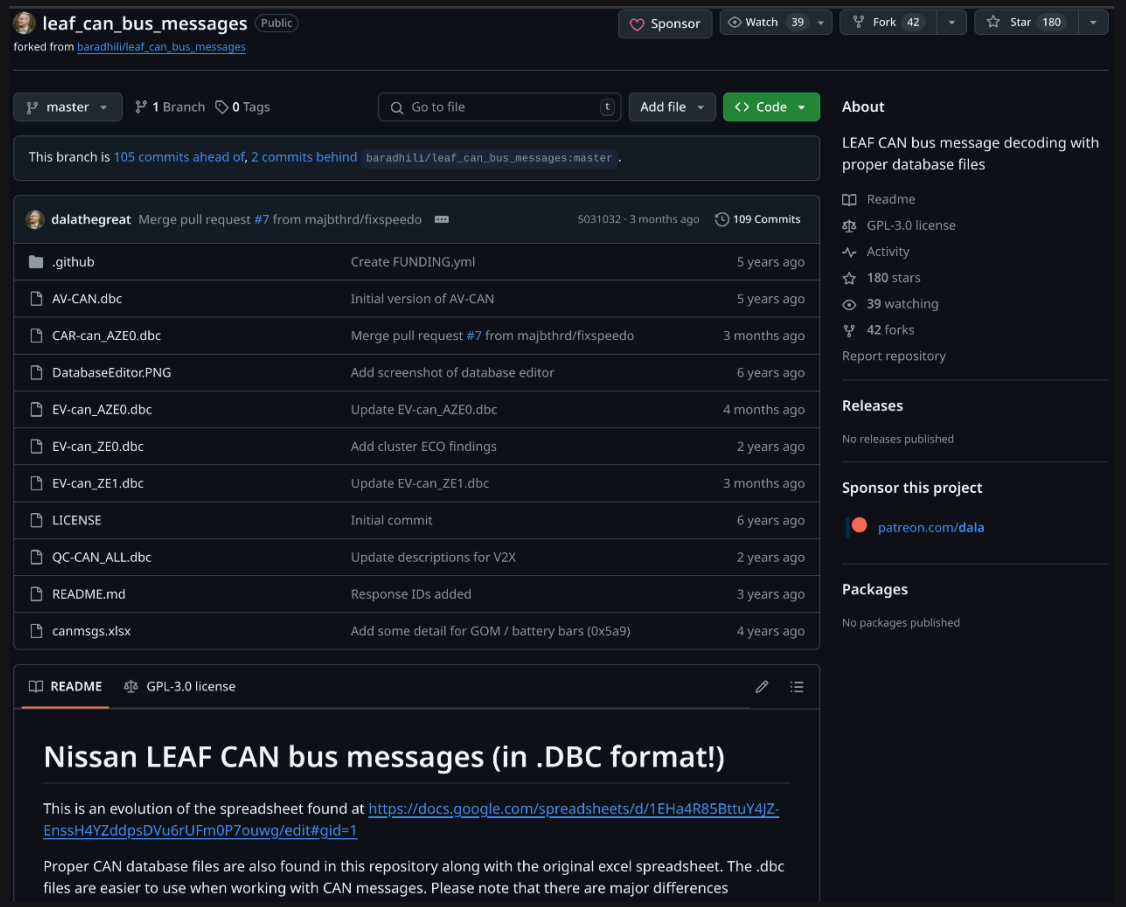


CAN bus tools

- USB2CAN: An USB-to-CAN device
- SavvyCAN: a CAN bus reverse engineering and capture GUI.
- socketcand: a daemon that provides access to CAN interfaces on a machine via a network interface.

Step 1
Reverse the motor RPM frame

- Thankfully, there is a large community of Nissan Leaf tinkerers that already reverse engineered the Leaf frames and published it on the internet.



https://github.com/dalathegreat/leaf_can_bus_messages

0x1DA

Inverter Status Frame

Signal Parameters

Name: MG_OutputRevolution

Start Bit:

BYTES	BITS -> 7	6	5	4	3	2	1	0
0	7	6	5	4	3	2	1	0
1	15	14	13	12	11	10	9	8
2	23 MG_ErrorCodes	22	21	20	19	18 MG_EffectiveTorque	17	16
3	31	30	29	28	27	26	25 MG_ErrorCodes	24
4	39 MG_OutputRevolution	38	37	36	35	34	33	32
5	47	46	45	44	43	42	41	40
6	55	54	53	52	51	50	49 MG_Clock	48
7	63 CRC_1DA	62	61	60	59	58	57	56

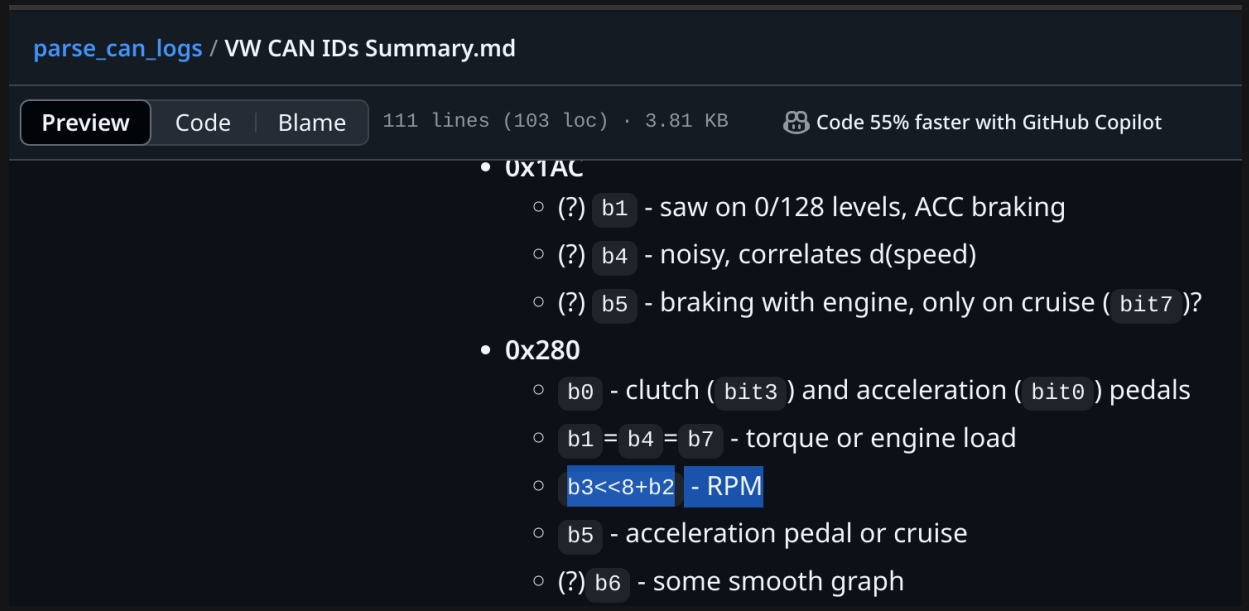
Bit Length:	15
Byte Order	LSB First (Little Endian)
Type:	UNSIGNED INTEGER
Scale:	1
Bias:	0
Min Value:	-16382
Max Value:	16382
Units Name:	rpm
Receiving Node:	Vector__XXX
Multiplexing	None / Multiplexed / Multiplexor / Extended
Multiplex Low Value	0
Multiplex High Value	0
Multiplex Parent	
Comment:	

Value Table:

Step 2

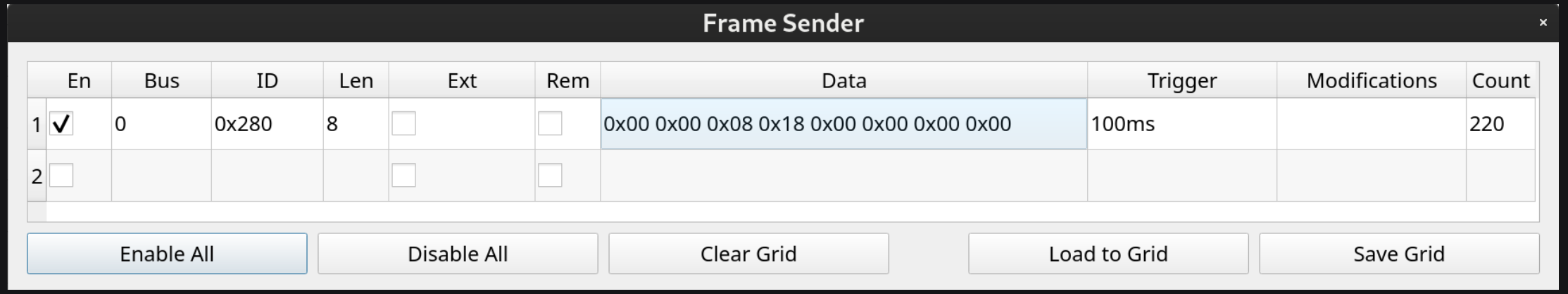
Reversing the Polo RPM Frame

- Pro tip: Do not remove the original engine before recording a complete CAN trace
- We could not find a DBC File for the 2007 Polo online
- VW is using similar frames on multiple cars of the same generation.



https://github.com/v-ivanyshyn/parse_can_logs

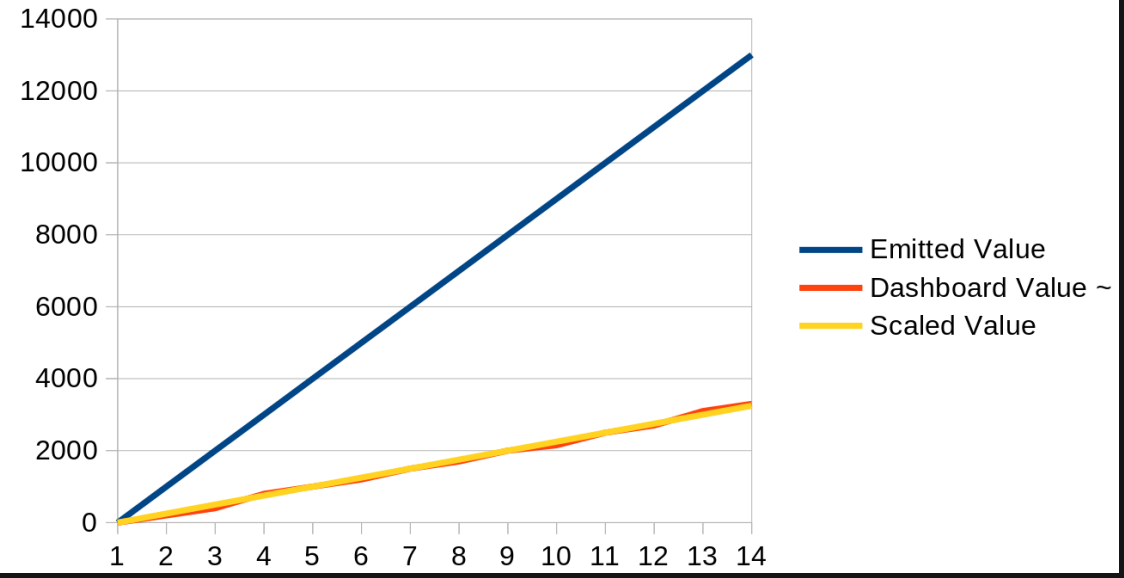
Emitting a custom frame with SavvyCAN



- Based on these findings, we tried to emit the 0x280 frame on the VW CAN bus at various frequencies.
- Some values of bytes 2 and 3 were actuating the RPM needle at 10Hz!
- Still need to translate real RPMs into the VW format
- physical value = offset + scale * raw_decimal_value

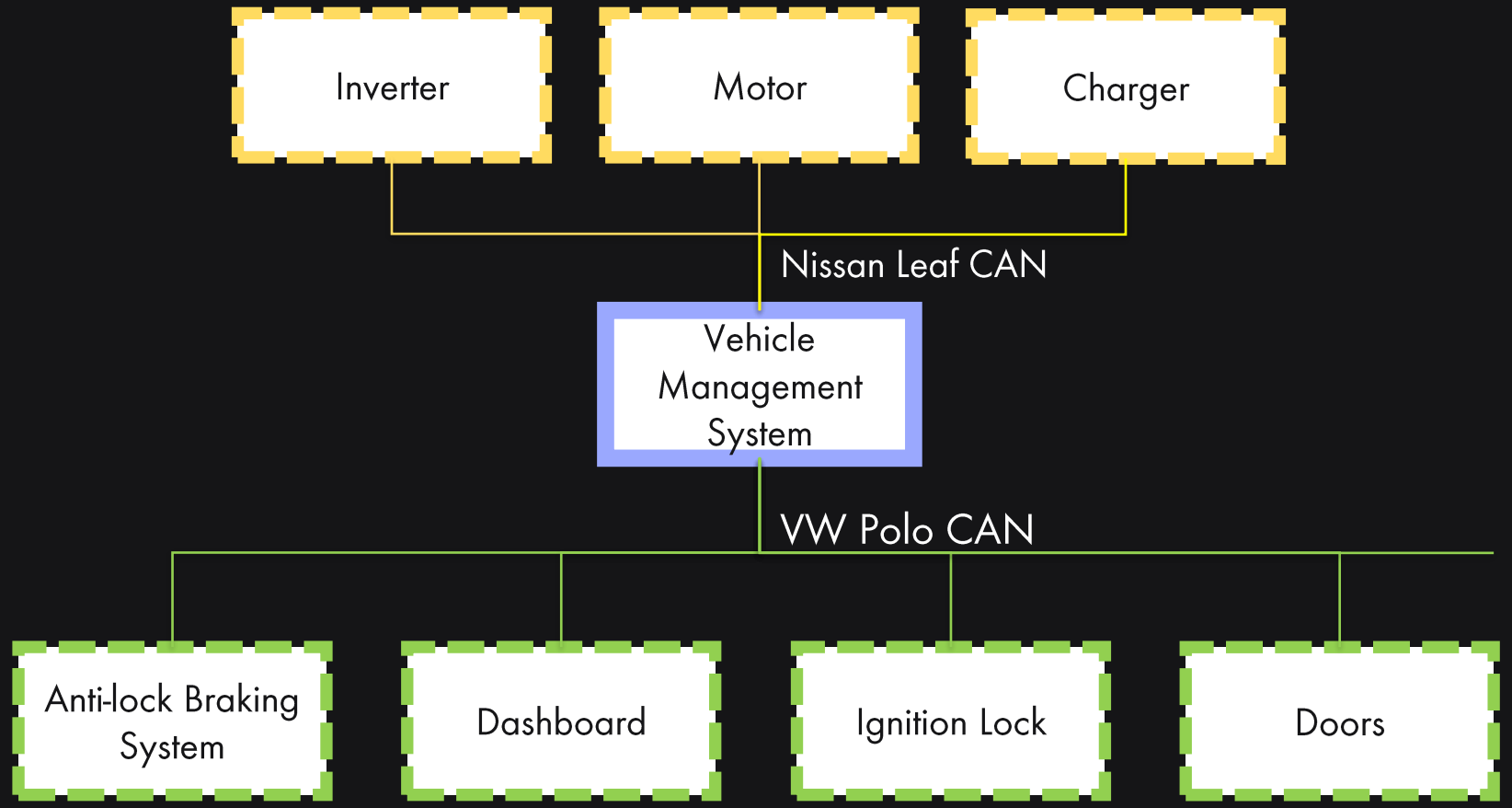
Reversing the RPM scale value

	A	B	C	D	E
1	Emitted Value	Dashboard Value ~	Scaled Value	Scale	
2	0	0	0	0.25	
3	1000	200	250		
4	2000	400	500		
5	3000	800	750		
6	4000	1000	1000		
7	5000	1200	1250		
8	6000	1500	1500		
9	7000	1700	1750		
10	8000	2000	2000		
11	9000	2150	2250		
12	10000	2500	2500		
13	11000	2700	2750		
14	12000	3100	3000		
15	13000	3300	3250		



Step 3

Concrete implementation using Elixir



CAN on Elixir 

- Elixir is a dynamic, functional language running on the BEAM (aka the Erlang VM).
- The BEAM has been designed to build massively scalable soft real-time systems.
- Since Erlang was originally designed by Ericsson to build telephony switches, all the binary primitives are available out-of-the-box.
- Pattern matching is also helping a lot in parsing CAN frames.

Receiving and parsing one frame

```elixir
with {:ok, socket}  <- :socket.open(29, :raw, 1),
     {:ok, ifindex} <- :socket.ioctl(socket, :gifindex, "can0" |> String.to_charlist()),
     address        <- <<0::size(16)-little, ifindex::size(32)-little, 0::size(32), 0::size(32), 0::size(64)>>,
     :ok            <- :socket.bind(socket, %{:family => 29, :addr => address})
do
  {:ok, raw_frame} = :socket.recv(socket)

  <<id::little-integer-size(16),
    _::binary-size(2),
    byte_number::little-integer-size(8),
    _::binary-size(3),
    raw_data::binary-size(byte_number),
    _::binary>> = raw_frame

  case id do
    0x1DA ->
      <<_::bitstring-size(32), rpm::big-signed-integer-size(16), _::bitstring>> = raw_data
      IO.inspect(rpm)
  end
else
  {:error, error} -> {:error, error}
end
```

Introducing cantastic

- An Elixir library doing all the heavy-lifting for sending and receiving CAN frames.
- Clean YAML DSL to describe your entire CAN Network
- Automatically spawns:
  - 1 receiver process (GenServer) per CAN network
  - 1 emitter process (GenServer) per emitted frame

https://github.com/open-vehicle-control-system/cantastic

Receiving the Leaf RPM value

```
---
can_networks:
  leaf_drive:
    bitrate: 500000
    received_frames:
      name: inverter_status
      id: 0x1DA
      frequency: 10
      signals:
        - name: rotations_per_minute
          kind: integer
          endianness: big
          sign: signed
          value_start: 32
          value_length: 16
          unit: rpm
  polo_drive: ···
```

```elixir
defmodule Leaf.Inverter do
  use GenServer
  ...
  def init(_) do
    Cantastic.Receiver.subscribe(self(), :leaf_drive, "inverter_status")
    {:ok, %{}}
  end

  def handle_info({:handle_frame,
    %Cantastic.Frame{name: "inverter_status", signals: signals}},
    state)
  do
    %{
      "rotations_per_minute" => %Cantastic.Signal{value: rotation_per_minute}
    } = signals

    Polo.Dashboard.set_rotation_per_minute(rotation_per_minute)
    {:noreply, state}
  end
  ...
end
```

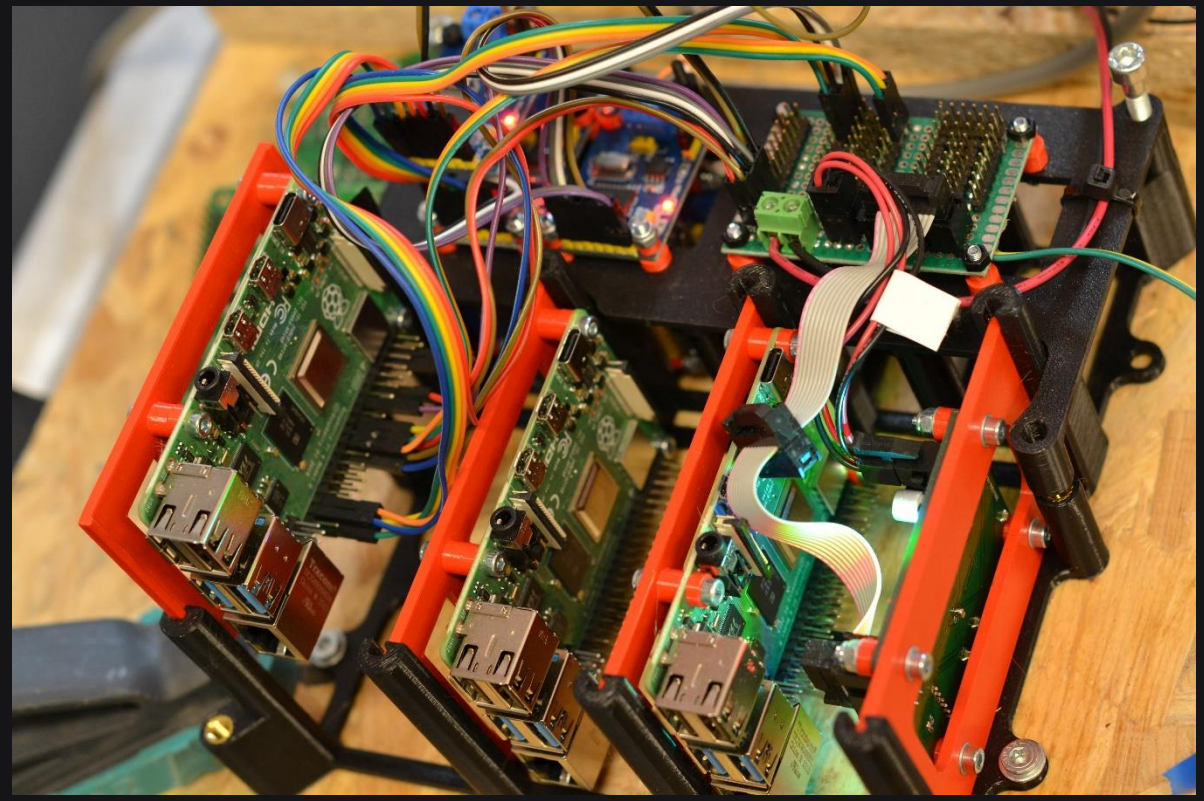
Emitting the Polo RPM value

```
---
can_networks:
  leaf_drive: ...
  polo_drive:
    bitrate: 500000
    emitted_frames:
      name: engine_status
      id: 0x280
      frequency: 100
      signals:
        - name: rotations_per_minute
          kind: integer
          unit: rpm
          value_start: 16
          value_length: 16
          scale: "0.25"
```

```
defmodule Polo.Dashboard do
  use GenServer
  ...
  def init(_) do
    :ok =  Cantastic.Emitter.configure(:polo_drive, "engine_status", %{
      parameters_builder_function: :default,
      enable: true,
      initial_data: %{
        "rotations_per_minute" => 0
      }
    })
    {:ok, %{}}
  end

  def set_rotation_per_minute(rotation_per_minute) do
    Cantastic.Emitter.update(:polo_drive, "engine_status", fn (data) ->
      %{data | "rotations_per_minute" => abs(rotation_per_minute)}
    end)
  end
  ...
end
```

Let’s push this on a Raspberry PI



NERVES

And... It works!

Spin42

Want some more?

- Loïc will present more details about the robotic part tomorrow at 13:35 in UB2.147.
- Marc will present the OVCS project tomorrow at 16:00 in K.1.105.
- ovcs.be