# E-Team Squadra Corse

## Introduzione al software

# Architettura Hardware

- Sensori: temperatura, pressione, velocità (“foniche”), PPS, IMU
- Attuatori: Inverter (+ motore), pompa, contattori HV
- Battery Pack + BMS + IMD
- Vehicle Control Unit (VCU)
- Shutdown Chain (SDC)
- Display, Logging, Telemetria

# Architettura di Comunicazione

- Uno o più bus di comunicazione basati su protocollo CAN
- EV-A: 3 can bus
  - CAN-S: signal bus, CAN-I: inverter can, CAN-B: BMS CAN
- ET-16: bus unificato (il BMS usa anche un bus interno)
- LoRA per la comunicazione telemetria

# Software “in-house”

- can_common: generazione DBC, bindings C, documentazione, analisi del carico del bus
- can_bootloader: STM32 and AVR
- config_lib: configurazione dei parametri senza ricompilare
- etdv_dashboard: visualizzazione
- Telemetria: CAN2Radio, Radio2CAN
- Firmware MCU

# Software utilizzato

- Git/Github per il controllo di versione e collaborazione
- PlatformIO: programmazione dei micro
- Meson e Cmake: compilazione su linux
- STM32CubeMX: Configurazione degli STM32 (VCU)
- FreeRTOS: scheduler real-time che gira sulla VCU
- cantools, asammdf, can-utils
- Linux (sviluppo e raspberry)

# CAN Bus Design

# Design space

- **CAN bus speed**: as high as 1Mbit/s (up to 40m cable length). Higher speed is associated to less noise immunity.
- **Can Messages** (Standard Frame):

**Message Period**

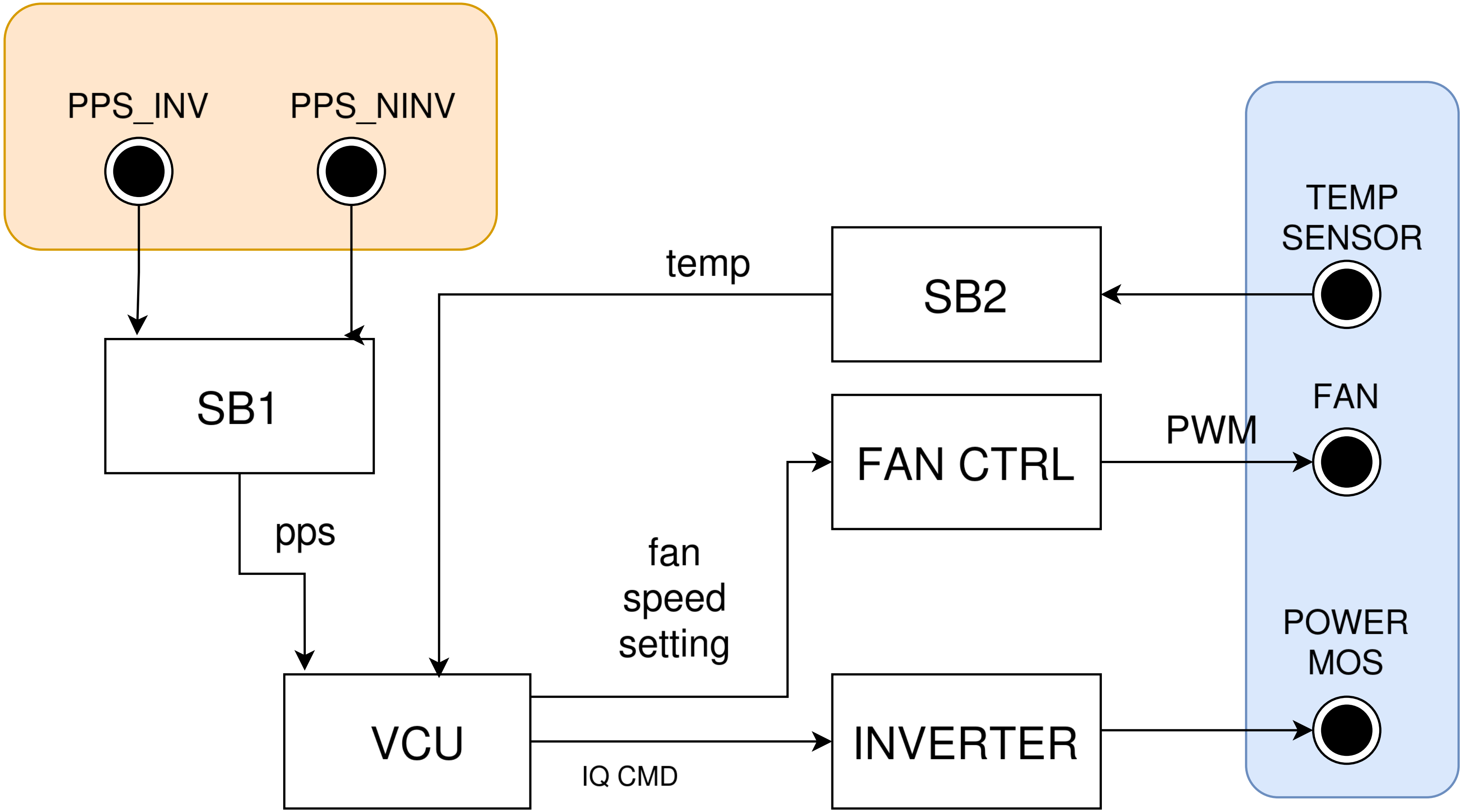
| ID (11bit) | Payload (up to 8 bytes) |
| --- | --- |

Affects bus utilization and worst case delay of lower-priority messages.

Lower ID means higher priority.
Affects worst case message delay.

Affects total transmission time (MCU)
Affects bus utilization

# Design example



# can_common

Python based, generates a bunch of stuff:

- DBC for reading the CAN bus
- C bindings to automate packing/unpacking messages
- Bus documentation in PDF
- Bus load and timing analysis

# using can_common to describe the bus

**can_def.py**

```
vcu_messages = [
    {
        'name': 'iq_cmd',
        'senders': [NodeId.VCU],
        'receivers': [NodeId.INVERTER],
        'message_id': MsgId.IQ_CMD,
        'comment': 'Quadrature current command',
        'cycle_time': None,
        'signals': [
            {
                'name': 'iq_rms',
                'scale': 0.1,
                'range': (0, 200),
                'unit': 'Arms',
                'comment': 'RMS value of quadrature current',
            },
        ]
    },
    ...
```

# Generating the DBC and bindings

```python
from dbc_dict_processor import create_dbs, fail_on_overlapping_ids
from dbc_bindings_generation import write_dbc_files, write_nodes_header
from can_def import main_messages, NodeId, DBC_VERSION
import cantools

main_db, main_bindings_db = create_dbs(main_messages(), NodeId, NodeId, DBC_VERSION)
fail_on_overlapping_ids([main_db])
cantools.database.dump_file(main_db, r'dbc/main.dbc')

write_dbc_files(main_bindings_db, 'main', 'bindings/main_db.h', 'bindings/main_db.c')

write_nodes_header([
    ('main', DBC_VERSION, NodeId),
], 'bindings/nodes.h')
```

# Firmware development
# Concepts and tools

# HAL (Hardware Abstraction Layer)

- Abstracts hardware details from implementation
  - CAN sending/receiving, sensor reading, pin control, timers, etc.
- Allows simulating the firmware code on a Linux PC ( posix_hal)
- Used by almost all our firmwares (VCU, sensorboard, pdu)

# HAL for design example SB

```c
#pragma once

#include <stdbool.h>
#include <stdint.h>

bool hal_can_init();
extern bool hal_send_can_message(uint32_t addr, const uint8_t* data, uint8_t length);
extern void hal_handle_can_message(uint32_t addr, const uint8_t* data, uint8_t length);

extern void analog_sensor_conversion_ready(int num_sensor, float value);

```

Called from interrupt
on real hardware

# VCU HAL

| | |
|---|---|
| bool | **vcu_can_init** ()<br>start CAN controller/communication |
| bool | **vcu_send_can_message** (uint32_t addr, const uint8_t *data, uint8_t length)<br>send out a CAN message |
| bool | **vcu_add_can_id_filter** (uint32_t id)<br>add a CAN id to the reception filter; must be called before **vcu_can_init** |
| void | **vcu_handle_can_message** (uint32_t addr, const uint8_t *data, uint8_t length)<br>handle CAN message inside the CAN handler task |
| void | **vcu_watchdog_start** ()<br>enable watchdog |
| void | **vcu_watchdog_stop** ()<br>stop the watchdog |
| void | **vcu_reset_watchdog** ()<br>refresh the watchdog |
| bool | **vcu_watchdog_has_reset** ()<br>check if the VCU started after a watchdog reset: will be true until next power cycle |
| void | **vcu_set_buzzer** (bool state)<br>set the buzzer state (true is on) |
| bool | **vcu_is_sdc_closed** ()<br>read shutdown-chain status |

read shutdown-chain status

| | |
|---|---|
| bool | **vcu_read_digital** (**vcu_digital_in_id** id)<br>read digital input from one of the four digital input pins |
| void | **vcu_write_digital** (**vcu_digital_out_id** id, bool value)<br>write digital state to one of the four digital output pins |
| void | **vcu_handle_digital_trigger** (**vcu_digital_in_id** id, bool value)<br>handle digital input trigger interrupt |
| float | **vcu_read_analog** (**vcu_analog_in_id** id)<br>read analog input |
| void | **vcu_handle_adc_conversion** (**vcu_analog_in_id** id, float value)<br>handle ADC conversion interrupt |
| void | **vcu_jump_to_bootloader** ()<br>reboot to CAN bootloader (soft hw reset) |

# Scheduling periodic tasks

- Simple approach: use a timer (fake_bms, sensorboard, pdu)
- When more control is needed, use a RT scheduler/OS (e.g. FreeRTOS)

# Heartbeats

- Periodic signals (usually CAN messages) generated by a board to signal it is alive
- Usually has some info attached (state, faults etc)
- Choose the period by estimating needed response time to a fault
- Some commands need to be issued periodically to be valid, e.g. inverter control or BMS command

# Critical Signals and CRC

- Critical signals are valid only if their last updated value is recent enough
- Example: PPS (pedal position signal) needs to be updated at a fast enough rate (15ms~20ms period) to ensure safety
- For stronger safety, CAN messages can have an additional CRC of few bits to check message intentionality (assuring that another board is not writing a message with wrong id)
- To ensure message ordering, add a *rolling counter* field to the payload

# VCU Critical Signals

Updating a critical signal (done in the can handler task):

```
void vcu_update_critical_signal (vcu_signal_id id, const void *data, TickType_t now, uint8_t counter, bool crc)
```

Update the critical VCU signal value identified by `id`.

Reading a critical signal: if value was not updated, dest value is garbage. Must either fail or provide an alternative safe value (e.g. 0 PPS)

```
bool vcu_read_signal_critical (void *dest, vcu_signal_id id, TickType_t now) __attribute__((warn_unused_result))
```

Read the signal value.

# State Machines

- Communicate design intent to others
- Reason about correctness
- Validation and code generation from state machine definitions is common in automotive (see Simulink + Stateflow)
- See VCU State Machine

# CAN bootloader

- Inspiration for the protocol from an AVR doc example/technical note
- Lives in a sector of the flash
- Each board has a unique identifier for flashing

```
[env:can_bootloader]
build_type = debug
can_baudrate=500000
canbl_id=0x06
canbl_node=0x02
```

Bootloader config is embedded in the platformio env.
Upload using

```
pio run -t can_upload_stm32
```

# config_lib

- Somewhat similar to standard XCP protocol for ECU
- Discoverable parameters with descriptions and bounds
- Supports bool, float, int datatypes
- Forward compatible storage on flash if used correctly
- Has a GUI… improvements or rewrites are welcome

# PlatformIO tips

- Use environments to have different configurations for the same project (e.g. different build time flags/options)
- Can be used in vscode, but try to use it with the command line at least a couple of time

# Firmware testing

- Unit testing: testing the single functions (see platformio documentation)
- Functional testing: testing of the functionality of a software unit (e.g. one particular firmware)
- Integration testing: testing the interaction between two ore more components (e.g. testing the whole firmware)

# Reproducible Testing

- My attempt: ETeam Test Framework

```python
def test_vcu_precharge_missing_signals(logging_manager):
    mgr = logging_manager.with_can_dispatcher(vcu=True)
    precharge_start = 3
    total_time = precharge_start + 5

    # create the following bms states sequence:
    # montoring -> precharge -> error
    bms_status_signal = mgr.main_can_signal("bms_BMS_Status.BmsMasterStatus")
    bms_status_signal.set_values(
        [0, precharge_start, precharge_start + 3, total_time],
        [
            "eFsmState_STANDBY",
            "eFsmState_PRECHARGE",
            "eFsmState_OPERATIONAL_LOAD",
            "eFsmState_OPERATIONAL_LOAD",
        ],
    )
    # ts button press for ~3 seconds
    button_press_signal = mgr.main_can_signal("STEERING_WHEEL_ButtonInputs.buttons")
    button_press_signal.set_value(0.2, 1)
    button_press_signal.set_value(precharge_start + 1, 0)

    mgr.set_main_can_signals(bms_status_signal, button_press_signal)
    logs = mgr.run(total_time)

    vcu_status_signal = logs.select(mgr.main_bus_name).extract_signal(
        "VCU_VCU_Heartbeat.Status"
    )

    assert (
        vcu_status_signal.predicate(lambda s: s == "IDLE" or s == "PARAM_UPDATE")
        .always()
        .eval()
    )
```

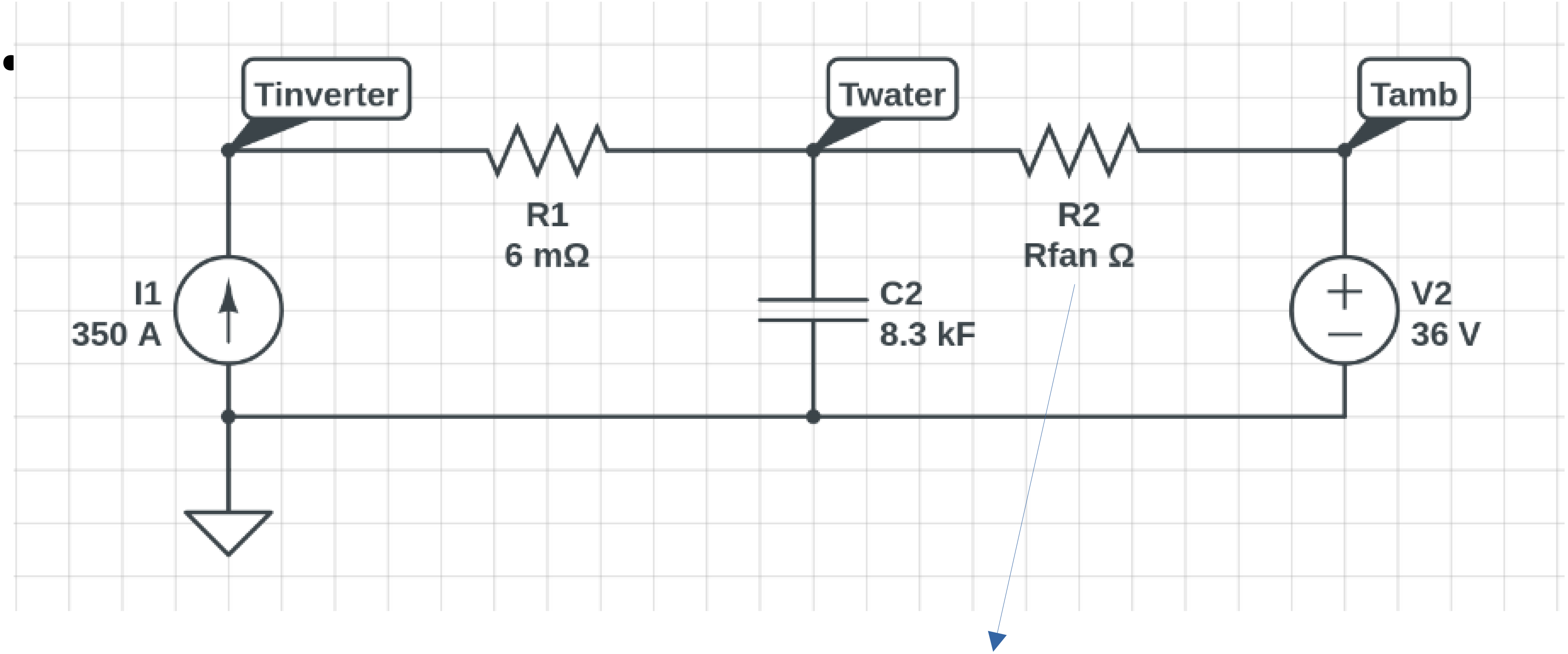
# Vehicle Model/Simulator

- el_vehicle_model: C++ implementation of a vehicle model (dynamics, powertrain, some control electronics)
- Preliminary support for export as FMU (Functional Mock-Up Unit), which can be used inside Simulink or Modelica
- Works alongside the other simulated firmwares, so structure needs to be coded directly in (C++, boost, not really easy)

# Homework: design example

- Control the fan speed to regulate the heatsink outflow and keep the inverter under 90 C
- Max power to dissipate: 70kW * 0.05
- Water temperature is measured

# ???




The value of Rfan must be controlled indirectly using VCU CAN messages

# Homework: design example

- Control the fan speed to regulate the heatsink heatflow and keep the inverter under 90 C
- Max power to dissipate: 70kW * 0.05
- Water temperature is measured
- Thermal model simulator is already there, follow README for compilation and further instruction
- <u>VCU pps signal should be critical, fix the bug!</u>
- <u>Sensorboard code for temperature reading is missing!</u>