Dashboard v2.0

Constructors: Michał Błotniak

(season 2025)

Table of contents

1. Project description

The main goal of the Dasha project for 2025 is to develop a new display, building on the previous version, which was excellently designed both in terms of software and hardware by Ignacy Kajdan. This year's display will be enhanced with a greater amount of data and further development of existing features, including the ability to operate in two modes: RACE and MAIN, active display of dangerous conditions for the car, SDC status, and much more. Additionally, the DASH LED board, which complements the display's functionality, will also be redesigned.

In the latest version, the project aims to display data such as:

MAIN SCREEN:

[1] Temperatures of individual motors

[2] Highest inverter temperature

[3] Fluid temperatures

[4] Fluid pressure

[5] LV battery temperature

[6] SDC status

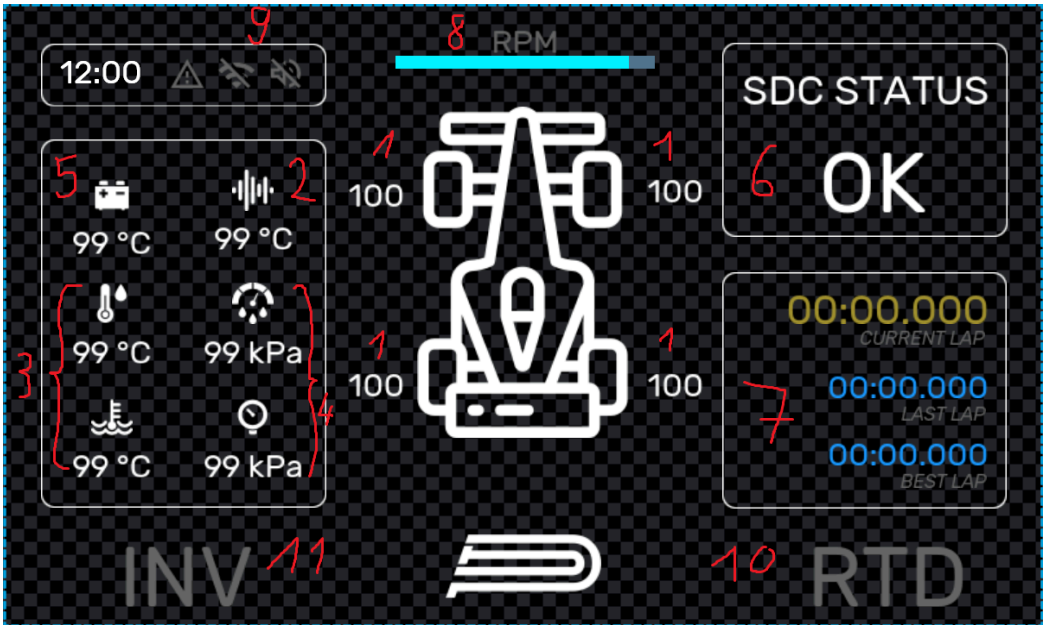
[7] Detailed lap timer data

[8] RPM

[9] Warnings, PC connection, and radio

[10] RTD

[11] Inverter status



RACE SCREEN:

[1] DRS status

[2] RTD

[3] SOC of HV battery

[4] SOC of LV battery

[5] Highest inverter temperature

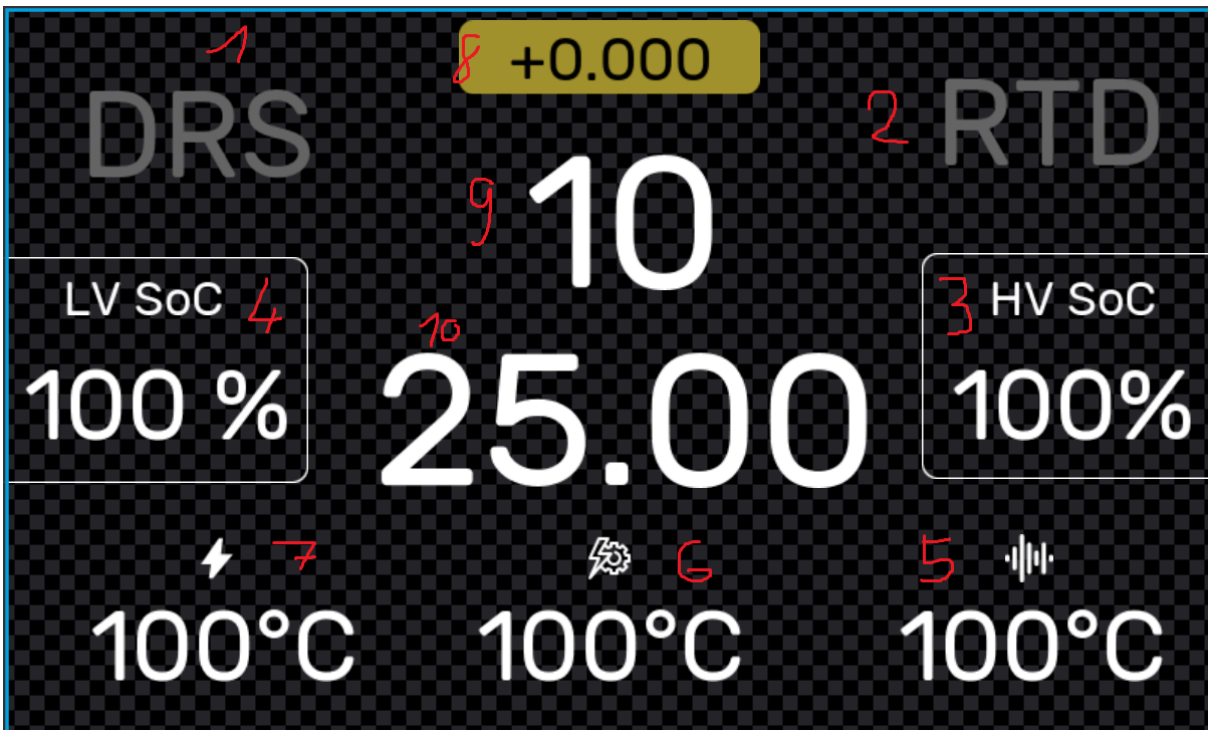
[6] Highest motor temperature

[7] HV battery temperature

[8] Pace

[9] Number of laps remaining

[10] Predicted range



NOTIFICATION SCREEN:

Displays dangerous conditions for the car or driver, such as: excessively high temperatures of motors, inverters, LV and HV batteries, or BSPD activation.

2. Regulation requirements

There are no requirements specified in the FS rules concerning the car's dashboard.

3. Research

Most dashboards used in Formula Student racing cars are based on a Raspberry Pi single board computer. That is because of the availability of reasonably priced HDMI displays from different manufacturers and the ease of development. However, there also exist projects of dashboards built solely on STM32 boards. Such development boards come with a built-in 4.3” display, which is just 0.7” less than the current display. Such size seems to be sufficient to display all the important information to the driver.

Figure 1: Team Bath Racing - 10" dashboard driven by a Raspberry Pi.

Figure 2: Approximately 5” dashboard - TU Graz.

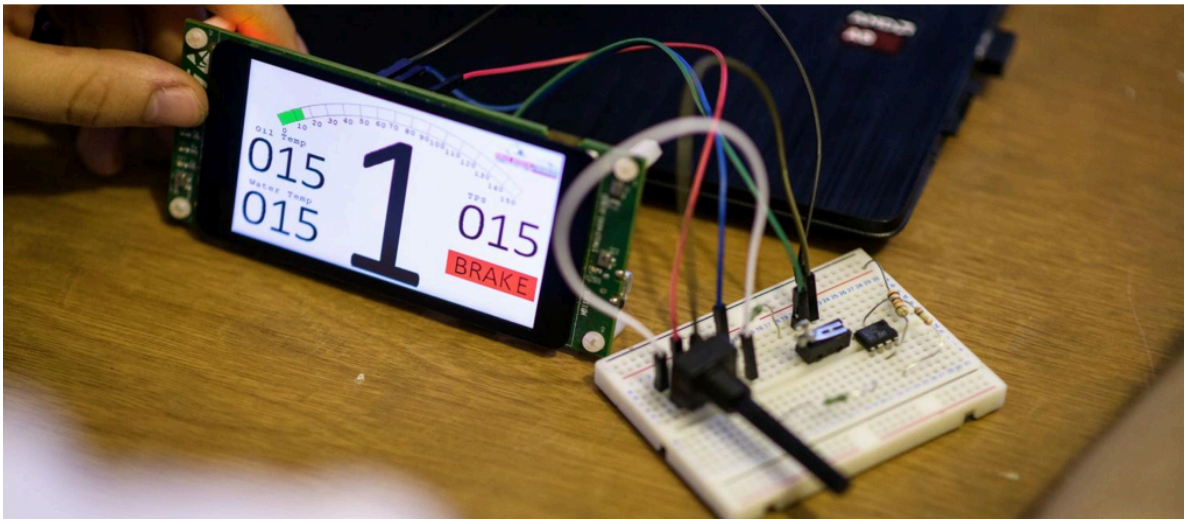


Figure 3: A prototype of an STM32 dashboard.

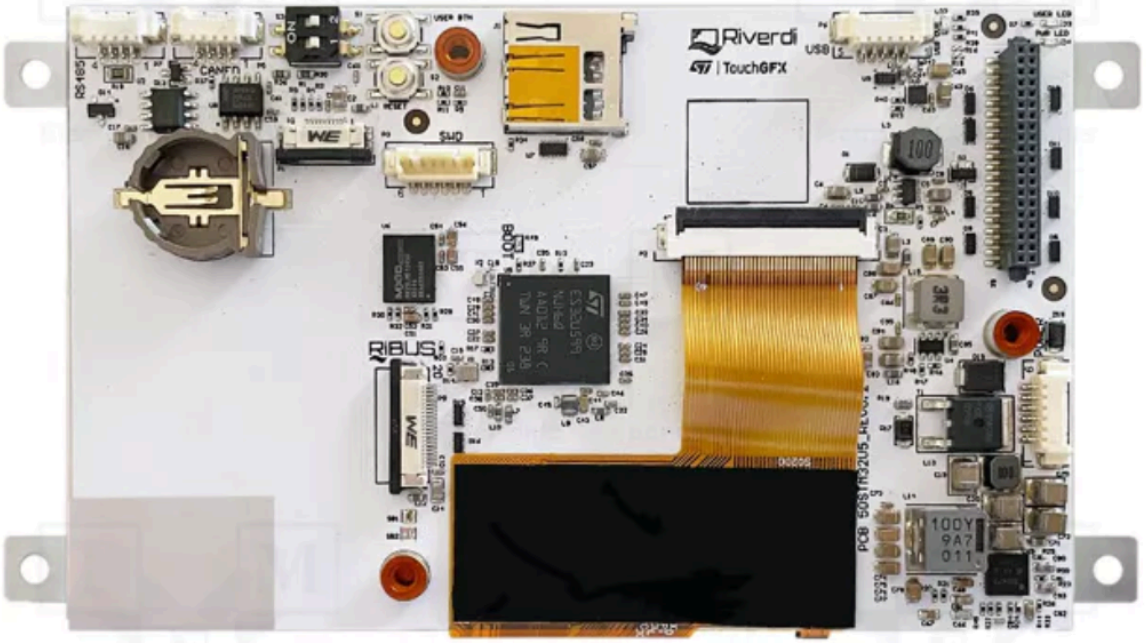

3
00:00.000
5879
3
0.00
125
+0.00 00:00.00
Race
3
01:54.231


+0.000
01:43:12
3
6926
78.8°
6.3
5
280.0
80.9°
14.1
153
08:18.402
01:53.988
01:53.988
TC
TC CUT
ABS
ENGINE
FUEL
1
47.9
25
1.9
13

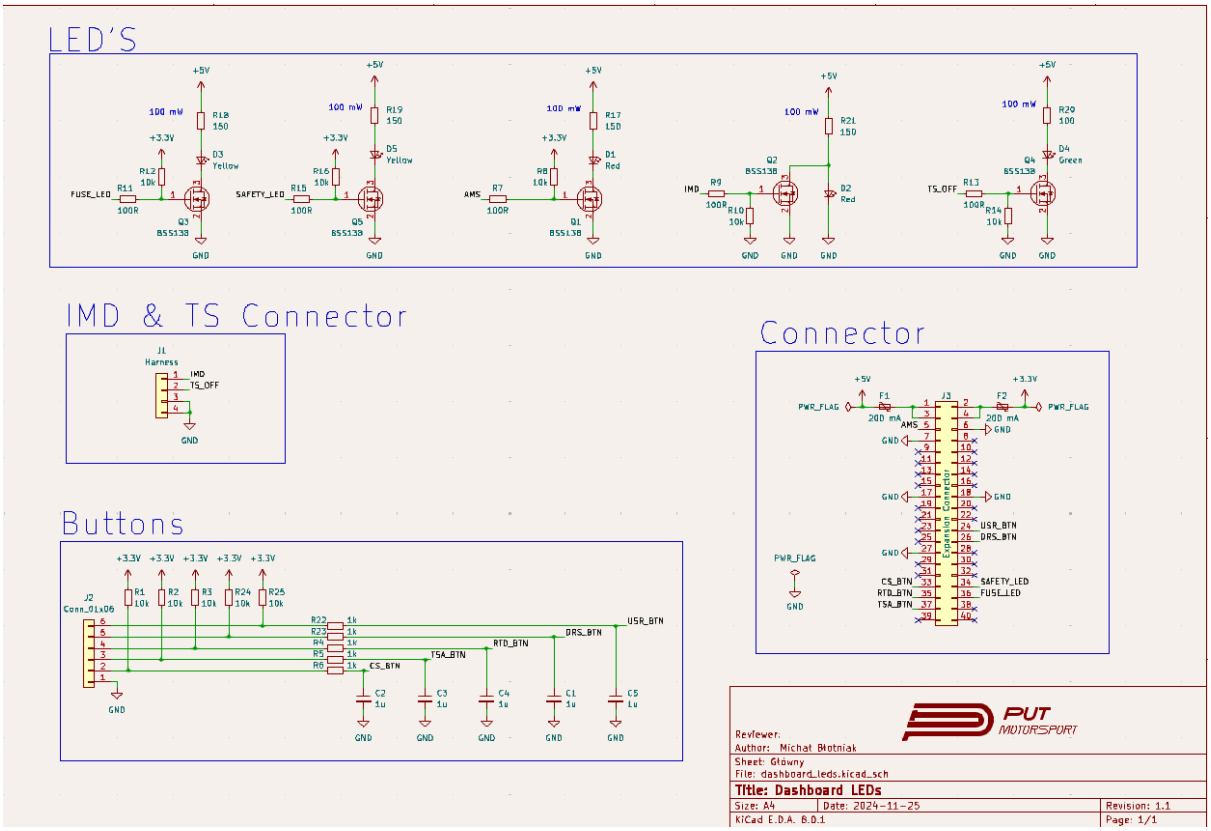
4. Hardware

The core of the dashboard is a board SM-RVT50HQSFWN00 Riverdi with a built-in display and an STM32 microcontroller. An additional advantage is the inclusion of outputs for the CAN bus and GPIO connections.

https://www.tme.eu/pl/details/sm-rvt50hqsfwn00/wyswietlacze-inteligentne/riverdi/



The completed board works in conjunction with the DASH_LEDS board, which connects to the main PCB via outputs and receives signals from the TS ON, RTD, and CHANGE SCREEN buttons. Additionally, it uses LEDs to signal the vehicle's status and provide information in compliance with regulatory requirements.

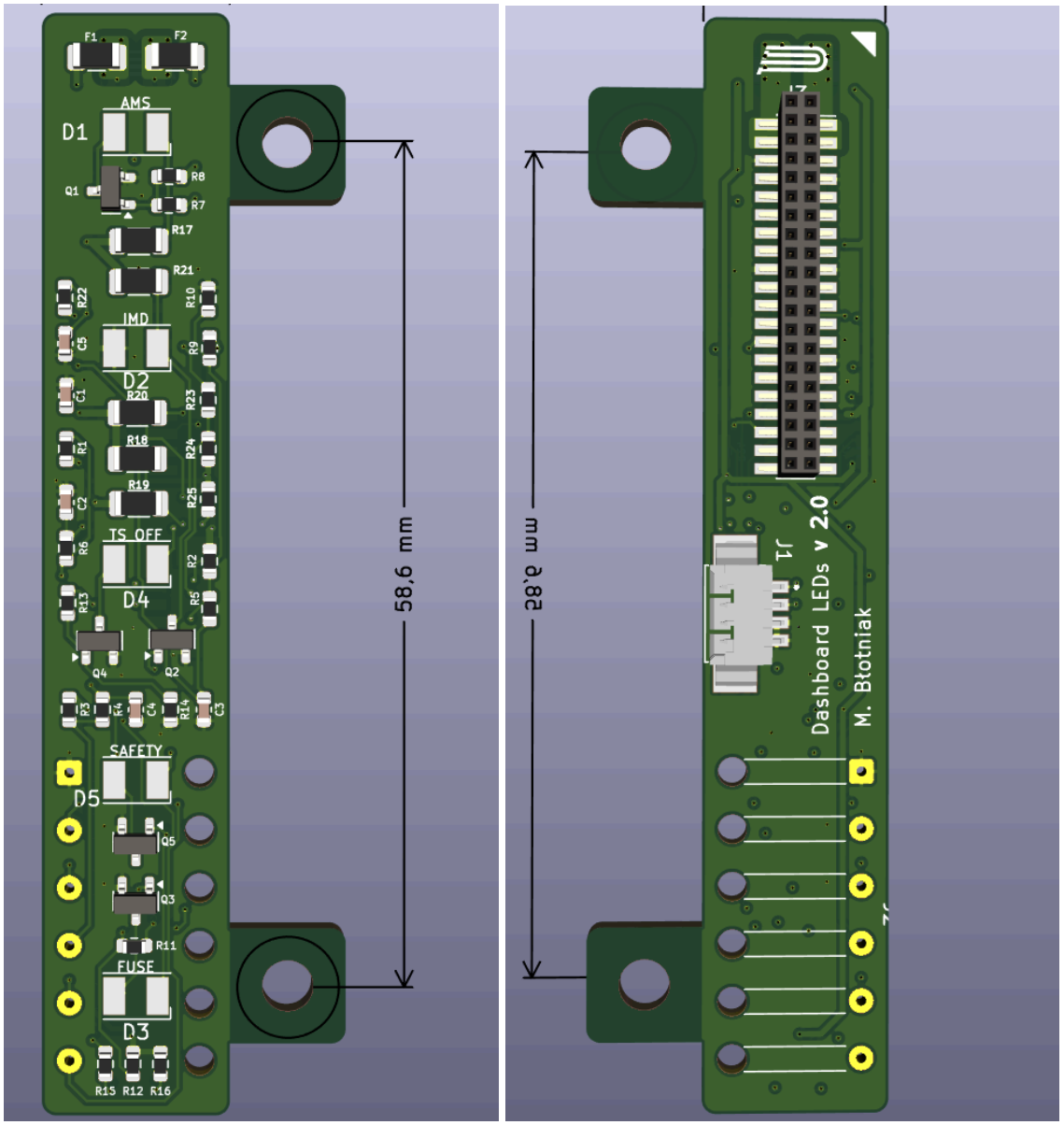


All details of the DASH_LEDS project are available in the team's repository:

https://github.com/PUT-Motorsport/PUTM_EV_DASHBOARD_LEDS_2024

The connector linked to the main board facilitates information exchange between the two. A low-pass filter has been added for button handling to eliminate contact bounce. The LEDs are controlled via transistors and have been selected to ensure their visibility is unquestionable, even in full sunlight.

The **IMD** and **TS_OFF** are inputs that control the LEDs independently of the software and the main control board, as required by regulations.


F1
F2
AMS
D1
Q1
R8
R7
R17
R21
R22
R10
IMD
C5
R9
D2
C1
R20
R23
R18
R1
R24
R19
C2
R25
TS OFF
R6
R2
R13
D4
R5
Q4
Q2
R3
R4
C4
R14
C3
SAFETY
D5
Q5
Q3
R11
FUSE
D3
R15 R12 R16
58,6 mm
28,82 mm
J1
Dashboard LEDs v 2.0
M. Błotniak

5. Software

The graphical interface is generated using the TOUCH GFX program, which, in collaboration with STM32CubeIDE, generates some of the files for us and creates a commonly used code structure for graphical projects known as MVP (Model-View-Presenter). Additionally, we use the FreeRTOS real-time operating system, which helps us manage the rest of the project.

Core Structure and description of the most important files:

Model:

- Represents the application logic and holds shared data for the system.
- Inherits ModelListener (indirectly binds the Model to the Presenter).
- Implements data synchronization using mutexes to ensure thread safety when interacting with shared hardware resources.

Example:

```
Model::Model() : modelListener(0) {}

void Model::tick() {

    if(modelListener != 0) {
        modelListener->toggleElements();
        modelListener->switchScreenMR();
        modelListener->switchScreenRM();


        if(osMutexAcquire(sharedDataMutexHandle, osWaitForever) == osOK) {
            m_sharedData = sharedData;

            m_sharedDataPrev.time = m_sharedData.time;
            modelListener->setClock(m_sharedData.time);

            m_sharedDataPrev.connection = m_sharedData.connection;
            modelListener->setConnection(m_sharedData.connection);

            m_sharedDataPrev.warning = m_sharedData.warning;
            modelListener->setWarning(m_sharedData.warning);

            m_sharedDataPrev.radio = m_sharedData.radio;
            modelListener->setRadio(m_sharedData.radio);

            m_sharedDataPrev.ready_to_drive = m_sharedData.ready_to_drive;
            modelListener->setReadyToDrive(m_sharedData.ready_to_drive);
```

ModelListener:

- Abstract base class defining interfaces for presenters to receive updates from the model.
- Acts as a bridge between the Model and the Presenter.
- Virtual methods allow for polymorphic behavior, enabling each specific screen (Main, Race, etc.) to customize its reaction to data changes.

Example:

```
class ModelListener {
  public:
    ModelListener() : model(0) {}
    virtual ~ModelListener() {}
    void bind(Model* m) { model = m; }

    //Main Screen
    virtual void setClock(uint32_t time) {}
    virtual void setConnection(bool status) {}
    virtual void setWarning(bool status) {}
    virtual void setRadio(bool status) {}
    virtual void setReadyToDrive(bool status) {}
    virtual void setInvertersStatus(bool inv_ready,
                                    bool inv_FL_status,
                                    bool inv_FR_status,
                                    bool inv_RL_status,
                                    bool inv_RR_status,
                                    bool inv_FL_error,
                                    bool inv_FR_error,
                                    bool inv_RL_error,
                                    bool inv_RR_error) {}
    virtual void setDrsStatus(bool status) {}
    virtual void setBatteryLVTemperature(uint8_t temperature) {}
    virtual void setOilTemperature(uint8_t temperature) {}
    virtual void setOilPressure(uint8_t pressure) {}
    virtual void setCoolantTemperature(uint8_t temperature) {}
    virtual void setCoolantPressure(uint8_t pressure) {}
    virtual void setCurrentLap(uint32_t time) {}
    virtual void setLastLap(uint32_t time) {}
    virtual void setBestLap(uint32_t time) {}
    virtual void setMotorFrontLeftTemperature(uint8_t temperature) {}
    virtual void setMotorFrontRightTemperature(uint8_t temperature) {}
    virtual void setMotorRearLeftTemperature(uint8_t temperature) {}
    virtual void setMotorRearRightTemperature(uint8_t temperature) {}
    virtual void setSDC(SafetyData_TypeDef& safetyData) {}

    virtual void setInverterTemperature(uint8_t inv_FL_temperature,
                                        uint8_t inv_FR_temperature,
```

Presenter:

- Acts as an intermediary between the Model and the View.
- Implements the concrete logic of the screen's behavior based on data from the Model.
- Inherits from ModelListener to receive updates and interacts with a specific View instance.

Example:

```
MainScreenPresenter::MainScreenPresenter(MainScreenView& v) : view(v) {}
void MainScreenPresenter::activate(){ screenStatus.MainScreen = true; }
void MainScreenPresenter::deactivate(){ screenStatus.MainScreen = false; }
void MainScreenPresenter::setClock(uint32_t time) { view.updateClock(time); }
void MainScreenPresenter::setConnection(bool status) { view.updateConnection(status); }
void MainScreenPresenter::setWarning(bool status) { view.updateWarning(status); }
void MainScreenPresenter::setRadio(bool status) { view.updateRadio(status); }
void MainScreenPresenter::setReadyToDrive(bool status) { view.updateReadyToDrive(status); }
```

View:

- Handles the visual representation and user interaction.
- Provides concrete implementations for rendering and reacting to events.
- Inherited classes like MainScreenView, RaceScreenView, etc., override or add methods for specific visual elements.

Example from MainScreenView:

```
void MainScreenView::updateInverterTemperature(uint8_t inv_FL_temperature,
                                               uint8_t inv_FR_temperature,
                                               uint8_t inv_RL_temperature,
                                               uint8_t inv_RR_temperature) {

    uint8_t inv_temp_highest = std::max({inv_FL_temperature, inv_FR_temperature, inv_RL_temperature, inv_RR_temperature});

    Unicode::snprintf(invTempTextBuffer, INVTEMPTEXT_SIZE, "%d", inv_temp_highest);
    if(inv_temp_highest > INVERTER_TEMPERATURE_MAX || inv_temp_highest < INVERTER_TEMPERATURE_MIN) {
        invTempIcon.setBitmap(Bitmap(BITMAP_INVERTER_CRIT_ID));
        invTempText.setColor(touchgfx::Color::getColorFromRGB(255, 0, 0));
    } else if(inv_temp_highest > INVERTER_TEMPERATURE_MID) {
        invTempIcon.setBitmap(Bitmap(BITMAP_INVERTER_WARN_ID));
        invTempText.setColor(touchgfx::Color::getColorFromRGB(163, 146, 46));
    } else {
        invTempIcon.setBitmap(Bitmap(BITMAP_INVERTER_ID));
        invTempText.setColor(touchgfx::Color::getColorFromRGB(255, 255, 255));
    }
    invTempIcon.setVisible(true);
    invTempText.setVisible(true);
    invTempIcon.invalidate();
    invTempText.invalidate();
}
```

Other important files, outside the MVP model, essential for the operation of the project:

1. communication_task.cpp

Purpose: Manages communication with external components via the CAN bus (Controller Area Network). It handles both data transmission (TX) and reception (RX) for various safety and system parameters.

Key Functions:

- Transmission:

- Sends button states (DRS, RTD, TSA)

- Uses the `PUTM_CAN` library to construct and send CAN messages.

- Reception:

- Receives data from components such as the frontbox, rearbox, BMS, and PC.

- Updates shared data structures (sharedData, safetyData) based on received messages, ensuring thread safety using mutexes.

- Monitors timeouts to ensure components respond within defined intervals, setting warnings if they don't.

```
// BMS HV
if(PUTM_CAN::can.get_bms_hv_main_new_data()) {
    timeoutData.bms_hv_last_frame_time = current_tick_time;
    auto bms_hv_main_data = PUTM_CAN::can.get_bms_hv_main();

    if(bms_hv_main_data.ok) {
        interfaceData.ams_led = false;
    } else {
        interfaceData.ams_led = true;
    }

    if(osMutexAcquire(sharedDataMutexHandle, osWaitForever) == osOK) {
        sharedData.warning = false;
        sharedData.soc_hv = bms_hv_main_data.soc / 10;
        sharedData.battery_hv_temperature = bms_hv_main_data.temp_max;

        osMutexRelease(sharedDataMutexHandle);
    }
} else if(current_tick_time - timeoutData.bms_hv_last_frame_time > DASH_TIMEOUT_DURATION) {
    interfaceData.ams_led = true;

    if(osMutexAcquire(sharedDataMutexHandle, osWaitForever) == osOK) {
        sharedData.warning = true;
        sharedData.soc_hv = 0;
        sharedData.battery_hv_temperature = 0;

        osMutexRelease(sharedDataMutexHandle);
    }

}
```

2. data.c

Purpose: Defines and initializes the shared data structures used throughout the application.

Example:

```
ScreenStatus_TypeDef screenStatus = {
    .MainScreen = false,
    .RaceScreen = false,
    .NotificationScreen = false,
};
```

3. interface_task.cpp

Purpose: Handles physical interface elements, such as buttons and LEDs.

Key Functions:

- LED Management:

- Updates LEDs based on system states (ams_led, safety_led, etc.).

- Button Handling:

- Reads button states using GPIO and implements debouncing logic to filter false triggers.

- Tracks specific buttons for special tasks:

- RTD Button

- TSA Button

- CS Button: Screen Change.

- USR Button

- DRS Button: Drag Reduction System toggle.

```
// Buttons
//RDT button
if(HAL_GPIO_ReadPin(RTD_BTN_GPIO_Port, RTD_BTN_Pin) == GPIO_PIN_RESET) {
    if(interfaceData.rtd_timer >= DASH_BUTTON_DEBOUNCING_TIME) {
        interfaceData.rtd_button = true;
    } else {
        interfaceData.rtd_timer += DASH_BUTTON_POOLING_RATE;
    }
} else {
    interfaceData.rtd_button = false;
    interfaceData.rtd_timer = 0;
}
```

You need to be careful when working in this file because it is responsible for reading TSA and RTD which are crucial to start the car

File:	*Function:*
MainScreenPresenter.cpp	Handles data for the main screen (temperatures, warnings, screen switching).
MainScreenView.cpp	Displays visual data for the main screen.
RaceScreenPresenter.cpp	Handles race data (temperatures, DRS, screen switching).
RaceScreenView.cpp	Displays detailed race data (batteries, DRS, motors).
NotificationScreenPresenter	Manages critical errors and notifications.
NotificationScreenView	Displays warning and error messages.
Model.hpp / Model.cpp	Application logic, synchronizes data between hardware and presenters.
ModelListener.hpp	Interface for communication between the model and presenters.
communication_task.cpp	It deals with CAN communication: transmitting and receiving data
data.c	Initializes and manages shared data structures used across the application.
interface_task.cpp	Manages physical interface components (buttons and LEDs) with debouncing and state control.

6. How start

The entire project is on github:
https://github.com/PUT-Motorsport/PUTM_EV_DASHBOARD_2024.

The ReadMe contains a detailed description of what to do to download the project and what you need to start working.

If there are errors related to the CAN library:

- check if the project has the latest version

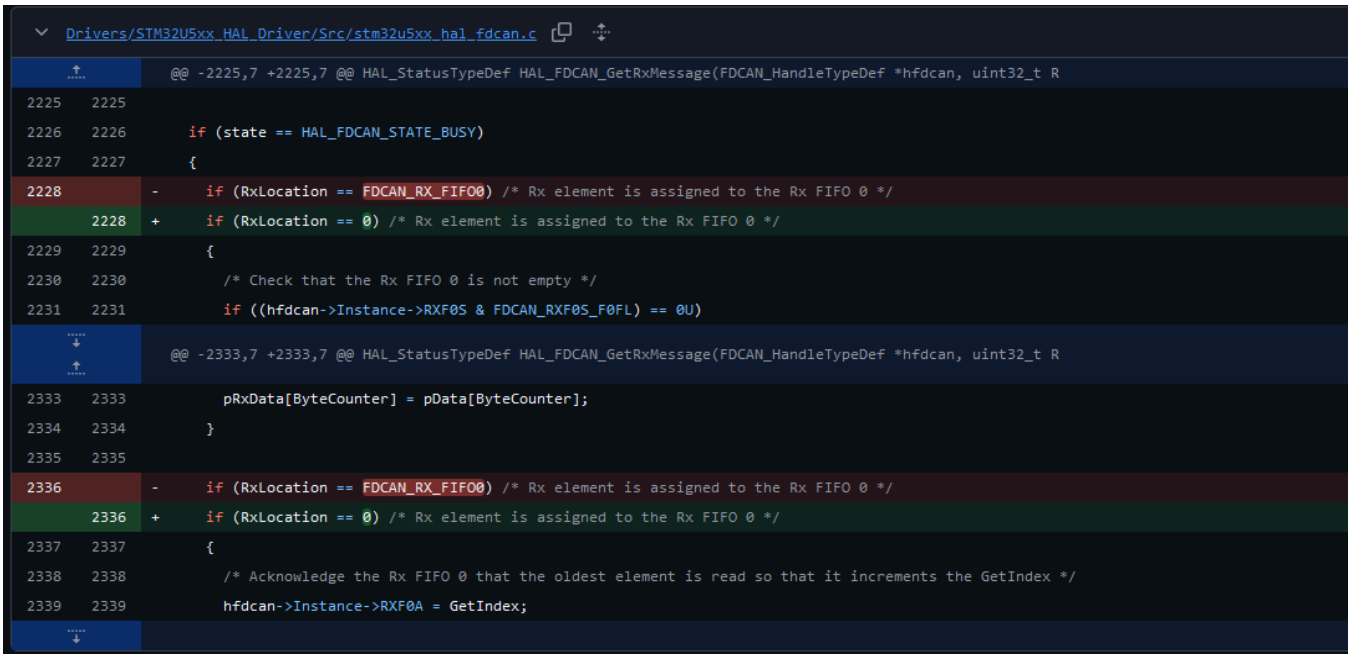
If there are errors related to the lack of files from TOUCH GFX:

- open the Dashboard.touchgfx file and generate the code
- clean and refresh the project in CubeIDE
- try to build it again

If there are errors related to the lack of files generated by CubeMX:

- generate the project again using CubeMX, clean it, refresh it and try to build it again

Remember that generating a project by CubeMX introduces changes to the file responsible for CAN communication. Additionally, it creates a main.c file, which you can delete



On the main branch there is only a stable version that guarantees the car's operation. There are code version markings on it, e.g. v 1.0. Therefore, before you start working, create your own branch and work on it until the code is brought to a stable, working version.

7. Conclusions

TODO