An overhaul of the previous TruControl project:

TruControl - custom

An Arduino project for making truck simulators even more immersive.

Csongor ’csongoose’ Csaszar & Baran Ismen
2023.01.29.

Table of Contents

Introduction **2**
What is TruControl and why was it made? 2
Hardware parts 2
Software parts 2
The Arduino software 2
The python software 3
Installation **4**
Using the compiled executable 4
Using the python script files 7
Wiring information 7
The software **8**
In-depth look into the python code 8
Config files 8
Telemetry reading 8
Data handling 8
Outputs 8
Inputs 9
Different python files 10
In-depth look into the Arduino code 10
Arduino input handling 10
Arduino output handling 11
Arduino code flowchart 12
Communication 13
How does serial communication work? 13
Establishing connection 13
Encoding 14
Decoding 14
Serial Communication Flowchart 15
Hardware **16**
Editing the controls.sii file **19**
Conclusion **20**
Default pinout chart **21**

Authors:
- Csongor ’csongoose’ Csaszar (: csongoose#9557, : csongoose)
- Baran Ismen (: Captain619)

Introduction

What is TruControl and why was it made?

TruControl is a project for the popular driving game Euro Truck Simulator 2. It helps provide a more immersive environment for enjoying the simulation experience. The software part is somewhat universal – it can be modified by changing the configuration files and the definitions in the Arduino sketch. Although the hardware part is fully custom planned and made by Baran, the core hardware parts are also universal.

The system includes a custom-made button box, a stalk panel and a dashboard, which is attached to a Logitech G27 wheel. It can control most of the functions on a truck (from indicators to axle lifting and suspension), and the dashboard can display most of the in-game indicator lights. The hardware is powered entirely by an Arduino MEGA board via USB.

Hardware parts

- Steering Wheel Set (obviously), a Logitech G27 in our case
- Arduino Mega 2560 (CH340 or 2560 chipset)
- Bunch of electronic parts (will be mentioned below in detail)
- Bunch of 3D-Printed parts

Software parts

The software consists of two main parts, the program run by Arduino and the program run by the computer.

The Arduino software

The Arduino runs a code which is written in C++ and only does basic functions. The main goal is doing message and input handling. The code sends messages over serial according to the inputs given and gives outputs according to the messages received. You can read more about the communication on page 13. The software on the board is equipped with several safety stops, which can be used in case one of the pins float or gives false input. The running code can be stopped at startup with the use of the software emergency pin, or by pressing the desync button/switch when the main loop is already running. When the desync button is released, the main loop continues on. An emergency stop is not needed in this case, because the USB cable can be pulled anytime, which stops both the python and the Arduino program, acting as another safety stop. The Arduino will reset its internal variables at every reset and/or power loss.

Every safety feature will show a unique error message, which can be read from the dashboard lights lighting up and/or flashing. The message can be decoded with the help of the *arduino_errorcodes* file included with the documentation.

The python software

The python software is included both as a compiled exe and as a python script. In both cases it depends on the two config files, *config.yml* and *keybinding.yml* (config files). If any of the files is missing, the program will produce an error message and exit. The python code is interlocked with the telemetry server, the game and the Arduino board, when starting it up. If any of these dependencies is non-existant ot not started, the program will not start. In case the telemetry server or the Arduino dependency is missing, the software will exit and give you the option to start the setup when you run the program the next time. This feature is included to easily change options in case something changed on your computer since the last time the program was started.

The keyboard emulating and input handling module only starts if all of the dependencies are met. If the *keybinding.yml* file is not edited with the syntax which the keyboard module requires, the software will exit when pressing the button.

IMPORTANT NOTE: All keyboard events can be disabled by setting the *inputsEnabled* variable to *false* in the *config.yml* file.

The config files are only read on startup. This means that you cannot change values while the software is running, as it will disregard the changes until the read function is called again.

Python modules

The program is built on several different publicly available python libraries, which need to be downloaded and installed if the python script is run. All of these modules are already integrated into the compiled executable and as so, included with the zip file.

The modules used are the following:

-json
-pyyaml
-time
-keyboard

None of these modules are modified in any way, which means that the one you can download from PyPi or github will work.

Installation

TruControl can be installed and used in one of two ways.

Using the compiled executable

The *executable.zip* file on github has a compiled version of the software with all internal dependencies (python modules) included.

In this case, the user only needs to flash their arduino and install funbit's telemetry server. You can install the software this way without any coding knowledge.

Step 1: Download everything you need

executable.zip	link
Funbit's telemetry server updated by dakar2008*	link
Arduino project file	link
Arduino button library from madleech	link

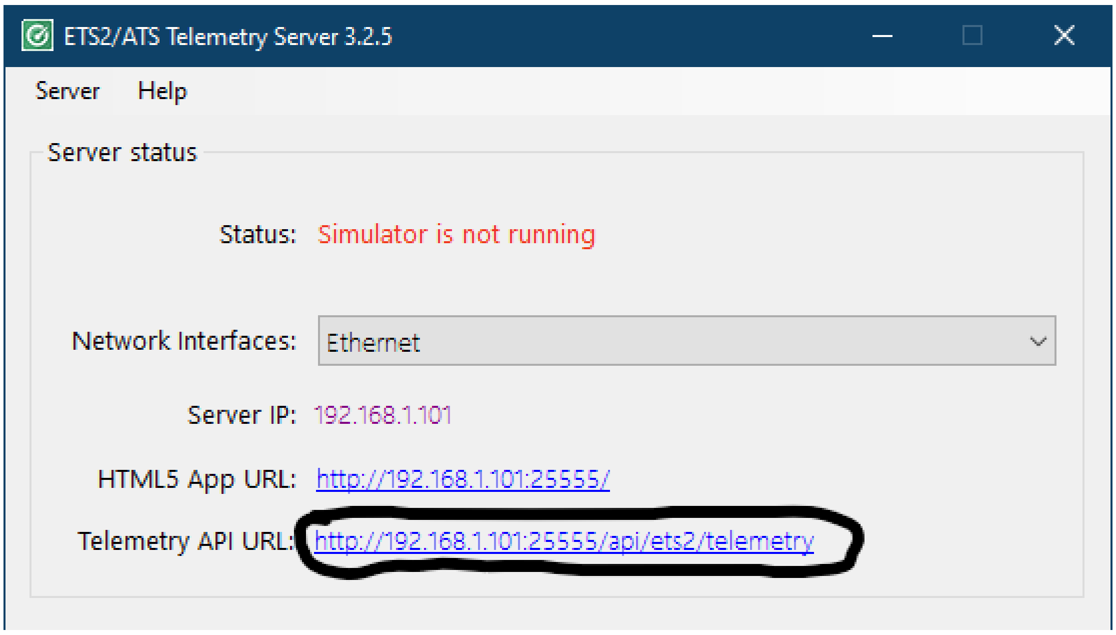
*: The software should work with Funbit's original release, but the original project is no longer supported.

Step 2: Flash, unzip and install!

Open the downloaded telemetry server. It should automatically find your game and put the required modifications in it. If it fails to find your game, you need to add it manually. The software has a setup included which will guide you through the installation steps.

You will need to flash your Arduino board with the sketch file downloaded (*trucontrol_custom_arduino.ino*). If you need to modify the pinout, or have a different board, please check page 10.

The zip file you downloaded from the TruControl github page needs to be unzipped. You can do this in any folder you want, the program doesn't care about its own directory. If you are running the software for the first time, the setup wizard will start. Plug in your Arduino board and start the telemetry server. You will need the telemetry server URL from the telemetry server software which can be found here:



Status: Simulator is not running

Network Interfaces: Ethernet

Server IP: 192.168.1.101

HTML5 App URL: http://192.168.1.101:25555/

Telemetry API URL: http://192.168.1.101:25555/api/ets2/telemetry

Fig. 1: The location of the Telemetry API URL inside Funbit's software.

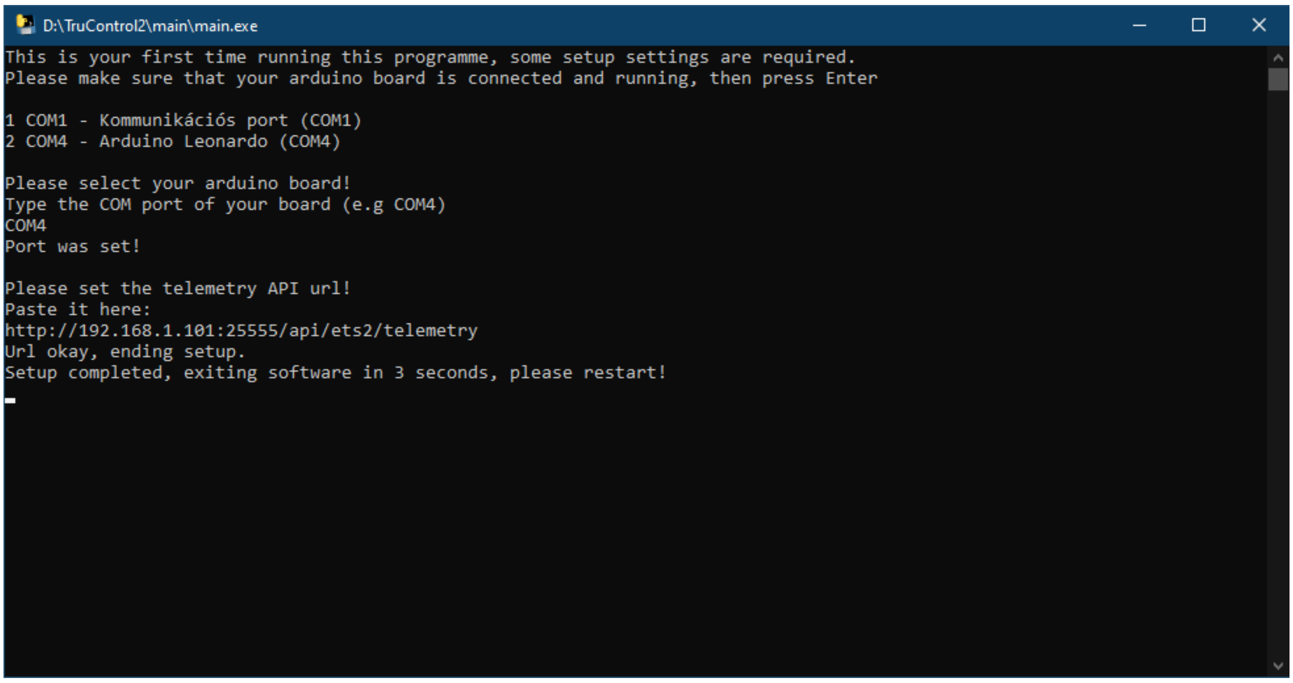
Run the *main.exe*! If you are running the software for the first time, the setup wizard will start.

Step 3: The setup

Give the Arduino COM port number to the software. It will list the available COM ports on your device with names. Arduino boards can either show up by their name (e.g. COM4 Arduino Leonardo), as an unknown device (e.g. COM3 Unknown device), or by their chip name (e.g. COM4 AtmegaXX). If you are not sure which device you need to use, close the software, pull your Arduino cable from your computer, and start the software again. Whichever device disappears should be the board you are looking for. The syntax used while setting the COM port is *COMx*, e.g *COM4*. If you are using the wrong syntax, the software will warn you.
NOTE: You can select any device from your COM list, and as long as a message over serial can be written to it, it will pass the check, so be careful setting this device.

Set the telemetry URL when asked, by simply pasting the telemetry URL into the command prompt and press enter. The software will perform a check, which consists of a request to your URL. NOTE: This means that any valid URL will pass the check and does not necessarily mean that the given URL is correct.

If you have done everything right, the software will close after 3 seconds, and on your next startup, you should be good to go. Make sure you're running the main.exe file in Administrator Mode, ensuring that the communication between software & hardware is not interrupted.



```
This is your first time running this programme, some setup settings are required.
Please make sure that your arduino board is connected and running, then press Enter

1 COM1 - Kommunikációs port (COM1)
2 COM4 - Arduino Leonardo (COM4)

Please select your arduino board!
Type the COM port of your board (e.g COM4)
COM4
Port was set!

Please set the telemetry API url!
Paste it here:
http://192.168.1.101:25555/api/ets2/telemetry
Url okay, ending setup.
Setup completed, exiting software in 3 seconds, please restart!
```

Fig. 2: The setup wizard

Step 4: Keys, keys, keys!

IMPORTANT NOTE: A default keybinding.yml file is provided for you, although it is heavily recommended to change and customize these keybindings.

Set up your preferred key settings in the *keybinding.yml* file. If you are only using the software for dashboard operations, set the *inputsEnabled* variale in *config.yml* to *false* and skip this step. The inputs are enabled by default.

The syntax of the *keybinding.yml* file should be written according to the rules of the keyboard module. This means that most of the key names you can think of will work. If you want to cross-check with the official canonical names, see the *validkeys.md* file.

Please note that due to scan code interference, the keyboard module does not handle the numpad, and as so, any numpad keys given will be translated to their alphanumerical values
(E.g. *num 1* will be equal to pressing the alphanumerical 1 key. To put it simply, *"num 1" = "1"*)

Adding key combinations to your keybinding

The keyboard module can handle key press combinations (e.g. Ctrl + L). In this case the keybinding file should be written as the default key combination syntax of the keyboard module. This is as follows:

```
<key1>+<key2>
```

Key1 and key2 are canonical names of keys. Here is an example:

```
retOffKey: shift+1
```

This example will make it press the shift and the 1 key at the same time.

If you want to use key combinations in game, you need to edit the *controls.sii* file. A guide to this can be found on page 19.

Step 5: Using the software after installation

If you have completed the installation steps, you can run the software normally. Start your telemetry and connect your Arduino before running the main.exe. If you fail to do so, the software will produce an error and prompt you to run setup. If this happens, you can reject the setup function by simply answering *n*.

If anything goes bad when starting up the software, for example something has changed since the last time you ran it (e.g.: Arduino board is on a different port), the software will prompt you to run setup the next time you start the software.

The software will wait for your game to load before continuing into the main loop. This includes loading into your truck. The main loop starting is indicated by a message in the terminal. After the *„You are good to go!"* line, the software starts functioning.

If - for any unknown reason - your Arduino board or your computer hangs when starting the software, you can short the software emergency pin with GND on your Arduino, and/or kill the process by pressing Ctrl+C in the command line.

Using the python script files

If you are using the python script files, you obviously have some coding knowledge. All of the .py scripts and the *.yml* files need to be downloaded and put into the same folder. The code can run under python3, if you have all of the dependencies installed. The dependencies can be found under the *python modules* section on page 3. Other than that, the software should work the same as the compiled executable. Check steps 3, 4 and 5 in the previous installation section for reference.

Wiring information

The Arduino board has the input pins set as INPUT_PULLUP. This means that the pins are HIGH state (+5 or+3,5 V) by default. When you want to activate one of the inputs, this signal needs to be pulled down to GND. A resistor is not required, as the board uses its internal pullup resistors to achieve the signal, which means that shorting it to the GND will not result in an actual short circuit.

The outputs on most of the boards available can easily handle a bunch of LEDs on every pin, but make sure you read the parameters of your own board before directly soldering everything and possibly breaking your device. You need to use resistors before LEDs to protect them from overheating and overcurrent! In this project, we used 150 Ohm resistors before every LED. If the board is only powered via USB, the power limitations of said USB connector apply.

The TruControl team is not liable to any damage caused by faulty wiring or disregard to the safety guidelines - according to the license. If you do not feel comfortable assembling your device, ask a professional to help.

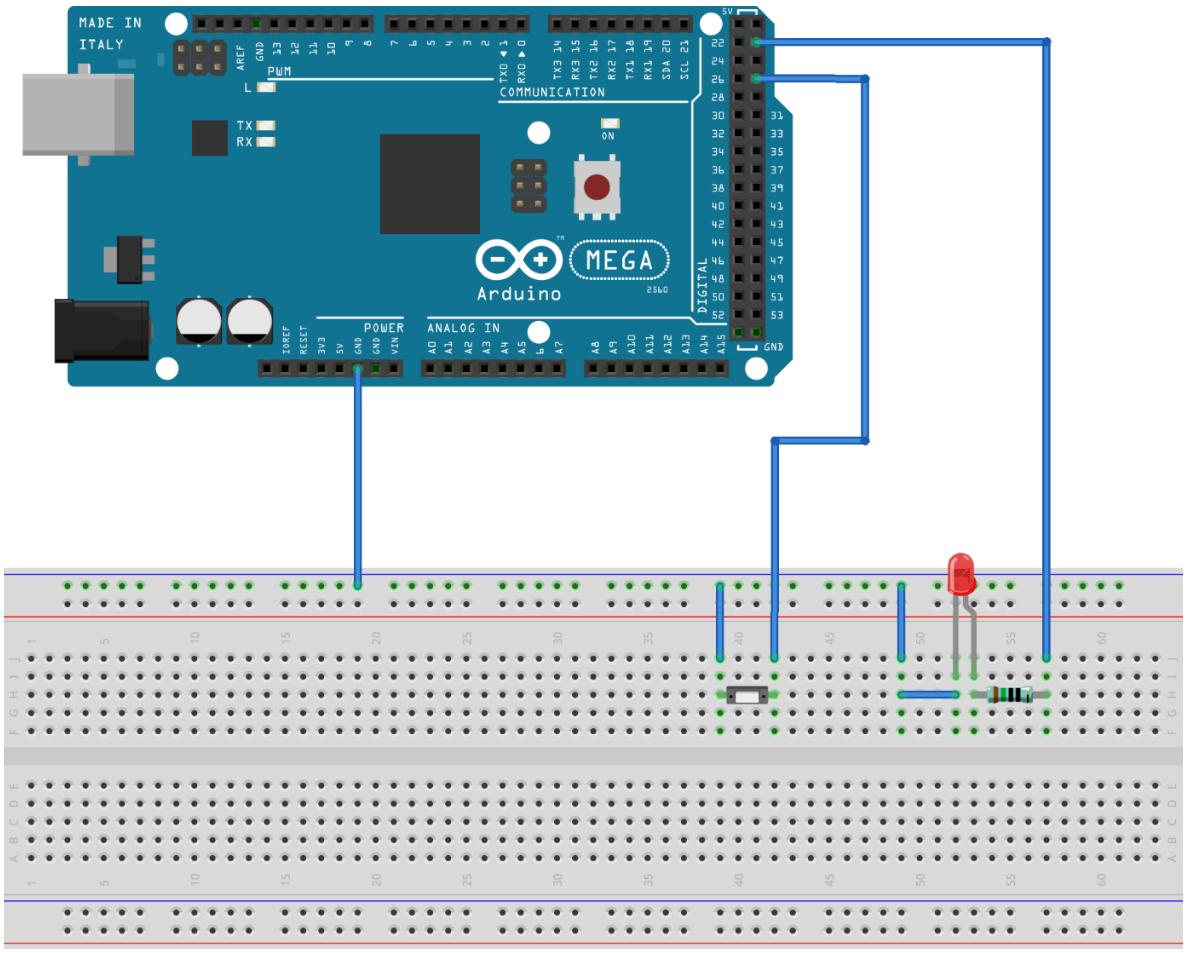


Fig. 3: Wiring example for an LED and a pushbutton.

The software

In this part, we will take an in-depth look into the software parts. The two different software parts use two different programming languages. If we are talking about the python code, the language is obvious, and the Arduino code is written in a heavily modified version of C++.

In-depth look into the python code

The python code handles most of the heavy work. It is composed from several python files, which all integrate into a main loop.

Config files

Several settings are stored in external files, which have the .yml file extension. These are YAML files and the decoding of them is handled by the pyYaml module. This allows the user (or developer) to change a few options without having to deal with the code itself. It is worth noting that while the config files can be modified while running the software, the code itself only reads most of the values on startup, which means that any option change should always end with restarting the program itself.

Problems with the config files

If the user decides to change something in *config.yml* to a value that is not valid, or the user inputs an invalid syntax into *keybinding.yml* and/or changes something, which is not recognized in the code, the software will exit with an error message. When changing any of the config files, it is a good habit to create backups if something fails. It is worth noting that the default config files can be redownloaded from the github repo.

Telemetry reading

The telemetry data is supplied by Funbit's telemetry server. It has a JSON output on a local server, which can be accessed via a local URL. The data reading is achieved with opening the URL and parsing it. The data updates faster than the python code runs. The parsed JSON file is stored in a local variable.

Data handling

The incoming data is handled by the classes found in *inputhandling.py*. This is for the ease of accessing the data through the software. Every single key has its own line that can be called. Since the data is in plain text at this point, the keyboard handling module can be changed relatively easily.

Outputs

The python code reads the handled data constantly. If the code detects a change in any of the values, it will send a message through the established USB interface between the Arduino board and the computer. This message is always unique and designed in a way that it can be read by humans. The code has built-in stops, which prevent the code from spitting out messages over the serial bus. The messages going over the serial bus are encoded, because the Arduino does not use the same serial communication protocol as python. These messages can be sent in every code cycle, so the only limitation on the speed is the cycle frequency and the baud rate. An in depth-look into the communication is available on page 13.

Inputs

Similar to the sent codes, the Arduino board can also send codes to the computer. These codes need to be decoded too, as mentioned before. The decoded codes are stripped from the junk characters by a partitioning function and are read into an internal variable. The codes received are also readable by humans.

How does the game detect this?

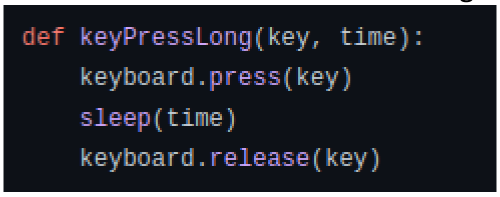
Unfortunately, the easiest way to get the game to process this data was keyboard emulation. The emulation is handled by python's keyboard module. The keybindings for this module can be set in the *keybinding.yml* file. A guide to set up these keybindings can be found in the keys section (page 6). **All inputs can be turned off by setting the inputsEnabled variable in config.yml to false.**
There are three types of key pressing emulation utilized by the software:

1. **The press_and_release command**

   The keyboard.press_and_release command does exactly what the name refers to. It hits a key on the emulated keyboard for a very short amount of time. This is used by inputs which are controlled by buttons. In these cases, the code does not care about if the button is held, nor when it's released. To put it simply, the code part runs when the button is pushed, or when the signal goes from HIGH to LOW (keep in mind that the inputs are HIGH by default and go LOW when the button is pressed or held down). For example, this command handles the axles. This is fine for most of the simple functions, although in some cases, the very short time delay between the press and release of the key can result in the game not detecting it.

2. **The keyLongPress function**

   The keyLongPress function is a modified version of the press_and_release command, with the addition of a time parameter. This is needed in cases where the simulator does not detect keypresses if they are too short. This is hard-coded in ETS2 game engine for the settings which have different detent options apart from their basic On-Off toggles; such as Wiper, Retarder, Light and Ignition. For example, this function is used with a 0.05 second delay when handling the individual retarder lever detents. The signal detection is the same as the press_and_release command, the only difference is there's a tiny delay in between.
   The function looks like the following:



```
def keyPressLong(key, time):
    keyboard.press(key)
    sleep(time)
    keyboard.release(key)
```

3. **Two-state (Momentary) keys**

   Some functions in-game require keys to be held down. An example for this is the suspension adjusting system. In these cases, the Arduino board sends a message both when the key is pressed and when it is released. The software, similar to this, has both a press and a release function. In these cases, there is no key release command until the user releases the button. It is important to know here, that a disconnect may temporarily cause your key to be stuck in the down position, but the next cycle the software will exit with an error message. In very extreme cases (caused by a damaged cable or high levels of electrical noise) the release message coming from the Arduino board can be lost. If this happens, and the software does not exit automatically, the key can be stuck in the down position. This can be solved by pressing the same button on your switchboard again.

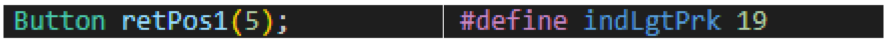
Different python files

For ease of programming, the software's code is split in between a few python files. These contain functions according to their names and are referred to in the import section of the main code. The files are the following:

- *main.py*: Contains the main code. This file needs to be run when running the software from PowerShell.
- *firstrun.py*: Contains the setup function. This will run if the *firstRun* variable in *config.yml* is set to true (default when downloading). After the code in this file runs, the variable will be set to false. Note: The variable will be set to true again if an error occurs while running the code and you answer *y* to run the setup again.
- *inputhandling.py*: Contains the classes that store the input keys read from *keybinding.yml*

In-depth look into the Arduino code

The software was originally tested on an Arduino MEGA board but it should not have any problems running on any other board which has the required amount of pins. The Arduino sketch file is written in a way that every pin number can be changed by the user through the use of the define and button functions.



```
Button retPos1(5);
#define indLgtPrk 19
```

The numbers in these functions refer to the allocated pin number. Change these numbers if you want to change the pinout.

As mentioned before, the python code does the heavy lifting, and as so, the Arduino board is only doing two basic tasks. The first of which is checking inputs and sending messages through the USB interface regarding those inputs. The second one is receiving messages through the Serial Bus and adjusting its outputs.

Arduino input handling

The inputs set on the Arduino board by the Button function are treated as INPUT_PULLUP pins. This means the common pins need to be connected to GND on the board. The INPUT_PULLUP pin mode activates a builtin pullup resistor, resulting in a much more stable input signal. This also means that the default stance of the input pins is HIGH and goes LOW when the button is pressed, or a switch is turned on. This information is all handled by the Button library, which makes coding a lot easier.

The Button library provides the code with basic debouncing features for the inputs, as well as a singular signal sent when the button or switch is activated or deactivated. These are the most used functions in the code, called with the .pressed() and .released() commands.

The inputs given prompt the Arduino code to send messages over the USB interfaces to the computer. The job of the Arduino board regarding the inputs ends here.

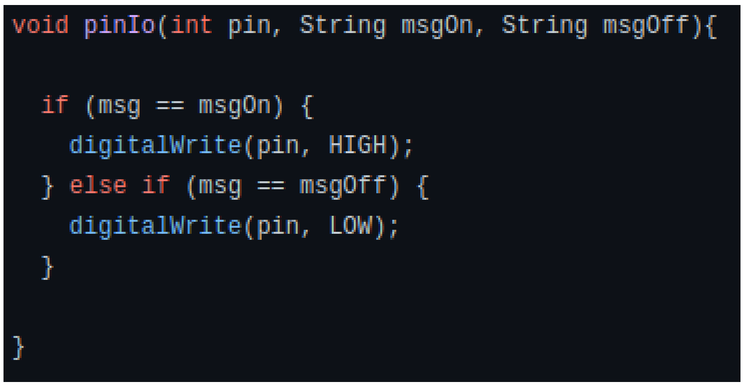
Arduino output handling

The OUTPUTS on an Arduino board can easily power a few LEDs without the need of an external power source. As such, the LEDs can be wired directly to the board, if needed.

The output handling consists of three basic parts. The Arduino only needs to compare the message to the predefined message list, and does not need to perform any logic regarding the outputs. It simply acts according to the received messages. A consequence of this is that you can manually test every output message by using the Arduino IDE serial monitor tool, and sending messages manually to the board.

The pinIO module

The pinIO module is used to shorten the code and avoid repetitive coding. It is a very simple block, which handles basic ON/OFF operations on outputs.



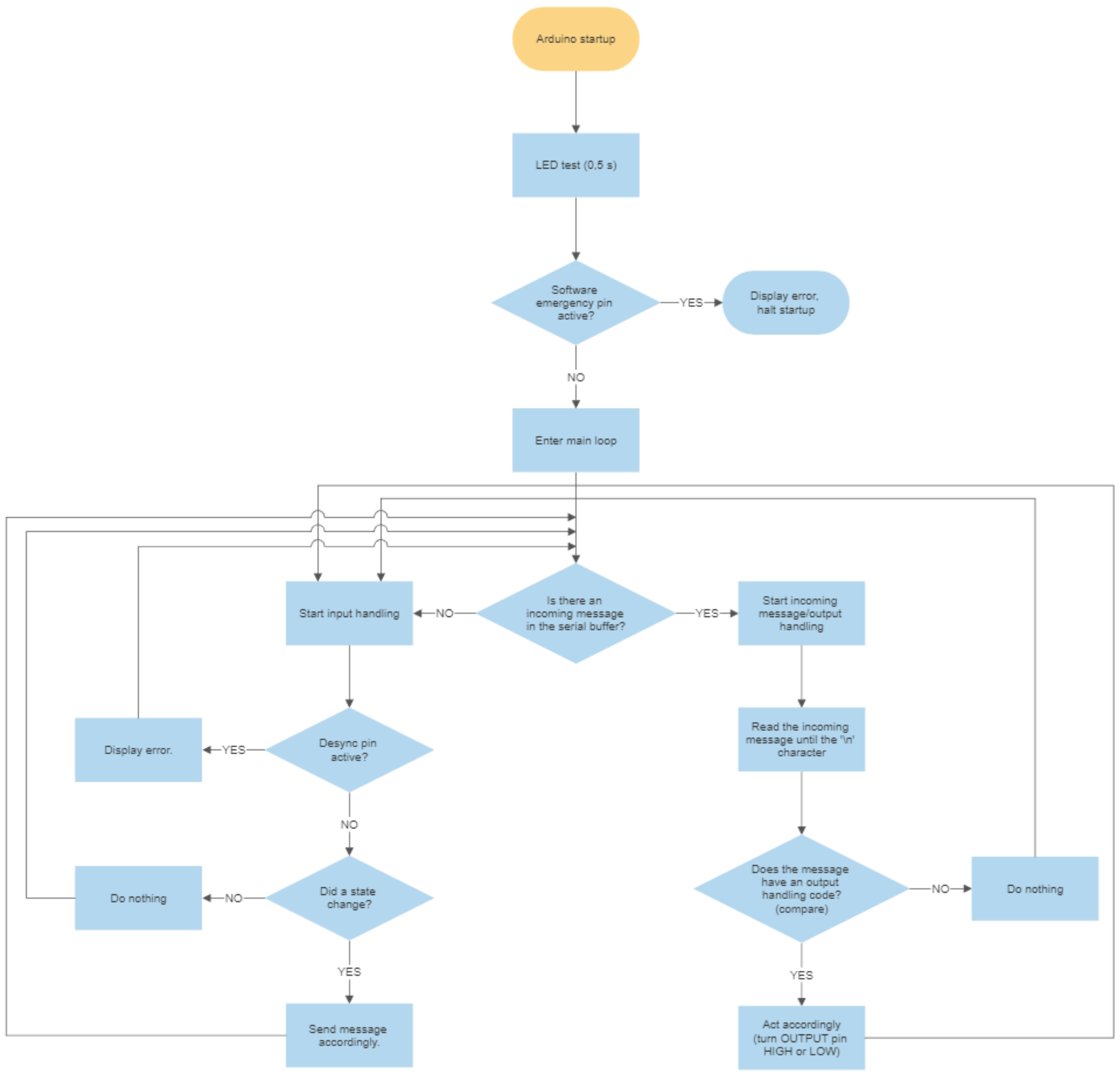
```
void pinIo(int pin, String msgOn, String msgOff){

  if (msg == msgOn) {
    digitalWrite(pin, HIGH);
  } else if (msg == msgOff) {
    digitalWrite(pin, LOW);
  }

}
```

It reads the global msg variable, and compares it to the msgOn and msgOff parameters given. If it matches either the msgOn or the msgOff parameter, it will turn the output set by the pin parameter either HIGH (on) or LOW (off).

Arduino code flowchart


Arduino startup
LED test (0,5 s)
Software emergency pin active?
YES
Display error, halt startup
NO
Enter main loop
Is there an incoming message in the serial buffer?
NO
Start input handling
YES
Start incoming message/output handling
Desync pin active?
YES
Display error.
NO
Did a state change?
NO
Do nothing
YES
Send message accordingly.
Read the incoming message until the '\n' character
Does the message have an output handling code? (compare)
NO
Do nothing
YES
Act accordingly (turn OUTPUT pin HIGH or LOW)

Communication

The main part of this project is the python and the Arduino communicating with each other. This utilizes the serial communication via USB. The communication code part is built from scratch.

How does serial communication work?

Serial communication sends data by deconstructing it to bytes and sending them bit-by-bit. Both the computer and the Arduino board have two serial buffers. An incoming and an outgoing one. The outgoing serial buffers store data that needs to be sent over the wire, until everything is sent. The bytes are then stored into the incoming serial buffers.

The incoming buffers have an important property: You can check if there is data in them (on the Arduino, you can even check how many characters are stored in the buffer currently). This allows us to create interlocks in the software, so it finishes reading the data before transmitting anything.

The rate the bytes are transmitted at is called the baud rate (bit/s). This needs to match on both devices. If the baud rates do not match, all transmitted info will be garbage bytes. The software uses a baud rate of 9600 by default.

Establishing connection

Before any message can be sent, the Arduino needs to connect to the computer and vice versa. This is done by accessing the serial port both on the computer and on the board.

Establishing connection in python

```
try:
    arduino = serial.Serial(port=port, baudrate=baudRate, timeout=.1)
except Exception as e2:
    print('Connection failed!')
    debugprint(e2)
    askForSetup()
```

The python code establishes a connection using the Serial function of the pyserial module. The port, baud rate and timeout parameters need to be set, although there are other parameters to this function. This function needs to be assigned to a variable, which can be referred to when transmitting messages, or doing anything with the serial device. The code module features basic error handling.

The parameters for this function are in the *config.yml* file and are set during the setup process.

Establishing connection in Arduino code

The Arduino coding allows you to simply initialize the serial communication with one single line.

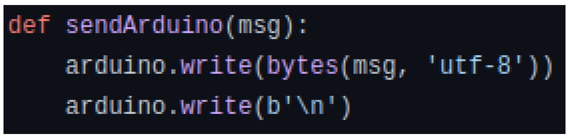
```
Serial.begin(9600);
delay(50);
```

Serial.begin() has one argument, which is the baud rate. The following delay always has to be equal or greater than 50, as the board needs time to actually start the serial module.

Encoding

The serial communication works perfectly without any encoding between two Arduino boards. There is a problem when trying to communicate with the python software though. The Arduino's Serial command uses UTF-8 encoding by default, which means that any message sent directly with pyserial will show up as garbage bytes for the Arduino. This means that the communication needs to use some kind of encoding.

The python function which handles the messages sent to Arduino is the following (the Arduino variable defined in the serial starting code module is used here):



```
def sendArduino(msg):
    arduino.write(bytes(msg, 'utf-8'))
    arduino.write(b'\n')
```

The message gets encoded to bytes as utf-8 and the code adds a '`\n`' character to the end of the message. This character is what the Arduino is looking for when reading the message, as it always means the end of an incoming string. The '`\n`' character needs to be encoded as bytes, too.

Encoding on the Arduino is included when using the *Serial.println()* command. This means that the Arduino does not need any encoding code written.

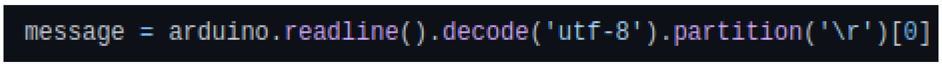
The problem with Serial.println()

The built in *Serial.println()* command suggests that the message sent will be ended with a new line character ('`\n`'). This is not exactly true. While on the serial monitor it shows up as a new line, and other boards can read the sent code fine, there is a hidden '`\r`' (carriage return) character before '\n'. This needs to be accounted for in the python code.

Decoding

As mentioned before, messages need to be decoded before doing anything with them. In this project, the term 'decoding' also means removing the unnecessary characters from the end of the message.

Decoding in python



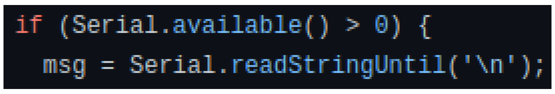
```
message = arduino.readline().decode('utf-8').partition('\r')[0]
```

This single line handles all the decoding needed in python, including the utf-8 and the removal of the '`\r`' and '`\n`' character. This is done by the partitioning function. A raw incoming but decoded message would look like this:

`<message>\r\n`

The partition function splits this into three parts: `<message>`, `\r` and `\n` these are referred to as `0`, `1`, and `2` partitions. The code needs to use the first part, so it's defined with `[0]`. After the process is done, the message variable will contain the readable incoming string used in the software.

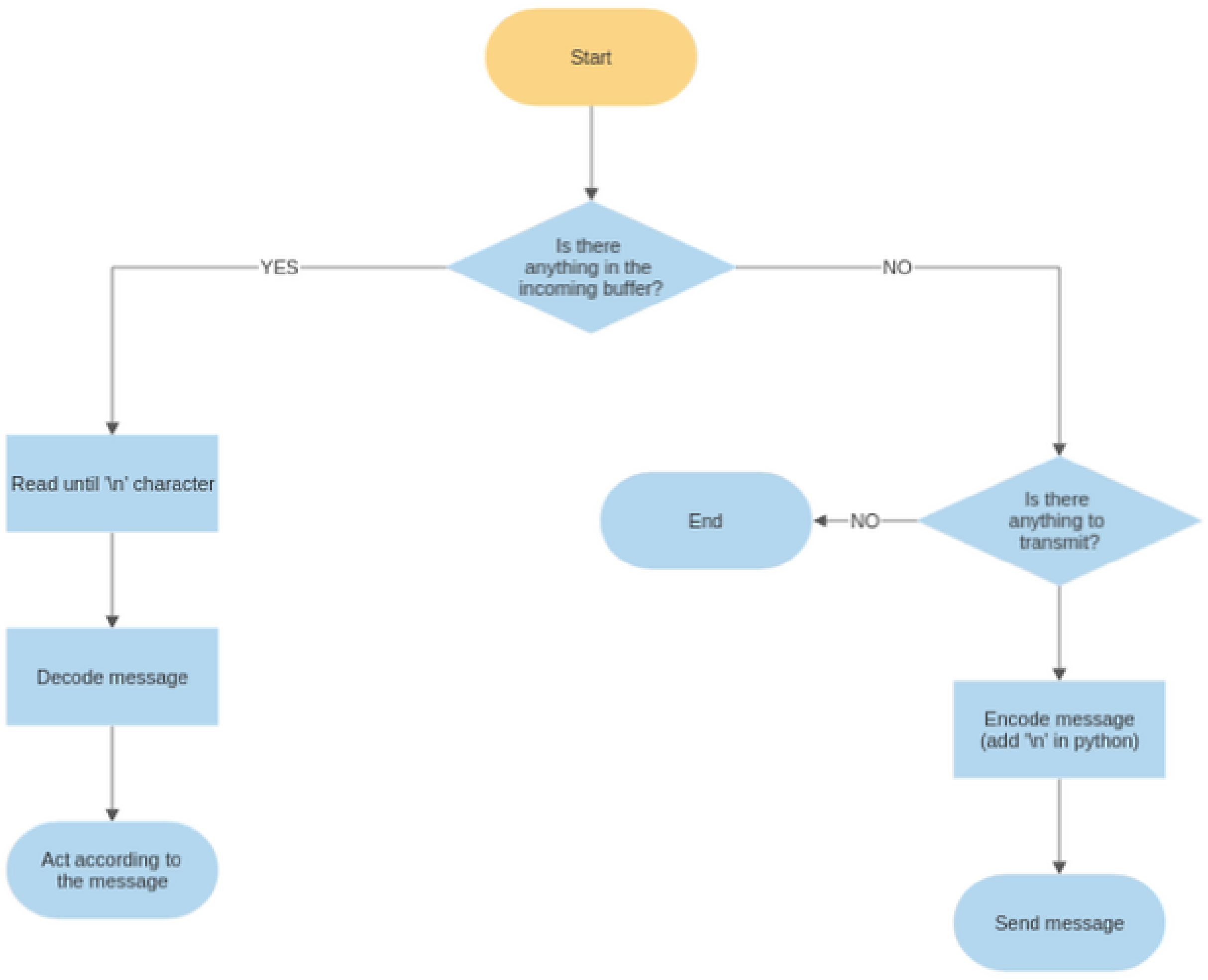
Decoding with Arduino



```
if (Serial.available() > 0) {
  msg = Serial.readStringUntil('\n');
```

The decoding process with Arduino is way easier. The board just needs to read the message until the '`\n`' character and store that in a variable. This is achieved with the code above.

Serial Communication Flowchart

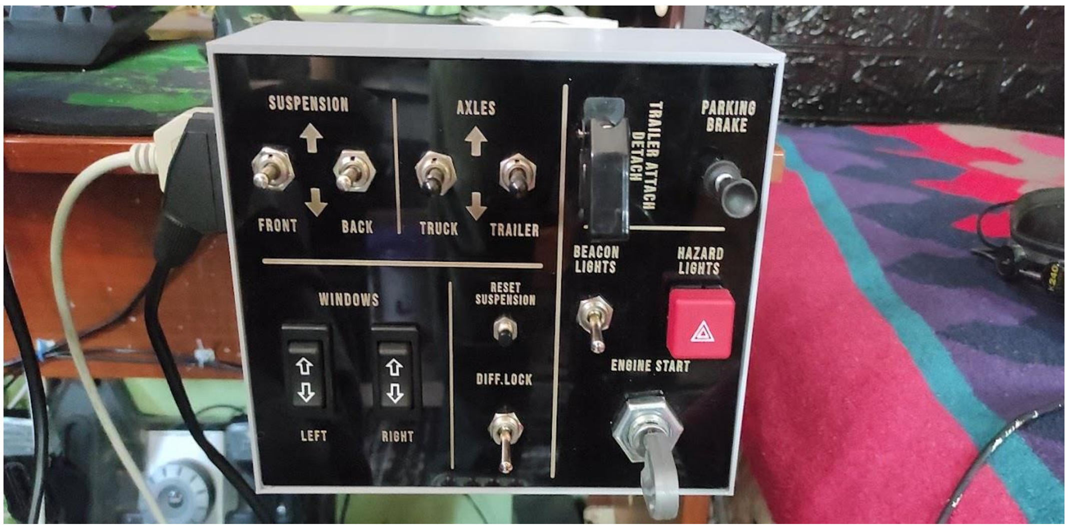

Start
Is there anything in the incoming buffer?
YES
NO
Read until '\n' character
Decode message
Act according to the message
Is there anything to transmit?
NO
End
Encode message (add '\n' in python)
Send message

Hardware

This project was actually thought of as only a switchbox first because after doing a deep research all over the internet, I could not find a similar project that merges inputs & outputs into a single one. Upon meeting csongoose on Reddit we discussed the possibilities and decided upon the functions that shall be introduced.

The hardware actually consists of 3 different modules that are modular: switchbox, wheel panel (stalk panel) and dashboard respectively. I'll go through the details one by one. Note that these are subject to change according to your own projects, this is just for general knowledge about what's going on.

A) **Switchbox:**
The main panel, housing the Mega board, along with some 2- or 3-way heavy duty momentary and permanent switches along with buttons and some real-car parts such as a hazard button and an ignition switch. I created a custom design on Illustrator, laser-cut and engraved the front panel, and used cables to make the connections. Main enclosure is also self-designed and 3D printed. It has 2 screw-type clamps behind, which can be removed if needed. The dashboard is connected directly with a SCART cable. The stalk panel is connected with a custom-designed cable hub (will be mentioned below) using a 25-Pin Serial Cable.



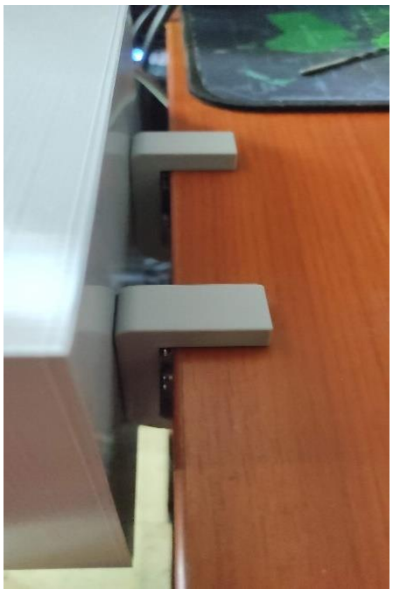
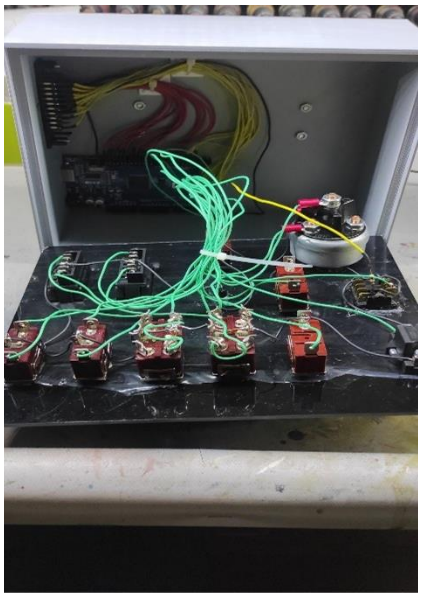


Fig. 4 (up): The main switchbox.

Fig. 5 (left): The clamps holding the panel to the table.

Fig. 6 (right): The insides of the switchbox.



B) **Wheel Panel (Stalk Panel):**
This is where the real magic is. Initially, I thought about using real car parts, but there were numerous handicaps such as the narrow space between the main body and the wheel itself, the inadequate surface area for such parts, the incompatible wheel hub size, etc. I needed to create a rather large surface to mount the parts, so I looked for and found a 3D-sketch of a Logitech G29 and designed a mould-type flat surface for the front side and 3D printed it. Using some super glue to fit small magnets on each side, I created a modular panel that can be removed if necessary. After that, I designed 3 different modules for the Retarder, Wiper and Signal/Light modules. For the retarder and the wiper, I used 12-way switches and some gearing to compensate for the turning angle of rotary switch itself. Then I used mini-MIC connectors and connected all of these 3 modules into a common hub. From there it connects to the switchbox with a serial cable. Hub is mounted under the desk using mini magnets as well.

The wiper switch has 4 detents (Off-Int-1-2), the retarder has 5 (Off-1-2-3-4). The indicator switch is a bit different. It has a left, a right, and a middle (off) position. At the end of the lever is a rotary switch for headlights (off-parking/marker lights-low beams). The lever also moves on the X axis. You push forward for High Beams and there is a spring-returned momentary position for high beam flashing if you pull it towards you (similar to those found on many present-day cars).

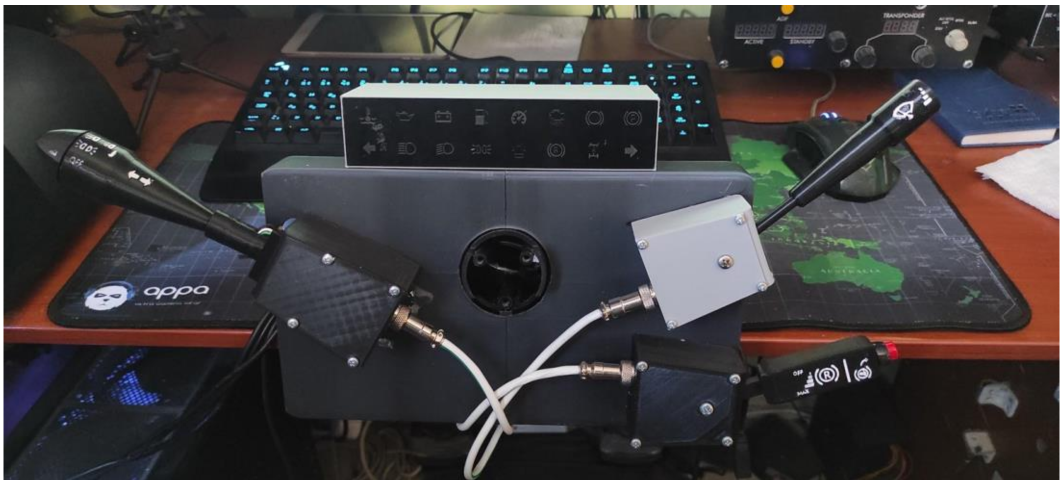
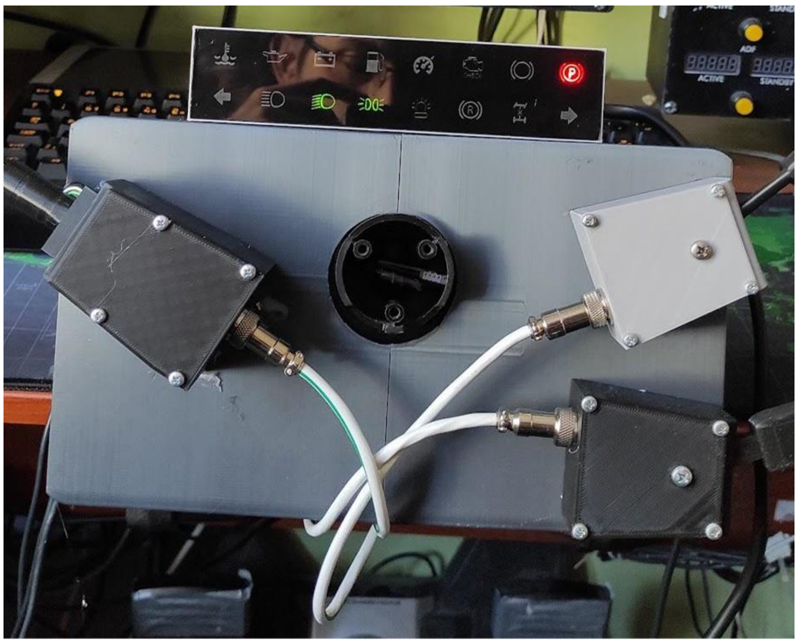


Fig 7 and 8: The stalk panel and the dashboard.



C) **Dashboard:**
Laser-engraved plexiglass panel with 16 different indicators. Each LED has a 150ohm resistor in series, and individually connected to the SCART connector. Box is 3D-printed and hot-glued to the wheel panel. This module is directly connected to the switchbox, as this was added after I designed everything for the input stuff.

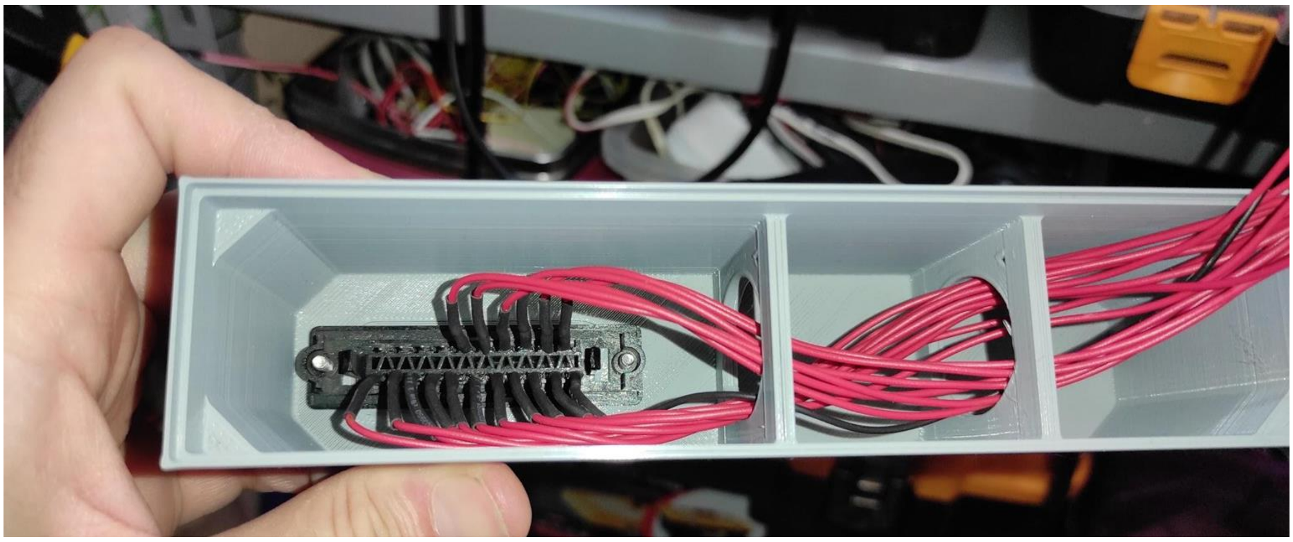
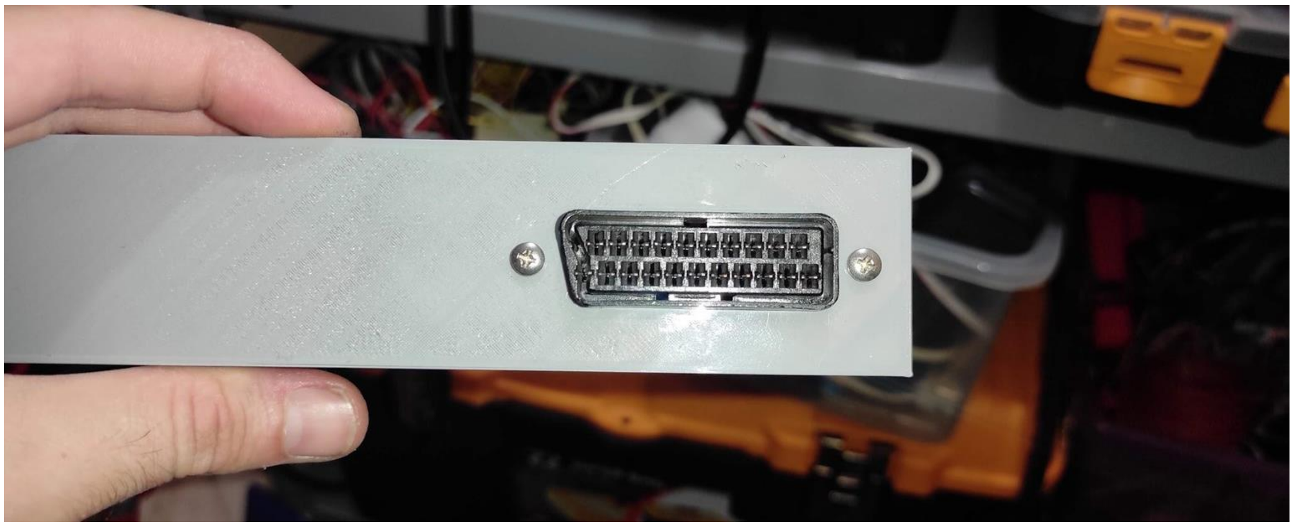
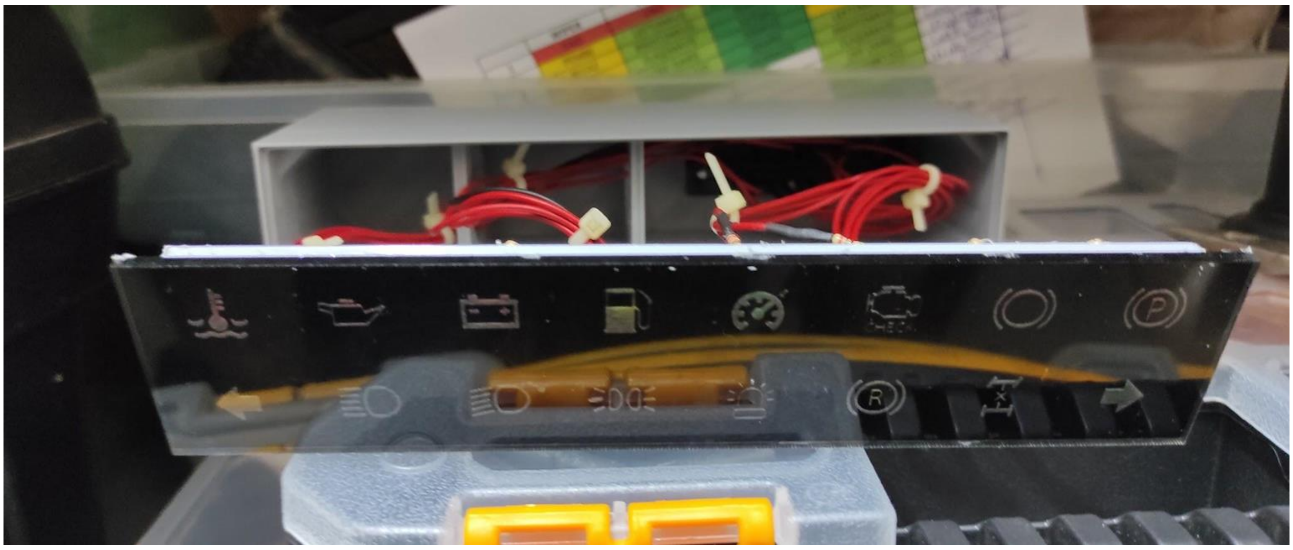




Fig 8, 9 and 10: The dashboard panel and its insides.

Editing the controls.sii file

It took some time to handle this, because it plays a key role in the rig being functional. There's a great tutorial here, which will help you through. Basically, even though we can use some of the already assigned keys as is by default (e.g diff.lock, flash, beacon light, hazards etc.), we'll eventually have to use combined inputs (key combinations) as nearly all individual buttons have a function assigned already.

As stated in the examples, if we use any modifier key such as ctrl, shift or alt with any other keys, we can NOT use them individually for other keybindings. This means if you use lshift & lctrl for shifting and want to use them for combinations, you'll have to sacrifice the shifting function. To put it simply, ctrl, alt and shift should not be used individually. Also you'll have to unassign the toggle functions of the assignments that you intend to use such as ignition, wipers, retarder and lights (e.g.: in ignitions case E), otherwise it will conflict with the manual ones you assign.

Next, we decide the combinations to be used. When these modifiers are used, there are many possibilities so you're basically free to choose any combination you want for your assignments.

The most important thing is that the common buttons (e.g F1-F2 etc, or numeric keys) are already used by the core functions of the game, so you'll have to define these separately as "no_modifier?0", otherwise game won't detect its main function.

For example, we already have camera change keybindings on 1-2-3-4-5 etc. and we want to define retarder lever as shift+1, shift+2, shift+3, shift+4 & shift+5. First, we have to define the camera keys as "non-modified" ones. Without doing this, the camera functions (or the individual presses of these keys whatever their assignment is) will NOT be detected. We'll have to do this first **(change any other assignments for the buttons that you are going to use in your combinations that are already assigned to something by default, like mirror settings etc.)!**

```
config_lines[191]: "mix cam1 `! keyboard.lshift?0 & keyboard.key1?0 | semantical.cam1?0`"
config_lines[192]: "mix cam2 `! keyboard.lshift?0 & keyboard.key2?0 | semantical.cam2?0`"
config_lines[193]: "mix cam3 `! keyboard.lshift?0 & keyboard.key3?0 | semantical.cam3?0`"
config_lines[194]: "mix cam4 `! keyboard.lshift?0 & keyboard.key4?0 | semantical.cam4?0`"
config_lines[195]: "mix cam5 `! keyboard.lshift?0 & keyboard.key5?0 | semantical.cam5?0`"
```

Camera keys defined as 'no lshift' – the '!' inverts the logic. (TRUE when NOT pressed)

We then decide which modifier we'll use and change the related config line as below. Note that shift, ctrl and alt buttons can be either separately or commonly assigned.

```
config_lines[330]: "mix retarder0 `keyboard.lshift?0 & keyboard.key1?0 | semantical.retarder0?0`"
config_lines[331]: "mix retarder1 `keyboard.lshift?0 & keyboard.key2?0 | semantical.retarder1?0`"
config_lines[332]: "mix retarder2 `keyboard.lshift?0 & keyboard.key3?0 | semantical.retarder2?0`"
config_lines[333]: "mix retarder3 `keyboard.lshift?0 & keyboard.key4?0 | semantical.retarder3?0`"
config_lines[334]: "mix retarder4 `keyboard.lshift?0 & keyboard.key5?0 | semantical.retarder4?0`"
```

Retarder keys defined with the lshift modifier.

Doing this, we tell the game that for retarder levels, we'll send a combined keyboard press command via Arduino to the game. You can simply test this in the game by pressing related buttons simultaneously.

It's vital to note that such modifier-based configurations will appear as "complex" in-game, which should not – under any conditions – be changed from the ingame settings, as they will corrupt your controls file, rendering the controls unusable. Decide everything beforehand, introduce them into *controls.sii* file and don't touch them in the game. It is also a good habit to make a backup of your *controls.sii* file both before editing, and after you are done. This way if anything goes wrong, you can just overwrite the *controls.sii* file with the backup.

Conclusion

TruControl – custom, while being similar to the TruControl project made before, is fundamentally different in both the intended goal and the tools used to get there. The whole project is made for having fun, and was completed by us in our free time. Many, many complications had to be overcome both in the code and hardware. This detailed documentation was made so that any devs wanting to make a similar project can more easily overcome issues that may come up in development.

In conclusion, I’d say we learned a lot. This whole project was made possible by the wonders of the web. The cooperation started on reddit and evolved into a full-blown development process. Even though I - the programmer - never saw the hardware in person, as we are several countries apart, we were still able to make a fully functioning simulator rig using remote testing and close cooperation. On that note, I’d like to thank Baran for providing me such an opportunity to test my skills and knowledge, and I’d also like to thank everyone who knowingly - or unknowingly contributed to the code. This includes some of my friends, and the many users of Github, Stackexchange, PyPI, Arduino forums, Reddit and so on, who were able to provide answers to questions either I, or another person before me asked.

It is possible, although unlikely that I will continue the development of this project. In the future, support will still be provided by either me or Baran. Don’t be afraid to contact us if anything comes up, either via email, reddit or twitter.

Special thanks go to the people who made the modules I used to make the code work. This includes Funbit, dakar2008, madleech, and the many contributors of the python modules.

Special thanks also goes to my Father, who got me into electronics when I was about 5 years old, and this guy, Devon Crawford, whose project eventually leaded to me buying an Arduino about 3 years ago.

This was a fun project to make, a fun journey.

Best regards,
csongoose

Default pinout chart

This does not contain all information. You can access the whole chart here.

Pin number	Function	Type	Software tag	Part type	Location	Note	Physical location	Buttons to Emulate	Has on/off state
0	*reserved*	-	serial0	-	-	-	-	-	-
1	*reserved*	-	serial1	-	-	-	-	-	-
2	engine brake	INPUT_PULLUP	engBrk	Button	Retarder lever tip	IC-191		B	•
3	retarder pos3	INPUT_PULLUP	retPos3				Steering wheel	shift + 4	
4	retarder pos2	INPUT_PULLUP	retPos2					shift + 3	
5	retarder pos1	INPUT_PULLUP	retPos1	Rotary	Retarder lever (right side)	1X12 ROTARY		shift + 2	
6	retarder off	INPUT_PULLUP	retOff					shift + 1	
8	retarder pos4	INPUT_PULLUP	retPos4					shift + 5	
9	low beam	INPUT_PULLUP	lgtLow					ctrl + l	
10	parking lights	INPUT_PULLUP	lgtPrk	Rotary	Multi-switch lever (left side)	1X12 ROTARY		shift + l	
11	lights off	INPUT_PULLUP	lgtOff					L	
12	high beam & flash	INPUT_PULLUP	lgtHgh	Switch	Multi-switch lever	IC-142		K	
13	blinkers right	INPUT_PULLUP	blnRgt	Switch	Multi-switch lever	IC-143		shift+q	•
14	blinkers left	INPUT_PULLUP	blnLft	Switch	(left side)	IC-143		q	•
15	wipers off	INPUT_PULLUP	wipOff					p	
16	wipers spd1	INPUT_PULLUP	wipSpd1	Rotary	Wiper lever (right side)	1X12 ROTARY		ctrl + p	
17	wipers spd2	INPUT_PULLUP	wipSpd2					shift + p	
18	wipers spd3	INPUT_PULLUP	wipSpd3					alt + p	
19	Parking lights indicator	OUTPUT	indLgtPrk	Green LED	Standalone			*Indicator*	
20	Low beams indicator	OUTPUT	indLgtLow	Green LED	Standalone			*Indicator*	
21	High beams indicator	OUTPUT	indLgtHgh	Blue LED	Standalone			*Indicator*	
22	Left blinker indicator	OUTPUT	indBlnLft	Orange LED	Standalone			*Indicator*	
23	Right blinker indicator	OUTPUT	indBlnRgt	Orange LED	Standalone			*Indicator*	
24	Beacon indicator	OUTPUT	indLgtBcn	Orange LED	Standalone			*Indicator*	
25	Diff lock indicator	OUTPUT	indDifLoc	Orange LED	Standalone			*Indicator*	
26	Handbrake indicator	OUTPUT	indHndBrk	Red LED	Standalone			*Indicator*	
27	Battery indicator	OUTPUT	indNoChrg	Red LED	Standalone	5mm LED	Dash	*Indicator*	
28	Oil indicator	OUTPUT	indNoOilp	Red LED	Standalone			*Indicator*	
29	Hazard light indicator	OUTPUT	indLgtHzd	Orange LED	Standalone			*Indicator*	
30	Air pressure low indicator	OUTPUT	indAirLow	Red LED	Standalone			*Indicator*	
31	Fuel low indicator	OUTPUT	indFueLow	Orange LED	Standalone			*Indicator*	
32	Check engine indicator	OUTPUT	indChcEng	Orange LED	Standalone			*Indicator*	
33	Retarder indicator	OUTPUT	indRetard	White LED	Standalone			*Indicator*	
34	Cruise Control indicator	OUTPUT	indCruise	Green LED	Standalone			*Indicator*	
35	Water Temp high indicator	OUTPUT		Red LED	Standalone			*Indicator*	
36	Ignition/pre-heat	INPUT_PULLUP	elcIgn					ctrl + e	
37	Engine start	INPUT_PULLUP	engSrt					shift + e	
38	Trailer attach	INPUT_PULLUP	trdWhl	Switch	Standalone	IC-152		T	
39	Beacon	INPUT_PULLUP	lgtBcn	Switch	Standalone	IC-152		O	
40	Hazards	INPUT_PULLUP	lgtHzd	Latching Button	Standalone	TOFAS		F	
41	Differential lock	INPUT_PULLUP	difLoc	Switch	Standalone	IC-152		V	
42	Handbrake	INPUT_PULLUP	hndBrk	Switch	Standalone	ASW-05		Space	
43	Left window down	INPUT_PULLUP	winLdn	Button	Standalone	PMX AN0020		ctrl + u	•
44	Left window up	INPUT_PULLUP	winLup					alt + u	•
45	Right window down	INPUT_PULLUP	winRdn	Button	Standalone	PMX AN0020		ctrl + k	•
46	Right window up	INPUT_PULLUP	winRup					alt + k	•
47	Front Suspension Down	INPUT_PULLUP	susFrtUp	Mom.Switch	Standalone	IC-160		alt + g	•
48	Front Suspension up	INPUT_PULLUP	susFrtDn	Mom.Switch	Standalone			shift + g	•
49	Rear Suspension Down	INPUT_PULLUP	susBckUp	Mom.Switch	Standalone	IC-160		alt + v	•
50	Rear Suspension Up	INPUT_PULLUP	susBckDn	Mom.Switch	Standalone			shift + v	•
51	Lift/Drop Truck Axle	INPUT_PULLUP	truAxl	Mom.Switch	Standalone	IC-160		y	
52	Lift/Drop Trailer Axle	INPUT_PULLUP	trlAxl	Mom.Switch	Standalone	IC-160		shift + y	
53	Reset suspension	INPUT_PULLUP	susReset	Button	Standalone	Button		ctrl + g	
54/A0	Desync/failsafe switch	INPUT_PULLUP	deSync	Switch/Button	Standalone		Not emulated (only used in the arduino software)		
7	software emergency	INPUT_PULLUP	emergency	-	Internal	-	Internal	-	-