K-Line (ISO 9141) Shift Indicator

A project by Ravi Shah

Overview

Many track-focused cars have complex gauge clusters with a wide range of information for the driver. One device that is commonly found in such cars is a gear-shift indicator, which is comprised of a set of lights mounted on the dashboard or steering wheel, directly in the driver's line of sight. These lights illuminate based on the engine speed (RPM) of the car and are extremely useful for knowing when to shift gears in high-pressure situations.

Figure 1.1 - Formula One steering wheel with sequential shift indicator [1]

Although my 1996 Mazda Miata is far from a track car, I wanted to add a shift indicator to my dashboard (maybe it'll actually be useful on the backroads). However, every product I found online was either incompatible with my Miata's older K-Line (ISO 9141) protocol or required splicing the tachometer wires, which I wasn't willing to do. So, I decided to design my own.

After much prototyping and debugging, I developed a K-Line shift indicator, with a custom PCB and full plug-and-play functionality as tested on my Miata.

Requirements

I set the following requirements for the device:

- Must display engine RPM using a set of seven lights
- Must have three distinct colors to show progression from low to high RPM
- Must begin illuminating at the lower end of shift range (3.5k) and fully illuminate by the higher end (6k), with equal steps in between
- Must mount to the dashboard without obstructing the driver's view
- Must not require any modifications to wiring/electronics of the car (i.e. must be plug-and-play)

Preliminary Design

Initial Sketch

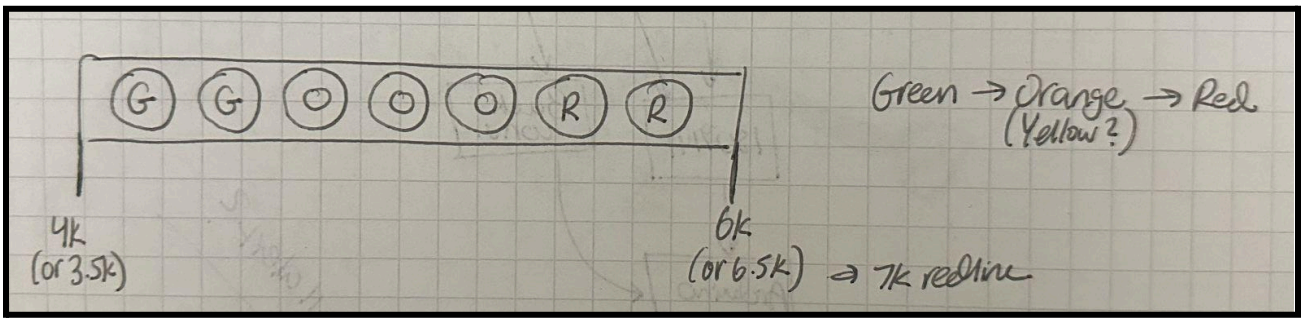


Figure 3.1.1 - Sketch of device display and LED colors

I initially decided to use the color sequence in Figure 3.1.1, with two green, three orange/yellow, and 2 red LEDs, as this is a common pattern in many commercially available devices. Note: the orange/yellow was replaced by blue in the final product for better contrast and daytime visibility.

Component Selection

At this stage, I had to make some critical decisions regarding the components used to make this device. Following initial research, I found a few LIN (Local Interconnect Network) bus transceivers that could communicate with a car's ECU (Electronic Control Unit) using the ISO 9141 protocol. Of these, I selected the TI SN65HVDA195 because of its low power usage, extensive documentation, and compatibility with microcontrollers like the Arduino Uno [2]. I chose an Arduino microcontroller board due to the integrated serial (and software serial) port, along with its ability to drive

several LEDs [3]. Additionally, a buck converter was needed to step down the car battery voltage (~12.6V) to an acceptable 5V for the Arduino. Finally, an OBDII harness was needed to make power (battery & ground) and data (ISO 9141 K-Line) connections between the device and the ECU.

Logic Flow

During operation, the device functions according to the logic diagram in Figure 3.3.1. The SN65HVDA195 transceiver consistently polls and receives RPM data from the LIN bus, which it sends to the Arduino, illuminating the relevant LED pattern.

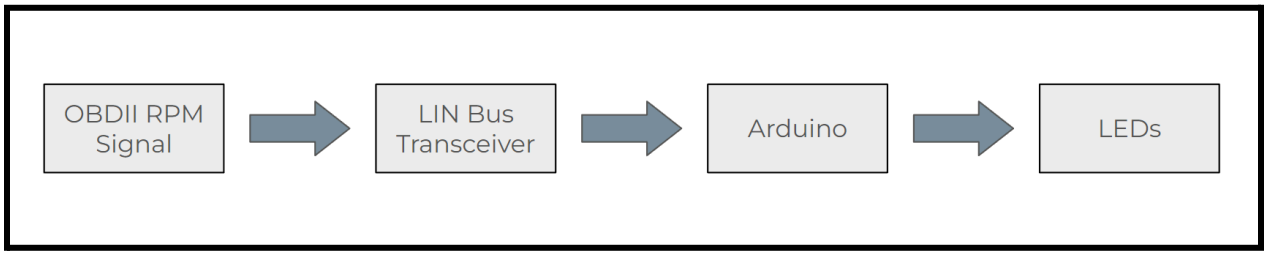


Figure 3.3.1 - Logic diagram for the K-Line Shift Indicator

Fritzing (Breadboard)

Once the components were selected, I designed the prototype's breadboard layout using Fritzing. The resistor, diode, and capacitor selections were made according to the typical application information in Figure 3.4.1 (provided by TI in the SN65HVDA195 datasheet).

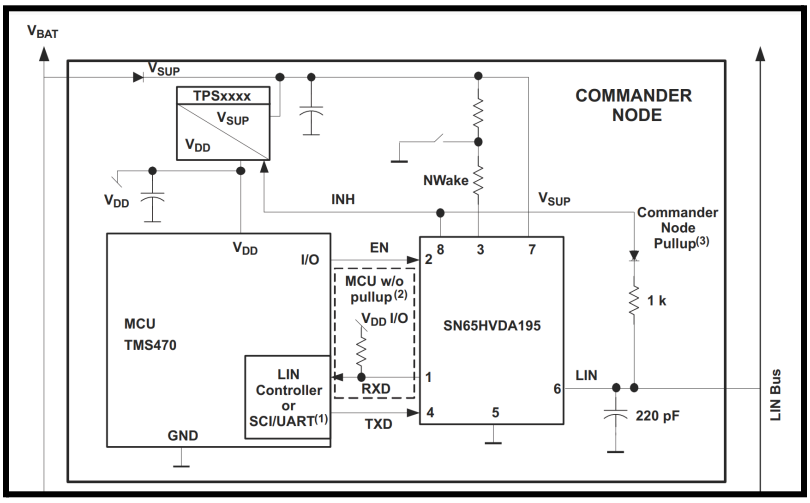


Figure 3.4.1 - Typical application diagram from TI datasheet [2]

I chose Arduino pins 8, 9, and 11 for the transceiver's RX (receive), TX (transmit), and EN (enable) pins, respectively. This ensured that the Arduino's software serial port (AltSoftSerial) would function properly [4]. OBDII pins 16 (battery voltage) and 4 (chassis ground) were split between the buck converter and the breadboard since the SN65HVDA195 IC operates on 12-14V [2]. Finally, the buck converter's output was connected to the Arduino's VIN (5V input) and GND pins as shown in Figure 3.4.2.

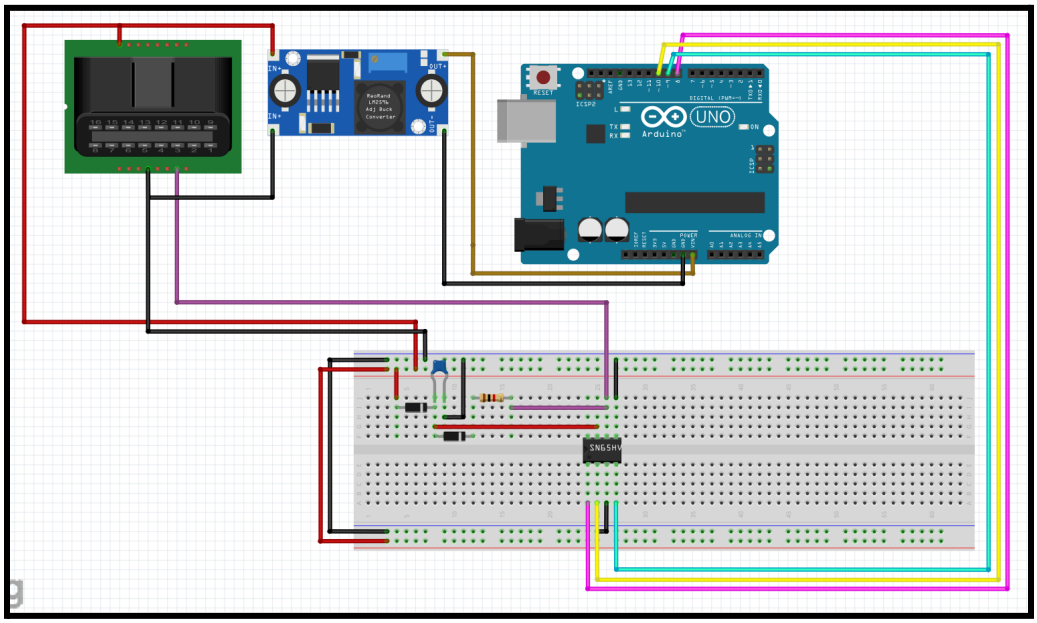


Figure 3.4.2 - Fritzing breadboard layout

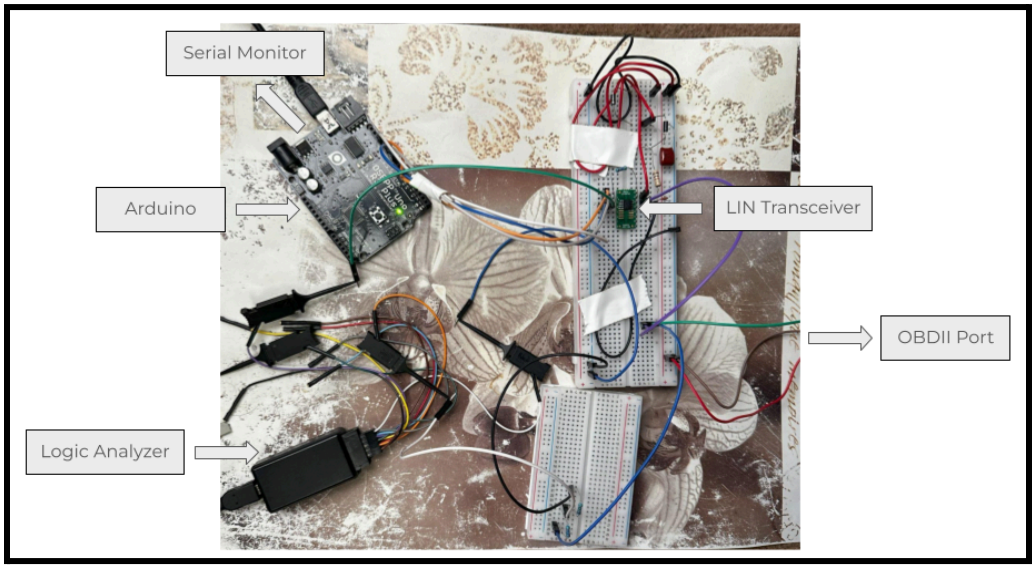


Figure 3.4.3 - Breadboard assembly for testing using Saleae logic analyzer

Final Designs

After testing basic functionality with the breadboard setup in Figure 3.4.3, I made a few changes for the final design of the device. Additionally, for the final product, I developed a custom PCB and enclosure that can be mounted on a dashboard.

Schematic

Based on the datasheet for the TI SN65HVDA195 LIN transceiver, along with some tweaks from breadboard testing, I used KiCad design software to create an electrical schematic for the device. As shown in Figure 4.1.1, it contains connection points for both the OBDII port and Arduino, along with a set of LEDs and a 200mA fuse for overcurrent protection.

This was my first time using KiCad, and I learned a lot throughout the process by following several guides [5] and teaching myself to use its extensive toolset.

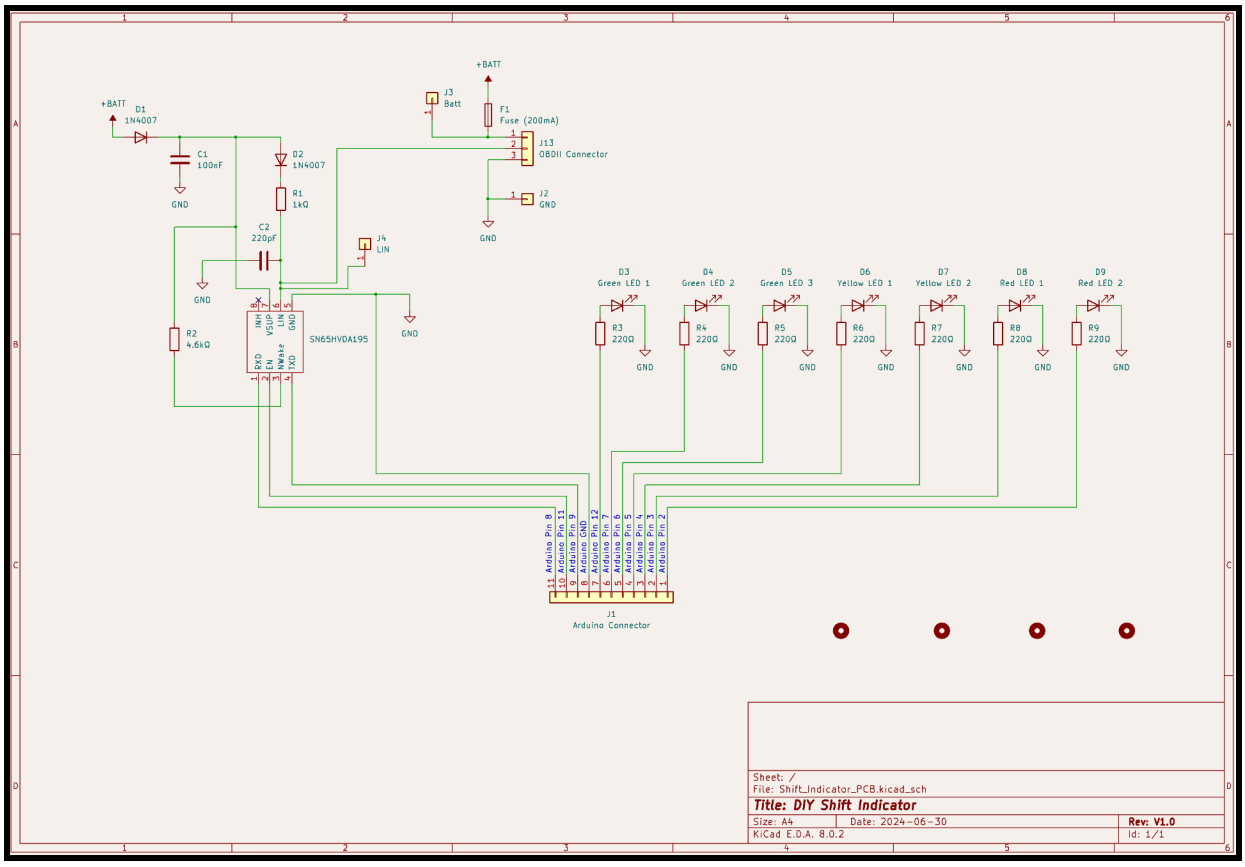


Figure 4.1.1 - KiCad Schematic for PCB with LIN transceiver and LEDs

PCB

To design the PCB, I assigned footprints to each component and exported the schematic netlist. I created a 2-layer PCB as shown in Figure 4.2.1, incorporating all the functionality of the breadboard design in a much more compact package. For the SN65HVDA195 transceiver, I followed TI's guidelines for traces and vias [2] to minimize total loop inductance.

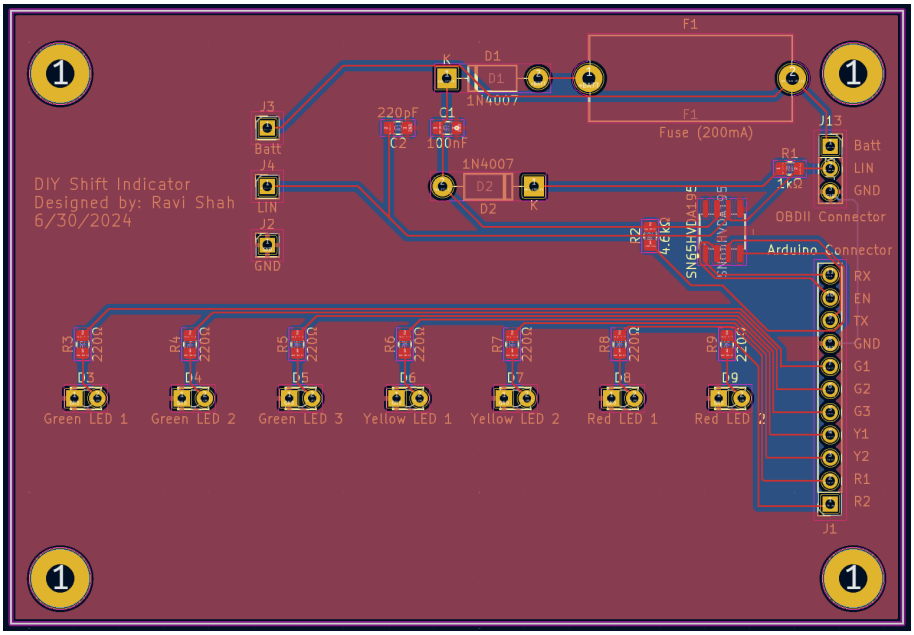


Figure 4.2.1 - KiCad PCB design based on schematic

After finalizing the PCB design, I generated Gerber files and uploaded them to PCBWay for fabrication. The bare PCB arrived as shown in Figure 4.2.2.

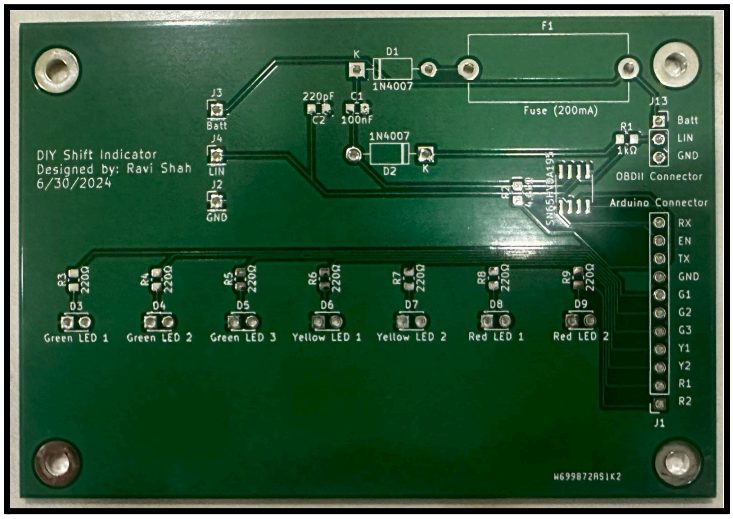
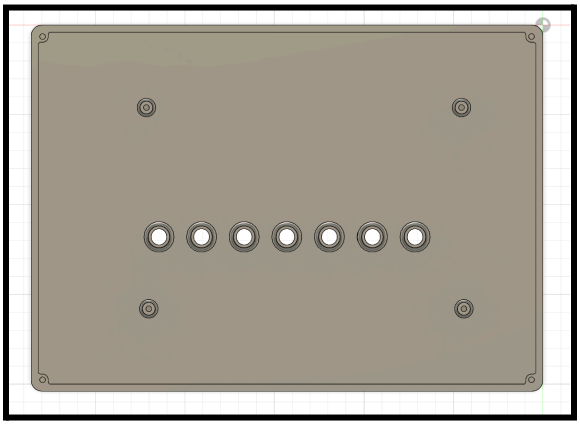


Figure 4.2.2 - PCB fabricated by PCBWay

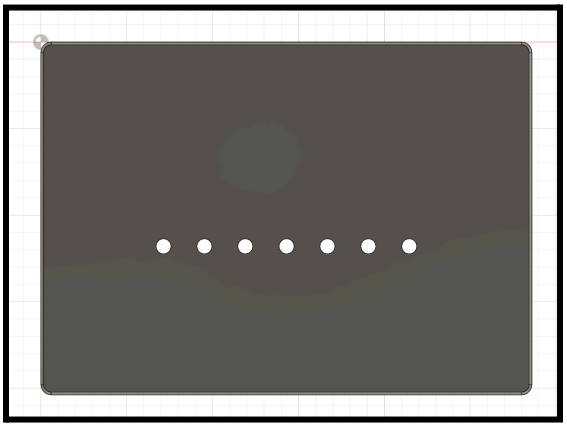
Enclosure

I designed an enclosure to house the PCB, Arduino, and buck converter. Using Autodesk Fusion, I created both the enclosure (with standoffs and LED channels) and the lid, which attaches using M2 screws. Figure 4.3.1 shows several views of the enclosure and lid design.

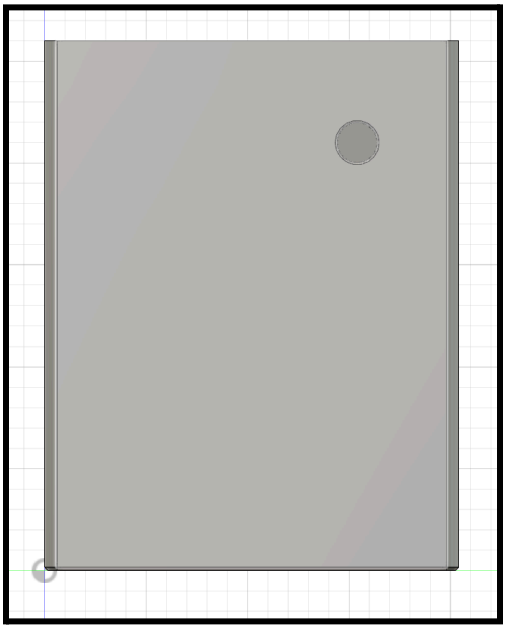
Back view (inside of enclosure)



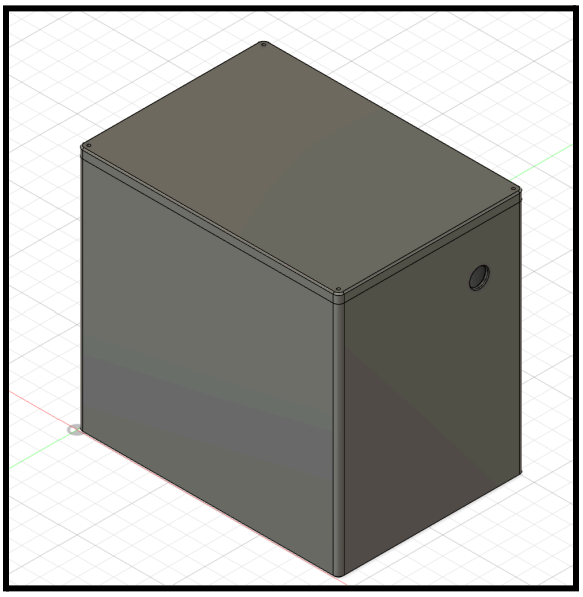
Front view (driver's view of device)



Side view with wire harness cutout



Isometric view of enclosure/lid assembly



Isometric view of lid

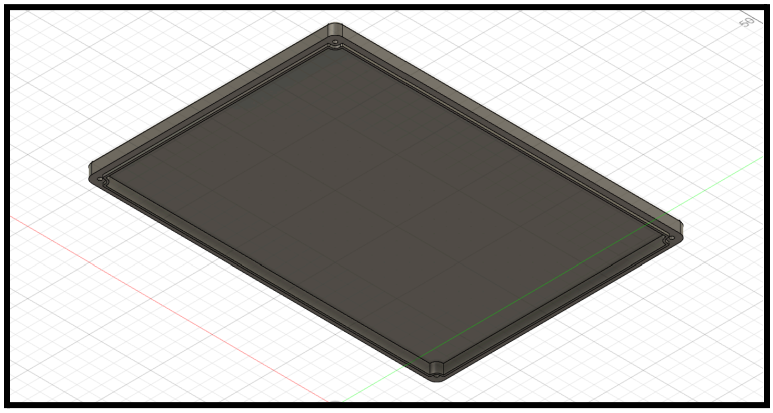


Figure 4.3.1 - Several views of enclosure and lid design

I then 3D-printed the enclosure and lid using a Markforged Onyx One with nylon/carbon fiber filament. I selected this material due to its high heat deflection temperature of 145 °C [6], allowing it to withstand extreme temperatures in hot vehicles.

Figure 4.3.2 - 3D-printed enclosure with LED channels and PCB standoffs

The enclosure can be mounted to a car's dashboard using velcro tape for convenient installation and removal.

Bill of Materials

The Bill of Materials (BOM) for this device includes all components required to assemble the PCB, along with the other boards, wiring, and enclosure. PCB

components are highlighted in green, other boards in orange, and miscellaneous components in purple.

Part Name	Part Number	Cost/Unit	Quantity	Total Cost
C2 (220pF Capacitor)	0805B221K500BD	$0.04	1	$0.04
C1 (100nF Capacitor)	0805B104K101CC	$0.06	1	$0.06
R3,R4,R5,R6,R7,R8,R9 (220Ω Resistor)	CRGCQ0805F220R	$0.10	7	$0.70
R1 (1kΩ Resistor)	RC0805FR-071KL	$0.10	1	$0.10
R2 (4.6kΩ Resistor)	ERJ-6ENF4991V	$0.11	1	$0.11
D1,D2 (Rectifier Diode)	1N4007-T	$0.17	2	$0.34
J1 (11 Position Female Connector Header)	PPTC111LFBN-RC	$0.73	1	$0.73
J2, J3, J4 (1 Position Male Connector Header)	PH1-01-UA	$0.10	3	$0.30
F1 (Fuse Holder)	0PTF0078P	$0.51	1	$0.51
200mA Fuse	0235.200HXP	$1.03	1	$1.03
D3, D4, D5 (Green LED)	LTL-709L	$0.37	3	$1.11
J13 (3 Position Female Connector Header)	PPPC031LFBN-RC	$0.38	1	$0.38
D8, D9 (Red LED)	LTL-709E	$0.41	2	$0.82
D6, D7 (Yellow LED)	LTL-709Y	$0.29	2	$0.58
SN65HVDA195 (LIN Bus Transceiver)	SN65HVDA195QDRQ1	$1.74	1	$1.74
PCB (from PCBWay Fab)		$1.00	1	$1.00
Arduino Uno	A000066	$27.60	1	$27.60
12V to 5V Buck Converter	B01NALDSJ0	$9.00	1	$9.00
3D Printer Filament (cost from slicer estimate for PETG)				$3.88
OBDII Wire Harness (J1962 Connector)	IKG-40198	$7.99	1	$7.99
Total				**$58.02**

Figure 4.4.1 - Device BOM for PCB, Arduino, enclosure, and wiring

Assembly

I assembled the device by soldering each component onto the fabricated PCB. I then prepared the wiring harness by splitting power and ground connections between the buck converter and PCB. Finally, I used jumper cables to connect the PCB, Arduino, and buck converter as shown in Figure 5.1.1.

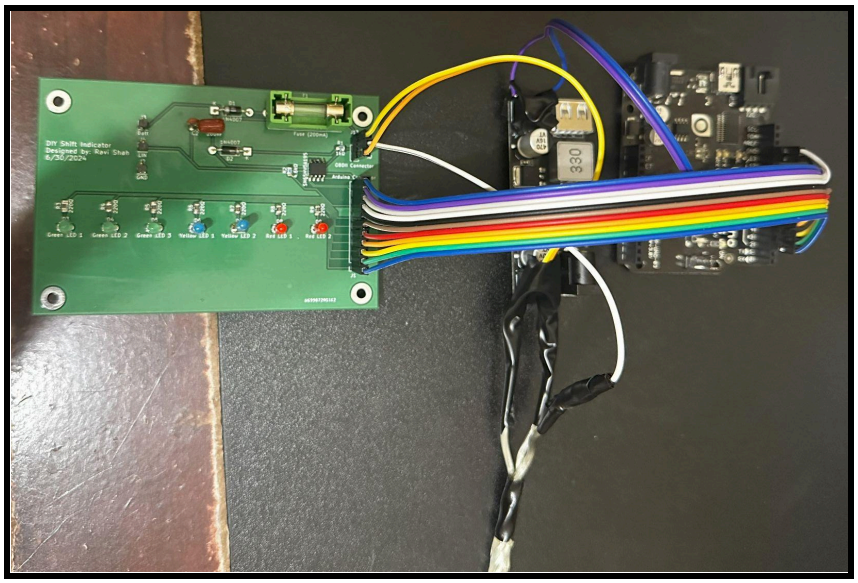


Figure 5.1.1 - Final product assembled outside of enclosure

After testing all the connections and verifying that the Arduino and LIN transceiver were functioning, I mounted all the components in the enclosure.

Figure 5.1.2 - Final product packaged inside enclosure with insulation

Code

I developed Arduino code to handle communication with the LIN transceiver, along with logic for the LEDs. By modifying the OBD9141 Arduino library by Ivor Wanders [7], I enabled the Arduino to communicate with the car's ECU and poll for data.

ISO 9141 Protocol

To understand the code used to communicate via the LIN bus (K-Line), one must first understand the ISO 9141 protocol. According to the International Organization for Standardization (ISO), this protocol functions to "[set] up the interchange of digital information between on-board emission-related Electronic Control Units (ECUs) of road vehicles and the SAE OBD II scan tool as specified in SAE J1978. [8]" In other words, it enables the reading of real-time vehicle data (RPM, wheel speed, temperatures), DTCs (diagnostic trouble codes), and other information.

To initialize a connection with the ECU, ISO 9141 provides a standard sequence of bits known as the "5-baud init." The OBDII device must send out a communication signal at a rate of 5 bits per second to the vehicle's K-Line [9]. This signal consists of setting the K-Line output to HIGH and LOW for specific time intervals as shown by the logic analyzer diagram in Figure 6.1.1. Specifically, the K-Line is set to *LOW* for 200ms, *HIGH* for 400ms, *LOW* for 400ms, *HIGH* for 400ms, *LOW* for 400ms, and finally *HIGH* for 200ms for the stop bit.

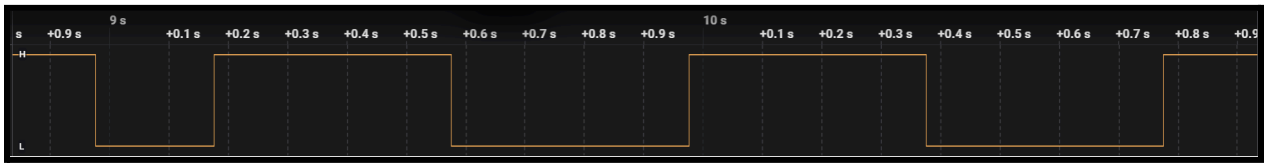


Figure 6.1.1 - Saleae logic analyzer data demonstrating 5-baud init

This sequence is implemented in C++ by the OBD9141 Arduino library as shown in Figure 6.1.2.

```
bool OBD9141::initImpl(bool check_v1_v2){
    use_kwp_ = false;
    // this function performs the ISO9141 5-baud 'slow' init.
    this->set_port(false); // disable the port.
    this->kline(true);
    delay(OBD9141_INIT_IDLE_BUS_BEFORE); // no traffic on bus for 3 seconds.
    OBD9141println("Before magic 5 baud.");
    // next, send the startup 5 baud init..
    this->kline(false); delay(200); // start
    this->kline(true); delay(400);  // first two bits
    this->kline(false); delay(400); // second pair
    this->kline(true); delay(400);  // third pair
    this->kline(false); delay(400); // last pair
    this->kline(true); delay(200);  // stop bit
    // this last 200 ms delay could also be put in the setTimeout below.
    // But the spec says we have a stop bit.
```

Figure 6.1.2 - OBD9141 implementation of 5-baud “slow” init

After this sequence, the standard allows us to treat the bus as a 10400 baud serial port [9].

```
// done, from now on it the bus can be treated ad a 10400 baud serial port.

OBD9141println("Before setting port.");
this->set_port(true);
OBD9141println("After setting port.");
uint8_t buffer[1];

this->serial->setTimeout(150);
```

Figure 6.1.3 - OBD9141 implementation to enable serial connection with ECU

The ECU should now send the byte *0x55*, followed by two identical bytes known as *v1* and *v2*. For my Mazda Miata, the value of these bytes is *0x08*, as shown in Figure 6.1.4.

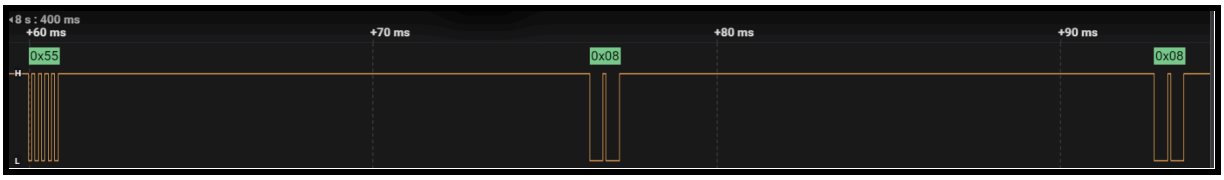


Figure 6.1.4 - Saleae logic analyzer data showing response from ECU

The *v1*/*v2* byte is then inverted (~) and sent on the K-Line. Finally, the ECU sends *0xCC*, indicating a successful initialization [9]. The serial port can then be used to send and receive data for vehicle information, real-time measurements, and diagnostic codes.

Loop Function

```
while (1) {

  res = obd.getCurrentPID(0x0C, 2); // request current RPM value
  if (res) {
    Serial.print("Result 0x0C (RPM): ");
    RPM = (obd.readUint16() / 4); // convert to usable value
    Serial.println(RPM);
    displayRPM(RPM); // display on LED series
  }

  Serial.println();

  delay(100); // wait 100ms to loop
}
```

Figure 6.2.1 - Code for main loop of program

The loop function in Figure 6.2.1 runs continuously to read RPM data from the car's ECU and display it on the LED series using the *displayRPM* function. It requests the PID (Parameter ID) for RPM (*0x0C*) from the ECU using the sequence of bytes shown in Figure 6.2.2 followed by *0x0C* [10].

Figure 6.2.2 - Saleae logic analyzer data for PID request byte sequence

Thus, the full request sequence for RPM is {*0x68, 0x6A, 0xF1, 0x01, 0x0C*}. The ECU then sends the current RPM value on the K-Line, and the Arduino formats it to a usable integer value.

LED Driver/Display

The *displayRPM* function is then called with this integer value as an input. According to the code in Figure 6.3.1, a certain number of LEDs (0-7) will illuminate based on this input.

```
void displayRPM(int RPM) {

  reset();

  // Use RPM value to determine which display option to use
  if (RPM <= 3000) {
    reset();
  } else if (RPM <= 3500) {
    one();
  } else if (RPM <= 3930) {
    two();
  } else if (RPM <= 4360) {
    three();
  } else if (RPM <= 4790) {
    four();
  } else if (RPM <= 5220) {
    five();
  } else if (RPM <= 5650) {
    six();
  } else {
    seven();
  }
}
```

Figure 6.3.1 - *displayRPM* function with specific RPM values for each pattern

Challenges Overcome

Initialization Failure

When testing the initial prototype (breadboard), I received constant timeouts during the initialization sequence, as the Arduino was unable to initialize communication with the ECU. To debug this, I connected a Saleae logic analyzer to the TX, RX, and K-Line, using a simple voltage divider to drop the voltage to a safe level.

The logic analyzer output, as shown in Figure 7.1.1, confirmed that both the LIN transceiver and ECU were functioning properly. The “5-baud init” and ECU response (*0x55, 0x08, 0x08* - small blips on the right side of the orange graph) were read by the logic analyzer. However, this response was never recognized by the Arduino.

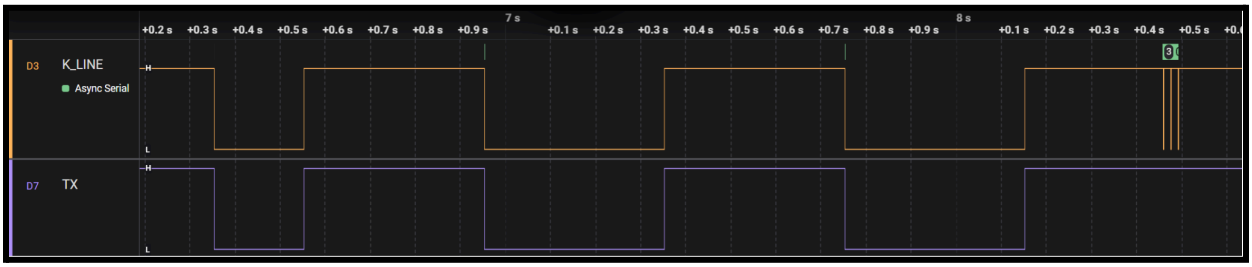


Figure 7.1.1 - Saleae logic analyzer data for K-Line and TX during failed initialization

I found that the cause of this issue was an insufficient timeout in the OBD9141 Arduino library. On average, the ECU took about 100ms to send a response, however, the default timeout was set to 20ms. After some trial and error, I increased this timeout to 150ms (as shown in the code in Figure 6.1.3), which fixed the problem and allowed for consistent initialization.

Lack of Symbols/Footprints for IC

The LIN transceiver IC used for this device (TI SN65HVDA195) is a fairly uncommon component, and thus doesn’t have built-in support in Fritzing or KiCad. To design the breadboard and PCB for this project, I had to learn how to create custom symbols and footprints for this IC. Now that I have experience creating components from scratch, I’ll be able to apply my skills to future electronics projects.

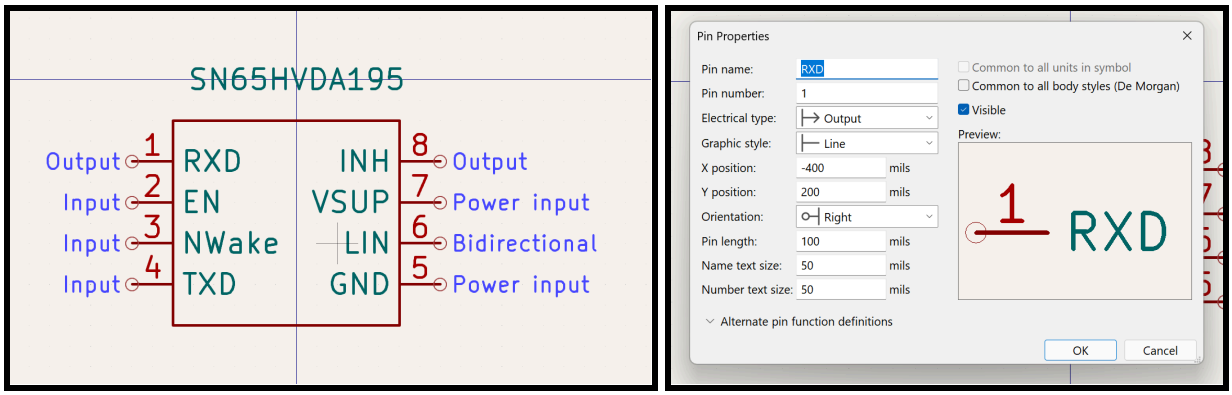


Figure 7.2.1 - Custom symbol for SN65HVDA195 IC in KiCad

Soldering SMD Components

This project was my first time soldering very small surface mount (SMD) components. I quickly learned that I had to be extremely precise with such components to ensure proper connection without causing damage. I learned to tin the pads of the PCB and use flux so the solder would flow/spread well. In this way, I soldered the LIN transceiver IC, along with all the surface mount resistors and capacitors. By the end, I was able to work much more quickly and precisely.

Evaluation

The final device, as pictured in Figure 8.1.1, meets the basic requirements I set for this project. It reliably displays engine RPM using a three-color progression with seven LEDs, illuminating from the lower end to the higher end of the shift range. It doesn't require any modifications to the car's wiring or electronics, since it's fully plug-and-play via the OBDII diagnostics port. Further, it attaches using detachable velcro, so it can be installed and removed easily.

However, the device is quite bulky and can obstruct the driver's view of the road under certain conditions. Additionally, the LEDs can be difficult to see at an angle, which could cause issues during use. Overall, this device provides the core functionality I was aiming for but falls short in terms of usability.

Figure 8.1.1 - Final device for the K-Line Shift Indicator

Future Plans

Although this device met or exceeded most of the initial requirements, further improvements could be made to improve usability, packaging, and cost. In future iterations, I aim to address concerns in each of these areas.

Usability

Future iterations of the device should be less bulky, ideally less than half the height of this iteration. This would allow the driver to easily see the road while gleaning information from the device. To achieve this, improvements in the selection and packaging of components will be required. Additionally, the next iteration should have diffusing material in the LED channels so that the lights can be seen easily from several angles.

Packaging

Instead of three separate boards for the LIN transceiver/display, Arduino, and buck converter, there should be one board incorporating their functionality. To implement this, the next iteration should use a custom microcontroller instead of an Arduino, allowing for a much simpler design. Additionally, the clearances between components on the PCB should be smaller (the current iteration has relatively large clearances since it was my first time designing a PCB). Alternatively, future iterations

could also move components like the buck converter into the OBDII harness itself to reduce wasted space inside the enclosure.

Cost

In addition to the cost benefits of replacing the Arduino with a microcontroller, improvements can be made to reduce the cost of this device. The Onyx filament I used is much stronger and much more heat resistant than necessary. It's also very expensive, costing about $35 per print - I only used it because the alternative I had (PLA) was too weak and had a relatively low heat deflection temperature of ~50°C [11]. The next iteration should use PETG filament, which is relatively cheap and has a sufficient heat deflection temperature of ~70°C [12].

Additional Features

In addition to addressing these concerns, I would like to implement the following features in future iterations:

- Stall Alert - Manual transmission cars run the risk of stalling (engine shutoff) when the engine turns below a certain RPM. Future iterations should implement a cue to warn the driver of such conditions. This could be a light flash or the illumination of a certain sequence of lights.

- Configurable lighting patterns - Many commercial devices offer customization in terms of the light pattern displayed. This allows the user to change the color of individual LEDs, along with the RPM ranges triggering each LED. Such a system could be implemented by simple buttons on the enclosure or a smartphone app connecting to the device.

References

The GitHub repository containing all development files for this project is available here. Below are citations for the datasheets and external sources in this document.

[1] UK Department for Business, Innovation and Skills, “Williams F1 steering wheel,” Flickr, https://www.flickr.com/photos/bisgovuk/4136431570 (accessed Jul. 21, 2024).

[2] “SN65HVDA195-Q1 Automotive LIN and Most ECL Physical Interface,” Texas Instruments, https://www.ti.com/lit/ds/symlink/sn65hvda195-q1.pdf (accessed Jul. 21, 2024).

[3] “Arduino UNO R3,” Arduino, https://docs.arduino.cc/resources/datasheets/A000066-datasheet.pdf (accessed Jul. 21, 2024).

[4] P. Stoffregen, “AltSoftSerial Library,” PJRC, https://www.pjrc.com/teensy/td_libs_AltSoftSerial.html (accessed Jul. 21, 2024).

[5] AnotherMaker, “Learn KiCad 8 in 45 minutes - From idea to upload in one video,” YouTube, https://www.youtube.com/watch?v=SFJRHFMOhQA (accessed Jul. 21, 2024).

[6] “Onyx,” Markforged Support Portal, https://support.markforged.com/portal/s/article/Onyx (accessed Jul. 21, 2024).

[7] I. Wanders, “Iwanders/OBD9141: A class to read an ISO 9141-2 port found in OBD-II ports.,” GitHub, https://github.com/iwanders/OBD9141/tree/master (accessed Jul. 21, 2024).

[8] “ISO 9141-2,” ISO, https://www.iso.org/obp/ui/#iso:std:iso:9141:-2:ed-1:v1:en (accessed Jul. 21, 2024).

[9] “K-line Communication Description,” National OBD Clearinghouse, https://www.obdclearinghouse.com/Files/viewFile?fileID=1380 (accessed Jul. 21, 2024).

[10] “OBD2 PIDs for programmers (technical),” Total Car Diagnostics, https://www.totalcardiagnostics.com/support/Knowledgebase/Article/View/104/0/obd2-pids-for-programmers-technical (accessed Jul. 21, 2024).

[11] “Heat-resistant 3D printing materials guide: Compare processes, materials, and applications,” Formlabs, https://formlabs.com/blog/heat-resistant-3d-printing/ (accessed Jul. 21, 2024).

[12] “PETG 3D printing filament,” BigRep Industrial 3D Printers, https://bigrep.com/filaments/petg/ (accessed Jul. 21, 2024).