Implementation of CAN bus in a concept solar car

DTU ROAST peripherals team - TOAST

Lukas Fredrik Cronje (s173919)
Louise J. Sternholdt-Sørensen (s173937)

Special Course

2020

DTU Electrical Engineering
Department of Electrical Engineering

Implementation of CAN bus in a concept solar car, DTU ROAST peripherals team - TOAST

Report written by:
Lukas Fredrik Cronje (s173919)
Louise J. Sternholdt-Sørensen (s173937)

Advisor(s):
Jens Christian Andersen
Claus Suldrup Nielsen

DTU Electrical Engineering
Technical University of Denmark
2800 Kgs. Lyngby
Denmark

elektro@elektro.dtu.dk

Project period:	31 August- 23 December
ECTS:	5
Education:	Special Course
Field:	Electrical Engineering
Class:	Public
Edition:	1. edition
Remarks:	This report is submitted as partial fulfillment of the requirements for graduation in the above education at the Technical University of Denmark.

Abstract

This paper will create an overview of the basic components required to set up a Controller Area Network (CAN) bus using the Standard CAN protocol (11-bit) and how it will be implemented in a concept solar powered car. A short introduction will be given as to how this fits into the DTU ROAST solar car project. This paper goes into detail on both the Hardware and Software side of things, on how to use and develop a CAN bus. The purpose of the paper is to create a guide that simplifies implementing a CAN bus in an Arduino platformed project, in order to be able to hand the project of to a new group of students to further innovate on our designs. Part of the simplification includes a PCB design for a CAN bus shield for the Arduino uno (or any other 5V arduino microcontroller), to make including additional micro-controllers to the CAN network more accessible.

Contents

CHAPTER 1

Introduction

As part of the work of getting the DTU ROAST solar car up and running, we had implement a CAN bus. In doing so, we have to establish a protocol for the communication in the car, choose a CAN bus standard and make a reliable and secure design.

DTU ROAST is the DTU roadrunners solar team. It has three subgroups, electrical, software and mechanical, to manage the different challenges of a solar powered car. We are the peripherals team (also dubbed TOAST), which is part of the electrical sub group. It is a newly established team, that was formed late last year, and therefor a lot of the groundwork needs to be laid before we have a car up and running. We aim to make this a comprehensible and clear-cut report, so future team members can easily understand and improve upon our work.

1.1 Objectives

Essentially a CAN bus is somewhat similar to the nervous system in a human. Since we have a lot of sensors, lights and other things on/or a part of our car, we need to have a way to transport the information around. This is where a CAN bus comes in handy, since it allows us to pass information around the car via electric wiring and CAN bus line driver.[Paz99]

Our goals were to:

- Implement a CAN bus in the DTU ROAST solar car
- Establish a protocol for communication
- Choose the CAN bus standard
- Make a reliable and secure design

CHAPTER 2

Analysis

2.1 Introduction

When we had to find a solution to fit the solar car, we had to consider different factors/limitations. For us the most important part was that it was easy to work with, and easy to get into, since it is something that is going to be improved upon by others in the future.

We considered two different ways we could make and set up our CAN bus prototype, The other we didn't choose, we will account for here.

Our CAN bus PCB is almost identical to the ecocars since we probably used the same resources (the datasheet), however how we set up the CAN controller plus the microcontroller/ECU and the software part is where we differ in our solutions.

2.1.1 Teensy ecocar

Since the roadrunners ecocar is a more established team, we talked with them about their solution, and how to do something similar in the solar car. They use a Teensy 3.6 (as of writing), and installed a program called TeensyArduino that adds Teensy support to the Arduino IDE. The pros of this is that the teensy is incredibly fast. Teensy 3.6 has a 180 mHz clock[PJR20] vs the Arduino Uno's 16 mHz[Ard20]. It also uses less power than the uno or mega. It is also more expensive, but since the roadrunners have bought in bulk that wouldn't be a problem.

However the teensy is less user friendly, and we value ease of use over speed, since 16 mHz is plenty enough for the solar car for now.

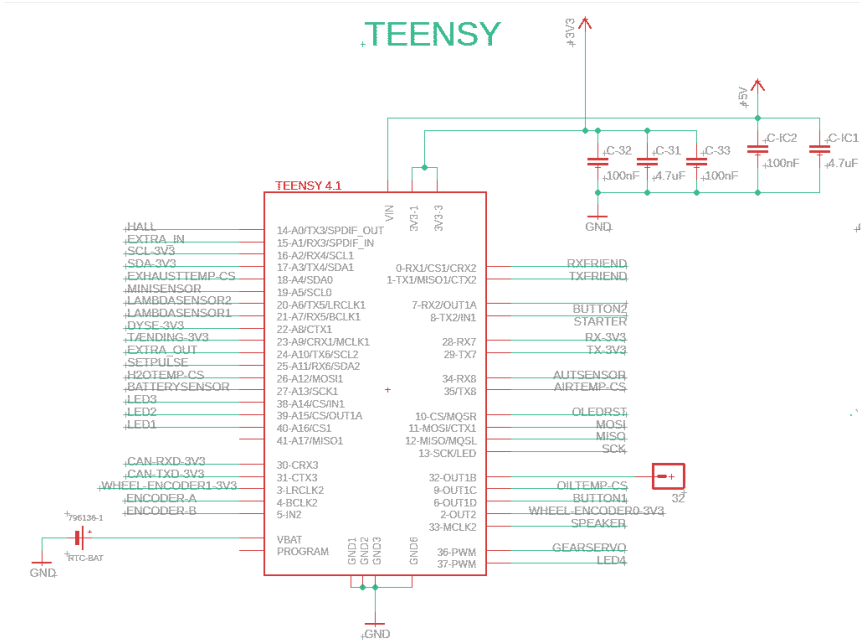

Figure 2.1: CAN transceiver PCB for a connecting the CAN network to a ECU/microcontroller (Revision 1)

2.2 Summary

The teensy definitely is superior to the arduino in almost every way, it is less user friendly and we were less familiar with it. So for the first iteration for the CAN bus, we decided to go with the arduino solution instead. We did so, since we ultimately have to hand it over to someone, and so for the first version of the solar car we were more focused on having something that runs and something that we document fulfillingly.

CHAPTER 3

CAN Bus

3.1 Introduction

A **Controller Area Network (CAN) Bus** is a reliable bus standard predominantly used in vehicles. It is designed to send and receive messages between micro-controllers and devices without the use of a host computer. A CAN Bus uses a message based protocol that transmits sequentially. If two devices transmit data simultaneously, then the device with the highest priority will continue, while the other waits. When multiple CAN devices are connected together they form a network, this network is comparative to the central nervous system of the human body. In a CAN Bus network all devices can receive messages, including the transmitter[con20].

CAN Bus has become a standard within the Automotive industry. Cars now a days are built with hundreds of sensors that carry critical information, and without a reliable message transmission system these cars would be unsafe to drive.

3.2 The CAN bus

Since **DTU ROAST** is building what is essentially an Electrical Vehicle, the car will therefore contain a lot of sensors and controllers that will need to be able to communicate with each other. One of the most vital systems of the car will be the battery management system, it is imperative that we can send and receive messages at a fast enough bandwidth and without errors.

The CAN Bus fulfills both of these requirements. A CAN Network consists of two wires **CAN High** and **CAN Low** for bi-directional data transmission. For CAN devices in a network the communication speed ranges from 50 Kbps to 1Mbps and the distance can range from 40 meters at 1Mbps to 1000 meters at 50kpbs. At this point we haven't made any estimates for how many meters of wires for data transmission will be in the car, but we do not expect it to exceed the limits of the CAN bus.

Speed [kbits/s]	Maximum Cable Length [m]
1000	40
800	50
500	100
250	250
125	500
50	1000

Table 3.1: Table showing maximum cable length for different bit rates[Tho14].

3.2.1 CAN bus electrical properties

As mentioned above the the nodes in a CAN network are connected to each other by two wires, **CAN High (CAN-H)** and **CAN Low (CAN-L)**. Normally a twisted pair cable is used and at each end the two wires are terminated to each other with 120Ω resistors. The wires need to be tied to the same potential and the resistors help balancing the lines. The two wires act as a differential line, which means that the **0** and **1** of the can signal is represented by the potential difference between **CAN-L** and **CAN-H**.

When a (1) is transmitted neither of the wires are driven high or low and they are both sitting at 2.5 V. Comparatively when a (0) is being transmitted **CAN-H** is driven 1 V higher at 3.5 V and **CAN-L** is driven at 1 V lower at 1.5 V. This creates a clear Voltage difference between the wires, which translates to a (0) being transmitted.

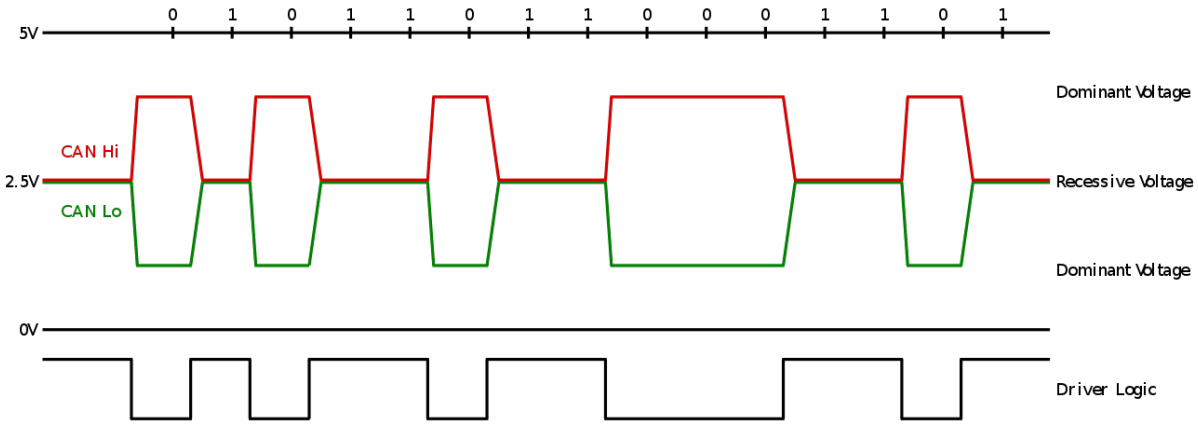

Figure 3.1: High-speed CAN signaling. ISO 11898-2[con20]

3.2.2 Format of a CAN message

A CAN message has a particular format consisting of eight different segments. Although each segment serves it's purpose, the two main segments we will be focusing on will be

the **identifier** and **data** segments.

Figure 3.2: Frame format of (a) Standard CAN and (b) Extended CAN.[Lun+10]

The Identifier is also known as the CAN ID. It is used to identify the devices present in the network. The length of the CAN ID can vary depending on the CAN protocol the network is using. The identifier can either be 11 or 29 bits long.

- Standard CAN: $0 - 2^{11} - 1$ (11-bit)
- Extended CAN: $0 - 2^{29} - 1$ (29-bit)

The Data segment of the CAN message is the actual values of each sensor or controller that are being sent from device to device. The size of the data can range from 0 to 8 bytes in length. Much like some programming languages when initiating arrays you have to declare the size of the array. For a CAN message the size of the data is declared by the **Data Length Code (DLC)**.

The CAN Bus can only read one message at a time. When one than one message is being sent the CAN Bus will prioritize the message with the lowest CAN ID.

In order to maintain an overview of all nodes in the newtwork, we created an Excel document listing all the sensors going from highest to lowest priority. The sensor priority is mainly determined by safety, which functions need to be read first in order to make the vehicle as safe as possible.

3.2.3 CAN Hardware

For this project we wish to use a CAN bus to communicate between micro-controllers. Since we are predominantly using Arduino based micro-controllers. Arduinos don't have any inbuilt CAN port, so we decided to use a **MCP2515 CAN Module** as our can device. This module interfaces with the Arduino through **SPI** communication.

Message ID 0xF1

7	6	5	4	3	2	1	0	bits / bytes
1	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	1	1
1	1	1	0	1	1	1	0	2
								3
								4
								5
								6
								7

	a
	b
	c
	d
	Not Used

Figure 3.3: Example of CAN Data with a DLC of 3.

	A	B	C	D	E
1	Component Group	Component	Bit-Assignment 8-bit (bx000000x00x)	Description	Priority
2	Battery	Battery Management System			1
3	Battery	Battery current (OUT)			1
4	Battery	Battery Current (IN)			1
5	Battery	Battery Voltage			1
6	Lights	Left Front Indicator			2
7	Lights	Left Side Indicator			2
8	Lights	Left Rear Indicator			2
9	Lights	RightFront Indicator			2
10	Lights	Right Side Indicator			2
11	Lights	Right Rear Indicator			2
12	Lights	Headlights			3
13	Lights	High Beam			3
14	Lights	Hazard Light			2
15	Lights	Rear Break Light (sides)			1
16	Lights	Rear Break Light (center)			1
17	Lights	Maximum power point tracking (MPPT)			1
18	Temperature	Battery Temperature (1 for each group of cells)			3
19	Temperature	Battery management system temp			3
20	Temperature	Motor Temp			3
21	Temperature	Motordriver temps (three phase inverter)			3
22	Temperature	Solar Cells Temp			3
23	Temperature	Cooling Fluid temp			3
24	Temperature	Maximum Power Point tracking (MPPT) temp			3
25	Temperature	Power converter		Power conversion from the Battery DC BUS to the Low Voltage BUs (48V)	3
26	Temperature	Cabin Air Temp			5
27	Tyres	Left Font Brake pressure		Oil pressure in tubes with brake fluid	3
28	Tyres	Left Rear Brake pressure		Oil pressure in tubes with brake fluid	3
29	Tyres	Right Front Brake pressure		Oil pressure in tubes with brake fluid	3
30	Tyres	Right Rear Brake pressure		Oil pressure in tubes with brake fluid	3
31	Tyres	Left Font Tyre pressure			3
32	Tyres	Left Rear Tyre pressure			3
33	Tyres	Right Front Tyre pressure			3
34	Tyres	Right Rear Tyre pressure			3

Figure 3.4: Excerpt from our ID priority list.

The MCP2515 CAN Module uses a MCP2515 CAN controller, which is a high speed CAN controller (**ISO 11898-2**). It supports operation speeds of up to 1 Mb/s. It has support for up to 112 nodes in the network, which is comfortably within our bounds as we currently have plans for 58 nodes within the network.[Tha19]

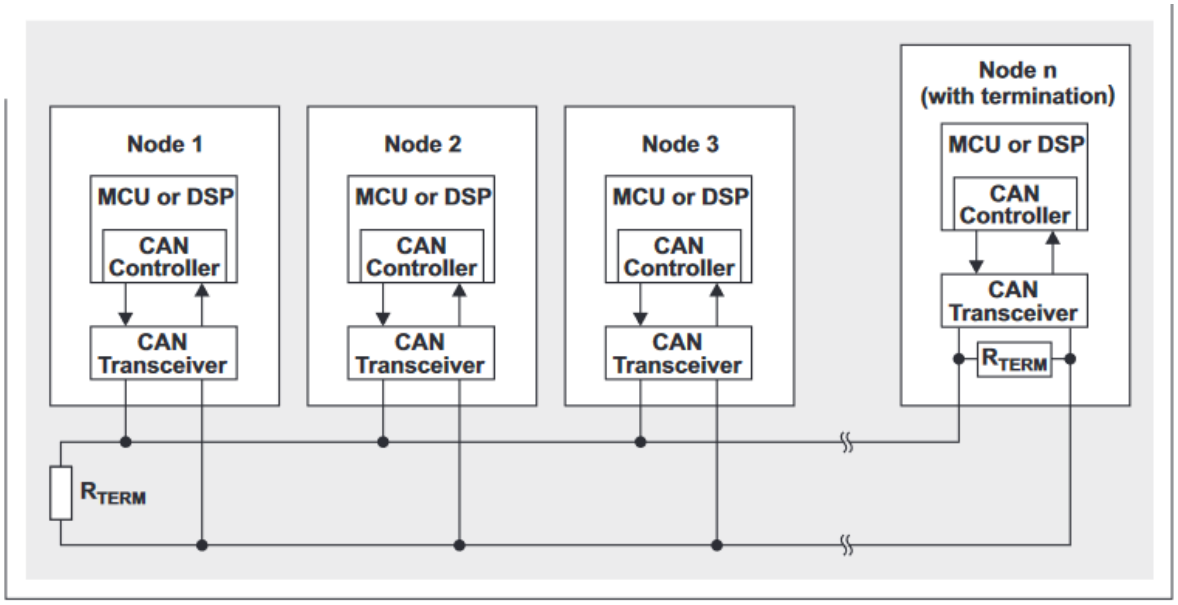

Figure 3.5: Figure from TI(Texas Instruments) for a classical CAN network[Mon13]

Another CAN module we use is the MCP2551, which is a CAN bus transceiver, and of the same brand and line as the MCP2515, so they work well together. It also supports operation speeds up to 1 Mb/s and is also compatible with the **ISO 11898-2** standard[Inc20]. We use it to make a standardized PCB to act as an interface between the microcontroller and the CAN High and Low.

3.3 Summary

So in summary, the CAN bus is robust, safe and reliable. We have laid out how the CAN bus works. We have chosen to use the **MCP2515 CAN bus Module** and we will be using the standard CAN protocol (11-bit) for communication.

CHAPTER 4

Prototype

4.1 Introduction

Now that we have become well acquainted with how the CAN Bus works, it is time to build a small prototype. For our prototype we will be focusing on acquiring data on one micro-controller and transmitting it via CAN Bus to another micro-controller where it will be displayed.

Our goal for our final product is to create a standard for sending and receiving data around the car, gathered from various sensors and ECU/micro-controllers.

4.2 Hardware

As a test, we have chosen to use a temperature sensor for our prototype. With a temperature sensor we will easily be able to manipulate the temperature (data) either by cupping it in our hands or using a flame. We want the micro-controller to send the temperature from the one micro-controller to other and to display an error if the temperature exceeds a threshold, that we have set in place.

For our display, we are using a TFT LCD display. This allows us to color code our displayed data, green for temperatures below the threshold and red for anything above the threshold.

In our setup, there is not that much that has to be done on the hardware side of things. For the most it is just connecting modules to the correct pins on the Arduino with jumper wires. This is a messy setup, that is why we will later on in this paper will discuss a custom PCB that we have designed.

4.3 Software

On the software side of things there are a few puzzle pieces that need to be put together. There are three pieces of code that need to be able to work together.

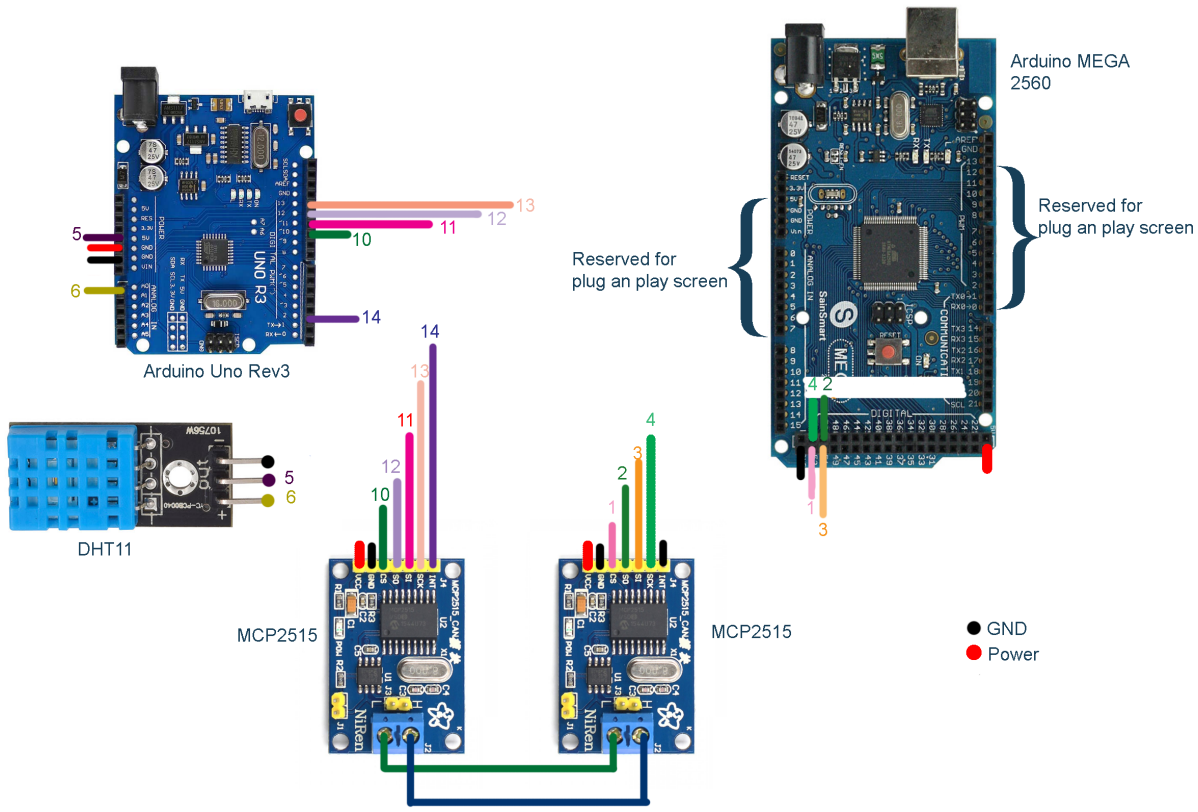

Figure 4.1: Wiring of the CAN prototype.(we don't have the numbers from 7-9, since we removed some of the unnecessary wiring)

- CAN Transceiver
- CAN Receiver
- LCD Display

4.3.1 CAN Transceiver

The **CAN Transceiver** part of the prototype, is the code that collects the data from the temperature sensor and prepares the data for being sent through the CAN Bus.

The first thing we have to do is setup the CAN Bus transceiver correctly. The **MCP2515 CAN Bus Module** interfaces with the Arduino boards through the on-board **SPI** pins. Therefore we will need two Arduino libraries, **<SPI.h>**[Ard19] and **<mcp2515.h>**[Dmi17].

First we need to set the pin number where the CS-SPI connection is, by default it is pin 10 on the Arduino Uno.

```
MCP2515 mcp2515(10);
```

To be able to store in a CAN message format we need to create a canMsg struct.

```
struct can_frame canMsg;
```

Next we move onto **void setup()**, here we will need to intialize SPI communication, reset the MCP2515 Module and set the bit rate. At the small distances we are working with a bit rate of 1Mbps should be possible, but just to limit the amount of data sent we used a bit rate of 125kbps. We also set the CAN mode to normal mode, to allow us to both send and receive messages.

```
SPI.begin();
mcp2515.setBitrate(CAN_125KBPS);
mcp2515.reset();
mcp2515.setNormalMode();
```

In the **void loop()** section of our code we can begin setting up our message. An important thing to remember is that CAN uses hexadecimal numbers. The first thing we need to declare will be the message ID, the lower the ID, the higher it will be on the priority list. Since we are only sending one message, we can use an arbitrary ID such as '0x0F1'. Next we need to declare the DLC (*Data Length Code*), we only nead a DLC of 2, for temperature and the error notification. To leave some bytes open for future use, we can increase the DLC and set the values to zero.

```
canMsg.can_id  = 0x0F1;
canMsg.can_dlc = 4;
canMsg.data[0] = t;       //Update temperature value in [0]
canMsg.data[1] = e;       //Update error value in [1]
canMsg.data[2] = 0x00;    //Rest all with 0
canMsg.data[3] = 0x00;
```

And finally to send the CAN message we use the following function.

```
mcp2515.sendMessage(&canMsg);
```

4.3.2 CAN Receiver

On the receiver side of things we want to use a TFT LCD Display, which would take up all the pins of a Arduino Uno, therefore we will use an Arduino Mega.

The receiver uses the same libraries as the Transceiver, **SPI.h** and **mcp2515.h**. In order to read the incoming message we need to create **struct data type** that will store the CAN message.

```
struct can_frame canMsg;
```

Since the Display takes up pin 10 on the Arduino, we need to use another SPI pin. The Mega can also communicate via SPI on pin 53, which we will use for initializing the MCP2515 Module.

```
MCP2515 mcp2515(53);
```

In the **void setup()** section it is important that the receiver has the same initialization as the transceiver, otherwise the data would look like it has been corrupted.

```
SPI.begin();
mcp2515.setBitrate(CAN_125KBPS);
mcp2515.reset();
mcp2515.setNormalMode();
```

Next in the **void loop()** we use an *if statement* to receive the message.

```
if (mcp2515.readMessage(&canMsg) == MCP2515::ERROR_OK)
```

Right now we are only receiving one message ID, once we begin to receive many messages with different ID's we will need some sort of function to filter out the messages that we want to view. We have created a filter that will look for all messages with the ID of **0x0F1**.

```
if (canMsg.can_id == 0x0F1) {
    int t = canMsg.data[0];
    int e = canMsg.data[1];
}
```

4.3.3 LCD Display

The final piece of the puzzle is the display where we print the measured temperature. This has been the most difficult part of the project, since it hasn't been easy to source a display and the documentation isn't readily available.

For the screen to work we will need to include the following libraries.

- Adafruit_GFX.h[Ind20]
- SPI.h
- MCUFRIEND_kbv.h[pre20]

First we create an object for the display, and lets call it *tft*.

```
MCUFRIEND_kbv tft;
```

Since this segment of code will be part of the receiver code, instead of making a **void setup** function, we create an initialization function that we can call in the setup for the receiver. This function will clear the screen and fill the background with our choice of color and set the orientation to landscape.

```
void ScreenInit()
{
    tft.reset();
    ID = tft.readID();
    tft.begin(ID);
    tft.setRotation(1);
    tft.fillScreen(BLACK);
}
```

Now we need a function that can take the values of our CAN message and display them on the screen. We've created a function that takes the temperature, error and last temperature as inputs. Since the refresh rate is of the screen is relatively slow, we only want the screen to update when the temperature changes, therefore we also want the last temperature reading as an input.

```
void Print_Temperature(int temperature, int error, int last) {

  if(last != temperature){
    tft.fillScreen(BLACK);
    tft.setCursor(0,120);
```

```
    if(error){
      tft.setTextColor(RED);
    }else{
      tft.setTextColor(GREEN);
    }

    tft.print("Temperature  ");
    tft.print(temperature);
    tft.print("C");
  }
}
```

We can use this function every time we read the CAN message for the temperature.

4.3.3.1 User Interface

After working with the CAN Bus we felt it was important to have an easy and clear way of visualizing the data from all the sensors around the car, therefore we have started work on a basic User Interface for the screen that will be in the car.

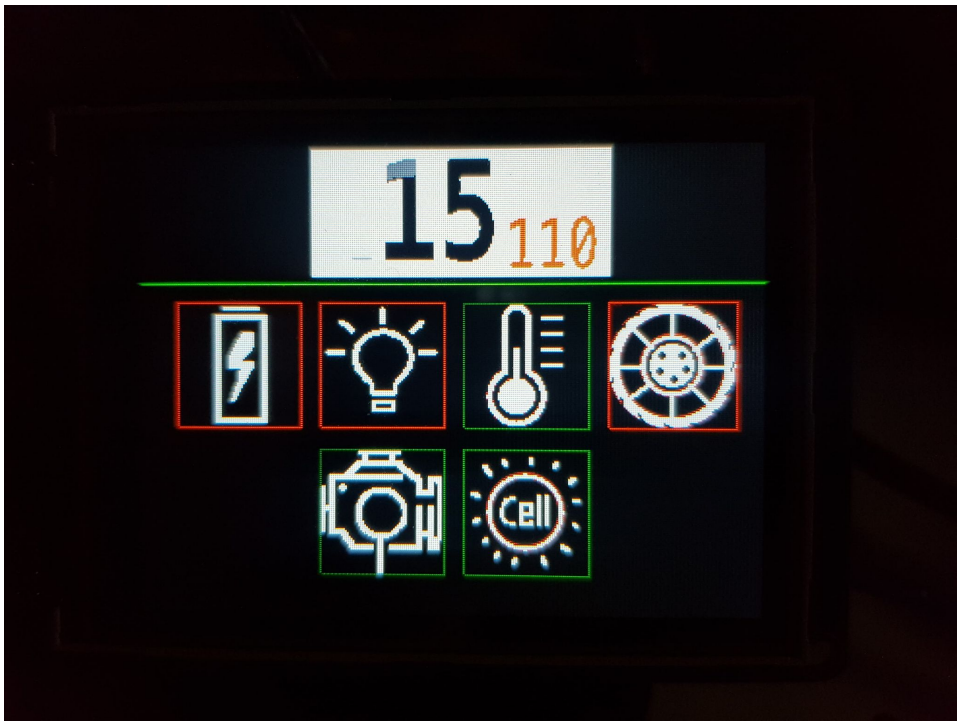

Figure 4.2: Work in progress car User Interface.

For the highest contrast we went with a black background. At the top center of the screen the speed of the car will be displayed along with a smaller recommended speed

next to it. Instead of having physical buttons, we made pictograms for each sensor, that the driver of the car can press to view the sensors in the corresponding category. If the pictogram has a green border it means there are no errors present, and if it is red then there are some errors, much in the same way a pictogram will light up on the dash in a normal car.

4.3.4 PCB

We were fortunate enough to be offered a PCB course by the the DANSTAR group at DTU, where they assist by helping you make, solder and pay for a 10x10cm PCB. We jumped at the chance, and so we where able to design a PCB that we now are waiting for their remarks and critique.

We wanted to create a PCB that could act as the interface between the CAN bus network and the microcontroller. One that we ideally could use over and over again for each of all the microcontrollers in the car, or at least most of them. A CAN bus shield. Furthermore, it can also act as buffer between the (relatively) high voltage of the CAN bus network, and the low voltage of a microcontroller[Inc03].

We are making the CAN transceiver as a PCB part and have microcontroller stand for interpreting the CAN message. So on our PCB we are going to have the CAN transceiver MCP2551. We connect the transceiver to the CAN network (CAN H and CAN L), and to the microcontroller via TXD and RXD (transmit and recieve data). All the other components as seen on the schematic 4.3, is what the manufacture recommended in the datasheet.[Inc03]

This is only the first iteration, and we have to submit our schematics to be revised by the DANSTAR volunteers, before they order it for us.

We can even have one PCB, where we replace the J1 header to a data link connector (such as the DB9), and place the PCB to be easily accessible in the car, so you could plug in some kind of car code reader, to see all error codes (in case of the display failing).[Ele20]

One thing to note that the first and last CAN bus device, has to have 120 Ohm resistor to terminate the bus. If not, the whole network won't work correctly, and therefor also the PCBs.[Mon13]

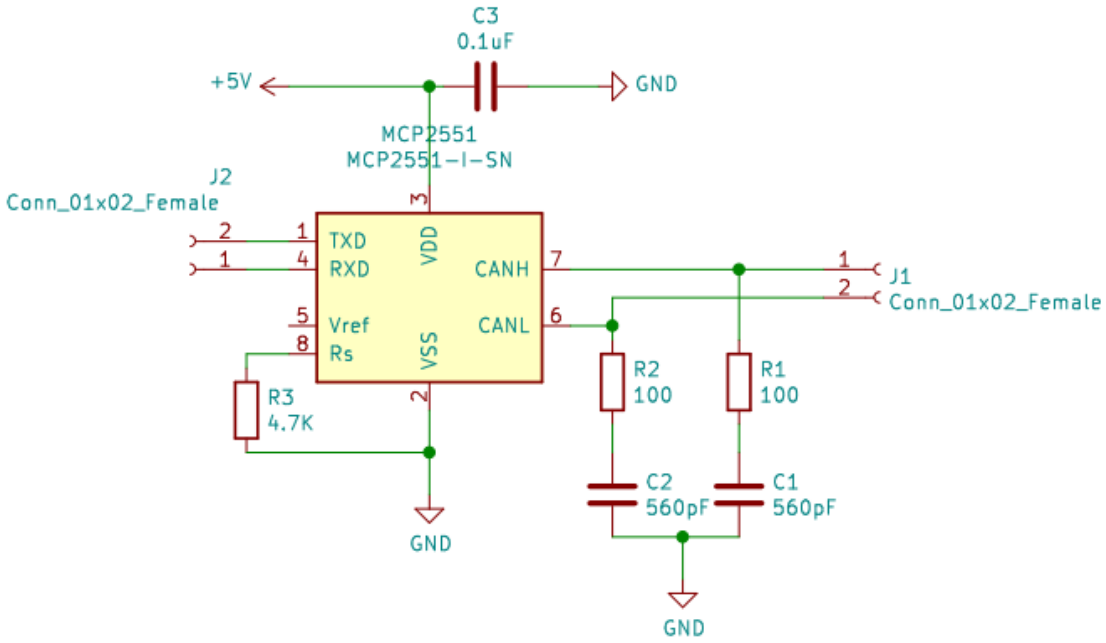

Figure 4.3: CAN transceiver PCB for a connecting the CAN network to a ECU/micro-controller (Revision 1)

4.4 Summary

After building our prototype and writing all the code it was time to test it. We set the threshold temperature to room temperature, 24℃, then we could easily cross the threshold by warming the sensor in the palm of our hand. The main part of the test was to see if the data we sent was the same as the data we received. By using the built in serial monitor of the Arduino IDE, we could see the exact non-formated data being sent from one micro-controller to the other. Since we only had to monitor one message ID, it was easy for us to spot unexpected values.

At first we did see some irregularities, but these were due to some wires not being grounded. After making the necessary adjustments, we started seeing the values that we expected.

CHAPTER 5

Conclusion

5.1 Conclusion

As part of the DTU ROAST, we had to implement a CAN bus. We have chosen the standard CAN bus protocol, with the 11 bit identifier. To have an overview over all of our sensors and controllers, we created an Excel document where gave each sensor/-controller a numerical score, between 1-5 (lowest score = highest priority), to see which items need to be prioritized highest.

We made a prototype that works, and made a standardised PCB design that can act as a CAN bus shield, and it works, and is documented so it can be implemented by others. It works and the CAN bus by itself is reliable, so we achieved our goal of implementing a reliable CAN bus.

Appendices

APPENDIX A

Scripts and schematics

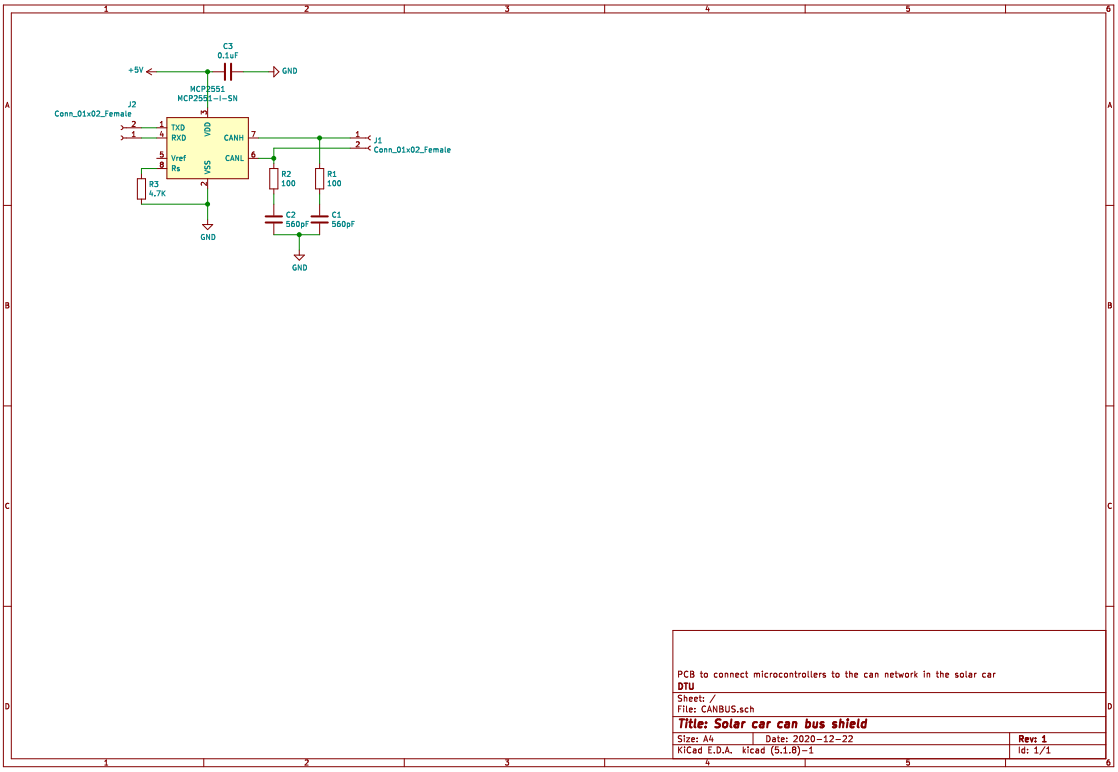

Figure A.1: Full pdf of our schematic for the CAN bus shield PCB

A.0.1 CAN Transmitter Script

```
1 #include <SPI.h>
2 #include <mcp2515.h>
3 #include <DHT.h>
4
5 #define DHTPIN A0
6 #define DHTTYPE DHT11
7
8 struct can_frame canMsg;
```

```
9
10 MCP2515 mcp2515(10);
11 DHT dht(DHTPIN, DHTTYPE);
12
13
14 void setup() {
15
16
17
18   while (!Serial);
19   Serial.begin(115200);
20
21   dht.begin();
22
23   mcp2515.reset();
24   mcp2515.setBitrate(CAN_125KBPS);
25   mcp2515.setNormalMode();
26
27   Serial.println("Send Temperature to CAN");
28 }
29
30 void loop() {
31   int t = dht.readTemperature();
32   int error = TemperatureError(t);
33
34   Serial.print("Temperature: ");
35   Serial.print(t);
36   Serial.print(", Error: ");
37   Serial.println(error);
38
39   CreateData(t,error);
40
41   mcp2515.sendMessage(&canMsg);
42   //delay(1000);
43
44 }
45
46 int TemperatureError(int t){
47   if (t >= 25){
48     return 1;
49   }else{
50     return 0;
51   }
52 }
```

```
53
54 void CreateData(int t, int error){
55   canMsg.can_id  = 0x0F1;
56 canMsg.can_dlc = 8;
57 canMsg.data[0] = error;       //Update error value in [0]
58 canMsg.data[1] = t;           //Update temperature value in [1]
59 canMsg.data[2] = 0x00;        //Set rest to 0
60 canMsg.data[3] = 0x00;
61 canMsg.data[4] = 0x00;
62 canMsg.data[5] = 0x00;
63 canMsg.data[6] = 0x00;
64 canMsg.data[7] = 0x00;
65 }
```

A.0.2 CAN Receiver Script

```
1  #include <SPI.h>
2  #include <mcp2515.h>
3
4  struct can_frame canMsg;
5  MCP2515 mcp2515(53);
6
7
8  void setup() {
9    Serial.begin(115200);
10
11   mcp2515.reset();
12   mcp2515.setBitrate(CAN_125KBPS);
13   mcp2515.setNormalMode();
14
15   ScreenInit();
16
17   Serial.println("------- CAN Read ----------");
18   Serial.println("ID  DLC   DATA");
19 }
20
21 void loop() {
22   if (mcp2515.readMessage(&canMsg) == MCP2515::ERROR_OK) {
23     Serial.print(canMsg.can_id, HEX); // print ID
24     Serial.print(" ");
25     Serial.print(canMsg.can_dlc, HEX); // print DLC
26     Serial.print(" ");
27
```

```
28      for (int i = 0; i<canMsg.can_dlc; i++)  {  // print the data
29        Serial.print(canMsg.data[i],HEX);
30        Serial.print(" ");
31      }
32
33      Serial.println();
34      WriteTemperature();
35    }
36    delay(1000);
37  }
38  int lastTemp = -1;
39  void WriteTemperature(){
40    if (canMsg.can_id == 0x0F1){
41
42      int t = canMsg.data[1];
43      int error = canMsg.data[0];
44
45      Print_Temperature(t,error,lastTemp);
46      Serial.print("Temp: ");
47      Serial.print(t,DEC);
48      Serial.print(", Error: ");
49      Serial.println(error);
50
51      lastTemp = t;
52    }
53  }
```

A.0.3 Write to Display Script

```
1  //Screen Stuff
2
3  #include <Adafruit_GFX.h>
4  #include <SPI.h>
5  #if defined(_GFXFONT_H_)           //are we using the new library?
6  #include <Fonts/FreeMono24pt7b.h>
7  #define ADJ_BASELINE 11            //new fonts setCursor to bottom of
   ↪  letter
8  #else
9  #define ADJ_BASELINE 0             //legacy setCursor to top of letter
10 #endif
11 #include <MCUFRIEND_kbv.h>
12 MCUFRIEND_kbv tft;
13
```

```
14 #define BLACK   0x0000
15 #define BLUE    0x001F
16 #define RED     0xF800
17 #define GREEN   0x07E0
18 #define CYAN    0x07FF
19 #define MAGENTA 0xF81F
20 #define YELLOW  0xFFE0
21 #define WHITE   0xFFFF
22
23 uint16_t ID;
24
25 void ScreenInit()
26 {
27
28     tft.reset();
29     ID = tft.readID();
30     tft.begin(ID);
31     tft.setRotation(1);
32     tft.fillScreen(BLACK);
33     tft.setFont(&FreeMono24pt7b);
34
35
36 }
37
38
39 void Print_Temperature(int temperature, int error, int last) {
40
41   if(last != temperature){
42     tft.fillScreen(BLACK);
43     tft.setCursor(0,120);
44
45     if(error){
46       tft.setTextColor(RED);
47     }else{
48       tft.setTextColor(GREEN);
49     }
50
51     tft.print("Temperature  ");
52     tft.print(temperature);
53     tft.print("C");
54   }
55
56
57 }
```

Bibliography

[Ard19] Arduino. *SPI library*. Arduino, December 2019. URL: https://www.arduino.cc/en/reference/SPI.

[Ard20] Arduino. *Arduino Uno Rev3 | Arduino Official Store*. 2020. URL: https://store.arduino.cc/usa/arduino-uno-rev3.

[con20] Wikipedia contributors. *CAN bus*. 2020. URL: https://en.wikipedia.org/w/index.php?title=CAN_bus&oldid=994118908.

[Dmi17] Dmitry. *Arduino-mcp2515*. autowp, May 2017. URL: https://github.com/autowp/arduino-mcp2515.

[Ele20] SparkFun Electronics. *Getting Started with OBD-II*. 2020. URL: https://learn.sparkfun.com/tutorials/getting-started-with-obd-ii.

[Inc03] Microchip Technology Inc. *High-Speed CAN Transceiver*. 2003. URL: http://ww1.microchip.com/downloads/en/devicedoc/21667d.pdf.

[Inc20] Microchip Technology Inc. *MCP2551 - Interface - Interface- Controller Area Network (CAN)*. 2020. URL: https://www.microchip.com/wwwproducts/en/en010405.

[Ind20] Adafruit Industries. *Adafruit-GFX-Library*. Adafruit Industries, October 2020. URL: https://github.com/adafruit/Adafruit-GFX-Library.

[Lun+10] Wei Lun et al. "Review of Researches in Controller Area Networks Evolution and Applications." In: *Proceedings of the Asia-Pacific Advanced Network* 30 (August 2010). DOI: 10.7125/APAN.30.3.

[Mon13] Scott Monroe. *Basics of debugging the controller area network (CAN) physical layer*. Texas Instruments Incorporated, 2013. URL: https://www.ti.com/lit/an/slyt529/slyt529.pdf?ts=1608722731147&ref_url=https%5C%253A%5C%252F%5C%252Fwww.google.com%5C%252F.

[Paz99] K. Pazul. *Controller Area Network (CAN) Basics*. Microchip Technology Inc., 1999. URL: http://ww1.microchip.com/downloads/en/Appnotes/00713a.pdf.

[PJR20] PJRC. *Teensy 3.6*. 2020. URL: https://www.pjrc.com/store/teensy36.html.

[pre20] prenticedavid. *MCUFRIEND$_{kbv}$*. July 2020. URL: https://github.com/prenticedavid/MCUFRIEND_kbv.

[Tha19] Pramoth Thangavel. *Arduino CAN Tutorial - Interfacing MCP2515 CAN BUS Module with Arduino*. Circuit Digest, July 2019. URL: https://circuitdigest.com/microcontroller-projects/arduino-can-tutorial-interfacing-mcp2515-can-bus-module-with-arduino.

[Tho14] Martin Thompson. *How can I determine a maximum run length for CAN-bus?* September 2014. URL: https://electronics.stackexchange.com/questions/130634/how-can-i-determine-a-maximum-run-length-for-canbus.

DTU Electrical Engineering
Department of Electrical Engineering
Technical University of Denmark

Ørsteds Plads
Building 348
DK-2800 Kgs. Lyngby
Denmark
Tel: (+45) 45 25 38 00

www.elektro.dtu.dk