

**Student: Cristiana Miranda de Farias**

**Supervisor: Dr. Deshinder Singh Gill**

# **REPORT ON CAN NETWORKS**

# TABLE OF CONTENTS

1. ABSTRACT
2. INTRODUCTION
3. CAN NETWORKS
4. ARDUINO
5. PLANNING
6. IMPLEMENTATION
7. CODE ANALYSIS
8. CONCLUSION
9. REFERENCES

## **1. ABSTRACT**

This project aims to make a brief study on Controller Area Networks (CAN) and highlight the most important aspects of its functionality. In addition, in order to explain and understand the protocol a prototype of a network have been made with Arduinos and CAN-BUS shields.

## **2. INTRODUCTION**

CAN (controller area network) is a protocol for peer to peer communication usually related to the data layer of the ISO/OSI protocol. It was initially created by Bosch to reduce the bulkiness of the wiring system connecting different components in cars, it then evolved and adapt to countless applications in automation. [1] It is defined, in its basic form, by the international standard ISO 11898-1 [2] [3] and variations in other protocols.

The system was innovative compared to most networks, it does not create a bulk of information to be transferred between two points, it create small chunks of information to be broadcast to the whole network, providing for data consistency. [4]

This report will be mainly divided in a session describing the theory behind CAN networks, its messages and how it works (item 3) then we will have a session describing and explaining the main functionalities of the platform used for the implementation of this project: the Arduino (item 4). Then there will be a session explaining the planning and the development in the project followed by the implementation of the physical part (items 5 and 6 respectively). Finally, there will be a more in depth analysis of the code used (item 7) and the discussion relating to the whole project (item 8).

## **3. CAN NETWORKS**

### **3.1. BASIC CONCEPTS**

CAN networks have a layered structure, composed of the object layer and transfer layer (corresponding to the data layer of the ISO/OSI model) and the physical layer. The object layer has the function of handling the status and messages as well as filtering what is to be received by the transfer layer. The transfer layer is responsible for bit timing and synchronization, message framing, arbitration, acknowledgement, error detection and signalling, and fault confinement. Finally we have the physical layer covers the electrical properties when transferring bits between nodes. In the most recent versions of CAN the object and transfer layer were replaced by the data exchange layer, subdivided in the logical link control sub-layer (LLC sub-layer) and the medium access control sub-layer (MAC sub-layer) [5]

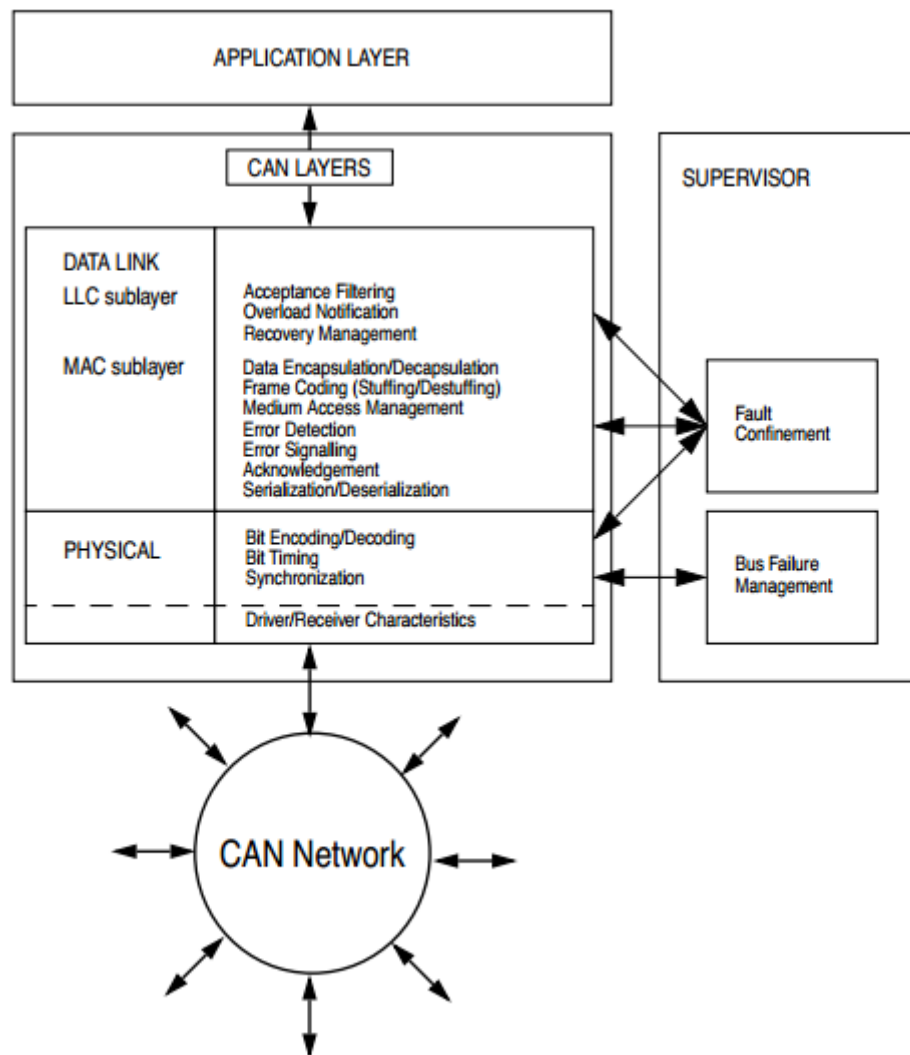


Figure 3. 1 - CAN LAYERS [5]

The CAN network is message-oriented, a property that gives the systems greater robustness as it does not describe the station contents, it only establishes a message identifier that defines contents and priority. A consequence of that is that all messages are broadcast at all times to the whole network and depending on its contents it can be filtered – any number of nodes can receive the message. The robustness and flexibility of the system can also be guaranteed as any node can be added without changing the software or hardware of the previous existing ones. [6]

Other features found on CAN are collision detection and arbitration. [4]

### 3.2. DATA TRANSFER

Every message has an identifier that serves to define the content of the message as well as its priorities. Figure 3.2 shows the frame for the standard CAN protocol.



Figure 3. 2 - Standard CAN frame [4]

The frame starts with the SOF marking the start of the file, it is used for bit synchronization in a bus after idle time. Following we have the 11bit ID that defines the priority of the message for arbitration. The ID with the lowed value will have the higher priority, in case of conflict the bus will perform bit wise arbitration [1].

The RTR flag then defines if there is need or not for a remote request - it will be set if the node sending it is requesting information. IDE is the flag that sets the frame as the standard one, DLC is the length of the data being transmitted, CRC has a checksum of the data transmitted and is used for error detection and ACK indicates an error free message has been send and then received. The data is an up to 64-bit message. [4]

Besides the data frame, described above, we have a remote frame, the error frame and the overload frame.

### 3.3. ERROR HANDLING

CAN aims to detect and signal errors as soon as they occur, the main mechanisms for that are the checksum after the data field of the frame, the checksum checks the size of the data field when it is transmitted and then recomputed on the receiver, if both values do not match an error will be computed.

There is also a frame check, that computes the frame format and the data size, and then compares each of the fields size to what they should be, any discrepancy will be signalled as error. Finally, we also have the ACK flag, the receiver sends it when the message arrives, I the transmitter does not receive an ACK frame an error is indicated [1].

### 3.4. EXTENDED FORMAT

The extended format is given by the frame used in figure 3.3.

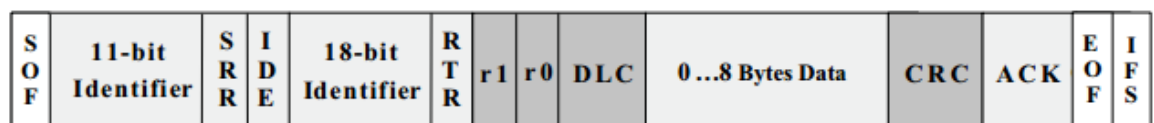


Figure 3. 3 Extended frame.

The main difference in this format is the extended 18-bit identifier.

## 4. ARDUINO

### 4.1. BASIC CONCEPTS

The Arduino is an open source microcontroller board based on either the 8-bit Atmel AVR or the 32-bit Atmel ARM architecture. With a great open source community and a varied range of applications. The board gained popularity especially in the DIY community, though it can

be used in the construction of many sophisticated embedded devices and interface a number of other platforms and APIs.



*Figure 4.1- Arduino UNO board*

There are many different Arduino boards in the market currently; the most popular is certainly the Arduino UNO, shown in figure 3.3. This board is the “standard model”, easier to learn (most projects that can be found online will be using it) and with a greater variety off shields (add-on components – see section 4.4). However, even though it is great for most uses it has a limited quantity of I/O pins and a limited memory, of only 2KB, which is certainly not enough if you are working with media such as audio, image, or even long strings. [7] Other options that may tackle those problems are the Arduino Due, Leonardo, Mega, Lilypad, or any other, each with specific characteristics that must be taken into account when developing a project.

#### 4.2. HARDWARE STRUCTURE

The basic hardware can be described by figure 4.2

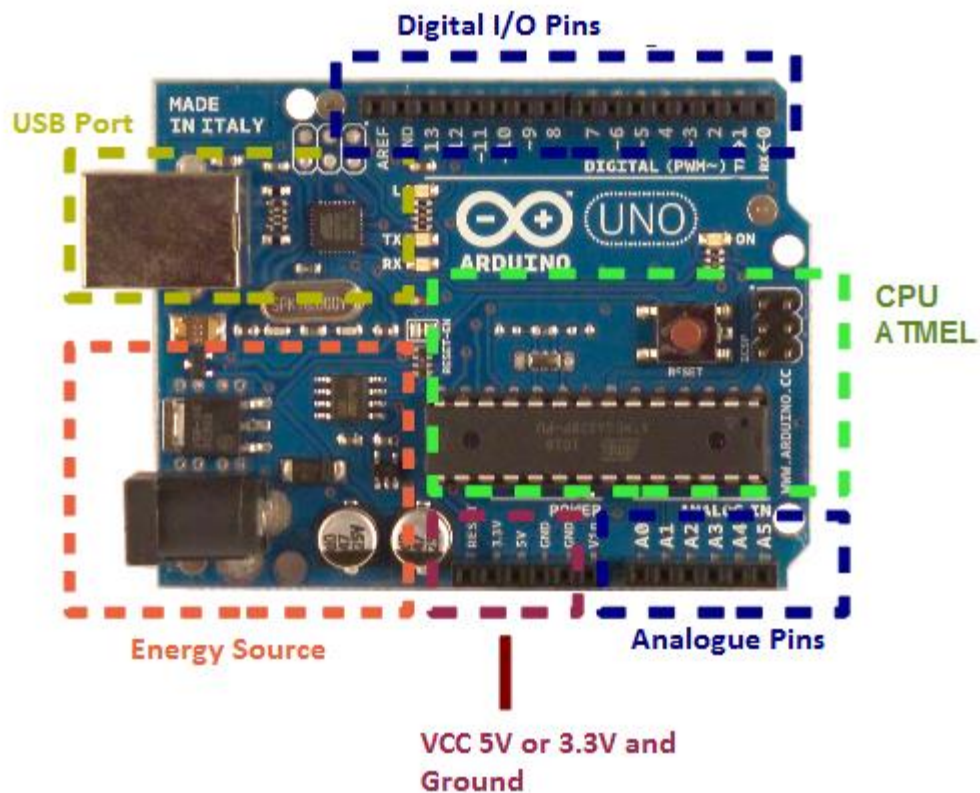


Figure 4. 2 - Arduino UNO Architecture

First we have the energy source, this component is usually used for applications that require greater input energy. When connected to an USB (which is needed for communication with the computer and downloading the codes) you are usually providing 5V input, allowing for the regulated 3.3V and 5V Vcc pins to work as output, as well as providing energy for the AVR chip. For components like some types of motor or relays you will need to provide greater input voltage, and that would come from the energy source.

You also have the I/O pins, the Arduino, being a digital component, cannot write a real analogue output, those can only be written using Pulse Width Modulation (PWM), and there are special pins for that, usually indicated in the board. Analogue input can be done via the analogue pins (A0-A5) and digital input or output can be done with any of the numbered pins, digital or analogue. [8]

Finally we have the AVR chip (in the UNO), the board processor. Most arduinos use chips of the ATmega family, particularly, the UNO is based on the ATmega328 architecture, with 32KB flash memory, SRAM of 2KB and EPROM of 1KB and a 16 MHz clock. Table 4.1 summarizes the main characteristics of the Arduino UNO.[9]

Table 4. 1 - Arduino UNO Features

Microcontroller	ATmega328
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limits)	6-20V

Digital I/O Pins	14 (of which 6 provide PWM output)
Analog Input Pins	6
DC Current per I/O Pin	40 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	32 KB (ATmega328) of which 0.5 KB used by bootloader
SRAM	2 KB (ATmega328)
EEPROM	1 KB (ATmega328)
Clock Speed	16 MHz
Length	68.6 mm
Width	53.4 mm
Weight	25 g

#### 4.3. PROGRAMMING

The Arduino can be programmed via an IDE that can be downloaded from the official Arduino website. Arduino UNO comes with a bootloader that allows code to be easily uploaded by selecting the board and the serial port your device is connected in the IDE, as shown in figure 4.3.[9]

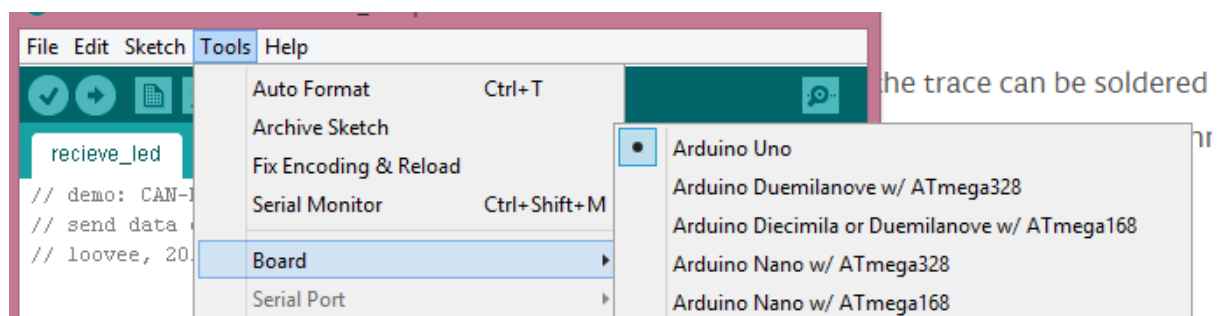


Figure 4. 3 - Board Selection and Serial Port

The language used for programming is C or C++, and the code is structured basically with a setup function, where the baudrate, pins and other variables are set and a loop function, that will be the one that will keep running and calling other functions repetitively.

The IDE for programming is shown in figure 4.4 and the most important buttons are marked.



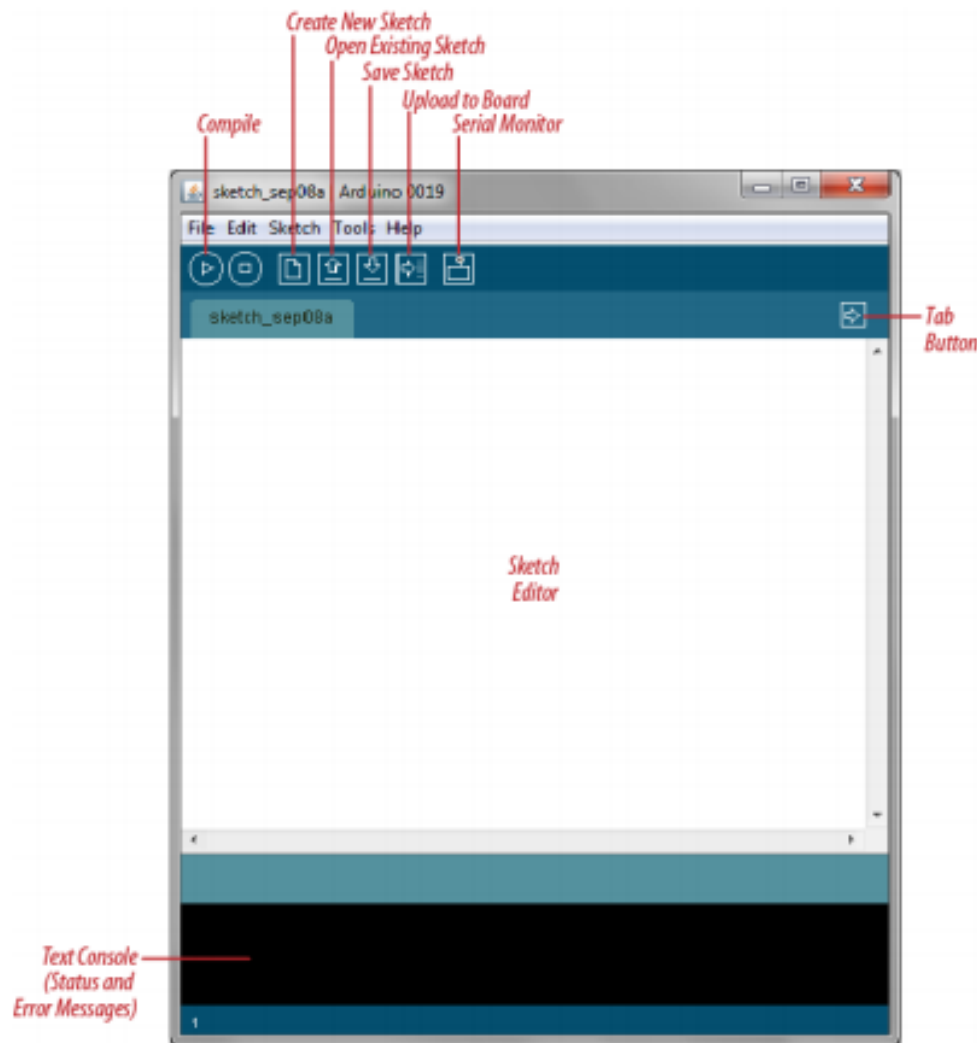


Figure 4. 4 - Arduino IDE [10]

One important feature of this IDE is the Serial Monitor, that can display data from any of the variables with the function `Serial.print()`.

#### 4.4. SHIELDS

Shields are accessories that have an interface with the Arduino board and extend its capabilities, the idea behind them is that they are easy to mount and cheap to produce. In the market a wide array of shields can be found, giving the Arduino an easy way to perform the most different functions, from communications, to motor control or joystick. In this project we will be using the CAN-BUS shield, shown in figure 4.5.

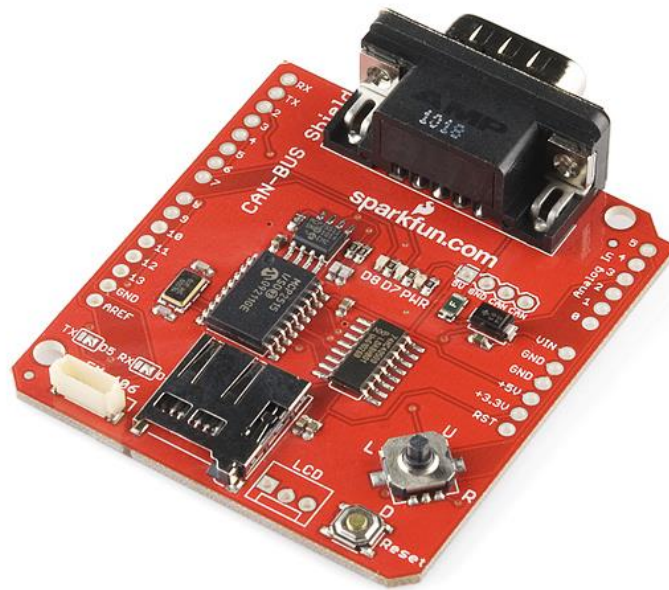


Figure 4. 5- CAN-BUS Shield

This Shield uses the microchip MCP2515 CAN controller with MCP2551 CAN transceiver and has a standard 9-way sub-D connector. It also has a micro SD card holder, GPS and LCD connectors, two LED indicator and a joystick for menu navigation. [11]

## 5. PLANNING AND DEVELOPMENT

The planning of this project was done with the aid of the Gantt Chart shown in figure 5.1.

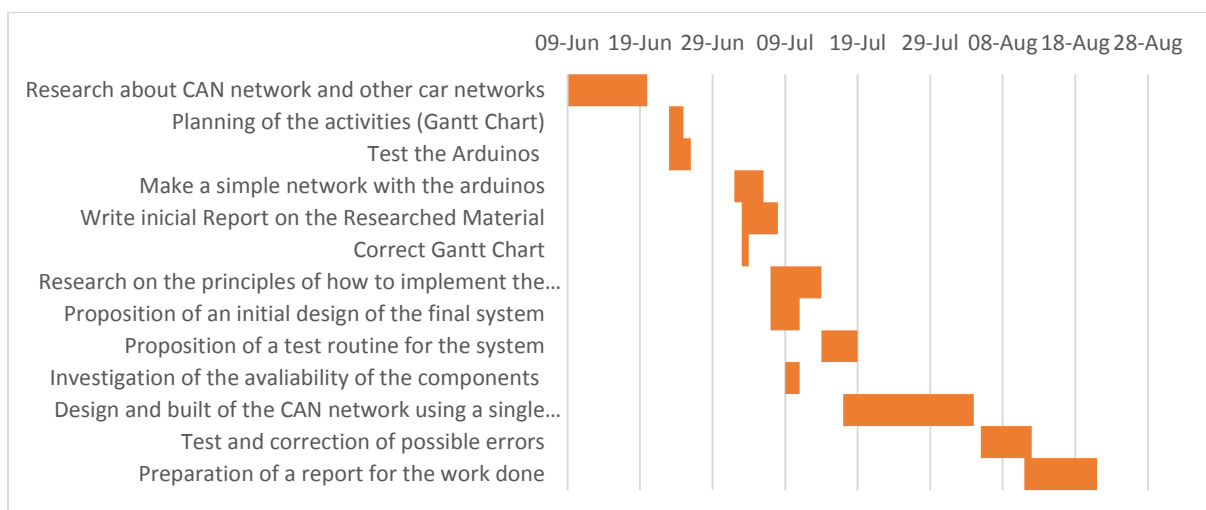


Figure 5. 1 - Gantt Chart

The project started with the research on how CAN networks work and a revision on the main structure of Arduino codes and usage. After this initial phase and after the main planning of the activities the

next step we took was make a few simple Arduino projects in order to review how to use the microcontroller.

When we were secure on how to work with the board, we started making tests with the CAN-BUS shield and the Arduino. Initially we build a simple circuit to send and receive packets from one board to the other, after that we started using the CAN Analyser to find a way to see the data on the network. In the time we were using the CAN Analyser we also started the final circuit design with three Arduinos and we ordered the needed components. Figure 5.2 shows the list of what was ordered.









	Product Name
	CAN-BUS Shield * This product is not available in the requested quantity and may mean that dispatch will be delayed. Usually available 5-10 working days.
	Arduino Uno - R3
	LM335AZ - Temperature Sensor
	Green 20x4 LCD Display
	Triple Axis Accelerometer Breakout - MMA8452Q
	Air quality sensor
	Hitec Servo Motor HS-55
	Twisted Servo Extension Cable 6" Female - Female

Figure 5. 2- Components

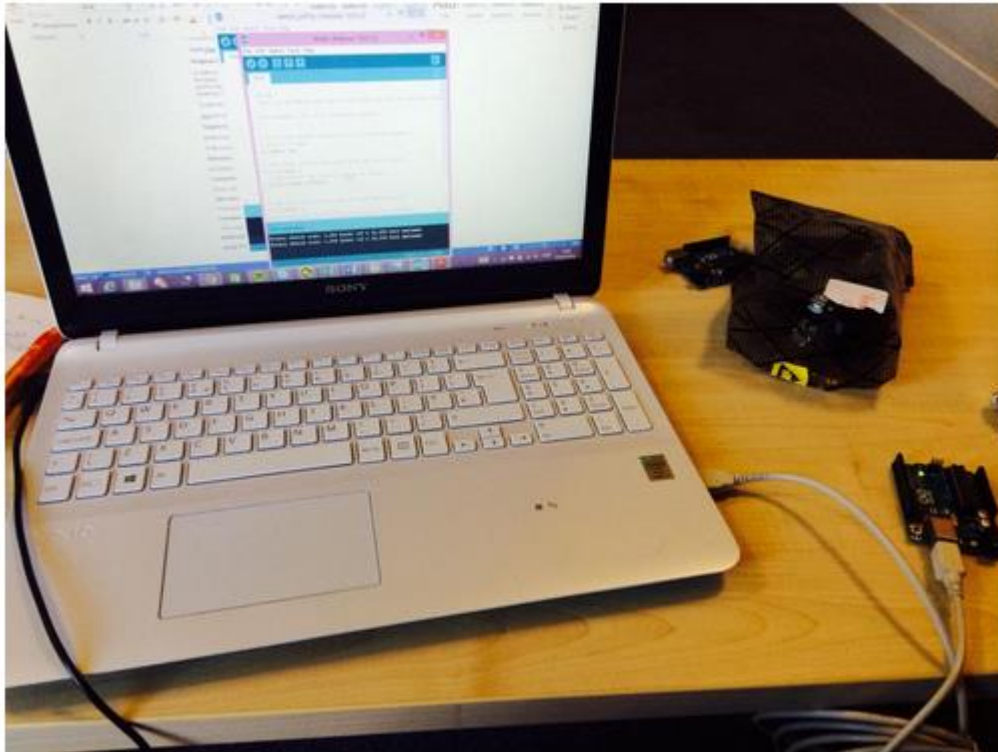
Because a long time was spent in trying to use the CAN analyser, the planned activities suffered a delay when compared to the proposed Gantt chart in figure 5.1, but ultimately the final proposed design was built and demonstrated.

## 6. IMPLEMENTATION

During the course of this project, many circuits were implemented and programmed before we moved to the final design. This section will look in how each of those worked and how each design helped us in the process of both understanding CAN networks and building the prototype.

### 6.1. BLINK FUNCTION

The first code we programmed in the Arduino was the Blink code, found in the example library. Our aim in using this code was to test if all the boards were working properly. Figure 6.1 shows the simple connection between the board and the computer.



*Figure 6. 1 - Arduino working with Blink function*

This simple example found in the Arduino library blinks a LED that can either be the one already found on the board or one that is connected in any chosen output pin.

Figure 6.2, bellow, shows the code for this function.

```

int led = 13;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);  // turn the LED on (HIGH is the voltage level)
  delay(1000);              // wait for a second
  digitalWrite(led, LOW);   // turn the LED off by making the voltage LOW
  delay(1000);              // wait for a second
}

```

Figure 6. 2- Blink Code

This function defines the in-board LED as the output pin and then writes HIGH and LOW values followed by a one-second delay.

## 6.2. TWO ARDUINOS

Knowing the arduinos were properly working we had to focus on making them communicate using the CAN-BUS shield. The first step in doing so was to make the connections between them, figure 6.3 shows the pin layout of the sub-D connector that would be linking the arduinos.

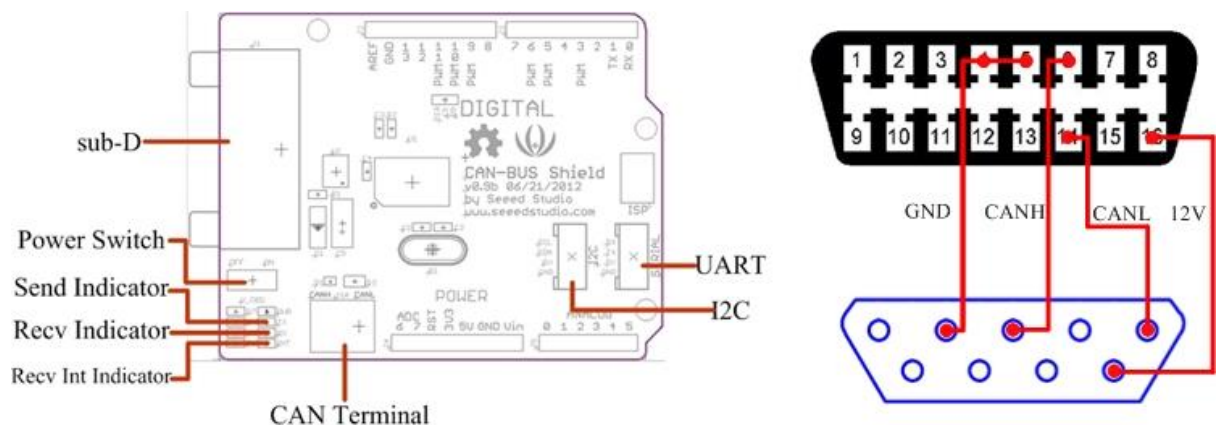


Figure 6. 3 - CAN-BUS layout [12]

The connection should be made by linking the CANL from one Arduino to the other, and the same must be done with CANH, also the ground and Vcc must be connected. It is important to remember that there must be a difference between CAN High and Low, to guarantee that difference a between the connections we put a resistor. Figure 6.4 shows the code for the receiver and figure 6.5 for the transmitter. The seed-studio CAN Bus shield library must be installed for this codes to work.

```

#include <SPI.h>
#include "mcp_can.h"

unsigned char Flag_Recv = 0;
unsigned char len = 0;
unsigned char buf[8];
char str[20];

void setup()
{
    Serial.begin(115200);

START_INIT:
|
    if(CAN_OK == CAN.begin(CAN_500KBPS))           // init can bus : baudrate = 500k
    {
        Serial.println("CAN BUS Shield init ok!");
    }
    else
    {
        Serial.println("CAN BUS Shield init fail");
        Serial.println("Init CAN BUS Shield again");
        delay(100);
        goto START_INIT;
    }
}

void loop()
{
    if(CAN_MSGAVAIL == CAN.checkReceive())           // check if data coming
    {
        CAN.readMsgBuf(&len, buf);    // read data, len: data length, buf: data buf

        for(int i = 0; i<len; i++)    // print the data
        {
            Serial.print(buf[i]);Serial.print("\t");
        }
        Serial.println();
    }
}

```

Figure 6. 4 – Receiver

```

#include <mcp_can.h>
#include <SPI.h>

void setup()
{
    Serial.begin(115200);

START_INIT:

    if(CAN_OK == CAN.begin(CAN_500KBPS))           // init can bus : baudrate = 500k
    {
        Serial.println("CAN BUS Shield init ok!");
    }
    else
    {
        Serial.println("CAN BUS Shield init fail");
        Serial.println("Init CAN BUS Shield again");
        delay(100);
        goto START_INIT;
    }
}

void loop()
{
    unsigned char stamp[8] = {0, 1, 2, 3, 4, 5, 6, 7};

    CAN.sendMsgBuf(0x00, 0, 8, stamp); // send data: id = 0x00, standard frame, data len = 8, stamp: data buf
    delay(10);                         // when the delay less than 20ms, you should use receive_interrupt
}

```

Figure 6.5 - Transmitter code

These two codes, once properly working, became the core of this project: all the following ones used them as base, making the necessary alterations. Figure 6.6 shows the data received in one of the tests when the string sent was made only of zeroes and the message ID was one.

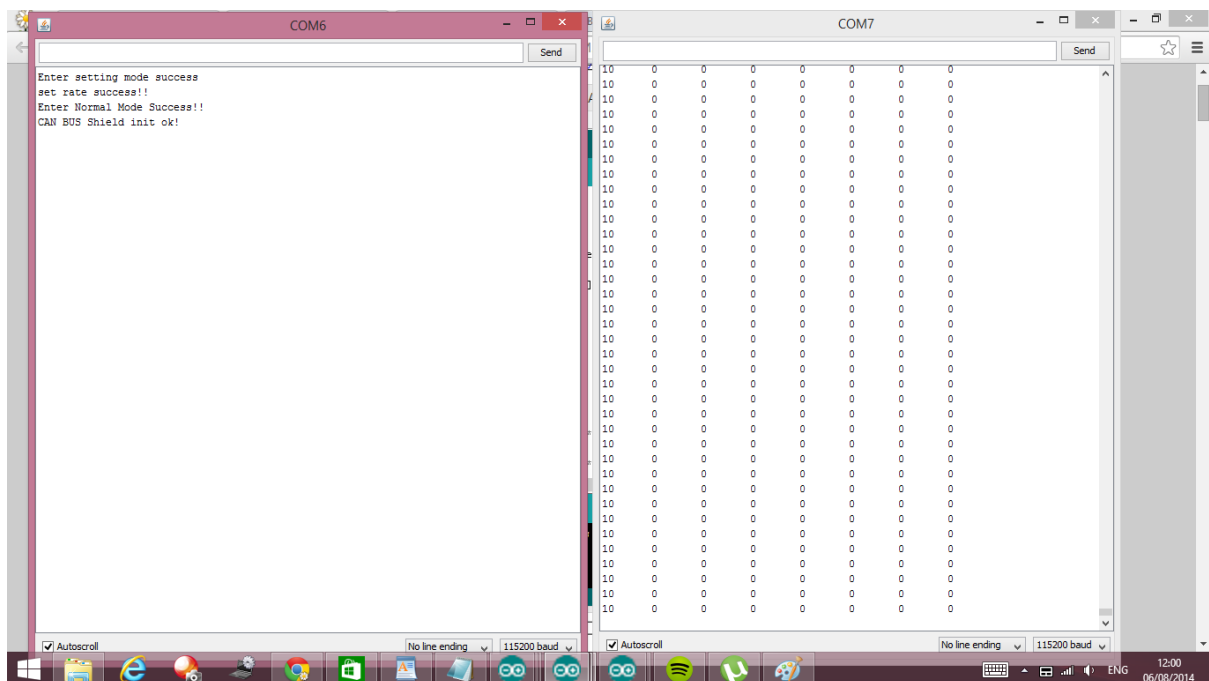


Figure 6.6 – Outputs

### 6.3. CAN ANALYSER

When the communication between the two nodes was established, we tried to visualise the packets in a kind of oscilloscope known as CAN analyser. This step of our implementation was the most problematic one yielding almost no result. Figure 6.7 shows the visualization of packets.



Figure 6. 7 - CAN Analyser visualization

All other times we tried to repeat the process for viewing the packets were unsuccessful, we could not configure the analyser to show us the results, even though we were sure the nodes were sending and transmitting.

### 6.4. SEND AND RECEIVE WITH LED AND POTENTIOMETER

When attempting to work with the CAN analyser we wanted to be sure that the nodes were transmitting information between each other. Initially we used the information on the serial monitor, as shown in section 6.2, but to visualize more easily and not to rely on the computer we decided to make a simple project where one node would read the data from a potentiometer and send the variations in the signal to the other node. The receiver would then read these variations and proportionally set the intensity of an LED's light, our system would work as a simple dimmer. Figure 6.8 shows the setup.



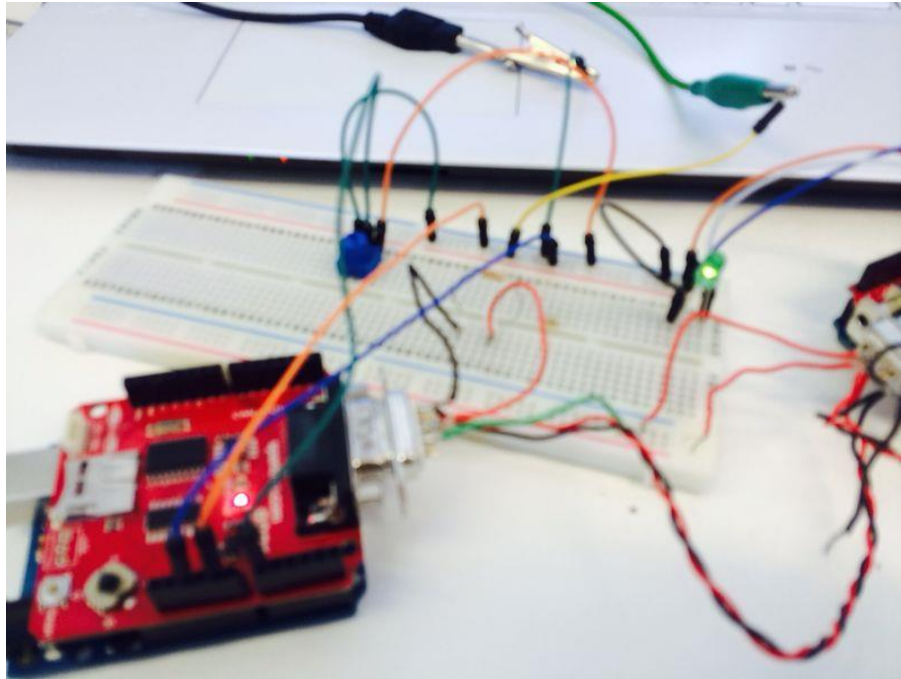


Figure 6. 8 - Setup for the LED and Potentiometer communication

Figure 6.9 shows a sketch of the schematic for the Arduino connection with the components, the connection between arduinos is done via the sub-D connector as shown in figure 6.3.

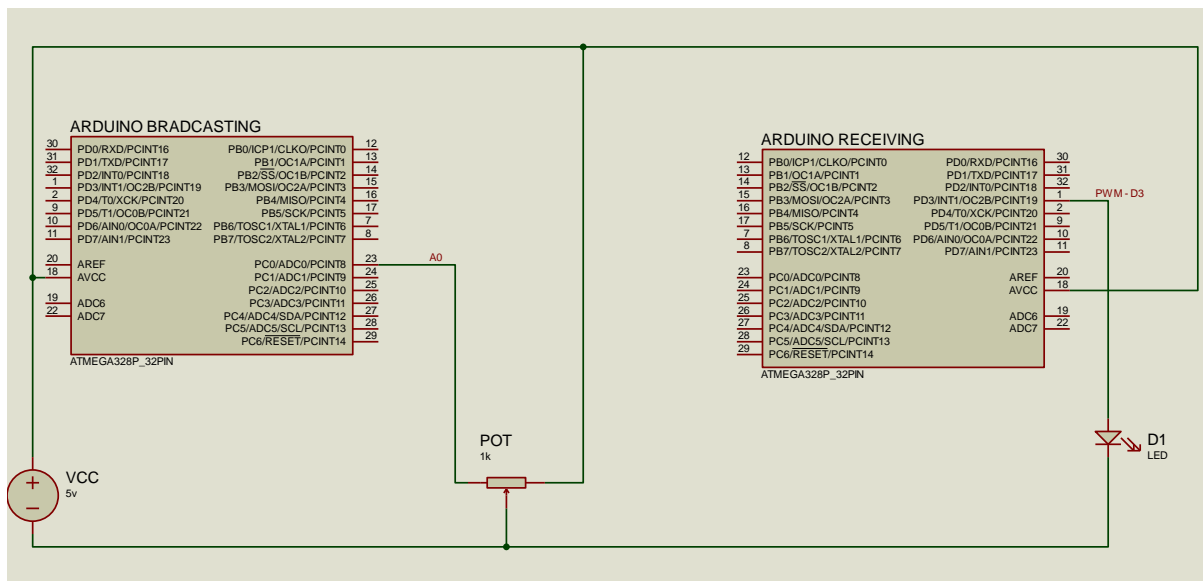


Figure 6. 9 – Schematic

Finally figures 6.10 and 6.11 show the codes for the transmitter and receiver respectively.

```

#include <mcp_can.h>
#include <SPI.h>
unsigned char stmp1[8] = {0, 200, 150, 250, 100, 50, 180, 190};
String str;
unsigned char stmp0[1];
int pot = A0;
int out_pot, pot_value;

void setup()
{
    Serial.begin(115200);

START_INIT:

    if(CAN_OK == CAN.begin(CAN_10KBPS))                // init can bus : baudrate = 500k
    {
        Serial.println("CAN BUS Shield init ok!");
    }
    else
    {
        Serial.println("CAN BUS Shield init fail");
        Serial.println("Init CAN BUS Shield again");
        delay(100);
        goto START_INIT;
    }
}

void loop()
{
    pot_value = analogRead(pot);
    out_pot = map(pot_value, 0, 1023, 0, 255);
    stmp0[0] = (unsigned char) out_pot;
    Serial.print(stmp0[0]);
    Serial.print("\n");
    CAN.sendMsgBuf(0x00, 0, 1, stmp0);    // send data: id = 0x00, standrad flame, data len = 8, stmp: data buf
                                         // when the delay less than 20ms, you shield use receive_interrupt
    delay(1);
}

```

Figure 6. 10 – Transmitter

```

#include <SPI.h>
#include "mcp_can.h"

unsigned char Flag_Recv = 0;
unsigned char len = 0;
unsigned char buf[1];
int led = 3;
int i = 0;
char str[20];
int outputValue = 0;

void setup()
{
    pinMode(led, OUTPUT);
    Serial.begin(115200);

START_INIT:

    if(CAN_OK == CAN.begin(CAN_10KBPS))                // init can bus : baudrate = 500k
    {
        Serial.println("CAN BUS Shield init ok!");
    }
    else
    {
        Serial.println("CAN BUS Shield init fail");
        Serial.println("Init CAN BUS Shield again");
        delay(100);
        goto START_INIT;
    }
}

void loop()
{
    if(CAN_MSGAVAIL == CAN.checkReceive())              // check if data coming
    {
        CAN.readMsgBuf(&len, buf);    // read data, len: data length, buf: data buf
        Serial.print(buf[0]);

        analogWrite(led, buf[0]);
        Serial.println();
    }
}

```

*Figure 6. 11 - Receiver*

## 6.5. THREE ARDUINOS

Our next step was a simple connection between three arduinos. There were two configurations possible in this case, the first one being two receivers and one transmitter, seeing we would not need any message filtering in this case we just recycled the codes from section 6.2 and implemented them.

The second possibility were two arduinos sending the message and one receiving, we made, in this case some small alterations in the codes of section 6.2. Figure 6.12 and 6.13 show the loop function for the transmitters and 6.14 the loop function for the receiver.

```

void loop()
{
    unsigned char stmp[8] = {0, 1, 2, 3, 4, 5, 6, 7};

    CAN.sendMsgBuf(0x01, 0, 8, stmp); // send data: id = 0x01, standrad flame, data len = 8, stmp: data buf
    delay(10);                       // when the delay less than 20ms, you shield use receive_interrupt
}

```

Figure 6. 12 - Transmitter 1

```

void loop()
{
    unsigned char stmp[8] = {0, 1, 2, 3, 4, 5, 6, 7};

    CAN.sendMsgBuf(0x02, 0, 8, stmp); // send data: id = 0x01, standrad flame, data len = 8, stmp: data buf
    delay(10);                       // when the delay less than 20ms, you shield use receive_interrupt
}

```

Figure 6. 13 - Transmitter 2

```

void loop()
{
    if(CAN_MSGAVAIL == CAN.checkReceive()) // check if data coming
    {

        CAN.readMsgBuf(&len, buf); // read data, len: data length, buf: data buf

        ID = CAN.getCanId();
        Serial.print (ID);
        if (ID == 0x01)
        {
            for(int i = 0; i<len; i++) // print the data
            {
                Serial.print(buf[i]);Serial.print("\t");
            }
        }
        else
            Serial.print ("Wrong ID \n");

        Serial.println();
    }
}

```

Figure 6. 14 - Receiver

## 6.6. FINAL DESIGN

Finally, we proposed one last design, where our three node bus would send meaningful data, similar to a real system. This design would have one node reading the temperature and broadcasting it and the two receiving nodes acting as actuators, one moving a servo after the temperature got higher than a pre-determined threshold and the other reading it and showing the data on a LCD display. Figure 6.15 shows a diagram for the final design.

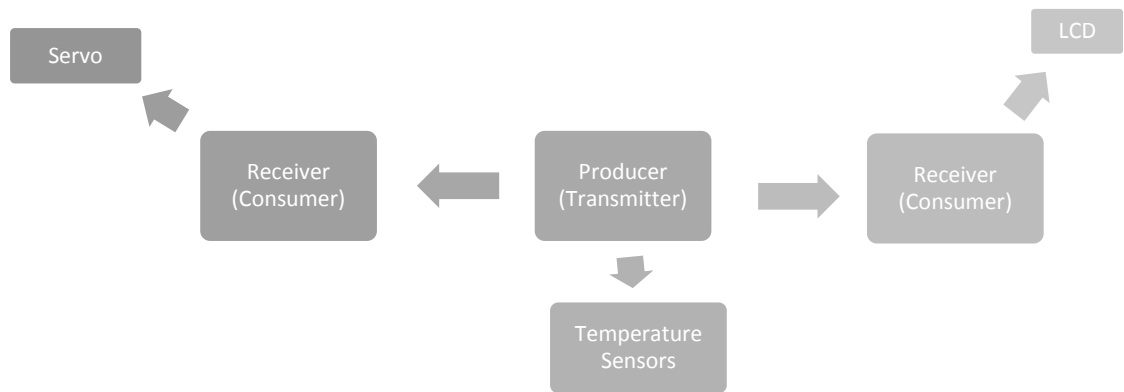


Figure 6. 15 - Diagram for the final design

Figure 6.16 shows the configuration we build with the LCD display and the servo motor, the temperature sensor is connected to the breadboard.

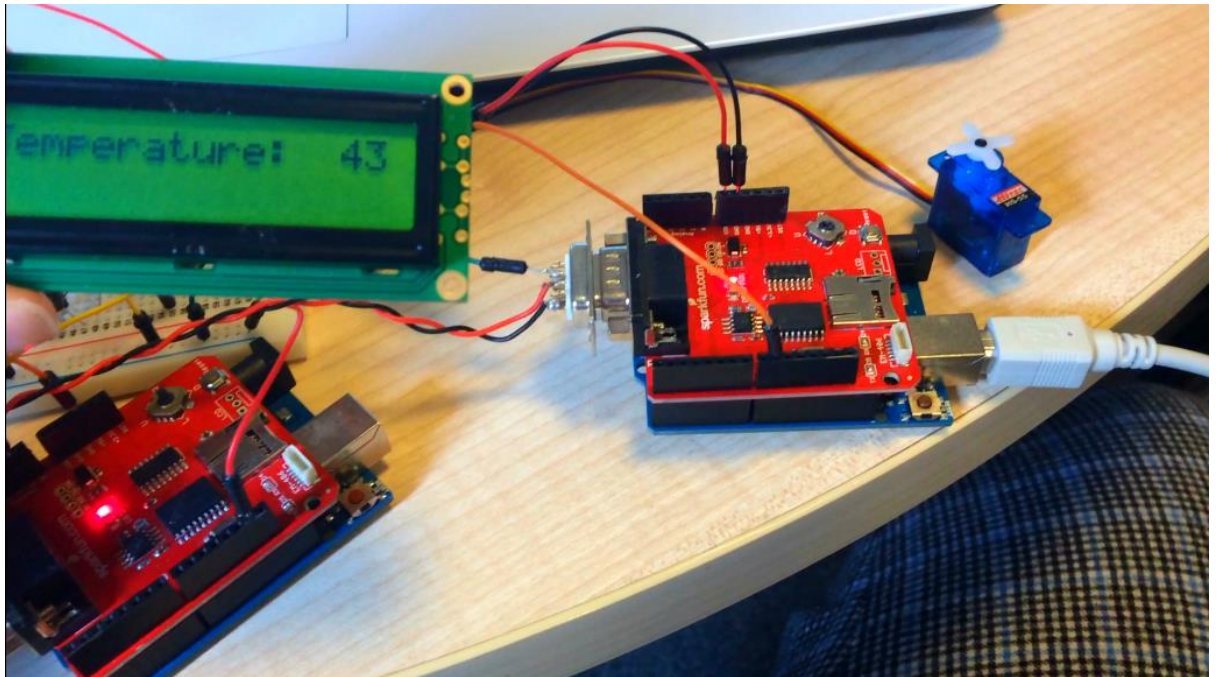


Figure 6. 16 - Circuit with the Arduinos, sensor and actuators

Figures 6.17 through 6.19 show the codes for the final design.

```

#include <mcp_can.h>
#include <SPI.h>
int pin1 = A1;    // select the input pin for the potentiometer
int pin2 = A2;
float tempV1, tempV2, tempK1, tempK2, tempC1, tempC2, tempM; // variable to store the value coming from the sensor
unsigned char temps[7];

void setup()
{
    Serial.begin(115200);

START_INIT:

    if(CAN_OK == CAN.begin(CAN_500KBPS))                // init can bus : baudrate = 500k
    {
        Serial.println("CAN BUS Shield init ok!");
    }
    else
    {
        Serial.println("CAN BUS Shield init fail");
        Serial.println("Init CAN BUS Shield again");
        delay(100);
        goto START_INIT;
    }
}

void loop()
{
    // read the value from the sensor:
    tempV1 = analogRead(pin1);
    tempK1 = (tempV1/1023)*500;
    tempC1 = tempK1 - 273.15;

    tempV2 = analogRead(pin2);
    tempK2 = (tempV2/1023)*500;
    tempC2 = tempK2 - 273.15;

    tempM = (tempC1 + tempC2)/2;

    temps[0] = (unsigned char) tempV1;
    temps[1] = (unsigned char) tempK1;
    temps[2] = (unsigned char) tempC1;
    temps[3] = (unsigned char) tempV2;
    temps[4] = (unsigned char) tempK2;
    temps[5] = (unsigned char) tempC2;
    temps[6] = (unsigned char) tempM;

    CAN.sendMsgBuf(0x00, 0, 8, temps); // send data: id = 0x00, standrad flame, data len = 8, stmp: data buf
    delay(1000);                       // when the delay less than 20ms, you shield use receive_interrupt
}

```

Figure 6. 17 - Temperature Broadcast

```

#include <SPI.h>
#include "acp_can.h"
#include <SoftwareSerial.h>

#define txPin 2
#include <Servo.h>

Servo myservo;
int pos = 0;

SoftwareSerial LCD = SoftwareSerial(0, txPin);

const int LCDdelay=2;

// wbp: goto with row & column
void lcdPosition(int row, int col) {
  LCD.write(0xFE); //command flag
  LCD.write((col + row*64 + 128)); //position
  delay(LCDdelay);
}

void clearLCD(){
  LCD.write(0xFE); //command flag
  LCD.write(0x01); //clear command.
  delay(LCDdelay);
}

void backlightOn() { //turns on the backlight
  LCD.write(0x7C); //command flag for backlight stuff
  LCD.write(128); //light level for off.
  delay(LCDdelay);
}

void serCommand(){ //a general function to call the command flag for issuing all other commands
  LCD.write(0xFE);
}

unsigned char Flag_Recv = 0;
unsigned char len = 0;
unsigned char buf[8];
char str[20];

void setup()
{
  Serial.begin(115200);
  pinMode(txPin, OUTPUT);
  LCD.begin(9600);
  clearLCD();

  START_INIT:

  if(CAN_OK == CAN.begin(CAN_500Kbps)) // init can bus : baudrate = 500k
  {
    Serial.println("CAN BUS Shield init ok!");
  }
  else
  {
    Serial.println("CAN BUS Shield init fail");
    Serial.println("Init CAN BUS Shield again");
    delay(100);
    goto START_INIT;
  }
}

void loop()
{
  if(CAN_MSGAVAIL == CAN.checkReceive()) // check if data coming
  {
    CAN.readMsgBuf(&len, buf); // read data, len: data length, buf: data buf

    Serial.println("Temperature C1");
    Serial.println(buf[2]);

    Serial.println("Temperature C2");
    Serial.println(buf[5]);

    Serial.println("Temperature Mean");
    Serial.println(buf[6]);
  }

  lcdPosition(0,0);
  LCD.print("Temperature:");
  lcdPosition(0,14);
  LCD.print(buf[6]);

  delay(100);
}

```

Figure 6. 18- LCD Display Temperature

```

#include <SPI.h>
#include "mcp_can.h"
#include <Servo.h>

Servo myservo;
int i=10, pos = 0;

unsigned char Flag_Recv = 0;
unsigned char len = 0;
unsigned char buf[8];
char str[20];

void setup()
{
    Serial.begin(115200);
    myservo.attach(9);

START_INIT:

    if(CAN_OK == CAN.begin(CAN_500KBPS))                // init can bus : baudrate = 500k
    {
        Serial.println("CAN BUS Shield init ok!");
    }
    else
    {
        Serial.println("CAN BUS Shield init fail");
        Serial.println("Init CAN BUS Shield again");
        delay(100);
        goto START_INIT;
    }
}

void loop()
{
    if(CAN_MSGAVAIL == CAN.checkReceive())                // check if data coming
    {
        CAN.readMsgBuf(&len, buf);    // read data, len: data length, buf: data buf
        Serial.println(buf[6]);
    }
    if (buf[6] >= 40)
    {
        if (pos>0 || pos<180)
        {
            pos=pos+i;
            myservo.write(pos);
            delay(250);
        }
        if (pos == 180)
            i=(-10);
        if (pos == 0)
            i=10;
    }
    myservo.write(pos);
    delay(5);
}

```

Figure 6. 19 - Servo Move



## 7 CODE ANALYSIS

### 7.1 MCP library

Before analysing the codes we must take a look at the library used for the CAN bus shield, in the sections below the main functions used on the codes in this report are described. [12]

#### 7.1.1 CAN.begin(*baud\_rate*)

Initializes the CAN shield and sets its baud rate. The library defines the available baud rates, so the user must choose among those values. This function return weather the initialization was successful or not.

- Baud\_rate – one of the pre-defined values listed bellow:

CAN\_5KBPS  
CAN\_10KBPS  
CAN\_20KBPS  
CAN\_40KBPS  
CAN\_50KBPS  
CAN\_80KBPS  
CAN\_100KBPS  
CAN\_125KBPS  
CAN\_200KBPS  
CAN\_250KBPS  
CAN\_500KBPS  
CAN\_1000KBPS

#### 7.1.2 CAN.sendMsgBuf(*INT id, INT ext, INT length, data\_buffer*)

This is the function to send a message.

- Id – the ID address of the message
- Ext – if the message is standard (0) or extended (1)
- Length – size of the data contained in the message.
- Data\_buffer – The contents of the message

#### 7.1.3 CAN.readMsgBuf(*unsigned char length, unsigned char data\_buffer*)

Function that receives a message in the buffer.

- Length – size of the data contained in the message.
- Data\_buffer – The contents of the message

#### 7.1.4 getCanID(*void*)

Function that returns the ID of a message received.

#### 7.1.5 CAN.checkReceive(*void*)

Function that checks if a message was received. Returns 1 if frame arrives and 0 if not.

## 7.2 SEND AND RECEIVE

Both codes start by including the libraries and setting up the global variables. Following that there is the setup code. Since we are just using a simple routine for receiving and broadcasting information the setup just has to initialize the CAN bus. We can note, as shown in figure 7.1, that both codes have those lines.

```
void setup()
{
    Serial.begin(115200);

    START_INIT:
    |
    if(CAN_OK == CAN.begin(CAN_500KBPS))
    {
        Serial.println("CAN BUS Shield init ok!");
    }
    else
    {
        Serial.println("CAN BUS Shield init fail");
        Serial.println("Init CAN BUS Shield again");
        delay(100);
        goto START_INIT;
    }
}
```

Figure 7. 1 - Setup function

First we start the serial communication link within the Arduino to allow information to be displayed on the serial monitor, we then set the baud rate and start the shield. If the initialization has any error it will give a message and keep retrying until it is successful.

If everything was successful the function will move to the loop function, figure 7.2 shows that function for the transmitter.

```
void loop()
{
    unsigned char stap[8] = {0, 1, 2, 3, 4, 5, 6, 7};

    CAN.sendMsgBuf(0x00, 0, 8, stap); // send data: id = 0x00, standrad flame, data len = 8, stap: data buf
    delay(10);                       // when the delay less than 20ms, you shield use receive_interrupt
}
```

Figure 7. 2 - Loop that sends a message

This simple function first defines a message, in figure 7.2 for example, this message is “0 1 2 3 4 5 6 7”. After the message is defined we use the CAN.sendMsgBuf to send it, the ID is defined as 0x00 we are using a standard frame and because the message is composed of 8 numbers the length is eight.

```
void loop()
{
    if(CAN_MSGAVAIL == CAN.checkReceive()) // check if data coming
    {
        CAN.readMsgBuf(&len, buf); // read data, len: data length, buf: data buf

        for(int i = 0; i<len; i++) // print the data
        {
            Serial.print(buf[i]);Serial.print("\t");
        }
        Serial.println();
    }
}
```

Figure 7. 3 - Receiving loop

The receiver loop, shown in figure 7.3, starts by checking if a message has come or not with the CAN.checkReceive() function, if nothing has come it repeats itself until something appears. Once a message is received it is read with the CAN.readMsgBuf function () and then printed in the serial monitor.

### 7.3 SEND AND RECEIVE WITH LED AND POTENTIOMETER

These codes are very similar to the original send and receive ones, the main difference are the contents of the message and the operations with the LED and potentiometer.

Initially we are going to analyse the transmitter code, it starts by defining its global variables, and including the libraries. It is in this part that we define the pin in which the potentiometer will be connected – A0 in this case. The rest of the initialization occurs exactly like the previous codes.

Next, we have the loop function, shown in figure 7.4.

```
void loop()
{
    pot_value = analogRead(pot);
    out_pot = map(pot_value, 0, 1023, 0, 255);
    stap0[0] = (unsigned char) out_pot;
    Serial.print(stap0[0]);
    Serial.print("\n");
    CAN.sendMsgBuf(0x00, 0, 1, stap0); // send data: id = 0x00, standrad flame, data len = 8, stap: data buf
    // when the delay less than 20ms, you shield use receive_interrupt
    delay(1);
}
```

Figure 7. 4 - Potentiometer transmitter

This function starts by reading the device and mapping its values (with a resolution of 1024) to 256, we do that to divide the possible values in sections that will be written in the LED. After the mapping we send the message in the way described in section 7.2.

We then have the receiver function, in this case we are writing on a LED so its initialization as an output must be done in the setup function. By defining the pin to be used as a global variable and adding the following line to the setup function we can operate in the LED.

```
pinMode(led, OUTPUT);
```

The rest of the setup is similar to the one in figure 7.1.

```
void loop()
{
    if(CAN_MSGAVAIL == CAN.checkReceive()) // check if data coming
    {
        CAN.readMsgBuf(&len, buf); // read data, len: data length, buf: data buf
        Serial.print(buf[0]);

        analogWrite(led, buf[0]);
        Serial.println();
    }
}
```

Figure 7. 5 - LED receiver

Figure 7.5 shows the loop function for the receiver, almost the same as the original one, but in this case we don't have a string of values to print so we don't need to use the repetition

structure. Also there is the `analogWrite()` function to write on the LED (modulating the analogue value with PWM).

#### 7.4 THREE ARDUINOS

For the three arduinos with two of them transmitting we use the original codes for transmitting, only changing the ID for one of the messages, and for test purposes, the message contents. In our code one of the IDs was 0x01 and we wanted to receive only that.

Figure 7.6 shows our function.

```
void loop()
{
    if(CAN_MSGAVAIL == CAN.checkReceive())           // check if data coming
    {

        CAN.readMsgBuf(&len, buf);    // read data, len: data length, buf: data buf

        ID = CAN.getCanId();
        Serial.print (ID);
        if (ID == 0x01)
        {
            for(int i = 0; i<len; i++)    // print the data
            {
                Serial.print(buf[i]);Serial.print("\t");
            }
        }
        else
            Serial.print ("Wrong ID \n");

        Serial.println();
    }
}
```

*Figure 7. 6 - Three Arduinos Receiver*

The code is basically like the original receiver, but in this case the wanted to filter the message. What happened was that we had the `readMsgBuf` reading at all times, so we kept getting the IDs of the message read and by using a conditional clause only printed the message if the ID was the desired one.

#### 7.5 FINAL DESIGN

For the final design we will first look at the transmitter. This node was reading the temperature of a sensor at all times and transmitting it to the other two nodes. Figure 7.7 shows the loop function.

```

void loop()
{
    // read the value from the sensor:
    tempV1 = analogRead(pin1);
    tempK1 = (tempV1/1023)*500;
    tempC1 = tempK1 - 273.15;

    tempV2 = analogRead(pin2);
    tempK2 = (tempV2/1023)*500;
    tempC2 = tempK2 - 273.15;

    tempM = (tempC1 + tempC2)/2;

    temps[0] = (unsigned char) tempV1;
    temps[1] = (unsigned char) tempK1;
    temps[2] = (unsigned char) tempC1;
    temps[3] = (unsigned char) tempV2;
    temps[4] = (unsigned char) tempK2;
    temps[5] = (unsigned char) tempC2;
    temps[6] = (unsigned char) tempM;

    CAN.sendMsgBuf(0x00, 0, 8, temps); // send data: id = 0x00, standard frame, data len = 8, stamp: data buf
    delay(1000);                      // when the delay less than 20ms, you should use receive_interrupt
}

```

*Figure 7. 7 - Temperature read and broadcast*

The temperatures are read from the sensor as a voltage value, we used two sensors (by calculating the average) so we could have a better accuracy when getting the temperature value. After reading the voltage value we calculate the temperature in Kelvin and then we subtract 273.15 to get it in Celsius. After getting the two Celsius values we calculate the average and send all those values to the CAN bus.

After transmitting the message we receive it in our two actuator nodes. First we will analyse the LCD node. Before the setup function we create and define some functions to operate with the LCD, as shown in figure 7.8.

```

SoftwareSerial LCD = SoftwareSerial(0, txPin);

const int LCDdelay=2;

// wbp: goto with row & column
void lcdPosition(int row, int col) {
  LCD.write(0xFE); //command flag
  LCD.write((col + row*64 + 128)); //position
  delay(LCDdelay);
}
void clearLCD(){
  LCD.write(0xFE); //command flag
  LCD.write(0x01); //clear command.
  delay(LCDdelay);
}
void backlightOn() { //turns on the backlight
  LCD.write(0x7C); //command flag for backlight stuff
  LCD.write(157); //light level.
  delay(LCDdelay);
}
void backlightOff(){ //turns off the backlight
  LCD.write(0x7C); //command flag for backlight stuff
  LCD.write(128); //light level for off.
  delay(LCDdelay);
}
void serCommand(){ //a general function to call the command flag for issuing all other commands
  LCD.write(0xFE);
}

```

---

*Figure 7. 8 - LCD functions*

Before those functions to use the LCD we must include the SoftwareSerial library, with that we can assign the txPin to the LCD, as shown in the first line of the figure.

Then we have some routines to perform the most common actions on the display, such as turning the backlight on and off, clearing the data and positioning the cursor. After them we initialize the LCD and the CAN in the setup function, figure 7.9 shows the LCD initialization, the CAN's is done as in figure 7.1.

```

Serial.begin(115200);
pinMode(txPin, OUTPUT);
LCD.begin(9600);
clearLCD();

```

*Figure 7. 9 LCD Initialization*

Finally, in the loop function, shown in figure 7.10, we receive the message and use the serial communications to print the temperature on the display.

```

void loop()
{
    if(CAN_MSGAVAIL == CAN.checkReceive())           // check if data coming
    {
        CAN.readMsgBuf(&len, buf);    // read data, len: data length, buf: data buf

        Serial.println("Temperature C1");
        Serial.println(buf[2]);

        Serial.println("Temperature C2");
        Serial.println(buf[5]);

        Serial.println("Temperature Mean");
        Serial.println(buf[6]);
    }

    lcdPosition(0,0);
    LCD.print("Temperature:");
    lcdPosition(0,14);
    LCD.print(buf[6]);

    delay(100);
}

```

*Figure 7. 10 - LCD recieving and displaying the temperature*

The last code is the receiver that will actuate on the servo. This program will read the temperature and if it is higher than the set threshold the servo will move.

First we had to include the Servo library in the code, to be able easily write on it. Then in the setup we wrote the following line to begin operation with the actuator:

```
myservo.attach(9);
```

Where 9 is the pin where it is connected.

In the main loop, shown in figure 7.11, we set the threshold for 40 degrees Celsius (the variable holding the average temperature is buf[6] as can be seen in figure 7.7), if the temperature reaches this value the servo will be incremented until it reaches its maximum value, then it will start the inverse movement. When the temperature falls below 40 the servo will stop.

```

void loop()
{
    if(CAN_MSGAVAIL == CAN.checkReceive())           // check if data coming
    {
        CAN.readMsgBuf(&len, buf);    // read data, len: data length, buf: data buf
        Serial.println(buf[6]);
    }
    if (buf[6] >= 40)
    {
        if (pos>0 || pos<180)
        {
            pos=pos+i;
            myservo.write(pos);
            delay(250);
        }
        if (pos == 180)
            i=(-10);
        if (pos == 0)
            i=10;
    }
    myservo.write(pos);
    delay(5);
}

```

*Figure 7. 11 - Servo movement*

## 8 CONCLUSION

This project focus was to learn and implement a CAN network, and the results were very satisfactory, as we could implement the communications between the nodes in many different formats and we could learn many function of the Arduino.

The biggest drawback we found was being able to use the CAN analyser to see the packets being transmitted, and, while a close study of the results displayed by it would enrich the our learning experience, we still could understand a lot of what was happening with the data as it navigate to the bus.

In addition to all the practice and theoretical knowledge we gained there was also the planning stages and the weekly reports on the student folio platform, a practice of the utmost importance for any work in engineering. Keeping track of what we did proved to be also useful for later reference.

In overall, this project was successful and for future works there should be done an expansion of the system build with a more complex use of masks and filters for data arbitration, as well as a bigger time investment on understanding the setting up of the CAN analyser and its usage and study.

## 9 REFERENCES

- [1] <http://www.can-cia.org/index.php?id=systemdesign-can-protocol>- CAN IN AUTOMATION
- [2] [http://www.iso.org/iso/catalogue\\_detail.htm?csnumber=33422](http://www.iso.org/iso/catalogue_detail.htm?csnumber=33422) – ISO 11898



- [3] <http://www.kvaser.com/about-can/can-standards/> - CAN ISO Standard
- [4] <http://www.kvaser.com/about-can/can-standards/> - Texas Instruments
- [5] [http://www.freescale.com/files/microcontrollers/doc/data\\_sheet/BCANPSV2.pdf](http://www.freescale.com/files/microcontrollers/doc/data_sheet/BCANPSV2.pdf) - Freescale document on CAN
- [6] <http://esd.cs.ucr.edu/webres/can20.pdf> - BOSCH CAN specification
- [7] <http://www.tested.com/tech/robots/456466-know-your-arduino-guide-most-common-boards/> - Guide for most common boards
- [8] [http://www.robotizando.com.br/curso\\_arduino\\_hardware\\_pg1.php](http://www.robotizando.com.br/curso_arduino_hardware_pg1.php) - Arduino Basics, in Portuguese.
- [9] <http://arduino.cc/en/Main/arduinoBoardUno> - Arduino UNO characteristics
- [10] [http://cdn.oreillystatic.com/oreilly/booksamplers/9780596802479\\_sampler.pdf](http://cdn.oreillystatic.com/oreilly/booksamplers/9780596802479_sampler.pdf) - Arduino Cookbook
- [11] <https://www.sparkfun.com/products/10039> - CAN-BUS Shield Sparkfun
- [12] [http://www.seeedstudio.com/wiki/CAN-BUS\\_Shield](http://www.seeedstudio.com/wiki/CAN-BUS_Shield) - CAN-BUS library reference
- [13] [https://github.com/Seeed-Studio/CAN\\_BUS\\_Shield](https://github.com/Seeed-Studio/CAN_BUS_Shield) - Repository with the source codes for the CAN Bus library
- [14] <https://folio.brighton.ac.uk/group/view.php?id=476> - Folio