**CHAPTER 3: TIRE TEMPERATURE AND GPS DATA LOGGER**

**System Goal**

**System Need**

When suspension parameters were initially discussed, instantaneous measurements of camber was discussed. There was no way to package a system of sensors to measure camber in relation to the track without having a major impact on the dynamics of the suspension due to increased un-sprung weight or hanging off the sides of the car which could potentially clip cones at an auto‐cross course. Instead, a method to validate if a current camber setting was desired. As a car turns, the outside wheel is loaded, and the tire sidewalls flex, effectively pulling the contact patch of the tire under the car, causing more work to be done on the outside edge of the tire if the car was not set up with enough negative camber. With the increase in friction, the surface temperature of that area of the tire increases proportionally. Ideally, the contact patch of the tire would be flat at the most extreme lateral acceleration to get the most grip possible out of the tire. This would cause an even temperature across the entire width of the tire.



**Fig 3.1**

**MLX90614 Infrared Temperature Sensor**

**System Design**

In order to measure the temperature of the surface of the tire, without affecting the tire, a non‐contact temperature sensor was needed. Many tire temperature sensor arrays are available commercially, which are designed for use with commercial Data Acquisition systems, costing over 200 dollars per sensor. A more cost effective solution was found from Melexis. The MLX90614 infrared temperature sensor has a built in 16‐bit ADC, in a package that is less than 10 mm in diameter. The standard model measures surface temperatures in a 90 degree cone emanating from its center. By placing the sensors approximately 1.5 to 2 inches away from the surface of the tire, three MLX90614 sensors can be aimed at the surface of the tire with no overlap in order to measure the temperature in three radial bands around the inside, middle, and outside of the tire surface.

The MLX90614 an I2C interface to communicate with the microcontroller. This method of interface requires only two wires in addition to the power and ground wires for the sensor. Additionally, the two communication wires are common to all sensors on the I2C bus, which allows the sensors to be daisy chained in line, whereas the SPI interface on the MCP320X ADC chips requires a chip select signal unique to each chip in addition to its 3 common communication lines. The MLX90614 is precise to ±0.02 Celsius, which is more than necessary for noting the difference in temperature on the tire.

**Programming protocol**

**MLX90614 Infrared Temperature Sensor**

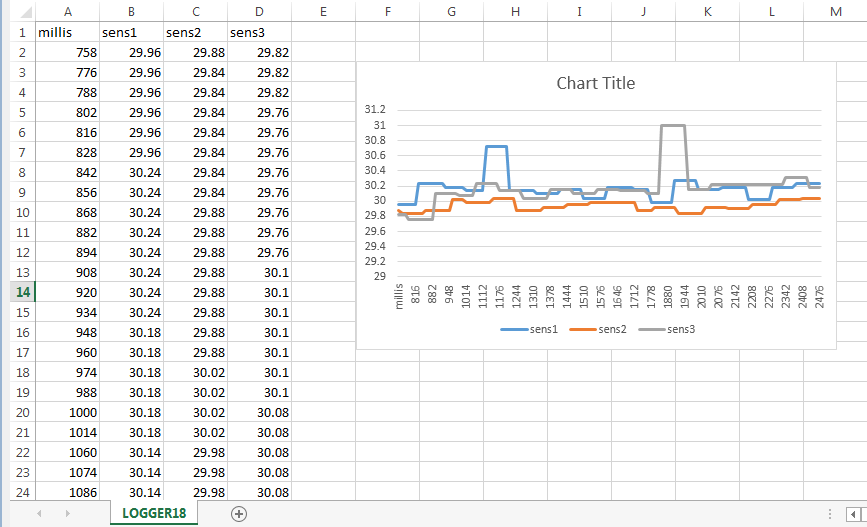
The MLX90614 infrared temperature sensors report their values in a 16‐bit unsigned value. Each tick here is the equivalent of 0.02 Kelvin. In order to scale the value from the raw data form into Celsius degrees the following operation is performed:

**Temperature Sensor scaling calculation**

[(unsigned reading) ∗ 0.02] − 272.0 = Celsius degrees

This allows the temperature gradient across the tire to be easily seen as differences as slight at 0.02 degrees Celsius are recorded.

The major trouble with working with multiples of the MLX90614 is the communication. This unit uses an I2C communication protocol. I2C is a very common and very useful protocol as it uses only 2 wires in addition to the power and ground for the chip it is connecting to, as opposed to 4 wires for SPI. However, since I2C does not have a chip select function, every I2C device has a Slave Address set in its EEPROM (Electronically Erasable Programmable Read Only Memory), and in the case of the MLX90614 and most I2C devices, it is set to a default value from the factory. In order to change the slave address of the sensor, it must be first reset to 0x000 (0 in hexadecimal, representing 12 zeros in binary) and then set to the desired new address. The MLX90614 must then be power cycled for the new address to take effect. This process may seem simple, however, in order to write to the EEPROM in the MLX90614, a Cyclic Redundancy Check (CRC) is required. The CRC is an additional byte of data that is calculated based on the bytes preceding it in the packet. This is used to check for any sort of transmission error. The sensors were programmed to change their address. The addresses were from 0x01 to 0x0c and arranged accordingly on a breadboard.



**Fig 3.1**

Code

#include <i2cmaster.h>

void setup(){

Serial.begin(115200);

Serial.println("Setup...");

i2c\_init(); //Initialise the i2c bus

PORTC = (1 << PORTC4) | (1 << PORTC5);//enable pullups

}

void loop(){

Serial.print("time:");

Serial.print(millis());

for(int i =1;i<13;i++)

{

int dev = i<<1;

int data\_low = 0;

int data\_high = 0;

int pec = 0;

i2c\_start\_wait(dev+I2C\_WRITE);

i2c\_write(0x07);

// read

i2c\_rep\_start(dev+I2C\_READ);

data\_low = i2c\_readAck(); //Read 1 byte and then send ack

data\_high = i2c\_readAck(); //Read 1 byte and then send ack

pec = i2c\_readNak();

i2c\_stop();

//This converts high and low bytes together and processes temperature, MSB is a error bit and is ignored for temps

double tempFactor = 0.02; // 0.02 degrees per LSB (measurement resolution of the MLX90614)

double tempData = 0x0000; // zero out the data

int frac; // data past the decimal point

// This masks off the error bit of the high byte, then moves it left 8 bits and adds the low byte.

tempData = (double)(((data\_high & 0x007F) << 8) + data\_low);

tempData = (tempData \* tempFactor)-0.01;

float celcius = tempData - 273.15;

// float fahrenheit = (celcius\*1.8) + 32;

//Serial.println(i);

//Serial.print("time:");

// Serial.print('\t');

//Serial.print(millis());

Serial.print('\t');

//Serial.print("Celcius: ");

Serial.print(celcius);

Serial.print('\t');

//Serial.print("Fahrenheit: ");

//Serial.println(fahrenheit);

//delay(1000); // wait a second before printing again

}

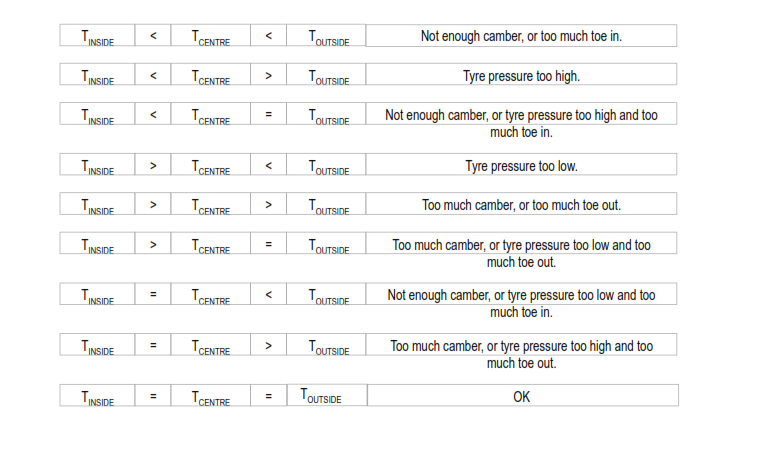
Serial.println();

}

**Hardware**

Four PCBs for four tires were manufactured and the sensors were soldered according to their addresses. Four pin relimate connectors were used to connect the sensors from four PCBs to the data logger.

Tire Data Analysis



**Fig 3.2**

**GPS**

The GPS data logger wirelessly transmits the NMEA data to the computer. This data can be used to view live position of the car and get a trace of the entire lap. The GPS module works on UART. It sends out data serially which is read by the same data logger.

The data logger has an FTDI chip onboard which converts UART to USB so that the data can sent to the computer via the telemetry system which has USB ports on it.



**Fig 3.3**

**Code**

#include <SoftwareSerial.h>

#include <TinyGPS.h>

/\* This sample code demonstrates the normal use of a TinyGPS object.

It requires the use of SoftwareSerial, and assumes that you have a

4800-baud serial GPS device hooked up on pins 3(rx) and 4(tx).

\*/

TinyGPS gps;

SoftwareSerial ss(3, 4);

void setup()

{

Serial.begin(115200);

ss.begin(9600);

//Serial.print("Simple TinyGPS library v. "); Serial.println(TinyGPS::library\_version());

//Serial.println("by Mikal Hart");

// Serial.println();

}

void loop()

{

bool newData = false;

//unsigned long chars;

//unsigned short sentences, failed;

// For one second we parse GPS data and report some key values

for (unsigned long start = millis(); millis() - start < 1000;)

{

while (ss.available())

{

char c = ss.read();

Serial.write(c); // uncomment this line if you want to see the GPS data flowing

//if (gps.encode(c)) // Did a new valid sentence come in?

//newData = true;

}

}

/\*

if (newData)

{

float flat, flon;

unsigned long age;

gps.f\_get\_position(&flat, &flon, &age);

Serial.print("LAT=");

Serial.print(flat == TinyGPS::GPS\_INVALID\_F\_ANGLE ? 0.0 : flat, 6);

Serial.print(" LON=");

Serial.print(flon == TinyGPS::GPS\_INVALID\_F\_ANGLE ? 0.0 : flon, 6);

Serial.print(" SAT=");

Serial.print(gps.satellites() == TinyGPS::GPS\_INVALID\_SATELLITES ? 0 : gps.satellites());

Serial.print(" PREC=");

Serial.print(gps.hdop() == TinyGPS::GPS\_INVALID\_HDOP ? 0 : gps.hdop());

}

gps.stats(&chars, &sentences, &failed);

Serial.print(" CHARS=");

Serial.print(chars);

Serial.print(" SENTENCES=");

Serial.print(sentences);

Serial.print(" CSUM ERR=");

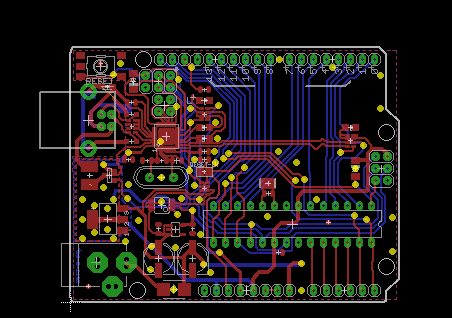
Serial.println(failed);

\*/}

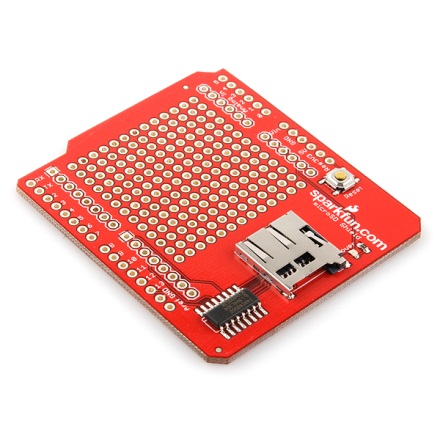
**DATA LOGGER**

The data logger uses AVR Atmega328 the schematic for the PCB was referred from arduino.cc. The arduino IDE was used for coding purpose. To log the data on a micro SD card a PCB was required with an SD card socket on it. Since the manufacturing cost of a new PCB for SD card was same as that available from sparkfun we decided to buy the PCB from sparkfun.com. This PCB also had enough space to accommodate the GPS module on it.

The tire temperature harness had relimte connectors at both the ends connecting the tire temperature sensor PCBs and the data logger.



**Fig 3.4**



**Fig 3.5**