# Vehicle Condition, Driver Behavior Analysis and Data Logging Through CAN Sniffing

Adnan Shaout, Dhanush Mysuru, and Karthik Raghupathy
The Electrical and Computer Engineering Department, the University of Michigan, USA

**Abstract:** *modern vehicles nowadays have many Electronic Control Units (ECU) and sensors. Information (data) within an automobile is transferred via the Control Area Network (CAN) of a vehicle. This data may not be important to the driver, but if the CAN messages are analyzed, then driver behavior and vehicle conditions can be determined. This data can determine ahead of time any possible future failure and the driver can be alerted about it. This paper presents standalone real time embedded system that could analyze the CAN data were valuable lives may be saved in real time. The proposed hardware system acquires CAN data from a vehicle (CAN sniffing). The data is then processed and an appropriate message is then displayed for the driver. The proposed system also stores the CAN messages on to an industrial grade SD card (CAN logging) for future analysis. The proposed system can also perform driver behavior and driving terrain analysis.*

**Keywords:** *vehicle condition analysis, CAN sniffing, CAN logging, CANoe 9.0, driver behavior.*

*Received September 9, 2018; accepted January 22, 2019*

## 1. Introduction

Control Area Network (CAN) [4] is used to exchange data between different Electronic Control Units (ECU) in a vehicle. It was introduced by Robert Bosch in the year 1983.

There are several Electronic Control Units (ECU's) in a vehicle, namely engine control unit, electric power steering unit, power train control module and brake control module. These ECU's are connected to the CAN bus. CAN bus basically has two lines, CAN High (CANH) and CAN Low (CANL).CAN bus can provide data speeds up to 1Mbps. CANH and CANL are terminated by a 120 ohm resistor to avoid signal reflection. The basic construction of the CAN bus is shown in Figure 1.

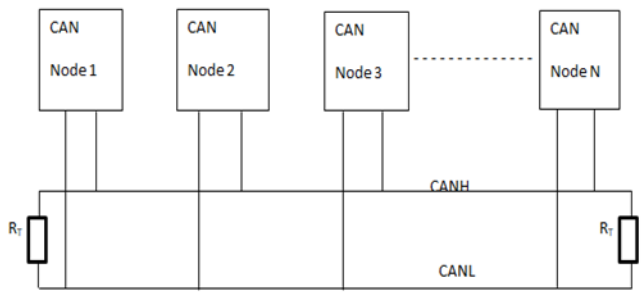
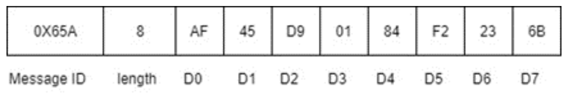


Figure 1. Simple CAN network.

The ECUs communicates by sending packets of data called CAN messages. These CAN messages have specific structure. The first part of the CAN message has the message address which corresponds to the ID of the sender which sends out the message. The receiver looks for message with this specific ID on the CAN bus and picks only the right messages ignoring other messages with different ID coming out of different ECUs. The second part of the message contains the length of the CAN message. This could be either 8 or 22 bytes long which indicates the actual data length. The next field is the actual data. A CAN message also has other data such as Cyclic Redundancy Check (CRC), acknowledgement and several other bits which are not used in our system. Figure 2 shows a typical CAN message.



| Message ID | length | D0 | D1 | D2 | D3 | D4 | D5 | D6 | D7 |
|---|---|---|---|---|---|---|---|---|---|
| 0X65A | 8 | AF | 45 | D9 | 01 | 84 | F2 | 23 | 6B |

Figure 2. Typical CAN message.

CANoe [8] is a software tool used for development, testing, simulating and analyzing of the ECU networks. It was developed by Vector Informatik GmbH. CANoe supports CAN, LIN, flexRay and other vehicle bus architectures. Vector 1610 hardware provides bus interface to CANoe software. Vector 1610 is an interface tool for fast access of CAN data and has two transceivers.

ArduinoUNO is a general purpose prototyping board that houses an ATMega328p microcontroller which is a 28 pin IC with 14 Digital I/O pins and 6 Analog pins. CAN shield is an addition hardware that mounts on the Arduino board which provides CAN interface capabilities to the Arduino. The CAN shield has an MPC2515 CAN controller. Teensy 3.6 is also a prototyping board which has an ARM Cortex-M4 processor that is ARM's high performance embedded processor [7, 9]. Teensy by itself has an on board SD card slot, which is the main reason of choosing Teensy over Arduino in the receiver part of the proposed system.

The paper is organized as follows: section 2 presents literature survey, where a comparison of the existing state of art with the proposed system implementation; Section 3 presents the system description and setup, where a discussion of the hardware and software setup of the proposed design; Section 4 presents the implementation and results; and section 5 presents testing and conclusion, where the tests performed on the system will be presented.

## 2. Literature Review

Several attempts were done in the past to acquire CAN data from a vehicle using different techniques. Johanson and Karlsson [3] discussed how data was sniffed by a wireless Diagnostic Read-Out (DRO) which used two main components: VIDA and dynamically linked library. A trigger button which was connected to a mobile unit present in a car was used to initiate the wireless DRO manually. An internet connection is established when the button is pushed via a General Packet Radio Services (GPRS) modem.

A Transmission Control Protocol (TCP) connection is set-up to a dispatcher and a public IP address is generated that is reachable from all the mobile units in a vehicle to run on a server. In the proposed system in this paper, the system would be directly tapping into the CAN bus of the vehicle to access to the CAN messages. A similar CAN simulation model proposed by Zhou *et al*. [10] was adopted in this paper. Zhou *et al*. [10] focused on simulating a CAN bus using the CANoe by taking the network topology, hierarchy and the baud rate into consideration.

Vehicle data can be logged using data-loggers like GL1000/2000/3000/4000 and CAN-log 3/4 and can be captured live using an On-Board Diagnostics (OBD) connectors when the vehicle is in motion. Using such data loggers can be expensive. Shah *et al*., [6] have proposed inserting the SD card directly into the ECU of the vehicle. In our proposed design we have inserted the SD card in the Teensy (CPU2) to store the incoming CAN data.

There are lots of security issues with the CAN bus [1, 2]. The data can be easily hacked using an OBD II tool. A Cyber-physical system can give access to the CAN bus network [1]. There have been lots of attempts to secure the CAN data in automobiles. In [5], authors have proposed a Cyber-Security mechanism by designing an architecture which helps in using the bus as low as possible. This mechanism keeps the bus utilization as low as possible and can achieve high security levels.

## 3. System Description and Setup

The basic system setup is as shown in Figure 3. It consists of CANoe and the vector 1610 for the CAN bus simulation. The CAN high and the CAN low from vector 1610 are connected to the CAN high and CAN low of the CAN shield, respectively.

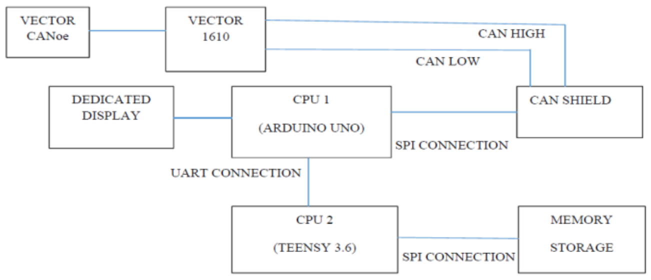


Figure 3. The proposed system architecture.

The CAN shield is connected to the Arduino UNO using SPI connection. Arduino is our first CPU (CPU1). There is a dedicated display interfaced with the Arduino for displaying alerts to the driver. Further, the Arduino is connected to the Teensy 3.6 using UART connections. Teensy is our second CPU (CPU2). Teensy has an onboard micro SD card slot.

A memory card is inserted into this slot in which the CAN data is stored. The actual system set up is shown in Figure 4.

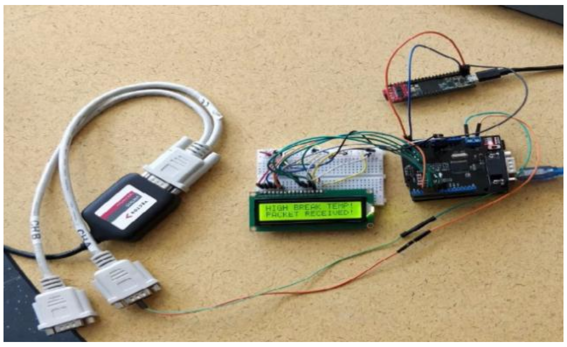
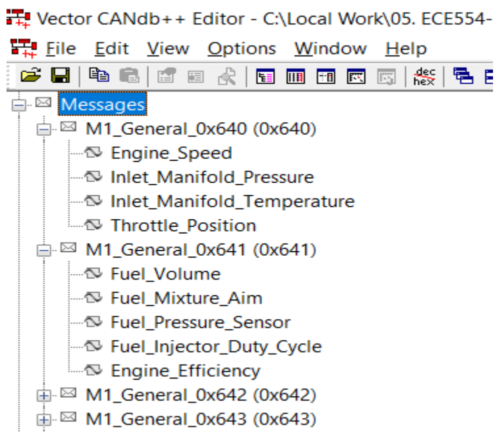


Figure 4. The actual proposed system prototype.

## 4. Implementation and Results

### 4.1. Can Simulation

Instead of sniffing the data from an actual vehicle, the CAN bus will be simulated using CANoe and vector 1610. In CANoe a database file provided by MOTEC USA was loaded which has the description of each message. These messages can be viewed and edited using theCANdb++ editor. Figure 5 shows the message structure used where the message ID and the parameters associated with it are shown. For example, by reading the message with the ID 0X640 the following parameters associated with it would be read: Engine speed, Inlet manifold pressure, Inlet manifold temperature and throttle position.



Vector CANdb++ Editor - C:\Local Work\05. ECE554-

File Edit View Options Window Help

- Messages
  - M1_General_0x640 (0x640)
    - Engine_Speed
    - Inlet_Manifold_Pressure
    - Inlet_Manifold_Temperature
    - Throttle_Position
  - M1_General_0x641 (0x641)
    - Fuel_Volume
    - Fuel_Mixture_Aim
    - Fuel_Pressure_Sensor
    - Fuel_Injector_Duty_Cycle
    - Engine_Efficiency
  - M1_General_0x642 (0x642)
  - M1_General_0x643 (0x643)

Figure 5. Message Structure.

A simple network was created that could generate CAN messages. A baud rate of 500 KBPS was selected as shown in Figure 6. The network has a basic interactive generator block.

CANoe has the option to set the rate at which CAN messages are sent out on the CAN bus.

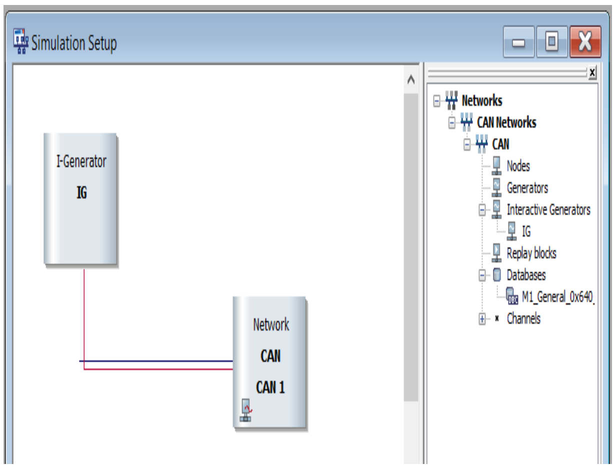
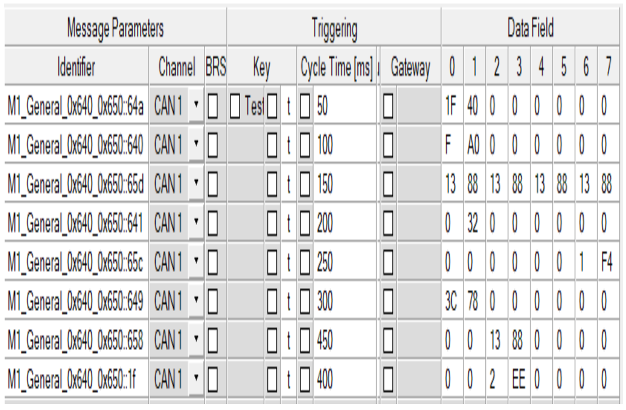


Figure 6. Simulated Network.

Round robin scheduling technique was used with a time slice of 50ms to manage the CAN messages. So messages are sent periodically with a time difference of 50ms between them. The CAN messages that were chosen in the proposed system are the ones which are required to analyses the condition of the vehicle, behavior of the driver and vehicle terrain. Only 10 messages were selected to show that the system works (proof of concept) and other messages can be added as needed. The selected messages that this proposed system and their respective ID are as follows:

- 0x64A-Exhaust temp (50ms).
- 0X640-engine speed (100ms).
- 0x65D- Brake temperature (150ms).
- 0x641 -Fuel level (200ms).
- 0x65C-power steering temp (250ms).
- 0x649 -Engine coolant temp (240).
- 0x658 - vehicle longitudinal accel (250ms).
- 0x1F- vehicle speed (300ms).
- 0x10- Vehicle Angle (350ms).

Figure 7 shows the loaded messages used in the proposed system with their schedule.



| Message Parameters | | | Triggering | | | Data Field | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Identifier | Channel | BRS | Key | Cycle Time [ms] | Gateway | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| M1_General_0x640_0x650::64a | CAN 1 | ☐ | ☐ Test ☐ t | ☐ 50 | ☐ | 1F | 40 | 0 | 0 | 0 | 0 | 0 | 0 |
| M1_General_0x640_0x650::640 | CAN 1 | ☐ | ☐ t | ☐ 100 | ☐ | F | A0 | 0 | 0 | 0 | 0 | 0 | 0 |
| M1_General_0x640_0x650::65d | CAN 1 | ☐ | ☐ t | ☐ 150 | ☐ | 13 | 88 | 13 | 88 | 13 | 88 | 13 | 88 |
| M1_General_0x640_0x650::641 | CAN 1 | ☐ | ☐ t | ☐ 200 | ☐ | 0 | 32 | 0 | 0 | 0 | 0 | 0 | 0 |
| M1_General_0x640_0x650::65c | CAN 1 | ☐ | ☐ t | ☐ 250 | ☐ | 0 | 0 | 0 | 0 | 0 | 0 | 1 | F4 |
| M1_General_0x640_0x650::649 | CAN 1 | ☐ | ☐ t | ☐ 300 | ☐ | 3C | 78 | 0 | 0 | 0 | 0 | 0 | 0 |
| M1_General_0x640_0x650::658 | CAN 1 | ☐ | ☐ t | ☐ 450 | ☐ | 0 | 0 | 13 | 88 | 0 | 0 | 0 | 0 |
| M1_General_0x640_0x650::1f | CAN 1 | ☐ | ☐ t | ☐ 400 | ☐ | 0 | 0 | 2 | EE | 0 | 0 | 0 | 0 |

Figure 7.CAN Message Scheduling.

## 4.2. Can Sniffing and Analysis

An incoming CAN message is picked up by the Arduino (CPU1) with the help of the CAN shield. The entire processing of Vehicle condition analysis, Driver behavior detection, CAN logging and the vehicle terrain detection is split between the two CPUs to achieve actual parallel processing. Firstly, on receiving a CAN message, CPU1 (Arduino) sends a copy of the CAN data to CPU2 via UART where the driver behavior, CAN logging and the terrain detection is performed. The Arduino (CPU1) performs the vehicle condition analysis.

In the Arduino pooled loop with interrupt type of Real Time Operating System (RTOS) was used, where the system continuously checks for the CAN messages on the CAN line. Whenever it encounters a CAN message an interrupt is raised and all the tasks are sequentially executed. The interrupt is then cleared and the system goes back to looking for a new CAN message. The entire process that the CPU1 is supposed to perform is broken into several tasks. For example, once a CAN message is received, CPU1 is supposed to send a copy of data to CPU2. This is made as a task. Another task is to perform message filtering by checking the address of the message. If the message is one of the required messages then it will be processed. If it is not required, then it will be filleted out. Checking the output of the calculation and displaying a proper message on the LCD display is also made as a task.

Figure 8 presents the flowchart that shows the control flow for CPU1.

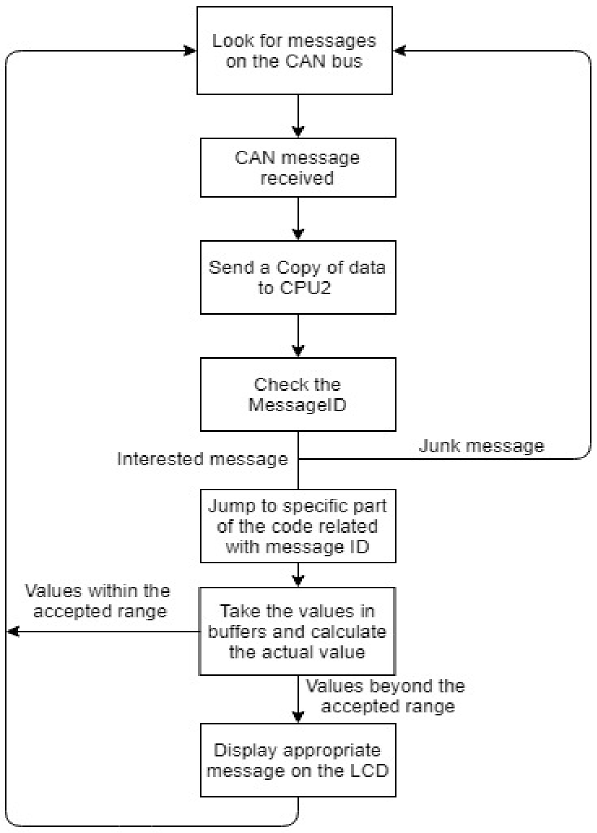

Look for messages on the CAN bus
CAN message received
Send a Copy of data to CPU2
Check the MessageID
Interested message
Junk message
Jump to specific part of the code related with message ID
Values within the accepted range
Take the values in buffers and calculate the actual value
Values beyond the accepted range
Display appropriate message on the LCD

Figure 8. The Logical Flow followed by CPU1.

When the data is received, a copy of the data is sent to CPU2 for logging and other analysis. Later, the message address is checked to determine if the packet is of interest or not. Then the actual data in HEX is converted to decimal. If the incoming data is a big number such as 0xAFD2, then it will be stored in 2 buffers (8 bits for each buffer). The actual values are calculated from these 2 buffers. For every checked parameter, a manually acceptable limit is set. If the values are above the acceptable limit then a suitable caution message is displayed on the 16*2 LCD display device.

In reading a sensor value, due to some external condition, the sensor might give erroneous values. The proposed system would not react to these erroneous values and display a wrong message to the driver. A state machine was implemented so that an erroneous value is detected if and only if 5 off-limit values are encountered consecutively for the system to recognize that there is an issue. Figure 9 shows the implemented state machine where 1 denotes faulty values (off-limit) and 0 denotes in-range values.

For Example, if we are checking the brake temperature, where the manually set accepted range is 0C to 600C, if the sensor send out one off-limit value then the system wouldn't alert the driver. But if 5 continuous off-limit values are obtained consecutively then only the system would alert the driver by displaying “HIGH BREAK TEMP!” error.

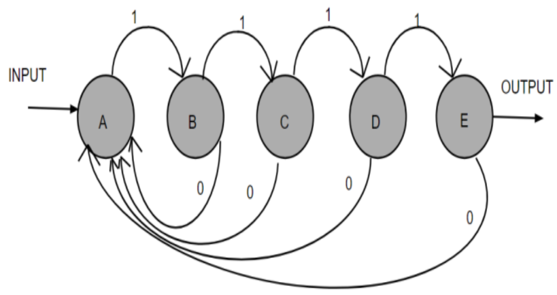
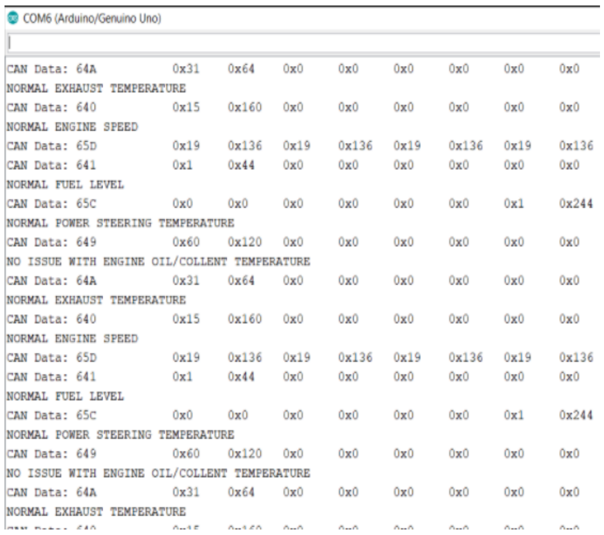


Figure 9. Mealy State Machine.

The output at the Arduino’s serial monitor is as shown in the Figure 10. It can be seen that when the message is received, then an appropriate message is printed.



```
CAN Data: 64A        0x31    0x64    0x0     0x0     0x0     0x0     0x0     0x0
NORMAL EXHAUST TEMPERATURE
CAN Data: 640        0x15    0x160   0x0     0x0     0x0     0x0     0x0     0x0
NORMAL ENGINE SPEED
CAN Data: 65D        0x19    0x136   0x19    0x136   0x19    0x136   0x19    0x136
CAN Data: 641        0x1     0x44    0x0     0x0     0x0     0x0     0x0     0x0
NORMAL FUEL LEVEL
CAN Data: 65C        0x0     0x0     0x0     0x0     0x0     0x0     0x1     0x244
NORMAL POWER STEERING TEMPERATURE
CAN Data: 649        0x60    0x120   0x0     0x0     0x0     0x0     0x0     0x0
NO ISSUE WITH ENGINE OIL/COLLENT TEMPERATURE
CAN Data: 64A        0x31    0x64    0x0     0x0     0x0     0x0     0x0     0x0
NORMAL EXHAUST TEMPERATURE
CAN Data: 640        0x15    0x160   0x0     0x0     0x0     0x0     0x0     0x0
NORMAL ENGINE SPEED
CAN Data: 65D        0x19    0x136   0x19    0x136   0x19    0x136   0x19    0x136
CAN Data: 641        0x1     0x44    0x0     0x0     0x0     0x0     0x0     0x0
NORMAL FUEL LEVEL
CAN Data: 65C        0x0     0x0     0x0     0x0     0x0     0x0     0x1     0x244
NORMAL POWER STEERING TEMPERATURE
CAN Data: 649        0x60    0x120   0x0     0x0     0x0     0x0     0x0     0x0
NO ISSUE WITH ENGINE OIL/COLLENT TEMPERATURE
CAN Data: 64A        0x31    0x64    0x0     0x0     0x0     0x0     0x0     0x0
NORMAL EXHAUST TEMPERATURE
```

Figure 10. Output at the Arduino Serial Monitor.

Figure 11 shows the actual output message shown to the driver when 5 off-limit sensor values are read from the brake temperature sensor.

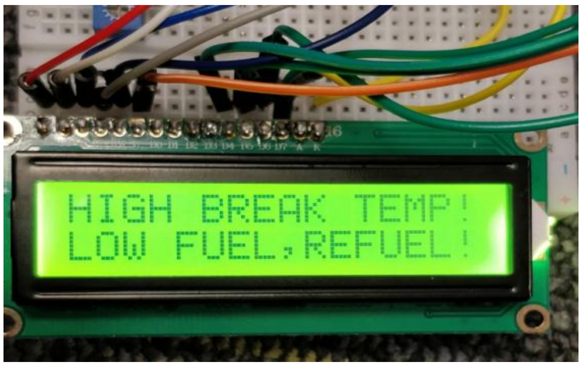


Figure 11. Actual message shown to the driver.

## 4.3. Can Logging Driver Behavior Analysis and Terrain Detection

A copy of all received CAN messages are sent to the Teensy (CPU2). The driver behavior analysis, CAN logging and the terrain detection are all processed by the Teensy processor.

Similar to the Arduino, a pooling loop with interrupt type RTOS was implemented for the Teensy which checks for the data through the UART. When a CAN message is coming, then an interrupt is raised and the message is read. All of the incoming CAN messages are directly written to the SD card in a form of a text file.

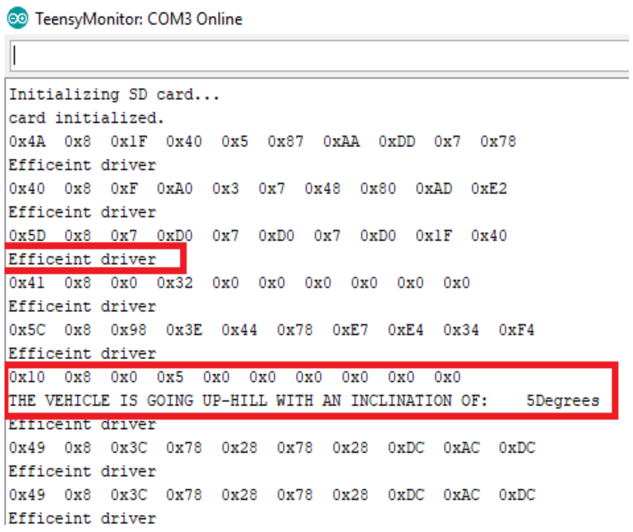
When a turn is made in a vehicle then there would be an opposite force in the other direction of the turn. This force is called Longitudinal Acceleration force (LA). This force is sensed by longitudinal acceleration sensor. By looking into the value read from this sensor, a turn by the vehicle can be detected. If the value of this sensor is high, then it means that the vehicle is making a steep turn. By combining this sensor value with the vehicle speed the behavior of the driver can be determined.

As a proof of concept 3 limits were manually set to differentiate the driver as efficient, moderately efficient and reckless driver. To avoid the erroneous values and to reduce the miss-prediction at a point of time, the last 10 sensor values are considered. The three limits that were selected in this paper are as follows:

- If speed is below 75 MPH and the LA is below 0.2G, then efficient driver.
- If speed is between 75 and 120 MPH and LA is between 0.2 and 0.6G, then moderately efficient driver.
- If speed is above 120 MPH and LA is above 0.6G, then inefficient driver.

Teensy in this paper is programmed to lookout for messages sent from the angle sensor (LA) of the vehicle. By reading this sensor the vehicle going uphill or downhill can be determined.

Figure 12 shows a sample run messages received by Teensy and the analysis (prediction) done by the Teensy processor in the serial monitor.

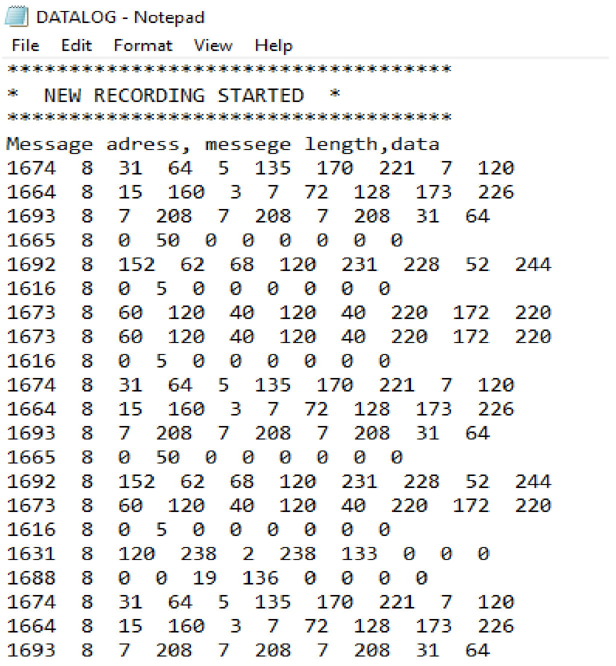


```
Initializing SD card...
card initialized.
0x4A  0x8  0x1F  0x40  0x5  0x87  0xAA  0xDD  0x7  0x78
Efficeint driver
0x40  0x8  0xF  0xA0  0x3  0x7  0x48  0x80  0xAD  0xE2
Efficeint driver
0x5D  0x8  0x7  0xD0  0x7  0xD0  0x7  0xD0  0x1F  0x40
Efficeint driver
0x41  0x8  0x0  0x32  0x0  0x0  0x0  0x0  0x0  0x0
Efficeint driver
0x5C  0x8  0x98  0x3E  0x44  0x78  0xE7  0xE4  0x34  0xF4
Efficeint driver
0x10  0x8  0x0  0x5  0x0  0x0  0x0  0x0  0x0  0x0
THE VEHICLE IS GOING UP-HILL WITH AN INCLINATION OF:    5Degrees
Efficeint driver
0x49  0x8  0x3C  0x78  0x28  0x78  0x28  0xDC  0xAC  0xDC
Efficeint driver
0x49  0x8  0x3C  0x78  0x28  0x78  0x28  0xDC  0xAC  0xDC
Efficeint driver
```

Figure 12. Output at the Teensy with the prediction.

The CAN messages are stored as a text file in the SD card.

Figure 13 show a sample of the logged CAN data in decimal.



```
************************************
*  NEW RECORDING STARTED  *
************************************
Message adress, messege length,data
1674  8  31  64  5  135  170  221  7  120
1664  8  15  160  3  7  72  128  173  226
1693  8  7  208  7  208  7  208  31  64
1665  8  0  50  0  0  0  0  0  0
1692  8  152  62  68  120  231  228  52  244
1616  8  0  5  0  0  0  0  0  0
1673  8  60  120  40  120  40  220  172  220
1673  8  60  120  40  120  40  220  172  220
1616  8  0  5  0  0  0  0  0  0
1674  8  31  64  5  135  170  221  7  120
1664  8  15  160  3  7  72  128  173  226
1693  8  7  208  7  208  7  208  31  64
1665  8  0  50  0  0  0  0  0  0
1692  8  152  62  68  120  231  228  52  244
1673  8  60  120  40  120  40  220  172  220
1616  8  0  5  0  0  0  0  0  0
1631  8  120  238  2  238  133  0  0  0
1688  8  0  0  19  136  0  0  0  0
1674  8  31  64  5  135  170  221  7  120
1664  8  15  160  3  7  72  128  173  226
1693  8  7  208  7  208  7  208  31  64
```

Figure 13. The logged CAN data in Decimal.

## 5. Testing

A single CPU was initially used in designing the proposed system. The system functioned properly when the time between the messages was equal to or greater than 50ms. But when the time between the messages was reduced by decreasing the time slice in the scheduler in the CANoe to match the real life conditions, the initial system started to drop messages. That is, the system would still be processing a previous message when a new message is in. This was happening because writing the message on to the SD card was taking too much time.

A decision was to split the number of tasks between two processors to reduce the workload and to meet the timing requirements. The enhanced proposed system with two CPUs has been tested for extreme cases by reducing the inter message time to as small as 5ms and the system still did not drop any messages. The proposed system in this paper has been tested for different CAN speeds such as 125Kbps, 250Kbps and 500Kbps and it did work fine without any issues.

## 6. Conclusions

The proposed real time system mainly focuses on vehicle condition analysis, driver assistance and behavior analysis aiming to make the vehicle efficient and improve road safety. The paper introduced a method of simulating and analyzing vehicle CAN data. The system used CANoe 9.0 to simulate the ECU data and CAN 1610 hardware tool to send the data on to the CAN bus.

The data was read by the first CPU (Arduino) with the help of the CAN shield for processing it and then it was sent to the second CPU (Teensy 3.6) over UART to store the data. The stored CAN data could then be used for fault analysis. Achieving the same results with a single CPU by adopting interrupt based techniques would be our future scope of work.

## References

[1] Abbott-McCune S. and Shay L., "Techniques in Hacking And Simulating A Modern Automotive Controller Area Network," *in Proceedings of IEEE International Carnahan Conference on Security Technology*, Orlando, pp. 1-7, 2016.

[2] Buttigieg R., Farrugia M., and Meli C., "Security Issues in Controller Area Networks in Automobiles," *in Proceedings of 18th International Conference on Sciences and Techniques of Automatic Control and Computer Engineering*, Monastir, pp. 93-98, 2017.

[3] Johanson M. and Karlsson L., "Improving Vehicle Diagnostics through Wireless Data Collection and Statistical Analysis," *in Proceedings of IEEE 66th Vehicular Technology Conference*, Baltimore, pp. 2184-2188, 2007.

[4] Know-How & Solutions for CAN/CAN FD https://vector.com/vi_can_solutions_en.html, Last Visited 2018.

[5] Lin C. and Sangiovanni-Vincentelli A., "Cyber-Security for the Controller Area Network (CAN) Communication Protocol," *in Proceedings of International Conference on Cyber Security*, Washington, pp. 1-7, 2012.

[6] Shah N., Cho B., Geth F., Clement K., Tant P., and Driesen J., "Electric Vehicle Impact Assessment Study Based on Data-logged Vehicle and Driver Behavior," *in Proceedings of Vehicle Power and Propulsion Conference*, Chicago, pp. 1-6, 2011.

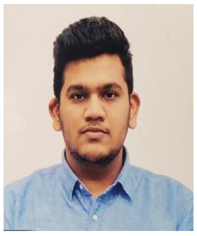
[7] Teensy USB Development Board https://www.pjrc.com/store/teensy.html, Last Visited 2018.

[8] Testing ECUs and Networks with CANoe https://vector.com/vi_canoe_en.html, Last Visited 2018.

[9] VN1600 - Network Interfaces with USB for CAN FD, LIN, K-Line, J1708 and IO https://vector.com/vi_vn1600_en.html, Last Visited 2018.

[10] Zhou F., Li S., and Hou X., "Development Method of Simulation and Test System for Vehicle Body CAN Bus Based on CANoe," *in Proceedings of 7th World Congress on Intelligent Control and Automation*, Chongqing, pp. 7515-7519, 2008.

**Adnan Shaout** is a full professor and a Fulbright Scholar in the Computer Science Department at the Electrical and Computer Engineering Department at the University of Michigan-Dearborn. His current research is in applications of software engineering methods, cloud computing, embedded systems, fuzzy systems, real time systems and artificial intelligence. Dr. Shaout has more than 36 years of experience in teaching and conducting research in the Computer Science, Electrical and Computer Engineering fields at Syracuse University and the University of Michigan-Dearborn. Dr. Shaout has published over 250 papers in topics related to Computer Science, Electrical and Computer Engineering fields. Dr. Shaout has obtained his B.S.c, M.S. and Ph.D. in Computer Engineering from Syracuse University, Syracuse, NY, in 1982, 1983, 1987, respectively.



**Dhanush Mysuru** is a graduate student in the Electrical and Computer Engineering department at the University of Michigan-Dearborn. He is a Software Developer at Navitas Systems, LLC Electrical.

**Karthik Raghupathy** is a graduate student in the Electrical and Computer Engineering department at the University of Michigan-Dearborn. Karthik was a Software Engineer Intern at Continental Automotive Systems.