

Vehicle Data Acquisition

Senior Design - ECE397
University of Illinois at Chicago
April 20, 2015

Cristian Valadez – cvalad5@uic.edu
Justin Czok – jczok2@uic.edu
Stephen Jeung – sjeung2@uic.edu




Academic Advisor: Dr. Vahe Caliskan – vahe@uic.edu

I. Project Goals

The objective of this product is to provide a vehicle data acquisition system with semi-professional quality designed for the hobbyist budget. The system will be fully available, adaptable and customizable allowing full control over its features to the user.

II. Product Need

Since this product is targeted for teams and hobbyists with limited budgets, price is of major concern with available vehicle data acquisition systems. Below is a chart detailing three of the most well-known and available solutions:

Manufacturer		Unit	Price
Motec		Base Unit (17 channel)	\$7500+
		Channel Expansion	\$2000+
Aim		Base Unit (5 channel)	\$1400
		Channel Expansion	\$400
AEM		Base Unit (8 channel)	\$500
		Channel Expansion	Not Available

Our estimated Base Unit Build Cost will be preferably under \$100 with the option of adding channel expansion units or external units like a driver dashboard display.

The data acquisition systems listed above all offer similar basic features like analog inputs, digital inputs, CAN bus, GPS input and serial communication. Our proposed design will include the basic analog and digital inputs as well as support different types of communication protocols. Please refer to Appendix C.

III. Technical Specifications

Analog I/Os:

The majority of available sensors can be connected to the acquisition system using the MCU's analog inputs. The MCU will convert the analog voltages to a digital code and store that raw data into memory. The user will be able to specify what each analog input is and assign a formula or curve that will be used to convert raw data to data with meaningful units (F, PSI, REV/MIN, etc). The conversion will not be performed each time the data is read by the MCU, but rather the formula will be stored along with the corresponding data and be executed in external analysis software. This will greatly increase MCU overhead time and performance.

Digital I/Os:

There are two digital inputs: Log Enable and Time Marker. The Log Enable signal is connected to a single pole switch which will trigger the data logger to start logging. Only until the switch is turned off with the logging be stopped. The Time Marker signal should be connected to a pushbutton. This signal will be polled by the software and saved along with the time data. A time marker is useful for pinpointing certain events.

Serial Communication:

SPI: The SPI is used by the MCU to communicate with the SD card adaptor which saves the data. This can be potentially tapped into if the user needs to connect more devices that use SPI. Software would have to be added to allow this feature.

I2C: The protocol is used by some hardware sensors like accelerometers and gyroscopes. This protocol shares its clock and data signal (2 wires only) with the hardware and selects the slave by sending an address through the data signal. The hardware that has this address will then become the slave. Due to lack of time, the I2C protocol was not fully tested or implemented in software, but the 2 hardware pins required to utilize it are available (see Appendix B for pinout).

CAN: The CAN protocol is used by many ECUs found in commercial vehicles as well as in aftermarket products. Our acquisition system will be able to receive data from such devices using the CAN bus. Since different manufacturers use different types of CAN protocols (OBD II for example), software will have to handle these variations and retrieve the correct information.

GPS: Due to lack of time, GPS capabilities were not added to the system.

RTC: The Teensy 3.2 microcontroller has a RTC included. This chip is capable of keeping time accurately (it should compensate for temperature changes) for several years when connected to an external coin battery. This purpose of this function is to have time markers along with the raw data, in other words, a time axis. A separate code program was written to setup and or modify the RTC's time.

Software: Please refer to section X. for more detailed information.

Memory Storage: An SD card was used to store memory. The Teensy microcontroller has an abundant amount of support on the internet and we used an open source library written for the SD card. This library has a “sync” function that allows the system to save data periodically (every 100ms) so that in the case of power loss there would be no subsequent data loss.

IV. Engineering Design Alternatives

All of our designs revolve around similar macro plans. As we are designing a system for collecting sensor readings, and not the sensors themselves, we know that our system must accommodate a predetermined type of signal. All of our designs alternatives tackle the handling of this in differing ways. The primary variations are microcontrollers and software analysis solutions.

Hardware Alternatives:

Option 1:

Our first solution involves using the Arduino platform. The simplicity of programming and the variety of available libraries could serve to condense development time of the data collection portions of the package. With a 16 MHz oscillator, and accommodations for around 12 total sensor inputs, we see a per channel sampling of up to 30 Hz. CAN bus is supported on some Arduino platforms, but in a limited form. This limitation may be too much of a drawback.

Option 2:

The second solution uses the slightly newer Teensy microcontroller. The clock speeds on these controllers is advantageous, allowing us a sampling rate closer to 50 Hz per channel, which is our target minimum. CAN bus support is expanded compared to the Arduino platform, but pin count is still limited. Both Arduino and Teensy platforms have compatible libraries, keeping development effort comparable.

Option 3:

Our final option is to use an Atmel automotive specific controller. This comes with a number of hardware perks, including a much better CAN bus compatibility, related development language to Arduino and Teensy, up to 200MHz clock speeds, and the ability to accommodate up to 40 sensor inputs while maintaining a 40+Hz sampling rate per channel.

Software Alternatives

We have also opted to break down our software designs into 3 distinct options. All of our sensor data is collected as discrete values and have no significant meaning without some method of interpretation. We have considered 3 methods for analyzing the raw data which include an Excel spreadsheet, Matlab scripts, and a free-standing GUI software.

Excel:

The benefits to an excel sheet include the ease of development and simplicity of user tweaking. This would be of excellent value to other university or student teams (a market segment we

would be targeting). Excel is also compatible with things like google sheets, which would allow a very inexpensive implementation and the ability to continually refine the interface and capability without significant investment of time or a CS degree.

Matlab:

We are viewing custom Matlab scripts as our current preference for interpreting data. Much like excel, there is an inherent simplicity in defining a script along with the ability to make a very polished interface. The floating point precision capability of Matlab is also a boon when sensor values can often have very tiny variations to account for. Like spreadsheets, Matlab scripts are easier to update, but do require more knowledge to tweak. The availability of Matlab at a reasonable cost does hinder the accessibility, and that will need to be factored into a finished product.

Custom GUI Software:

Custom software is the industry standard for data acquisition systems. This method would allow for more of a proprietary feel and reduce or eliminate the possibility of the end user to make minor mistakes when changing algorithms. The major downside, and the ultimate reason for dismissing custom software is the time of development and the lack of experience creating full software packages.

V. Design Alternative Evaluation Criteria

All of our criteria will be graded on a scale 1-5 where 5 represents the best possible solution.

Hardware Design Criteria:

- Number of channels permitted
- Sampling Frequency per Channel
- Cost
- Development/Coding time
- CAN Bus support

	Arduino	Teensy	Atmel Automotive
# of Channels	2	2	4
Sampling / Channel	3	4	5
Cost	4	4	3
Dev Time	5	5	3
CAN Support	2	3	5
Total Score	16	18	20

VI. Selection of Design Alternative

Although Atmel comes with a few developmental difficulties, the overall capabilities of the controller far outweigh the drawbacks. In order to achieve the desired sensor input capability, we would have had to use multiple Arduino or Teensy chips and communicate between them over I2C. While this would have allowed for an increased sample rate per channel, it would have led to other complications when it came to coding and managing processing overhead. In addition, using multiple chips to increase input count also increases cost of production. Two Teensy boards end up costing very close to a single Atmel automotive chip.

Environmental factors also came into play. As this system is being designed for automotive use, the operating conditions for an automotive spec controller has a wider range of limits. This limit includes temperature, but also in shock ratings and over/under voltage conditions.

Finally, a fully featured CAN bus support was determined to be critical to interfacing with engine control modules that can provide additional data parameters beyond what sensors a user has added.

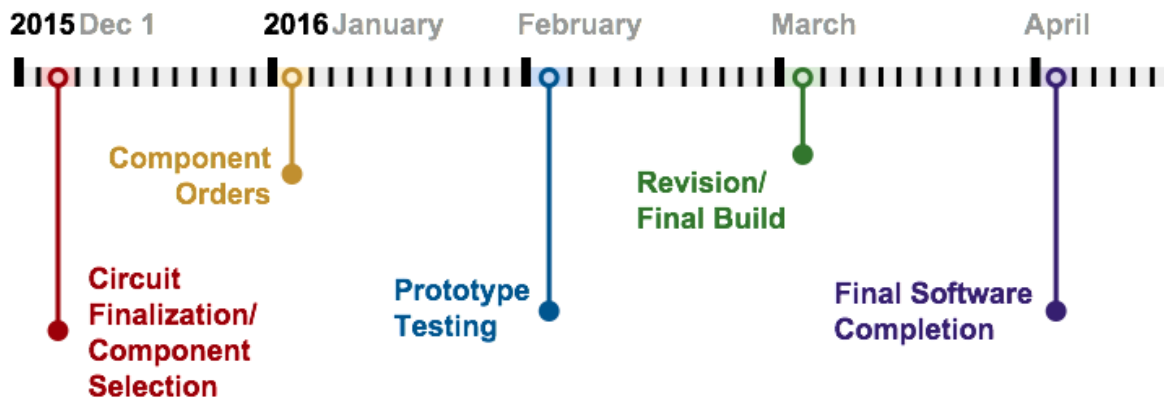
VII. Design and Production Cost Analysis

Bill of Materials:

Item	Qty	Component	Manufacturer	Manufacturer Part #	Vendor	Vendor Part #	Description	Price p/u	Link
1	1	MCU	n/a	n/a	PJRC	TEENSY32_PINS	Teensy 3.2, includes 2 pin header rows	\$ 22.80	T3.2
2	1	IC Chip	Microchip	MCP2561-H/P	Mouser	579-MCP2561-H/P	CAN Transceiver Chip	\$ 1.06	CAN
3	1	Crystal	Citizen FineDevice	CFS206-32.768KEZB-U	Mouser	695-CFS206-327KEZB-U	32.768KHz 12ppm 12.5pF Crystal for RTC	\$ 0.33	Crys
4	1	Battery	Panasonic	CR2032	Mouser	658-CR2032	CR2032 3V Coin Battery, 225mAh, 20x3.2mm	\$ 0.33	Batt
5	1	Battery	Renata	VBH2032-1	Mouser	614-VBH2032-1	CR2032 3V Coin Battery Holder	\$ 0.98	BattHo
6	1	LDO	Fairchild Semi.	KA7805ERTF	Mouser	512-KA7805ERTF	LDO - 5V, 1A, Max input = 35V, Package = TO-252	\$ 0.77	LDO
7	10	Cap	Murata	GCJ219R71H334KA12D	Mouser	81-GCJ219R71H334KA2D	0.33uF, 50VDC, 10%, 0805, buy multiples of 10	\$ 0.16	Cap1
8	10	Cap	Murata	GCJ21BR71H104KA01L	Mouser	81-GCJ21BR71H104KA1L	0.1uF, 50VDC, 10%, 0805, buy multiples of 10	\$ 0.114	Cap2
9	20	Res	Koa	RK73H2ATTD4701F	Mouser	660-RK73H2ATTD4701F	4.7K, 1%, 1/8W, 0805, buy multiples of 10	\$ 0.014	Res1
10	20	Res	Koa	RK73H2ATTD9101F	Mouser	660-RK73H2ATTD9101F	9.1K, 1%, 1/8W, 0805, buy multiples of 10	\$ 0.014	Res2
11	20	Res/Jumper	Koa	RK73Z2ATTD	Mouser	660-RK73Z2ATTD	0, 0805 (jumper), buy multiples of 10	\$ 0.012	Jump
12	10	Res	Koa	RK73H2ATTD1200F	Mouser	660-RK73H2ATTD1200F	120, 1%, 0805, 1/8W, buy multiples of 10	\$ 0.014	Res3
13	1	SD Adaptor	n/a	n/a	PJRC	SD_ADAPTOR	Micro SD Card Adaptor	\$ 8.00	SD
14	1	Enclosure	Deutsch	n/a	WireCare	PCM-DTM-1E-24-K	DTM 24 Contact PCB Enclosure Kit	\$ 65.66	Enc
15	1	Diode	Fairchild Semi.	ES1D-13-F	Mouser	621-ES1D-F	Diode, 200V, 1A, Vf=920mV	\$ 0.46	Dio
16	1	Board	OshPark	n/a	OshPark	n/a	Custom PCB board, \$5 per square inch, \$42.60 total for 3 boards (min) + expedited shipping	\$ 14.20	PCB
17	1	Pins	n/a	n/a	PJRC	HEADER_20X1	20 straight header pins, 0.1"	\$ 0.80	PIN1
18	1	Pins	3M	961210-6300-AP-PR	Mouser	517-9612106300ARRP	10 straight header pins, 0.1", 2 rows, SMD	\$ 2.03	PIN2
Total								\$ 121.10	

The total cost of the construction of this device totaled to \$121.10. This device provides comparable functionality to that of industry standard products while keeping a price that is only a fraction of what most other products in the market. Parts can be customized and can be swapped out or excluded entirely potentially bringing the cost down even more depending on the consumer's needs.

VIII. Task Allocation and Schedule



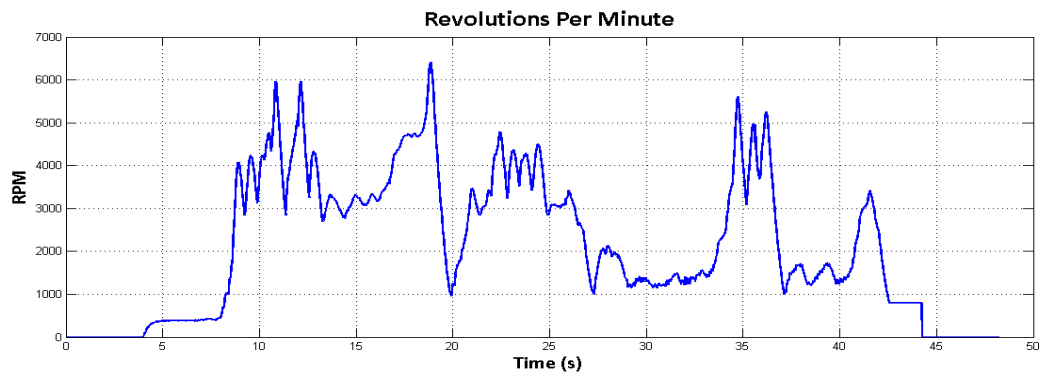
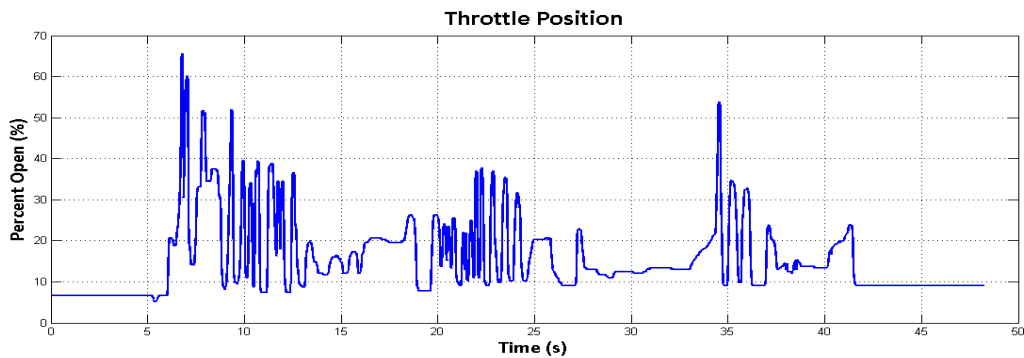
IX. Simulation/Modeling Results

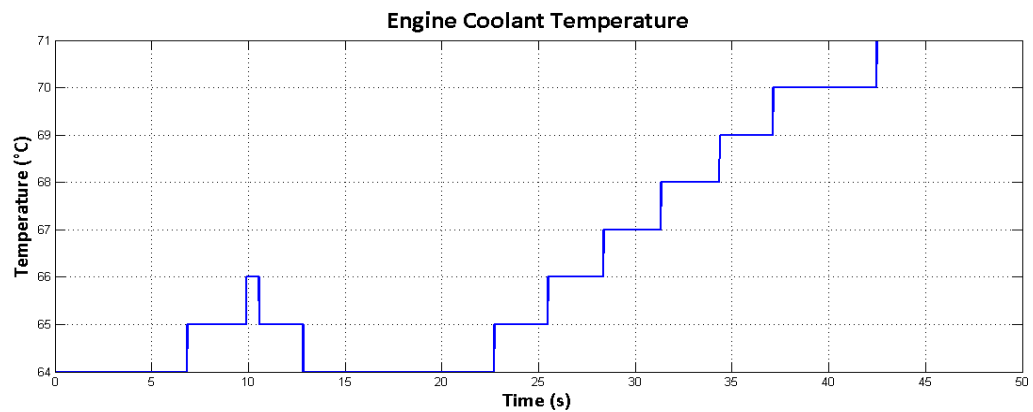
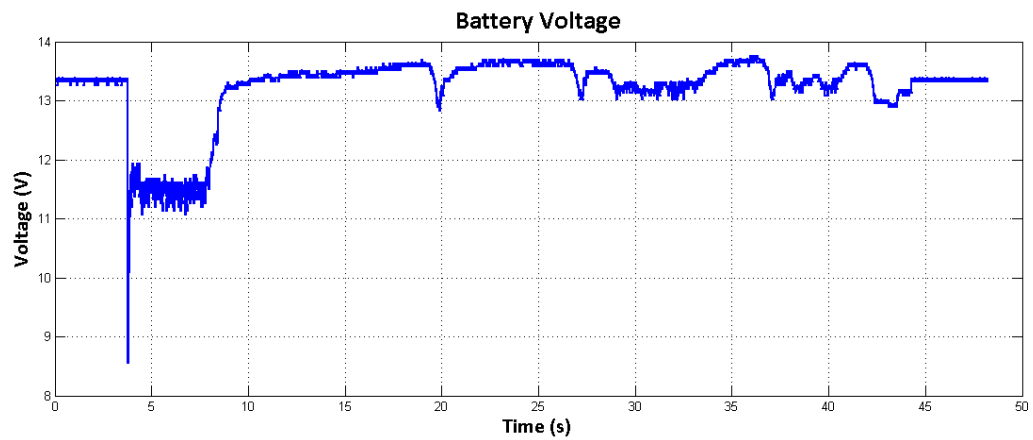
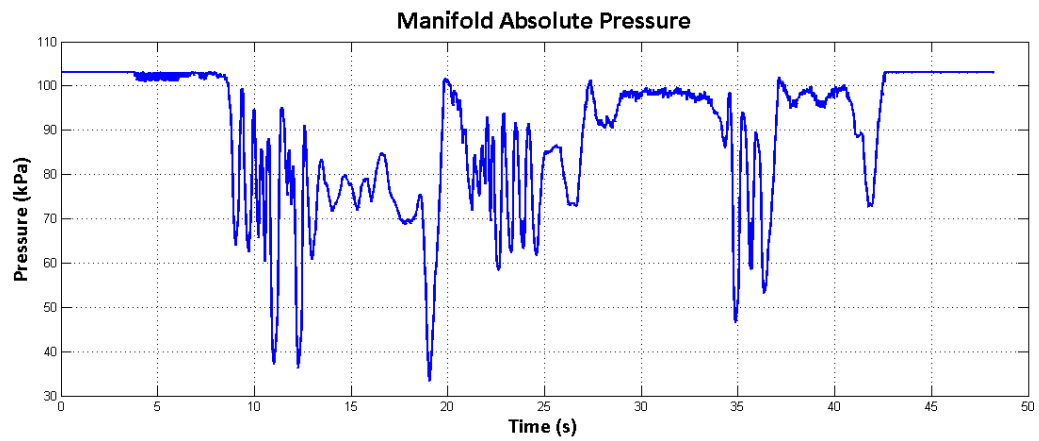
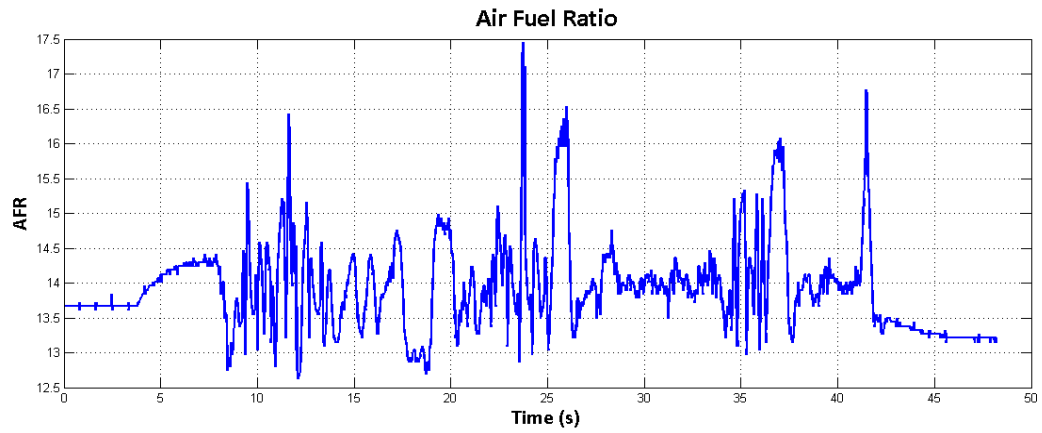
The results of testing analog sensors can be found in section X.

Below is shown the results of logging data from an Infinity 10 ECU installed on the SAE vehicle. The data was obtained from the CAN bus. The logging started before the vehicle was turned on and ended just after it was turned off. This data clearly shows the cranking as well as the revving.



Data Logger next to wiring harness: two shock travel sensors, 1 infrared temperature sensor





Since we are (SAE) making our own tunes for the vehicle, this data was able to give us insight on the vehicle's response. For example, we can clearly see in the data how long it takes for the RPMs to decrease from 6500 revolutions per minute to 1000 after the throttle is quickly released at about second 18.

X. Description of User Interface

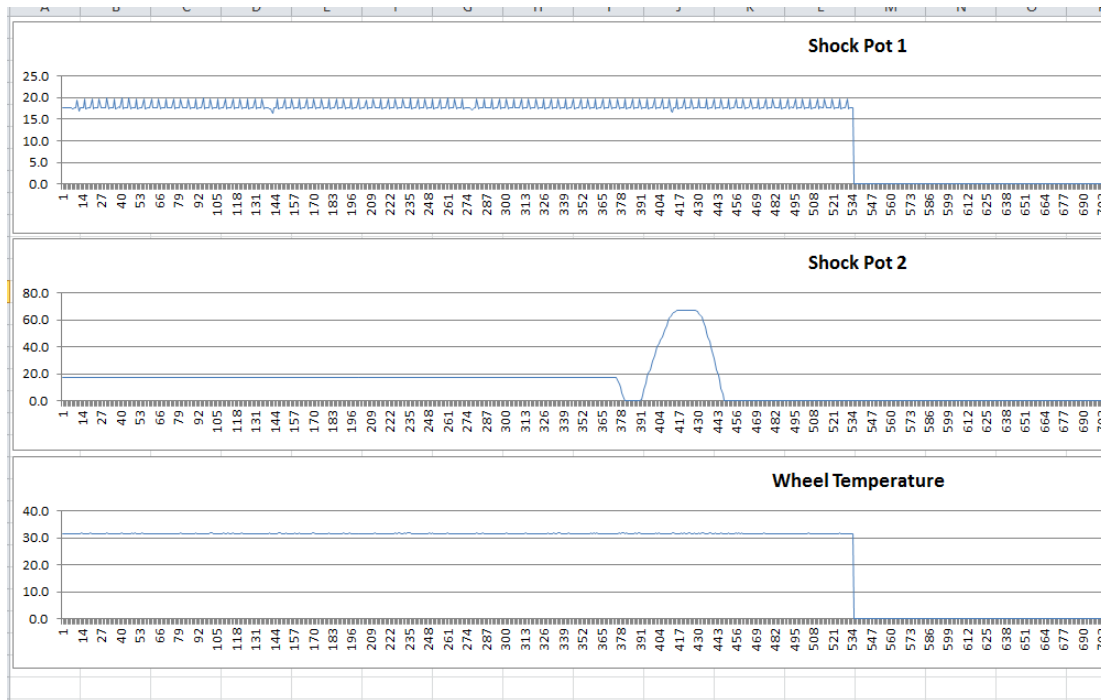
Software: The setup software was mention in the above sections several times. The software setup is needed to designate parameters in the MCU so that the correct settings are being used. The software should only communicate with the MCU and not necessarily reprogram it every time a setting is changed. This task might be difficult to achieve, so an interim “setup software” will be used. The interim method will involve generalizing code and adding condition blocks as well as data arrays storing user preferences. For the sake of completing the project, changing parameters in the code directly and reprogramming will be done rather than creating an interface between external software and the MCU.

The external software we opted to use for analysis was Microsoft Excel. Because we are saving our logged files as a Comma Separated Value file (.csv) Excel was a natural fit and a low cost solution.

The conversion formulas will be specified by the user in the excel sheet. Each channel will be mapped to specified values for each sensor. In this way, we can accommodate any analog output sensor commercially available or custom made. The sensor readings by default are a 10bit value (0-1023) corresponding to 0-3.3V. By simply giving the setup tab a value to map to the sensor, you can generate graphs with the right conversions.

	Name	Type	Sensor Reading		Value Mapping		Multiplier
			DLow	DHigh	Low	High	
Analog 1	Shock Pot 1	Position	0	1023	0	2	0.00195503
Analog 2	Shock Pot 2	Position	0	1023	80	300	0.29325513
Analog 3	Wheel Temperature	Temperature	0	1023	0	240	0.23460411
Analog 4							1
Analog 5							1
Analog 6							1
Analog 7							1
Analog 8							1
Analog 9							1
Analog 10							1
Analog 11							1
Analog 12							1
Analog 13							1
Analog 14							1
Analog 15							1
Analog 16							1

After setup, a simple import of a .csv file will generate the corresponding graphs to be used for analysis. This interface can be adjusted by the user for be display of the results suited to their needs. But all are graphed against the same time interval, so vertical alignment has data from exactly the same moment. By looking at multiple parameters simultaneously, design studies can be accomplished by future developers of specific vehicle components.



Future upgrades to this software are still being considered. Matlab and standalone software are still being evaluated. Since this project is being handed off to the SAE team here at UIC, we hope to be iterative and progressive with the applications for this product.

XI. Additional Issues

Economic: There are no foreseeable issues with economical factors impacting the manufacture and/or sale of this product. The end goal of this project is to have it available to anyone with an interest in the field and that require a device to acquire data for vehicle testing. The customers themselves will be able to build it on their own.

Environmental: All parts and manufacturing processes are subject to proper handling and disposal procedures of all standard electronics.

Political: The manufacturing and/or sale of this product abides by all government laws and regulations.

Global: This product isn't designed for use by those who are looking for an industry standard data acquisition device, but for anyone who has an interest for their own vehicle projects and testing for a much lower cost. This includes anyone in and out of the US.

Social: This product has no features that compromise anyone's private information. All data is distributed to intended parties, whoever the user chooses.

Ethical: The design, manufacturing, and sale of this product are subject to all rules, regulations, and responsibility outlined by the IEEE Code of Ethics.

Health and Safety: This product should be used in a manner that is appropriate and in compliance with specifications in the user manual. Normal usage does not pose any hazardous

or life threatening effects on the user. Power must be wired through a fuse to avoid permanent damage to the device in case of shorts.

Quality/Reliability: If used properly, this product can last for at least the lifetime of the vehicle it was intended for. It is to be considered, however, that this device is intended for low cost to the customer and should not be compared to industry grade equivalent.

Manufacturability: This product is intended for a print to build model. The user can use and customize the design and construct it on their own with parts readily available in the market.

Sustainability/Maintainability: The user must properly construct and maintain the device as they see fit. We are not liable for faulty construction by the customer. Because of the print to build model, no warranties will be issued. Software updates may be available through a website for download.

XII. Conclusion

Engineering work isn't as simple as making a design and implementing it. It also has a lot to do about how to convey your ideas to people. There are a lot of specification and regulations that needed to be accounted for that weren't brought up into initial brainstorming that are crucial to bringing a product into fruition. Going through this course has helped put us in that mindset of all the possibilities and all of the factors that need to be considered in the planning and design process.

Although the project was completed on time and fully implemented onto the vehicle, the extra features that would have made this device stand out from competition were not fully explored due to lack of time. The final stage of the project included a graphical user interface which would allow the user to easily analyze the data. A very specific MATLAB script was written to analyze the data logged from our customized Infinity 10 ECU. This proved quite effective, reliable and robust, but only to the SAE team. The issue with this data analyzer is that it cannot be easily ported to other devices, which in the end would prove useless to other users unless they implemented the data logger onto a very similarly equipped vehicle.

Luckily, this project will remain in the hands of the UIC SAE team and we are hopeful that future generations at SAE or ourselves individually can continue to improve this system and in the end create a user friendly data analyzer.

Appendix A: User Manual

Connect all sensors and external hardware to the appropriate connections. If a sensor uses +5V, make sure that both 4.7K and 9.1K resistors are soldered into the analog input used (refer to the schematic in appendix B). Note that this effectively converts a 0-5V scale to 0-3.3V scale, appropriate scaling must be done in the data analysis software if necessary. If a sensor uses +3.3V, a 0 ohm resistor should be placed in R1, R3, R5,..., or R23 (odd numbers).

Turn the Log Enable switch on to start logging and turn the switch off to stop logging. The Log Signal can be connected to an LED for example to visually see if the logger is actually logging. If it does not turn on, this means that there is an error which could be one of the following:

- No SD card connected
- SD card initialization error
- SD card write error

This error can be fixed by removing the SD card and plugging it back in. If this does not work, you can reset the power to the system (this should not be necessary). If this also does not work, there could be an internal software error or the SD card is malfunctioning.

Setup Software

A library has been written to perform all the required functions. To setup the sensors on the MCU, open the sketch called "DataLogger3.0.ino" using the Arduino IDE with the Teensyduino plugin installed. Plug in the logger using a USB cable and follow the instruction provided in the example code. Only the red boxes in the screenshot below should be modified:

```
// Create instances
VDAS_IO Sensors;

void setup() {
  delay(2000);
  Serial.begin(115200); delay(1000);
  Serial.print("Starting: ");

  // Setup the analog inputs: .setAnalogIn(int pin, int rate)
  //   int pin = analog pin to be setup (valid range: 1-12)
  //   int rate = desired read rate in Hz (valid range: 50, 25, 10, 5, 1)
  Sensors.setAnalogIn(1, 25);
  Sensors.setAnalogIn(11, 10);
  Sensors.setAnalogIn(12, 5);

  // Setup the CAN bus: .setCAN(uint32_t canID[], int size)
  //   uint32_t canID[0]: desired CAN message ID (usually in hex)
  //   int size: desired number of CAN messages to read (each contains up to 8 bytes)
  // NOTE: canID array size and parameter 2 of .setCAN MUST match
  uint32_t canID[2] = {0x5AE01, 0x5AE02}; // Wanted CAN messages
  Sensors.setCAN(canID, 2); // (CAN IDs array, # of messages to read)

  // NO NEED TO MODIFY CODE BELOW THIS LINE -----
  // -----
```

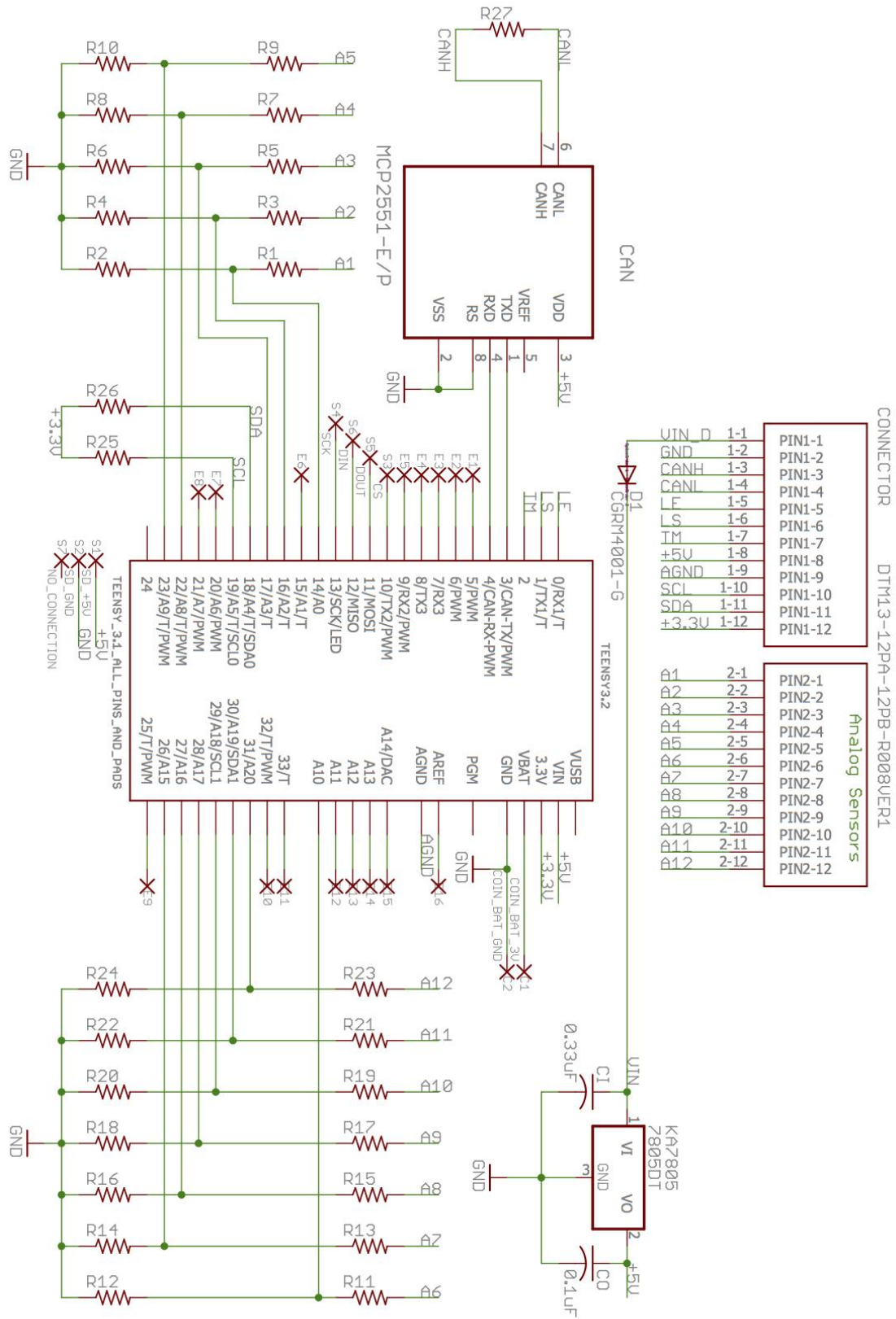
Appendix B: Schematics and Hardware

The pinout for the Vehicle Data Acquisition Device is shown below.

Connector 1			Connector 2		
Pin	Label	Description	Pin	Label	Description
1	VIN_D	Input voltage. Maximum input voltage is 40V	1	A1	Analog input 1
2	GND	Connect to ground (chassis or system ground)	2	A2	Analog input 2
3	CANH	CAN High	3	A3	Analog input 3
4	CANL	CAN Low	4	A4	Analog input 4
5	LE	Log Enable switch input	5	A5	Analog input 5
6	LS	Log Signal: digital output. High = logging, Low = not logging	6	A6	Analog input 6
7	TM	Time Marker push button input	7	A7	Analog input 7
8	+5V	+5V output for analog sensors	8	A8	Analog input 8
9	AGND	Analog ground. Connect sensors grounds to this ground	9	A9	Analog input 9
10	SCL	SCL connection for I2C bus	10	A10	Analog input 10
11	SDA	SDA connectin for I2C bus	11	A11	Analog input 11
12	+3.3V	+3.3V output for analog sensors	12	A12	Analog input 12

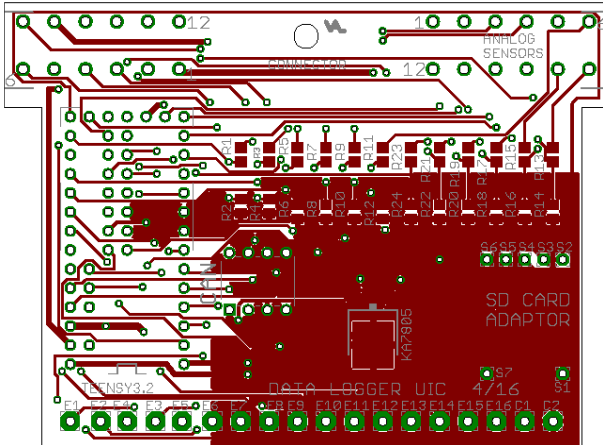
Due to our involvement in the Society of Automotive Engineers at the University of Illinois at Chicago, we have access to an array of sensors we currently use on our FSAE vehicle. These sensors were used to test all of our features.

Schematic

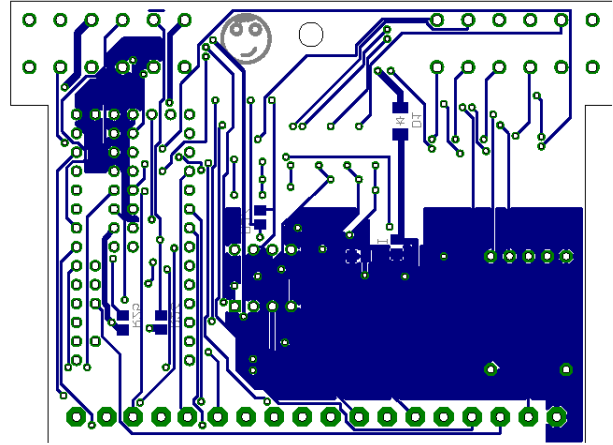


PCB Layout

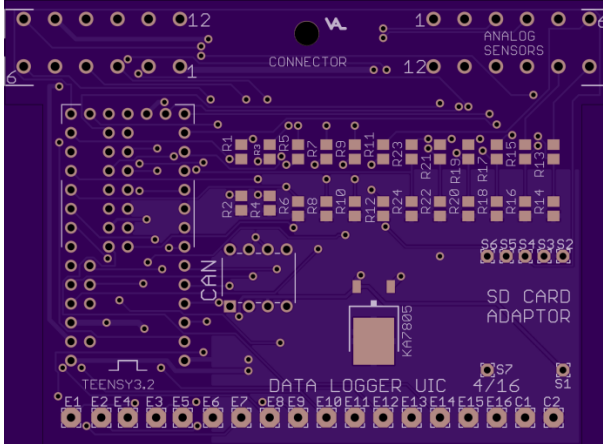
Front PCB layout design



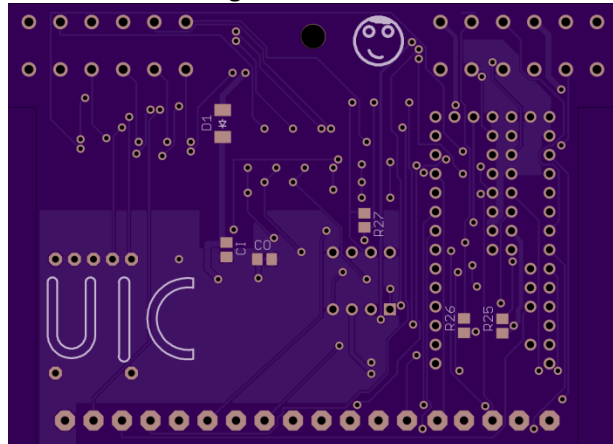
Back PCB Layout design



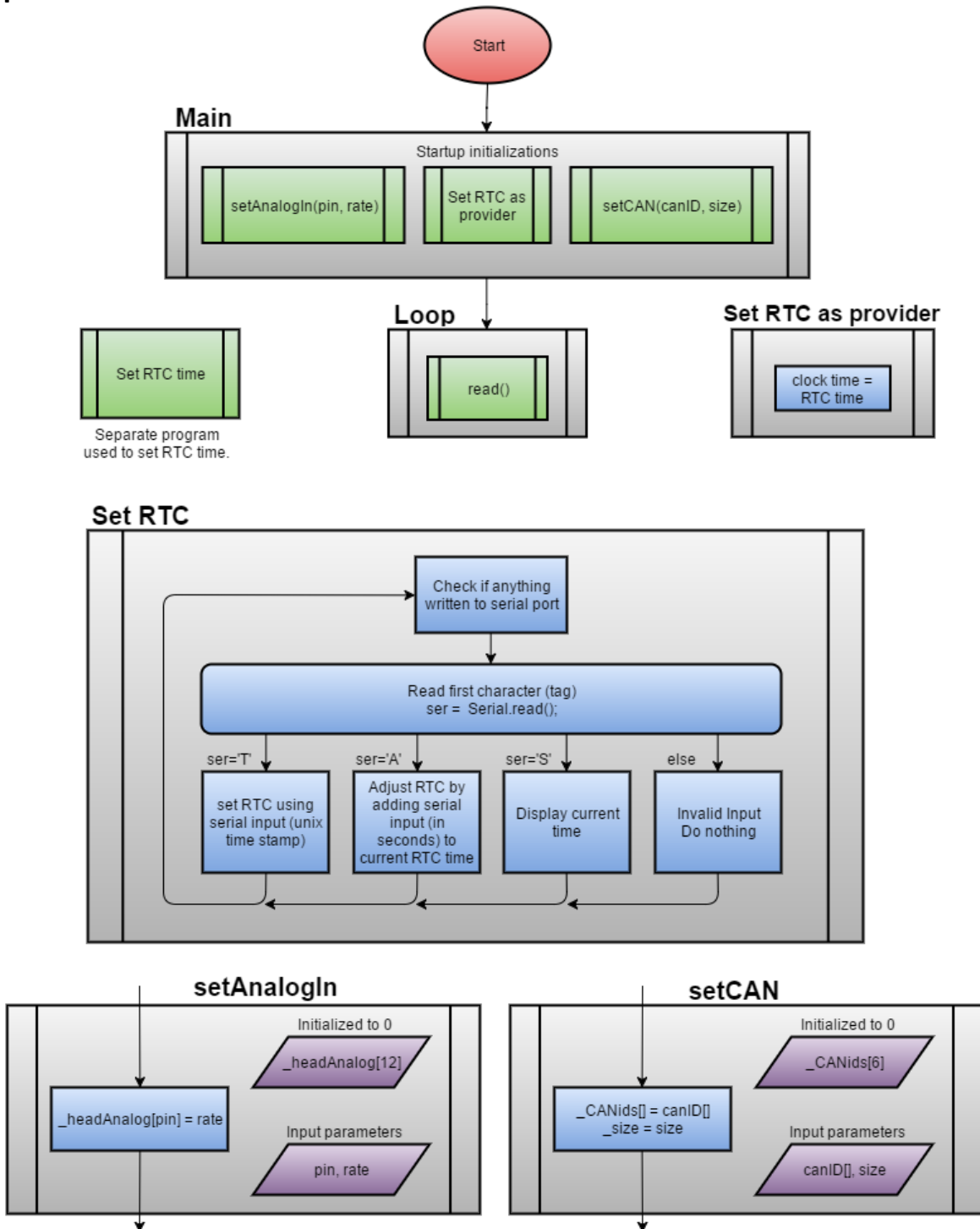
Front side rendering

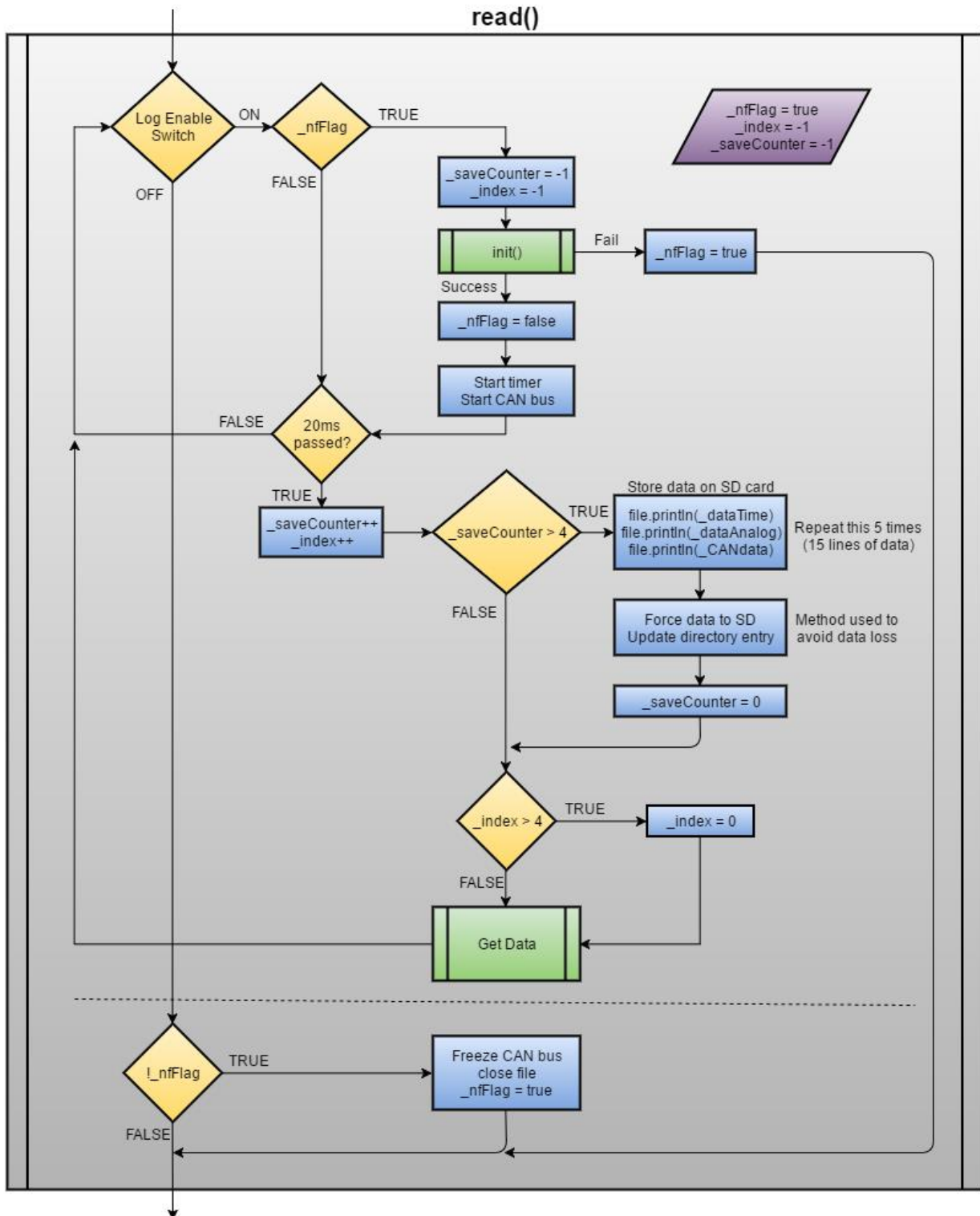


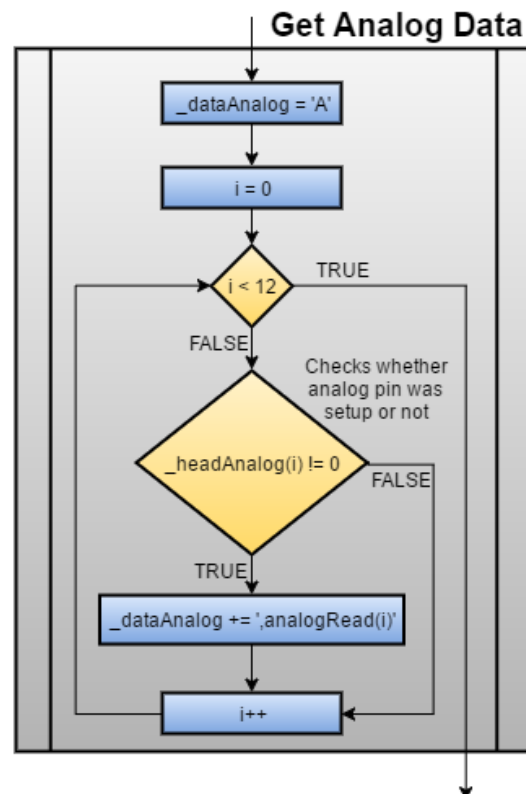
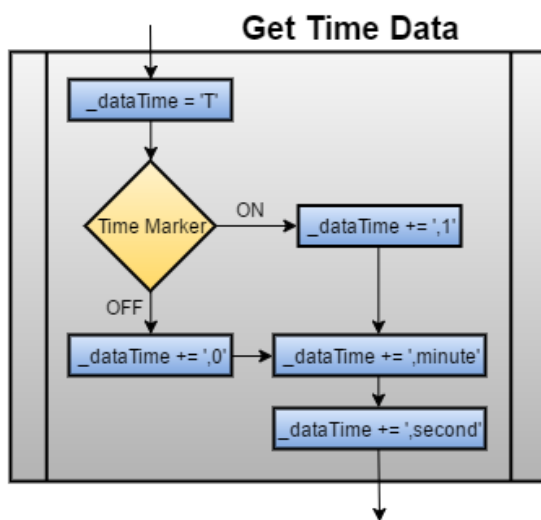
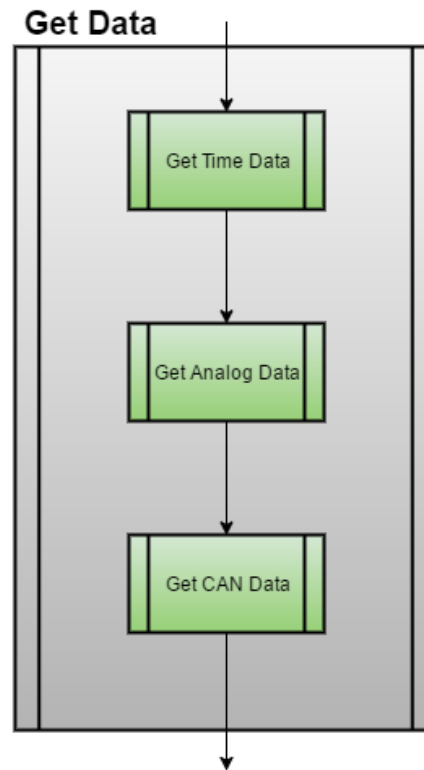
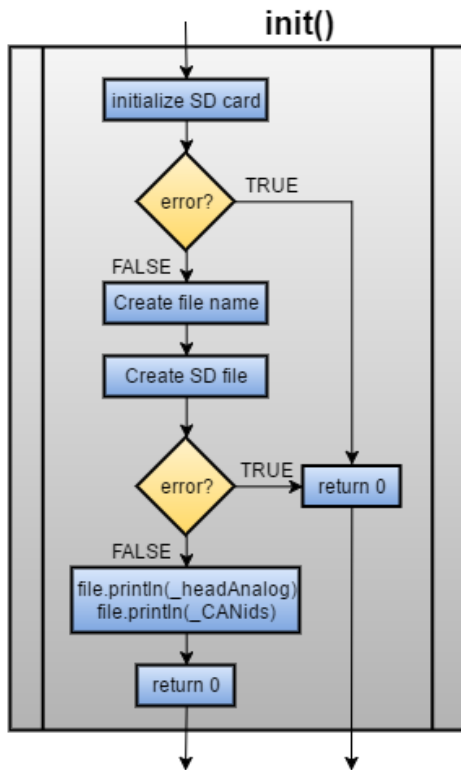
Back side rendering



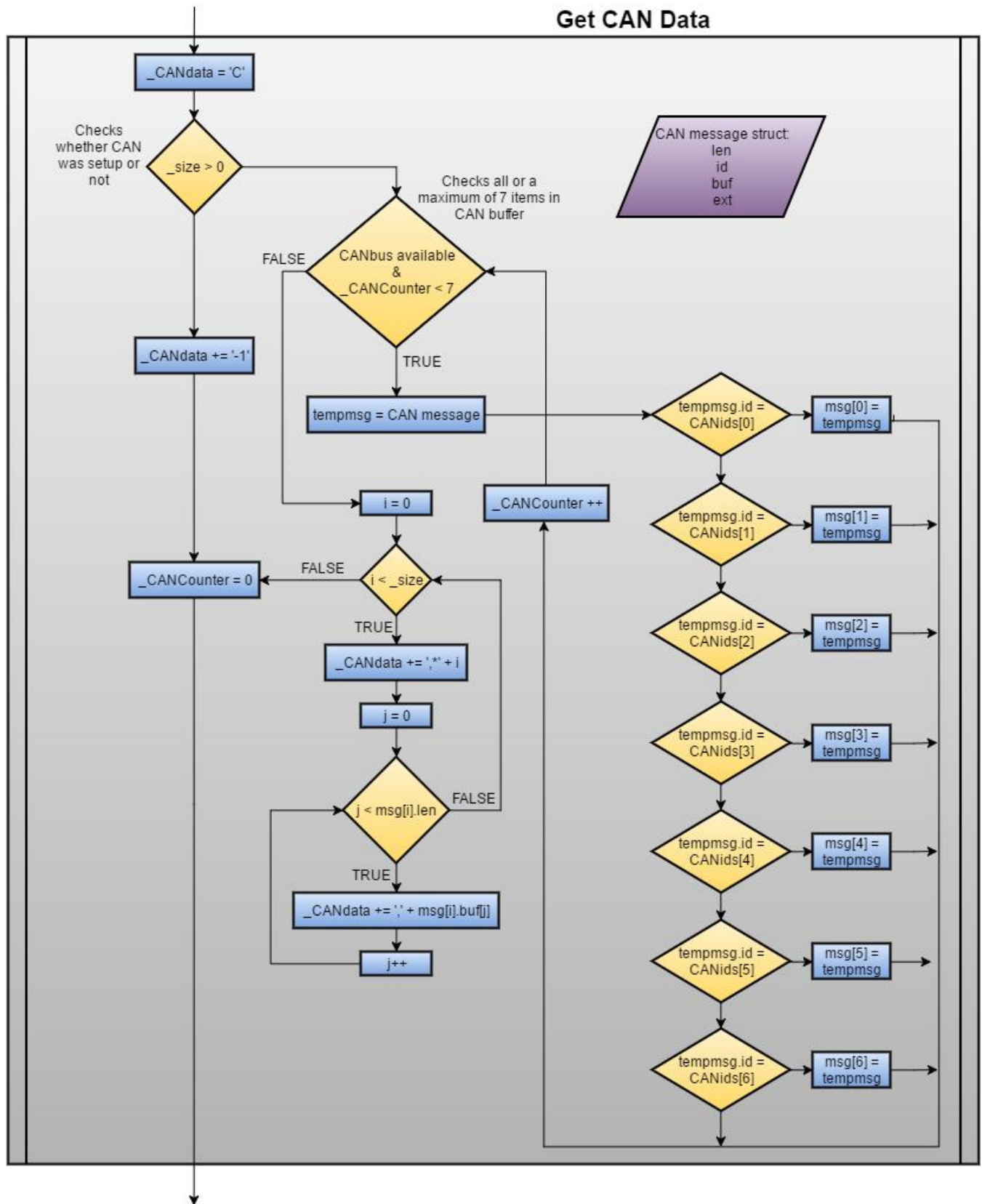
Appendix C: Software Flow







Get CAN Data



Appendix D: Requirements

The basic computer hardware requirements need to be met for the software to run. If the MATLAB script is used as the data analysis software, MATLAB and a valid license are required to run the script. The Arduino IDE and Teensyduino plugin can be found online and downloaded securely for free.

Appendix E: Software Code Listings

DataLogger3.0.ino

(user should use this code to setup the sensors and CAN bus)

```
#include <VDAS_TeensyRTC.h>
#include <VDAS_IO.h>
#include <Time.h>

/*
  Vehicle Data Acquisition System
  Revision: 3.0
  Start Date: 02/15/2016
  Last Updated: 04/15/2016

  ** Contributors:
    Cristian Valadez

  ** Purpose:
    The purpose of this program is to collect data from several sensors and
    peripherals. Analog sensors, CAN and I2C is supported, see datasheet.
    Data is written to an external memory device which the user can access.
  */

// Create instances
VDAS_IO Sensors;

void setup() {
  delay(2000);
  Serial.begin(115200); delay(1000);
  Serial.print("Starting: ");

  // Setup the analog inputs: .setAnalogIn(int pin, int rate)
  //   int pin = analog pin to be setup (valid range: 1-12)
  //   int rate = desired read rate in Hz (valid range: 50, 25, 10, 5, 1)
  Sensors.setAnalogIn(1, 25);
  Sensors.setAnalogIn(11, 10);
  Sensors.setAnalogIn(12, 5);

  // Setup the CAN bus: .setCAN(uint32_t canID[], int size)
  //   uint32_t canID[0]: desired CAN message ID (usually in hex)
```

```

// int size: desired number of CAN messages to read (each contains up to 8 bytes)
// NOTE: canID array size and parameter 2 of .setCAN MUST match
uint32_t canID[2] = {0x5AE01, 0x5AE02}; // Wanted CAN messages
Sensors.setCAN(canID, 2); // (CAN IDs array, # of messages to read)

// NO NEED TO MODIFY CODE BELOW THIS LINE -----
// -----
// Use Teensy 3.2 RTC to keep time
setSyncProvider(getTeensy3Time);
digitalClockDisplay();
}

void loop() {
  // Read all sensors and store data to file on SD card
  // Will only log when Log Enable switch is ON
  Sensors.read();
}

void digitalClockDisplay() {
  // display the time
  Serial.print(hour());
  printDigits(minute());
  printDigits(second());
  Serial.print(" "); Serial.print(day());
  Serial.print(" "); Serial.print(month());
  Serial.print(" "); Serial.print(year());
  Serial.println();
}

void printDigits(int digits) {
  // prints preceding colon and leading 0
  Serial.print(":");
  if (digits < 10)
    Serial.print('0');
  Serial.print(digits);
}

time_t getTeensy3Time() {
  return Teensy3Clock.get();
}

```

setRTCDataLogger3.0.ino

(user should only modify TIME_ZONE if necessary, use as reference only, see instructions in code below and perform operations in Serial Monitor)

/*

Vehicle Data Acquisition System : Set internal RTC time

Revision: 1.0

Start Date: 03/18/2016

Last Updated: 03/18/2016

** Contributors:

Cristian Valadez

** Purpose:

The purpose of this program is to set the internal Teensy 3.2 RTC clock. Note that a 32.768 kHz crystal MUST be soldered onto the board. The 3V coin battery retains time on the RTC.

To set the time, upload this program and open the Serial Monitor. Type the time you want preceded by 'T' to set the RTC to in Epoch time units. You can use <http://www.epochconverter.com/> to get current Epoch. Note that the epoch is always in GMT, so in setup code change the timeZone variable to correspond to your time zone.

Example:

In Serial Port type:

T1458325800

If variable timeZone = -6; (Chicago DST), the time set is
3/18/2016, 1:30:00 PM

RTC clock only needs to be set once, coin battery should last about 8 years. Occasionally it might be best to set the RTC clock to avoid drift in time and account for Daylight Savings Time.

*/

```
#include <TimeLib.h>
```

```
static const int TIME_ZONE = -5; // Time Zone
```

```
time_t t = 0L;
```

```
byte ser = 0;
```

```
void setup() {
```

```
// NO NEED TO MODIFY CODE BELOW THIS LINE
```

```

// -----
delay(1000);

// Use Teensy 3.2 RTC to keep time
setSyncProvider(getTeensy3Time);

Serial.begin(115200); // Start serial monitor
//while (!Serial);    // Wait for Serial Monitor to open
delay(1000);
Serial.println("To set RTC time, Enter the string: 'T' + 'epoch'");
Serial.println("To adjust RTC time, Enter the string: 'A' + 'seconds'");
Serial.println("To see current RTC time, Enter the string: 'S'\n");
}

void loop() {
// Continously check if something is entered in Serial Monitor
if (Serial.available()) {
    ser = Serial.read();    // Read first byte (header tag)
    if (ser == 'T') {        // If "T", set time
        t = Serial.parseInt() + (TIME_ZONE * 3600); // Get epoch and add time zone difference
        Teensy3Clock.set(t);    // set the RTC
        delay(100);            // short delay
        setTime(Teensy3Clock.get()); // Set Teensy time using RTC
        Serial.print("RTC set: ");
        digitalClockDisplay();    // Display time that was set
    }

    else if (ser == 'A') {    // If "A", adjust time in seconds
        t = Teensy3Clock.get() + Serial.parseInt(); // Get adjusted time
        Teensy3Clock.set(t);    // set the RTC
        delay(100);            // short delay
        setTime(Teensy3Clock.get()); // Set Teensy time using RTC
        Serial.print("RTC adjusted: ");
        digitalClockDisplay();    // Display time that was set
    }

    else if (ser == 'S') {    // If "S", display current time
        Serial.print("Current time: ");
        digitalClockDisplay();    // Display time that was set
    }

    else {                    // Invalid input
        Serial.println("Invalid input...\n");
    }
}

```

```
}  
}
```

```
void digitalClockDisplay() {  
    // display the time  
    Serial.print(hour());  
    printDigits(minute());  
    printDigits(second());  
    Serial.print(" "); Serial.print(day());  
    Serial.print(" "); Serial.print(month());  
    Serial.print(" "); Serial.print(year());  
    Serial.println("\n");  
}
```

```
void printDigits(int digits) {  
    // utility function for digital clock display: prints preceding colon and leading 0  
    Serial.print(":");  
    if (digits < 10)  
        Serial.print('0');  
    Serial.print(digits);  
}
```

```
time_t getTeensy3Time() {  
    return Teensy3Clock.get();  
}
```


VDAS_IO.h

Custom library header, use as reference only

```
/*
  Vehicle Data Acquisition System
  Input/Output Library (header file)
  Revision: 1.0
  Start Date: 02/20/2016
  Last Updated: 04/15/2016
  Contributors: Cristian Valadez
*/

#ifndef VDAS_IO_h
#define VDAS_IO_h

#include "Arduino.h"

class VDAS_IO {
public:
    VDAS_IO(void);           // Instance
    void setAnalogIn(int pin, int rate); // Specify analog pin and its read rate
    void setCAN(uint32_t canID[], int size); // Setup CAN
    void read();             // Reads from sensors
    int init();              // Create new file

private:
    // Private variables
    // Time variables
    String _dataTime[5];     // Time data

    // Analog variables
    String _headAnalog;      // Analog header
    String _dataAnalog[5];   // Analog data
    /*int _mapAnalog[12] = {    // Analog pin mapping, see datasheet
        1, 2, 3, 8, 9, 15, 16, 17, 18,
        19, 20, 10};

    // Actual mapping below..breadboard testing mapping above*/

    int _mapAnalog[12] = {    // Analog pin mapping, see datasheet
        0, 2, 3, 8, 9, 10,
        26, 27, 28, 29, 30, 31}; // Second row numbers correspond to
                                // digital pin numbers

    // CAN variables
```

```
int _size = 0;
String _CANdata[5];          // CAN data
String _CANids;              // CAN header
uint32_t _intCANids[6] = {0,0,0,0,0,0};

// Extra variables
boolean _nfFlag = true;      // Create new file? (flag)
uint8_t _CANCounter;         // CAN counter
int _saveCounter;            // Save to SD counter
int _index;                  // Data array index
};

#endif
```

VDAS_IO.cpp

Custom library source code, use as reference

Note: some lines of code were commented out to create a simpler version of the system for the demo performed at the Engineering Expo April, 2016 at UIC.

```
/*
  Vehicle Data Acquisition System
  Input/Output Library (source file)
  Revision: 1.0
  Start Date: 02/20/2016
  Last Updated: 04/15/2016
  Contributors: Cristian Valadez
*/

#include "Arduino.h"
#include "VDAS_IO.h"
#include "FlexCAN.h"
#include "Metro.h"
#include "TimeLib.h"
#include "SPI.h"
#include "SdFat.h"

const uint8_t chipSelect = SS;
//static CAN_filter_t filt0, filt1, filt2, filt3, filt4, filt5;  // CAN filters

Metro timer = Metro(20);      // 20ms timer
FlexCAN CANbus(500000);       // CAN instance
CAN_message_t tempmsg;        // Temporary CAN message
CAN_message_t msg[5][6];      // 6 FIFO CAN buffers
SdFat sd;                      // File system object
SdFile file;                   // SD log file

VDAS_IO::VDAS_IO(){
  // Create constructor
  // Setup digital pins (not configurable by user, set by hardware)
  pinMode(0, INPUT);           // Pin 0 dedicated to Log Enable switch
  pinMode(0, INPUT_PULLUP);    // Setup up pullup
  pinMode(1, INPUT);           // Pin 1 dedicated to Time Marker button
  pinMode(1, INPUT_PULLUP);    // Setup up pullup
  pinMode(2, OUTPUT);          // Pin 2 dedicated Log Enable out signal

  // Initialize
  _headAnalog = "0,0,0,0,0,0,0,0,0,0,0";
```

```

// Reset flags and counters
_nfFlag = true;           // Make sure flag initialized true
_saveCounter = -1;        // Reset save counter
_CANCounter = 0;          // Reset CAN counter
_index = -1;              // Reset index counter
}

void VDAS_IO::setAnalogIn(int pin, int rate){ // Set analog inputs
// Pin input accepts only integers 1-12
// Please see datasheet for hardware pin mappings
// Store user specified read rate into analog header array
// Rates: 50Hz=1 25Hz=2 10Hz=3 5Hz=4 1Hz=5
if (rate == 50){
    _headAnalog.setCharAt(2*(pin-1),'1');
} else if (rate == 25){
    _headAnalog.setCharAt(2*(pin-1),'2');
} else if (rate == 10){
    _headAnalog.setCharAt(2*(pin-1),'3');
} else if (rate == 5){
    _headAnalog.setCharAt(2*(pin-1),'4');
} else if (rate == 1){
    _headAnalog.setCharAt(2*(pin-1),'5');
} else{
    // Default is 50 Hz
    _headAnalog.setCharAt(2*(pin-1),'1');
}
}

void VDAS_IO::setCAN(uint32_t canID[], int size){
    _size = size;

    // Pass over canID elements to _CANids String (char array)
    // Used to send over to USB file
    _CANids = String(canID[0], HEX); // Pass first element
    for (int i=1; i < _size; i++){ // Pass remaining elements
        _CANids = _CANids + ',' + String(canID[i], HEX);
    }

    // Pass over canID elements to _intCANids uint32_t array
    // Used for internal checking
    for (int i=0; i < _size; i++){
        _intCANids[i] = canID[i];
    }
}

```

```

/*// Create CAN filters
filt0.rtr = 0; filt1.rtr = 0; filt2.rtr = 0; filt3.rtr = 0; filt4.rtr = 0; filt5.rtr = 0;
filt0.ext = 0; filt1.ext = 0; filt2.ext = 0; filt3.ext = 0; filt4.ext = 0; filt5.ext = 0;
filt0.id = _intCANids[0]; filt1.id = _intCANids[1]; filt2.id = _intCANids[2];
filt3.id = _intCANids[3]; filt4.id = _intCANids[4]; filt5.id = _intCANids[5];

// Set CAN filters
CANbus.setFilter(filt0, 0); CANbus.setFilter(filt1, 1);
CANbus.setFilter(filt2, 2); CANbus.setFilter(filt3, 3);
CANbus.setFilter(filt4, 4); CANbus.setFilter(filt5, 5);*/
}

void VDAS_IO::read() {           // Read all data

    while (digitalRead(0)) {     // Read if Log Enable switch is on

        // Create file unless already created ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** ** 
        if (_nfFlag) {           // If "new file flag" = 1, create new file
            // Set Log Enable digital pin output to HIGH
            digitalWrite(2, 1);  // LE signal: 1=on, 0=off
            _saveCounter = -1;    // Reset save counter
            _index = -1;         // Reset index counter
            if (!init()) {        // Create new file
                _nfFlag = true;   // Returned 1 = SD couldnt open
                return;           // Exit read()
            }
            _nfFlag = false;      // Set flag to false (file created)
            timer.reset();        // Start timer now
            CANbus.begin();       // Start the CAN communication
        }

        // Start logging. Save data if metro timer ticks ** ** ** ** ** ** ** ** 
        if (timer.check()) {      // 20ms passed, get data
            _saveCounter++;        // Increment save counter
            _index++;              // Increment index counter

            // Store every 5 iterations (100ms) ** ** ** *
            if (_saveCounter > 4) {
                // Store data
                // Line 1: Time data
                // Line 2: Analog data
                // Line 3: CAN data
                for (int i_s = 0; i_s < 5; i_s++) {

```

```

// Commented out for Engineering Expo Demo, only get 3 sensors
//file.println(_dataTime[i_s]);
file.println(_dataAnalog[i_s]);
//file.println(_CANdata[i_s]);
}
// Force data to SD and update directory entry to avoid data loss
if (!file.sync() || file.getWriteError()) {
    // Sync error
    Serial.println("SYNC ERROR!!!!!!!!!!");
    delay(100);
    _saveCounter = 0;
    _index = 0;
    return;                // Start over
}
_saveCounter = 0;          // Reset save counter
}
if (_index > 4) {           // Reset index counter if it reaches 5
    _index = 0;
}

// Get time stamp and check time marker ** ** ** **
_dataTime[_index] = "T," + String(digitalRead(1)) + "," +
    String(minute()) + "," + String(second());

// Get CAN data ** ** ** **
_CANdata[_index] = "C";    // Reset string contents
if (_size > 0){            // Read if CAN was setup

    while (CANbus.available() & (_CANCounter<7)){ // Read all buffers in FIFO a
maximum
                                // of 6 times (6 FIFO buffers)
        CANbus.read(tempmsg); // Read CAN message
        // Sort temporary message by matching ids
        if (_intCANids[0] == tempmsg.id){
            msg[_index][0]=tempmsg;
        } else if (_intCANids[1] == tempmsg.id) {
            msg[_index][1]=tempmsg;
        } else if (_intCANids[2] == tempmsg.id){
            msg[_index][2]=tempmsg;
        } else if (_intCANids[3] == tempmsg.id){
            msg[_index][3]=tempmsg;
        } else if (_intCANids[4] == tempmsg.id){
            msg[_index][4]=tempmsg;
        } else if (_intCANids[5] == tempmsg.id){

```

```

        msg[_index][5]=tempmsg;
    } else if (_intCANids[6] == tempmsg.id){
        msg[_index][6]=tempmsg;
    }

    _CANCounter += 1;          // Increment counter
}
// Store data in _CANdata using correct order
// i.e. Message1, Message2, ...
for (int i=0; i < _size; i++){
    _CANdata[_index] += "," + String(i);
    for (int j=0; j < msg[_index][i].len; j++){
        _CANdata[_index] += "," + String(msg[_index][i].buf[j], HEX);
    }
}
} else{                      // If CAN was not setup,
    _CANdata[_index] += "-1";    // set _CANdata to -1
}
_CANCounter = 0;              // Reset counter

// Get analog data ** ** ** **
_dataAnalog[_index] = "A";      // Reset string contents
for (int pin=0; pin < 24; pin += 2){    // Check if pin was setup by user
    if (!(_headAnalog.charAt(pin) == '0')){    // !(0) = pin used, read data
        // Read from mapped hardware pin
        _dataAnalog[_index] += "," + String(analogRead(_mapAnalog[pin/2]), HEX);

        // Expo version
        _dataAnalog[_index] += "," + String(analogRead(_mapAnalog[pin/2]));
    }
}

// Get I2C data ** NOT ADDED IN VERSION 1.0

// FOR DEBUGGING ** **
if (Serial){
    // Print time data
    Serial.println(_dataTime[_index]);
    // Print analog data
    Serial.println(_dataAnalog[_index]);
    // Print CAN data
    if (_size > 0) Serial.println(_CANdata[_index]);
}

```

```

    }
}

if (!_nfFlag){          // Executes only once right after log switch is turned off
    CANbus.end();        // Enter CAN freeze mode
    digitalWrite(2, 0);  // Log Enable switch off
    file.close();        // Close file
    delay(5);
    _nfFlag = true;      // New file needs to be created after LE switched on
}
}

// ** ** ** **
// ** ** ** **
// ** ** **

int VDAS_IO::init(){    // Create new file
    // Initialize SD card at SPI_FULL_SPEED
    if (!sd.begin(chipSelect, SPI_FULL_SPEED)) {
        Serial.println("SD begin error");
        delay(100);      // SD didn't respond
        return 0;         // Exit and try again
        //sd.initErrorHalt();
    }

    // File naming algorithm
    // Ex: 1212153020 = December 12, 15:30::20 (483FFCAB in hex)
    // This number is converted to hex since max FAT32 file naming size = 8.3
    uint32_t longfileName; // File name, unsigned 32 bits
    String fileName;       // File name
    char _fileName[13];    // File name (char array)

    longfileName = month()*100000000 + day()*1000000 + // Month and day
    hour()*10000 + minute()*100 + second();           // Hour, minute and second

    // Convert to hex and create file name, pad with 0 if necessary
    if (month() > 2){ // March produces 8 chars while anything before produces 7 chars
        fileName = String(longfileName, HEX) + ".csv";
    } else {
        fileName = '0' + String(longfileName, HEX) + ".csv";
    }

    // Convert to char array
    fileName.toCharArray(_fileName, 13);

    // Check if CAN was setup, if not, set _CANids = -1

```



```

if (_size == 0){          // CAN was not setup
    _CANids = "-1";
}

// Create file on SD card
if (!file.open(_fileName, O_CREAT | O_WRITE | O_EXCL)) {
    Serial.println("SD create file error");
    delay(100);          // File wasn't created
    return 0;            // Exit and try again
} delay(20);            // File created

// Send header files to file
// Comment out for Engineering Expo Demo
//file.println(_headAnalog); // Analog header
//file.println(_CANids);    // CAN message IDs header

// Check for write errors
if (file.getWriteError()) {
    //error("write failed");
    Serial.println("Write error: header lines");
    delay(100);
    return 0;
}

// FOR DEBUGGING: PRINT FILE NAME AND HEADER FILES
if (Serial){
    // Send fileName
    Serial.println();      // For cleaner display
    Serial.println(fileName);

    // Send analog header
    Serial.println(_headAnalog);

    // Send CAN message IDs header
    Serial.println(_CANids);
    Serial.println("*****"); // For cleaner display
}

return 1;                // File successfully created
}

```