Controller Area Network (CAN) and QFSAE

Ethan Peterson

December 31, 2019

1 CAN Bus Protocol

1.1 Background

Controller Area Network (CAN) is a communications protocol that is standard for electronics in the automotive industry. The protocol was introduced by the Society of Automotive engineers (SAE) in 1986. CAN is primarily used for communication between different electronics systems in different locations on the car. For instance, sensor systems on the back of the vehicle would be able to read data from the Engine Control Unit (ECU). In the case of the team's car, the ECU is the primary CAN device with which other systems communicate. The ECU provides a variety of metrics about the engine such as RPM, throttle position, ignition angle and many more. These metrics are read from the ECU over CAN by using their CAN ID. Each message must have a unique ID from the IDs in use by other devices on the car. It is also important to note that only one message can be on a CAN bus at a time. Messages with more "dominant" IDs will take precedence on the bus. As a result, there must be careful consideration when selecting an ID for a certain message depending on its overall priority.

1.2 Data Transmission and Addressing

CAN is two wire half-duplex serial-based communication protocol. Half duplex means that the CAN bus can only be used to send or receive messages in an alternating fashion and not at the same time as with full-duplex protocols. CAN messages consist of an identifier (CAN ID) and a data frame. There are several standards for the size of the message frames. The ECU on the QFSAE car implements the Society of Automotive Engineers (SAE) J1939 standard. This standard uses 29 bit message identifiers and 64 bit data frames. Since CAN is half-duplex, only one message can be on the bus at a time. However, CAN can still be used to create large networks of CAN devices despite this, namely through its implementation of a priority bus. CAN defines "dominant" and "recessive" bits where logical 0 is dominant. This allows to CAN to resolve conflicts on the bus. The table below summarizes this process.

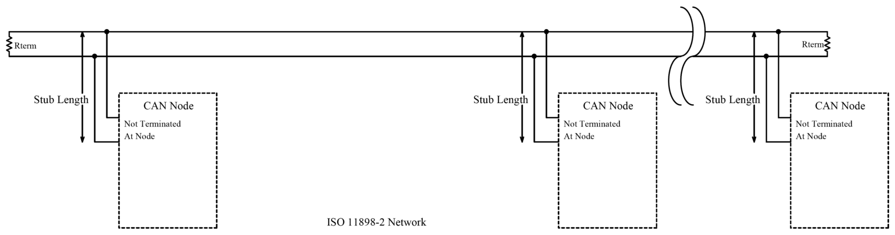
	CAN ID Bits				
	Start Bit	3	2	1	0
Node 3	0	0	0	0	1
Node 4	0	0	1	Stopped Transmitting	N/A
CAN Bus Data	0	0	0	0	1

Table 1: Demonstrating how CAN chooses the most "dominant" ID on the bus in the case of a conflict between messages.

As a result of 0 being the dominant bit on the bus, it is the smallest CAN IDs that have the highest priority on the bus. This property of the bus must be considered when selecting CAN IDs for custom messages outside of those broadcast by the ECU. For example, a message indicating to the dash that a gear shift happened is likely of a higher priority than updating the RPM.

1.3 Wiring

CAN consists of a two wire interface where the first two nodes connected to one another should have two 120Ω terminating resistors connected on either end of the bus. Nodes are defined as devices connected to the bus which can send or receive CAN messages. The diagram below is an example CAN network with three nodes.



2 CAN on the Q19 Car

In the case of the Formula car, the ECU has a terminating resistor. This is also the case for the end of the car's CAN line where the CAN to USB dongle is found. Thus, additional CAN peripheral should omit the terminating resistor in order to ensure thatthe bus keeps functioning properly.

2.1 Testbench Code

The Arduino code for a CAN test bench is provided below. The Arduino INO file can be found here.

```
// CAN Testbench
// runs between two CAN shields and adjusts the brightness of an LED
// connected to the receiver based off potentiometer readings from the sender

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

#define SPI_CS_PIN 9

// Set to 0 to run receiver code
#define SENDING 0

// LED and Potentiometer Definitions
#define LED_CATHODE 7
#define LED_PWM 6

#define POT_VCC A0
#define POT_WIPER A1
#define POT_GND A2

MCP_CAN CAN(SPI_CS_PIN);

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

  // SETUP LED FOR OUTPUT
  pinMode(LED_CATHODE, OUTPUT);
  pinMode(LED_PWM, OUTPUT);
  digitalWrite(LED_CATHODE, LOW);

  // SETUP POTENTIOMETER FOR BRIGHTNESS READ
  pinMode(POT_VCC, OUTPUT);
  pinMode(POT_WIPER, INPUT); // WIPER PIN
  pinMode(POT_GND, OUTPUT);
  digitalWrite(POT_VCC, HIGH);
  digitalWrite(POT_GND, LOW);

  while (CAN.begin(CAN_500KBPS) != CAN_OK) {
    Serial.println("CAN BUS init failure");
    Serial.println("Trying again");
    delay(100);
  }
  Serial.println("CAN Bus Initialized!");
}

void loop() {
  unsigned char len = 0;

  // messages have max length of 8 bytes
  unsigned char buf[8];

  // Check if there is a message available
  if (!SENDING && CAN_MSGAVAIL == CAN.checkReceive()) {
    CAN.readMsgBuf(&len, buf);
    unsigned long id = CAN.getCanId();
    Serial.print("Getting Data from ID: ");
    Serial.println(id, HEX);

    for (int i = 0; i < len; i++) {
      Serial.print(buf[i]);
      Serial.print("\t");
    }
    // Brightness of LED depends on potentiometer position on the sending CAN
    // node
    Serial.print("writing to LED: ");
    Serial.println(buf[0]);
    analogWrite(LED_PWM, buf[0]);
    Serial.print("\n");
    Serial.println("END OF MESSAGE");
  } else {
    unsigned char message[8] = {0, 0, 0, 0, 0, 0, 0, 0};
    unsigned long sendingID = 0x00;
    unsigned char reading = map(analogRead(POT_WIPER), 0, 1023, 0, 255);
    Serial.println(reading);
    message[0] = reading;
    CAN.sendMsgBuf(sendingID, 0, 8, message);
    delay(100);
  }
}
```

CAN Testbench

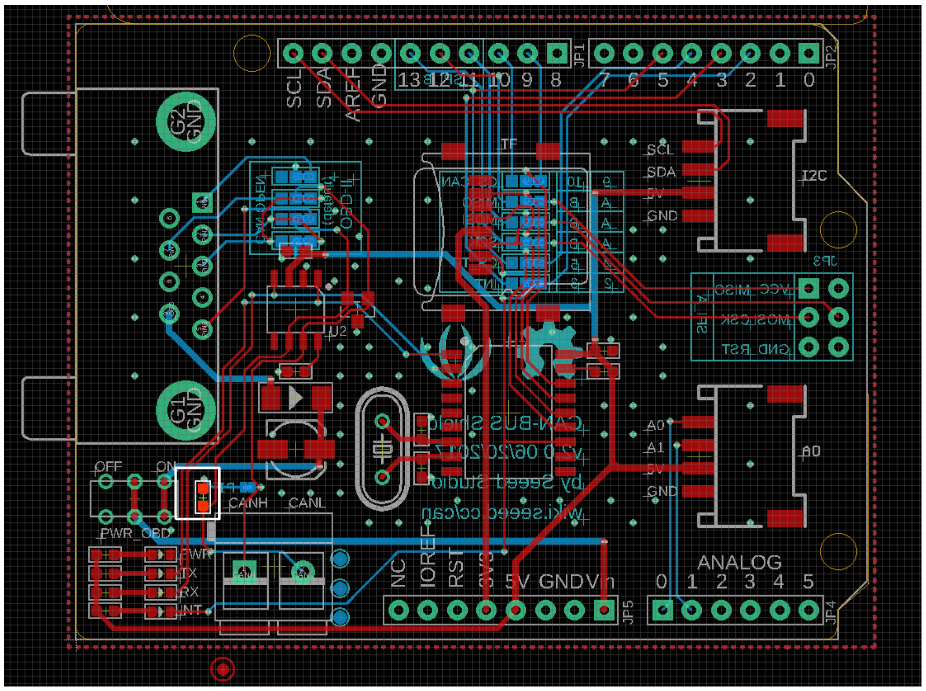
The testbench code utilizes two Arduino Unos with CAN Bus shields to form a two node network. `SENDING` is a boolean value which determines whether the Arduino will read or write data to the bus.

2.2 Testbench Setup

To get the CAN testbench up and running, the following items are required:

1. Two Arduino Uno Boards
2. Two Seeed Studio CAN Bus shields.
   - Both CAN shield version 1.2 and version 2.0 implement the SAE J1939 CAN standard used on the car.
   - The code assumes a version 2.0 CAN shield is in use, if the CAN shield v1.2 is used then `SPI_CS_PIN` definition should be set to 10.
   - The shield versions can be told apart by the SD card slot that is present on v2.0 and absent on v1.2.
3. USB Type B cable for programming.
   - It is recommended that two cables are used such that the serial output from both Arduinos can be viewed concurrently for easier debugging.
4. The Seeed Studio CAN Bus library, which can be installed here.
5. Two jumper wires for the CAN lines connecting the two shields.
   - The instructions to connect the shields are on the Seeed Studio Wiki.
6. Start the test bench by uploading the sender code to one Arduino and the receiver code to the other. The output can then be read from the Serial monitor. `SENDING` shall be set to 1 to obtain sender code and 0 to have receiver code.

The configured testbench serves as a “jumping off point” for other CAN related and testing in a broader network. An important note about building CAN networks using the test bench is to keep track of which nodes have a terminating resistor. By default, both versions of the CAN Bus shield have terminating resistors. Thus, nodes added to the same CAN bus should not have a terminating resistor. In the case that a node with a terminating resistor is being added (Ex. the ECU), then simply remove the terminating resistor from the CAN shield. On shield versions 1.2 and 2.0, R3 is the terminating resistor. The figure below shows the location of this resistor on both versions of the CAN shield. The resistor is marked by a white rectangle.



3 ECU Documentation

Attached below is an excerpt from the ECU datasheet which gives a list of the CAN IDs and the contents of the data section of those messages. This document should always be used as a reference when determining priority of existing messages being sent over the bus and when selecting CAN IDs for new messages.

AN400 Rev C– Application Note
CAN Bus Protocol for PE3 Series ECUs

Release Date 12/20/16

Firmware/Software Version:	PE3 V3.04.01 and higher
Relevant Hardware:	All PE3 controllers with installed CAN Bus
Additional Notes:	This document defines the CAN based parameters that the PE3 is broadcasting for the firmware listed above. The PE3 ECU contains a 120 ohm termination resistor.

CAN Bus Details

- 250 kbps Rate
- Broadcast parameters are based on SAE J1939 standard
- All 2 byte data is stored [LowByte, HighByte]
  *Num = HighByte*256 + LowByte*
- Conversion from 2 bytes to signed int is per the following:

```
  Num = HighByte*256+LowByte
  if (Num>32767) then
     Num = Num - 65536
  endif
```

CAN ID (hex)	Name	Rate (ms)	Start Position	Length	Name	Units	Resolution per bit	Range	Type
0CFFF048	PE1	50	1-2	2 bytes	Rpm	rpm	1	0 to 30000	unsigned int
			3-4	2 bytes	TPS	%	0.1	0 to 100	signed int
			5-6	2 bytes	Fuel Open Time	ms	0.1	0 to 30	signed int
			7-8	2 bytes	Ignition Angle	deg	0.1	-20 to 100	signed int
0CFFF148	PE2	50	1-2	2 bytes	Barometer	psi or kpa	0.01	0-300	signed int
			3-4	2 bytes	MAP	psi or kpa	0.01	0-300	signed int
			5-6	2 bytes	Lambda	lambda	0.01	0-10	signed int
			7.1	1 bit	Pressure Type			0 - psi, 1-kPa	unsigned char
0CFFF248	PE3	100	1-2	2 bytes	Analog Input #1	volts	0.001	0 to 5	signed int
			3-4	2 bytes	Analog Input #2	volts	0.001	0 to 5	signed int
			5-6	2 bytes	Analog Input #3	volts	0.001	0 to 5	signed int
			7-8	2 bytes	Analog Input #4	volts	0.001	0 to 5	signed int
0CFFF348	PE4	100	1-2	2 bytes	Analog Input #5	volts	0.001	0 to 5	signed int
			3-4	2 bytes	Analog Input #6	volts	0.001	0 to 5	signed int
			5-6	2 bytes	Analog Input #7	volts	0.001	0 to 5	signed int
			7-8	2 bytes	Analog Input #8	volts	0.001	0 to 22	signed int
0CFFF448	PE5	100	1-2	2 bytes	Frequency 1	hz	0.2	0 to 6000	signed int
			3-4	2 bytes	Frequency 2	hz	0.2	0 to 6000	signed int
			5-6	2 bytes	Frequency 3	hz	0.2	0 to 6000	signed int
			7-8	2 bytes	Frequency 4	hz	0.2	0 to 6000	signed int
0CFFF548	PE6	1000	1-2	2 bytes	Battery Volt	volts	0.01	0 to 22	signed int
			3-4	2 bytes	Air Temp	C or F	0.1	-1000 to 1000	signed int
			5-6	2 bytes	Coolant Temp	C or F	0.1	-1000 to 1000	signed int
			7.1	1 bit	Temp Type			0 - F, 1 - C	unsigned char
0CFFF648	PE7	1000	1-2	2 bytes	Analog Input #5 - Thermistor	C or F	0.1	-1000 to 1000	signed int
			3-4	2 bytes	Analog Input #7 - Thermistor	C or F	0.1	-1000 to 1000	signed int
			5	1 byte	Version Major		1	0-255	unsigned char
			6	1 byte	Version Minor		1	0-255	unsigned char
			7	1 byte	Version Build		1	0-255	unsigned char
			8	1 byte	TBD				

CAN ID (hex)	Name	Rate (ms)	Start Position	Length	Name	Units	Resolution per bit	Range	Type
0CFFF748	PE8	100	1-2	2 bytes	RPM Rate	rpm/sec	1	-10,000 to 10,000	signed int
			3-4	2 bytes	TPS Rate	%/sec	1	-3,000 to 3,000	signed int
			5-6	2 bytes	MAP Rate	psi/sec or kpa/sec	1	-3,000 to 3,000	signed int
			7-8	2 bytes	MAF Load Rate	g/rev/sec	0.1	-300 to 300	signed int
0CFFF848	PE9	100	1-2	2 bytes	Lambda #1 Measured	lambda	0.01	0 to 10	signed int
			3-4	2 bytes	Lambda #2 Measured	lambda	0.01	0 to 10	signed int
			5-6	2 bytes	Target Lambda	lambda	0.01	0 to 2.5	signed int
0CFFF948	PE10	100	1	1 byte	PWM Duty Cycle #1	%	0.5	0 to 100	unsigned char
			2	1 byte	PWM Duty Cycle #2	%	0.5	0 to 100	unsigned char
			3	1 byte	PWM Duty Cycle #3	%	0.5	0 to 100	unsigned char
			4	1 byte	PWM Duty Cycle #4	%	0.5	0 to 100	unsigned char
			5	1 byte	PWM Duty Cycle #5	%	0.5	0 to 100	unsigned char
			6	1 byte	PWM Duty Cycle #6	%	0.5	0 to 100	unsigned char
			7	1 byte	PWM Duty Cycle #7	%	0.5	0 to 100	unsigned char
			8	1 byte	PWM Duty Cycle #8	%	0.5	0 to 100	unsigned char
0CFFFA48	PE11	100	1-2	2 bytes	Percent Slip	%	0.1	-3000 to 3000	signed int
			3-4	2 bytes	Driven Wheel Rate of Change	ft/sec/sec	0.1	-3000 to 3000	signed int
			5-6	2 bytes	Desired Value	%	0.1	-3000 to 3000	signed int
0CFFFB48	PE12	100	1-2	2 bytes	Driven Avg Wheel Speed	ft/sec	0.1	0 to 3000	unsigned int
			3-4	2 bytes	Non-Driven Avg Wheel Speed	ft/sec	0.1	0 to 3000	unsigned int
			5-6	2 bytes	Ignition Compensation	deg	0.1	0 to 100	signed int
			7-8	2 bytes	Ignition Cut Percent	%	0.1	0 to 100	signed int
0CFFFC48	PE13	100	1-2	2 bytes	Driven Wheel Speed #1	ft/sec	0.1	0 to 3000	unsigned int
			3-4	2 bytes	Driven Wheel Speed #2	ft/sec	0.1	0 to 3000	unsigned int
			5-6	2 bytes	Non-Driven Wheel Speed #1	ft/sec	0.1	0 to 3000	unsigned int
			7-8	2 bytes	Non-Driven Wheel Speed #2	ft/sec	0.1	0 to 3000	unsigned int
0CFFFD48	PE14	100	1-2	2 bytes	Fuel Comp - Accel	%	0.1	0 to 500	signed int
			3-4	2 bytes	Fuel Comp - Starting	%	0.1	0 to 500	signed int
			5-6	2 bytes	Fuel Comp - Air Temp	%	0.1	0 to 500	signed int
			7-8	2 bytes	Fuel Comp - Coolant Temp	%	0.1	0 to 500	signed int
0CFFFE48	PE15	100	1-2	2 bytes	Fuel Comp - Barometer	%	0.1	0 to 500	signed int
			3-4	2 bytes	Fuel Comp - MAP	%	0.1	0 to 500	signed int
			5-6	2 bytes	-				
			7-8	2 bytes	-				
0CFFD048	PE16	100	1-2	2 bytes	Ignition Comp - Air Temp	deg	0.1	-20 to 20	signed int
			3-4	2 bytes	Ignition Comp - Coolant Temp	deg	0.1	-20 to 20	signed int
			5-6	2 bytes	Ignition Comp - Barometer	deg	0.1	-20 to 20	signed int
			7-8	2 bytes	Ignition Comp - MAP	deg	0.1	-20 to 20	signed int

3.1 ECU Arduino Library

QFSAE is developing a library to read these ECU message built atop the CAN shield library. Currently, the library can read all the messages on page 1 of the datasheet provided above. The library exposes an `ECU` class which has global variables containing each of the values transmitted by the ECU in CAN message IDs PE1 - PE7. The values are kept up to date with the latest ECU message transmission by passing the data in each CAN message from the ECU into the object's `update()` method. The class definition for the ECU object is given below followed by the code for an ECU testbench.

```
/*
  ECU Testbench
  This sketch collects all the Data the ECU broadcasts and prints it to serial
  using the library. There is method for performing a formatted print of each
  address listed on the ECU Datasheet and its message contents. This sketch must
  be used in tandem with a running PE3 ECU for testing the running vehicle.

  NOTE: Currently only the PE1 - PE7 addresses are supported by the ECU library
*/

#include "DEFS.h"
#include "ECU.h"
#include "mcp_can.h"

// Use pin 10 for CAN shield version 1.2
#define SPI_CS_PIN 9

MCP_CAN CAN(SPI_CS_PIN);
ECU ECU;

unsigned long id;
unsigned char buf[8];
unsigned char len;

void setup() {
  Serial.begin(115200);
  while (CAN_OK != CAN.begin(CAN_250KBPS)) {
    Serial.println("CAN INIT FAIL");
    Serial.println("TRY AGAIN");
    delay(10000);
  }
  Serial.println("Initialization Success");
}

void loop() {
  if (CAN_MSGAVAIL == CAN.checkReceive()) {
    CAN.readMsgBuf(&len, buf);
    id = CAN.getCanId();
    ECU.update(id, buf, len);
    ECU.debugPrint(id);
  }
  delay(100);
}
```

ECU Testbench

The code assumes that an Arduino and CAN shield are connected to the same CAN bus as the ECU. Using the CAN shield library, messages are received and passed into the `update()` function along with their ID. This updates the appropriate variables with their new values given in the message. The call to `debugPrint()` uses the ID to determine which values received an update from the ECU and prints them to the serial monitor.

3.2 Mock ECU

The Mock ECU library sends out CAN messages with the same IDs as the PE3 ECU. The library exposes a Mock ECU object which takes a structure containing the values to be sent. The following demo sketch sends a set of static ECU values in a loop, but the values could be easily populated dynamically by extending the code.

```
#include "MOCK.h"
#include "mcp_can.h"

#define SPI_CS_PIN 9

MockECU mock;
MCP_CAN CAN(SPI_CS_PIN);
ECUData input;

void setup() {
  Serial.begin(115200);
  while (CAN_OK != CAN.begin(CAN_250KBPS)) {
    Serial.println("CAN INIT FAIL");
    Serial.println("TRY AGAIN");
    delay(10000);
  }
  Serial.println("Initialization Success");
  // Populate ECU Data
  input.rpm = 2000;
  // add more test values as needed
}

// send input data on the correct CAN IDs for the PE3 ECU
void loop() { mock.sendData(input, CAN); }
```

Mock ECU Transmitter

4 CAN Interrupts

There are many cases on the Formula car where CAN messages need to be sent between devices on the car, based on an event such as a gear shift or throttle position threshold. Since CAN is a priority bus where the lower addresses have higher priority, a low address should be used for interrupt based messages since they are only fired once for a certain event taking place and must be able to overtake lower priority messages that will simply be sent again, such as an RPM update. Provided below is some example code which emulates the up and down shift CAN functionality on the car.

```
#include "mcp_can.h"
#define SPI_CS_PIN 9
#define SENDING 0
#define ADDR 0x00

volatile int gear = 1;
volatile bool interrupted = false;

MCP_CAN CAN(SPI_CS_PIN);
unsigned char len = 8;
unsigned char message[8];

void downShift() {
  if (gear > 1) {
    gear--;
  }
  message[0] = gear;
  if (SENDING)
    CAN.sendMsgBuf(ADDR, 0, 8, message);
}

void upShift() {
  if (gear < 6) {
    gear++;
  }
  message[0] = gear;
  if (SENDING)
    CAN.sendMsgBuf(ADDR, 0, 8, message);
}

void setup() {
  Serial.begin(115200);
  while (CAN_OK != CAN.begin(CAN_250KBPS)) {
    Serial.println("CAN INIT FAIL");
    Serial.println("TRY AGAIN");
    delay(10000);
  }
  Serial.println("Initialization Success");
  attachInterrupt(digitalPinToInterrupt(2), downShift, FALLING);
  attachInterrupt(digitalPinToInterrupt(3), upShift, FALLING);
}

void loop() {
  if (!SENDING && CAN_MSGAVAIL == CAN.checkReceive()) {
    CAN.readMsgBuf(&len, message);
    unsigned long id = CAN.getCanId();
    Serial.print("Getting Data from ID: ");
    Serial.println(id, HEX);
    Serial.print("gear = ");
    Serial.println(message[0]);
  }
}
```

CAN Pin Interrupt

The code maps buttons to each of the Arduino Uno's pin change interrupts. One button triggers the `downShift()` method, which decreases the gear and the other calls the `upShift()` method increasing the gear. The two interrupts are configured on Pins 2 and 3, which are the interrupt pins on Arduino Uno. This will need to be changed to the appropriate pin depending on the Arduino being used. The interrupt triggers on the falling edge of the signal from the button, since a pull-up resistor was used in the test circuit. However, this should be changed to a rising edge configuration if a pull-down resistor is used. Similarly to the CAN testbench code, the pin interrupt test code provides a `SENDING` boolean, which lets the user toggle whether the sketch is receiving or sending the CAN messages from the pin interrupts. See subsection 2.2 for in depth instructions for setting up a sender and receiver Arduino.