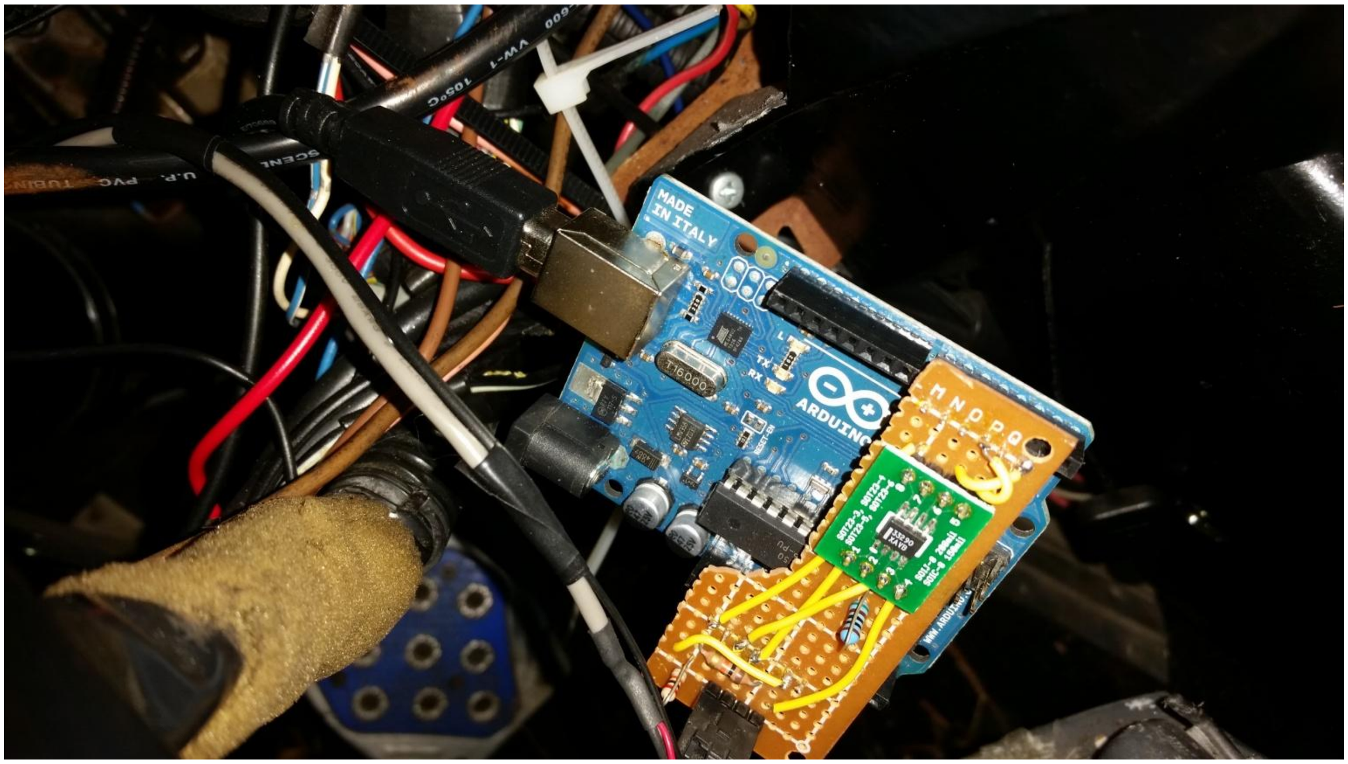
Vehicle On Board Diagnostic (OBD2) Interface

July 2014

Alex Vassallo



1 Abstract

This project was designed to allow a non-OBD2 vehicle to communicate with OBD2 diagnostic equipment using the standard OBD2 connection. The On-Board Diagnostics II (OBD2) vehicle standard specifies the connector, electrical signalling protocol, and the message format. For this application, the signalling protocol implemented was the ISO 9141-2, which uses a single K line to both transmit and receive data. Using this protocol, PID requests are sent by a tester unit, and the vehicle responds with the requested information. I used the MC33290 "ISO k Line serial link interface" chip to manage the half-duplex communications, and an Arduino Uno as the microcontroller to manage the data requests. These components were designed to mount together underneath a vehicles dashboard, and connect to an OBD2 port using a 4 pin connector. OBD2 testing devices which connect to that port receive vehicle information as if they were communicating with an actual vehicle's OBD2 system.

2 Hardware

2.1 System Schematic

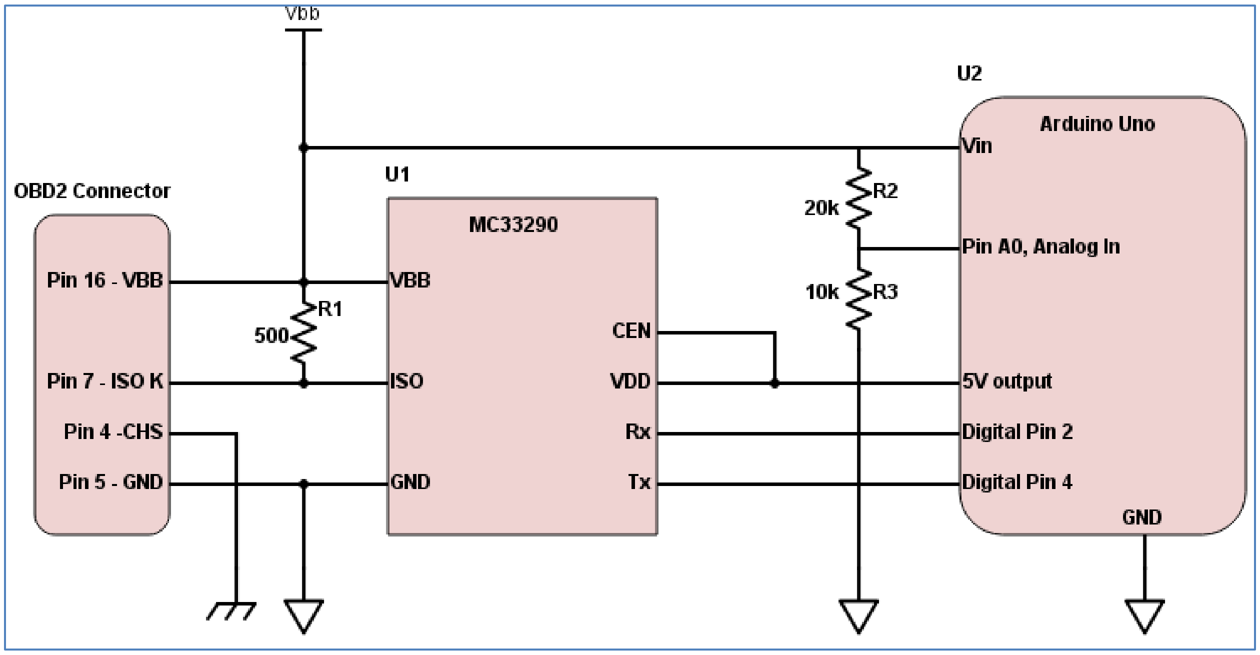
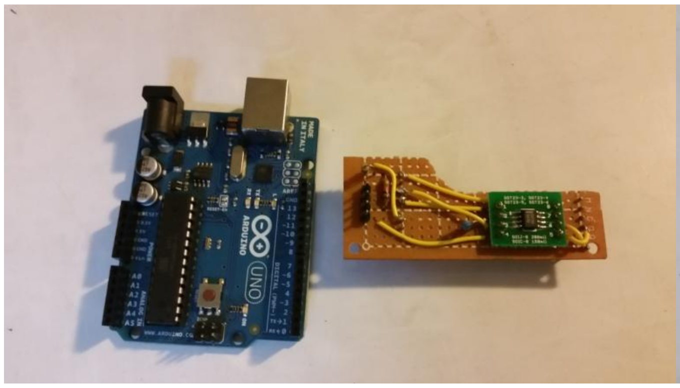


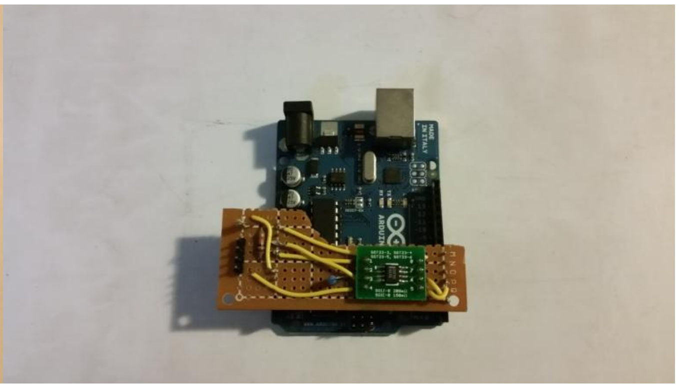
Figure 1: Schematic

2.2 Hardware Design

The hardware was designed to package together into a single unit that can be installed under the dashboard. The MC33290 is a serial link bus interface device designed to provide bi-directional half-duplex communication, specifically for the ISO9141 protocol. It receives power directly from the vehicle battery since the input voltage limit for the Arduino is 20V, and the MC33290 limit is 18V. To monitor the input voltage, a voltage divider scales down the voltage by 3 and feeds an input pin on the Arduino. There is a four pin harness, (not shown), that routes each necessary connection to the appropriate pin on the standard OBD2 connector.



Arduino Uno and Vehicle Interface



Complete Assembly

3 Software

3.1 Handshake

To begin communications, a handshake procedure must first be completed. The tester unit initiates a request by sending 0x33 (00110011) at a bit rate of 5 baud. When this is received by the vehicle, the vehicle waits for a time W1, switches to 10.4k baud, and responds with 0x55 (01010101) followed by two keywords, (typically 0x8 0x8), which describe the communication format. The tester unit then confirms those keywords by responding with the inverted value of the second keyword, followed by the vehicle sending the inverted value of the original address, 0xCC (11001100), to confirm the handshake is complete. PID 01 requests by the tester are then used as "keep alive" packets to avoid having to repeat this handshake.

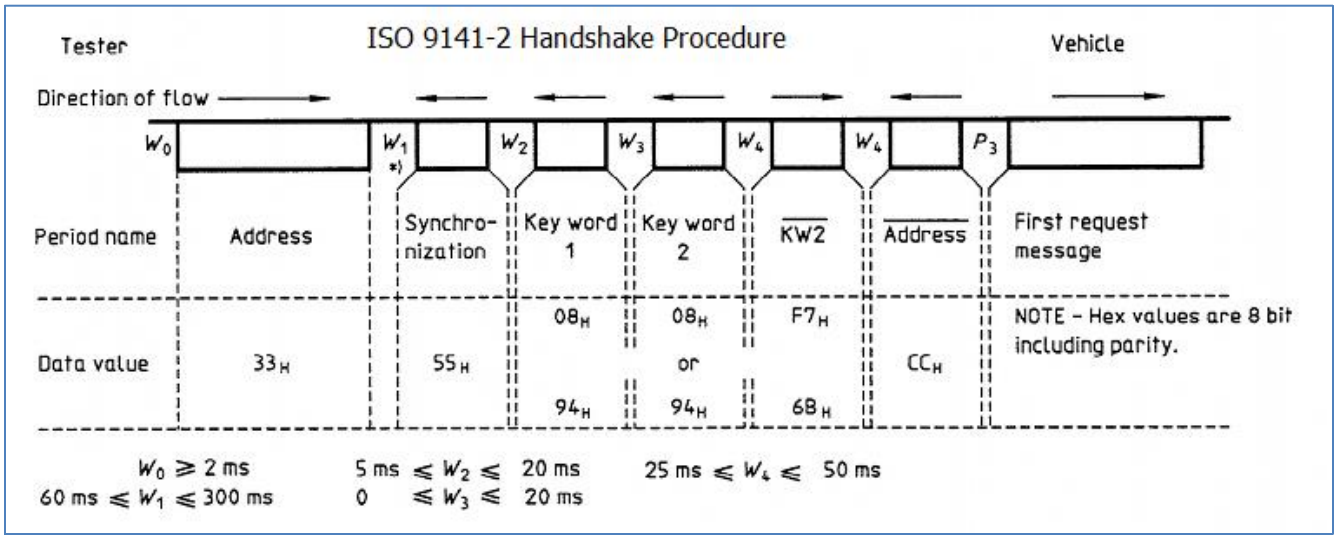


Figure 2: Handshake

3.2 Data Transfer

To transfer data, a tester sends a multi-byte request on the K line, and an ECU responds with a multi-byte reply containing the requested information. As shown in Figure 3, the timing marks are defined by the protocol, and imperative to follow since there is no other form of synchronization. Time P1 and P4 define the delay between bytes, P2 defines the time between packets, and P3 is the request timeout period. As Figure 3 indicates, it is possible for multiple ECU's to each transfer data in turn.

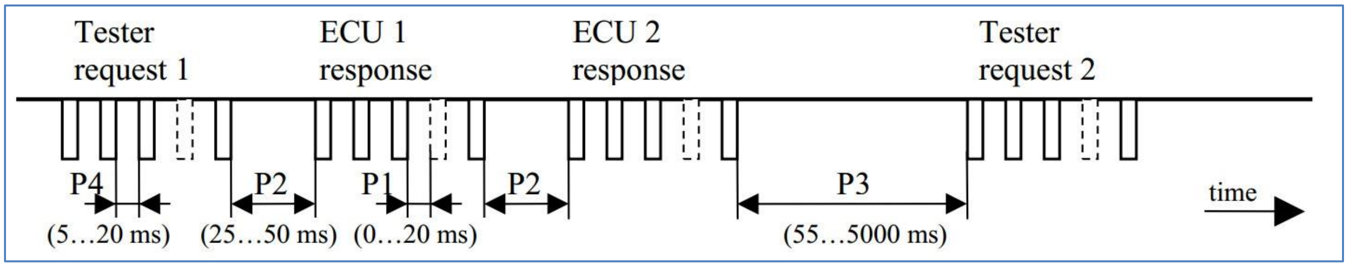


Figure 3: Data Transfer

3.3 Data Format

The data format of each packet is defined by the OBD2 specifications. Table 1 shows the definition of each byte by position. The first two bytes defines whether it is a request or response. The next byte is the source address of the equipment sending the message. The next byte defines which mode the ECU should be in when processing requests. Table 2 shows the various modes of operation. The next byte contains the PID, or Parameter ID, that specifies the exact data being requested. The bytes after the PID is the data section for responses, which must be between 1 and 8 bytes long. Large items such as VIN numbers are broken up into multi-packet messages, and the first byte of the data section contains the packet number. When responding to a request, the CRC checksum is always the last byte. It is simply a summation of the data area truncated to 8 bits.

Byte Definitions	
Byte[0]	0x68 for request, 0x48 for response
Byte[1]	0x6A for request, 0x6B for response
Byte[2]	Source address of sender
Byte[3]	Mode
Byte[4]	PID
Byte[5-12]	Data
Byte[last]	CRC checksum (sum of data)

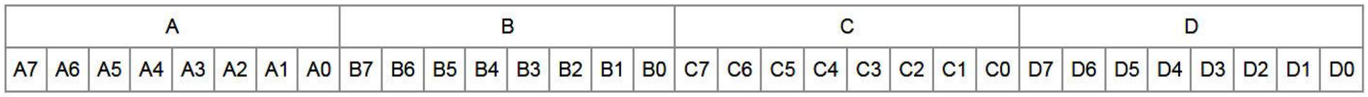
Table 1

Mode Definitions	
1	Show current data
2	Show Freeze frame data
3	Show stored Diagnostic Trouble Codes
4	Clear stored Diagnostic Trouble Codes
5	System Monitor (non-CAN)
6	System Monitor(CAN)
7	Show pending Diagnostic Trouble Codes
8	Control Onboard Systems
9	Request vehicle information
10	Show Permanent Diagnostic Trouble Codes

Table 2

Each mode of operation listed in Table 2 follows a specific format. Mode 3, (Show DTCs), requires 2 bytes of encoded data to define a DTC, and requires sending multiple packets for 3 or more DTCs. Mode 9 is used to verify the vehicle's identity and all responses require a multi-packet message. This includes information such as VIN numbers, serial numbers, or "tamper-proof" checksums. Modes 4 through 8 are not required for normal diagnostics, and are not used in this project.

Most of the normal PID requests are done using Mode 1, (Show Current Data), and require a specifically formatted data response. PIDs that request a sensor value each require a unique mathematical formula to scale and translate the values to be stored inside of 8 or 16 bits. PIDs that request status information will use an encoding map as shown in Figure 4, where bits A7 through D0 are uniquely defined for each specific PID. The data format for Mode 2 is identical, except the data being sent was all captured when first entering Mode 2.



A								B								C								D							
A7	A6	A5	A4	A3	A2	A1	A0	B7	B6	B5	B4	B3	B2	B1	B0	C7	C6	C5	C4	C3	C2	C1	C0	D7	D6	D5	D4	D3	D2	D1	D0

Figure 4: Bit Encoding

3.4 Software Outline

```
edgeTime += bitTime
if (txDataCount > 0) {
   transmitProc()
} else if (handshake > 0) {
   handshakeProc()
} else {
   receiveProc()
   if(data received and time W4 elapsed) {
      processData()}}
```

Figure 5: Main Loop Pseudo-Code

Main Loop()

- Synchronize data stream
- Transmit any queued data, or
- Perform handshake, or
- Listen to Rx signal
- process OBD2 requests

handshakeProc()

switch (handshake)

case 1: 0x33 @ 5 baud detected, wait time W1, switch to 10.4k baud, transmit 0x55
case 2: wait time W2 then transmit 0x8 0x8, prepare to measure time W4
case 3: call receiveProc() with handshake routines enabled, measures time W4
case 4: wait time W4, transmit 0x33 inverted (0xCC) as "ready to communicate"

receiveProc()

- Make sure it is time to sample a bit
- Sample the current bit
- check for 10 consecutive high bits (idle period) to enter idle state
- check for 10 consecutive low bits (handshake request) to prepare for handshake
- check for valid start bit and begin collecting data:
    - ❖ Align and insert bits into byte packet
    - ❖ store byte into local data structure
    - ❖ check for handshake conditions (handshake=3 or 0x33 @ 5 baud)
    - ❖ reset for next byte

transmitProc()

- Make sure it is time to send next bit
- transmit start bit
- transmit 8 bits of data
- transmit stop bit
- wait time W3
- repeat for each byte

processData()

- Verify valid request package was received
- Use Mode and PID values to build proper response packets
- Read sensor values on Arduino analog input pins, or...
- Measure battery voltage on pin A0, engine running if greater than 12.5V
- if engine running, use random numbers to create "dynamic" data(i.e. rpm, o2 voltage, etc)
- compute and append OBD2 CRC checksum byte
- send completed packet to transmit buffer

interrupt routine for Rx pin:

- synchronize data stream (edgeTime = [Time Now])