

OBD-II EMULATOR

DESIGN DOCUMENTATION

JONATHAN TOUSLEY

AARON KUNZ

DECEMBER 2015

Table of Contents

List of Tables and Figures.....	3
Introduction	4
Scope.....	4
Design Overview	4
Requirements.....	4
Dependencies.....	4
Theory of Operation.....	4
Design Alternatives	5
Design Details.....	5
Hardware Design.....	5
Hardware Selection.....	5
OBD-II Implemetation	5
Software Design	7
Testing.....	9
Test 1: Logic Analyzer.....	9
Test 2: Scan Tool	10
Conclusion.....	12
Scope.....	12
Optimality	12
Disclaimer.....	12
Appendix A: Hardware Schematic	13
Appendix B: Source Code.....	13
main.c.....	13
GPIO.h	17

List of Tables and Figures

Table 1: DIP Switch Mapping	6
Figure 1: OBD-II Pinout	6
Figure 2: Flowchart	8
Figure 3: Acknowledged Frame from Scan Tool	9
Figure 4: Acknowledged Frame from Microcontroller	10
Figure 5: County I/M Results Document.....	11
Figure 6: Scan Tool Menu	11
Figure 7: Scan Tool Emissions Results, pages 1 and 2.....	11

Introduction

This document discusses a product known as the OBD-II Emulator. The purpose of this device is to emulate the on-board diagnostics hardware on post-1996, CAN enabled vehicles. The OBD-II Emulator is able to communicate with emissions testing equipment in such a way that the emulator will appear to the emissions testing equipment as if it is the vehicle's on-board diagnostics hardware.

Scope

This document discusses the electrical and software makeup of the OBD-II Emulator. It describes how the OBD-II protocol is implemented. It does not, however, indicate how the hardware components will be packaged and supported in production. Also, this document will not cover the details of discreet installation, as this will vary from one vehicle to the next.

Design Overview

Requirements

The following is a detailed list of the system requirements:

1. The product shall be compliant with the OBD-II protocol.
2. The product shall be able to communicate over the Controller Area Network (CAN).
3. The product shall be implemented in such a way as to be indistinguishable from an actual vehicle, as viewed from the standpoint of a basic OBD-II automotive scan tool.
4. When emissions information is requested, product shall provide enough information to result in a conclusive, passing emissions test.
5. The user shall be able to select which emissions tests are 'supported.'

Dependencies

The design includes voltage regulators that will provide the individual hardware components with their proper voltage requirements, given that the design as a whole is provided with a 12 volt power source. The product is designed to interface with an automotive scan tool, thus a scan tool may be considered a dependency.

Theory of Operation

OBD-II is the automotive emissions communication standard required by the Environmental Protection Agency (EPA) on all post-1996 vehicles. There are three communication protocols over which OBD-II information may be transmitted: J1850, J-2234, and ISO 9141-2. On post-2008 vehicles, the EPA requires that CAN (J-2234) be the protocol over which the OBD-II information must be transmitted.

With these requirements in mind, this product transmits OBD-II emissions data over CAN. It includes an automotive Data Link Connector (DLC) and is designed to be discreetly installed in a vehicle in such a way that when a technician attaches emissions testing equipment to what he thinks is the vehicle's embedded system, he is really connecting to the installed OBD-II Emulator. In this way, if a vehicle cannot pass emissions on its own, the OBD-II Emulator can be installed to allow the vehicle to pass the emissions testing in spite of whatever emissions related malfunctions the vehicle has.

OBD-II is a query-response based system. In effect, during emissions testing, the scan tool asks the vehicle for specific information regarding which emissions tests are supported, and which have been

completed. The OBD-II Emulator is designed to respond to these emissions queries with data indicating that all supported tests have passed.

Design Alternatives

Another way this could have been implemented would have been to create a device to be installed between the vehicle's CAN and its Data Link Connector (DLC). In this way, the microcontroller could forward messages back and forth with or without alterations. This implementation would make the product more difficult to detect, since all the vehicle's CAN messages (such as live vehicle data, freeze frames, and manufacturer specific messages) would be available to the technician managing the testing equipment. In order for this approach to have worked, the product would have had to filter all the vehicle's CAN messages and decide which messages to alter before sending them to the scan tool.

This approach was not chosen because of its high level of complexity. A given vehicle sends thousands of messages in only a few seconds. Each of these messages would have to be filtered, and possibly altered. In the selected design, the product need only generate pre-determined responses standard OBD-II queries, and the CAN system of the vehicle remains untouched.

Design Details

Hardware Design

See Appendix A for the complete Hardware Schematic.

Hardware Selection

The Texas Instruments Tiva TM4C123GH6PM was selected as the appropriate microcontroller choice. It is a common, well documented microcontroller, and has an integrated CAN module, sufficient GPIO pins, and a high enough processing speed.

A CAN transceiver is necessary to integrate the microcontroller on the CAN bus. The microcontroller uses CAN Tx and Rx (send and receive), whereas automotive CAN busses use CAN High and Low. The transceiver is the interface between Tx/Rx, and High/Low. In effect, it is a voltage leveler.

Since the product is designed for automotive applications, its only power dependency should be 12VDC. The microcontroller requires 5VDC, and the CAN transceiver requires 3.3VDC. Thus, voltage regulators are included in the hardware design. In the case of the prototype, the 3.3VDC regulator is integrated on the TivaC Launchpad development board. The 5VDC regulator, however, is external hardware.

OBD-II Implementation

The user interface in automotive OBD-II systems is through the automotive scan tool. Similarly, the bulk of the user interface with the OBD-II Emulator will also be through the automotive scan tool. However, DIP switches are included to provide the user with setup options. When a vehicle owner takes his vehicle to state emissions testing, he receives a document from the service station regarding the results of the emissions test. The document lists twelve system monitors as either "READY," "NOT SUPPORTED," or "INCOMPLETE." Not all vehicles support all system monitors, therefore the design includes twelve DIP switches in order to allow the user to select which systems monitors the OBD-II Emulator will support. The Emulator then automatically lists these monitors as "READY" when queried by a scan tool.

For example, the 1996 2.2L Honda Accord does not support Evaporative System, Secondary Air System, A/C Refrigerant, or Heated Catalyst. So, when the owner of a 1996 2.2L Honda Accord intends to use this product to emulate his vehicle, he should shut DIP switches 1, 2, 3, 4, 5, 6, 10, and 11 and open DIP switches 7, 8, 9, and 12.

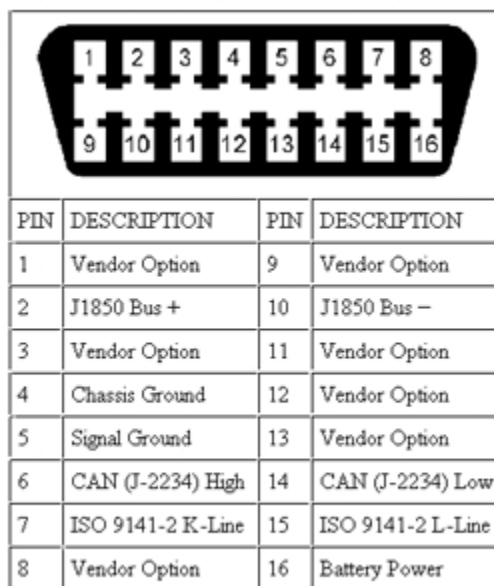
The table below illustrates the relationship between DIP switches, System Monitors, and GPIO Pins. Note that the Hardware Schematic (see Appendix A) routes these switches to the proper GPIO pins.

DIP Switch	System Monitor	GPIO Pin
1	Oxygen Sensor Heater	PA2
2	EGR System	PA3
3	Misfire Monitor	PA4
4	Fuel System Monitor	PA5
5	Comprehensive Component	PA6
6	Malfunction Indicator Lamp	PA7
7	Evaporative System	PC4
8	Secondary Air System	PC5
9	A/C Refrigerant	PC6
10	Oxygen Sensor	PC7
11	Catalyst	PE0
12	Heated Catalyst	PE1

Table 1: DIP Switch Mapping

Note that the Hardware Schematic (see Appendix A) routes these switches to the proper GPIO pins according to this table

The OBD-II standard includes a mandatory connector pinout. The figure below (as if looking at the vehicle's DLC) illustrates this pinout.



OBD-II Connector and Pinout

Figure 1: OBD-II Pinout

Note the location and pin numbers of CAN High (6), CAN Low (14), Signal Ground (5), Chassis Ground (4), and Battery Power (16). These pins are important in the hardware design of the product since it uses the CAN protocol. All the other pins are left unconnected. In the Hardware Schematic (see Appendix A), these pin numbers are listed on the DLC figure.

Software Design

Note: See Appendix B for the complete Source Code.

Each of the twelve possible modules that the microcontroller is capable of spoofing connects to a GPIO port which the microcontroller reads during initialization. Upon reading the inputs, the microcontroller stores the information in global variables that are used when the scan tool requests the specific information.

After initialization, the microcontroller simply watches the CAN bus for message objects sent on the broadcast address (0x7DF). When a matching message object occurs, the microcontroller reads the PID of the message, and if the PID matches any of the PIDs the microcontroller is programmed to respond to, it sends an appropriate message object in reply, assuming the role of the car's on-board computer (PCM) by sending the message object with an address of 0x7E8.

The figure below illustrates the described logic:

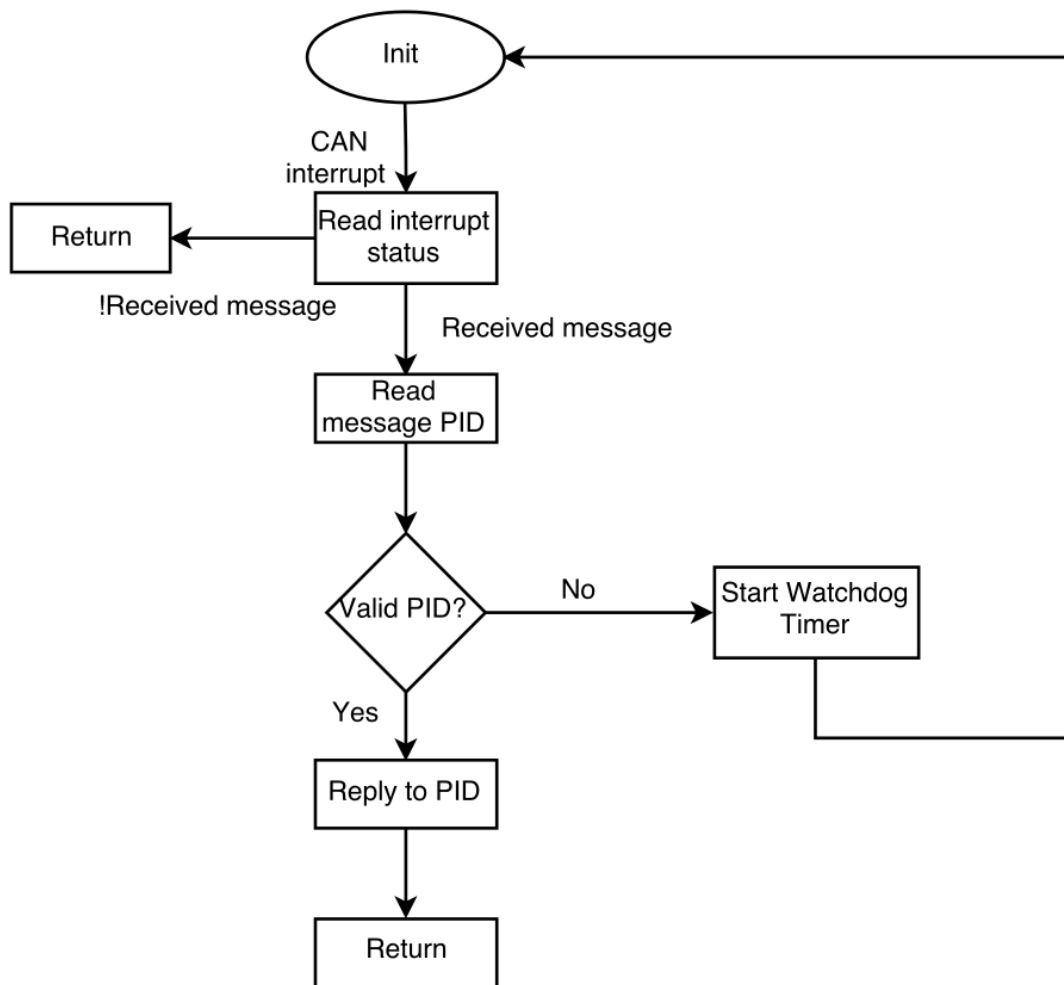


Figure 2: Flowchart

If the received PID does not match any the microcontroller is programmed to respond to, an error is assumed and a watch-dog timer is started. Since there is no handler, this leads to a software reset. This design removes the user even further from interfacing with the microcontroller as there is now no need to manually press the reset button, should something go wrong in communication. Furthermore, there is no penalty for this because:

1. The initialization process takes a very small amount of time.
2. The microcontroller needs to know nothing of consecutive messages, it simply responds to requests as the scan tool needs them.
3. The scan tool will continue to send requests until its queries are answered. So as long as the problem is not recurring, it will receive its answer.

Testing

Test 1: Logic Analyzer

Requirements 1 and 2 indicate that the product must be OBD-II compliant and operate over CAN. Figures 3 and 4 show verification that the OBD-II Emulator communicates properly using the OBD-II protocol via the Controller Area Network (CAN). These logic analyzer screenshots show the scan tool's query and the microcontroller's response using the standard CAN addresses. Thus requirements 1 and 2 have been verified.

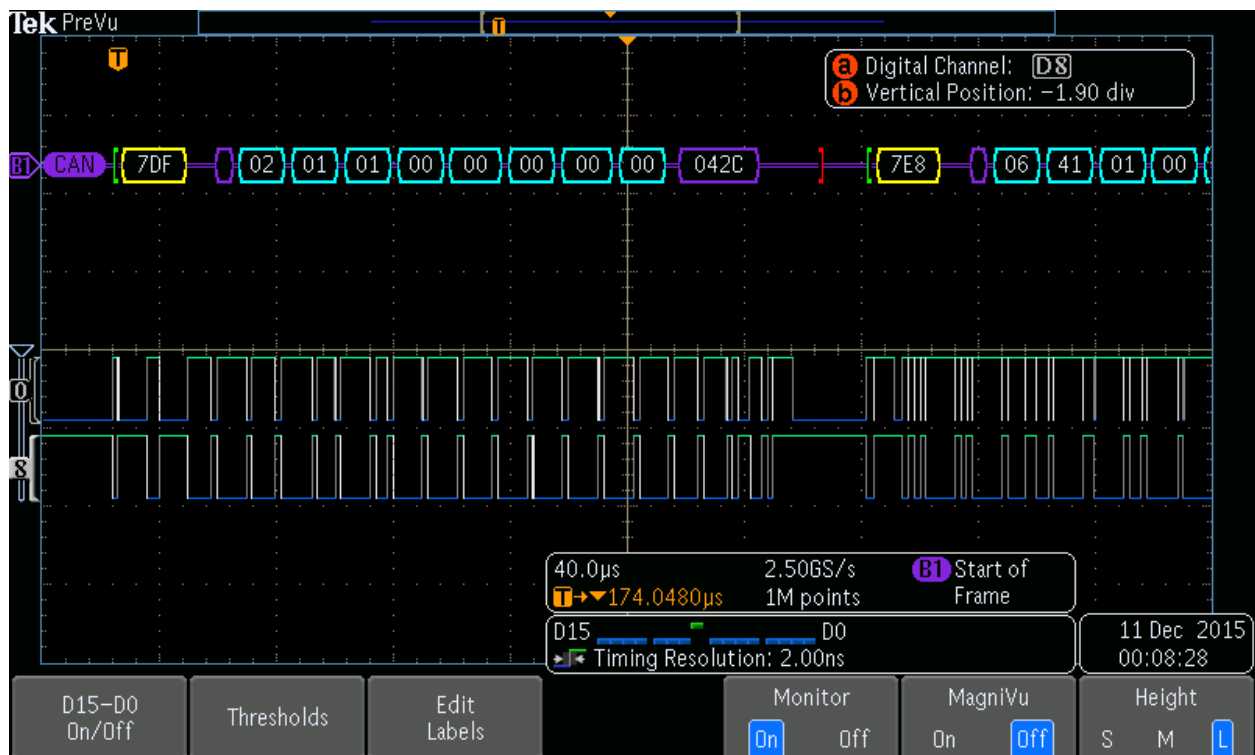


Figure 3: Acknowledged Frame from Scan Tool

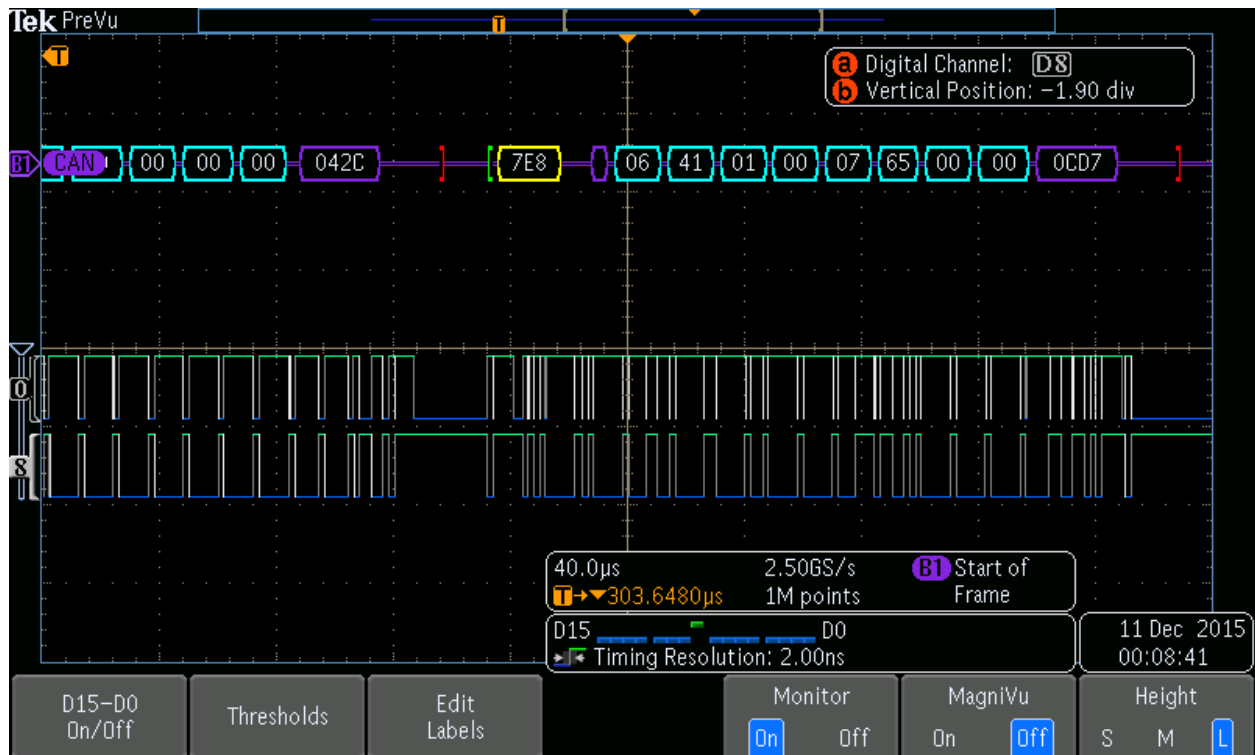


Figure 4: Acknowledged Frame from Microcontroller

Test 2: Scan Tool

Design requirements 3 and 4 indicate that the emulator must be indistinguishable from the actual vehicle, as viewed by the scan tool, and result in a passing emissions test. Going back to the example of the 1996 Honda Accord, figure 5 shows the official passing emissions test results for such a vehicle, performed in the usual way. Figure 5 represents the readiness monitors of the actual vehicle as viewed by the emissions testing equipment. Figures 6 and 7 show the readiness statuses of the OBD-II emulator. Note that all the same monitors are either both supported and ready or not supported. In other words, the results are the same. So, as viewed by the scan tool, there is no difference between the vehicle and the emulator as far as the I/M monitors are concerned. Also note that the OBD-II emulator indicates a passing emissions status. Thus requirements 3 and 4 have been verified.

Design requirement 5 indicates the user should be able to select which of the monitors are supported for the current test. In accordance with the document shown in figure 5, we chose to not support:

1. Heated Catalyst
2. Evaporative System
3. Secondary Air System
4. A/C Refrigerant

These monitors correspond to DIP switches 7, 8, 9, and 12, respectively. Therefore, those DIP switches were left open during this test, and the proper results are seen in figure 7. Thus, requirement 5 has been verified.

OBD TEST INFORMATION				
SYSTEM MONITORS				DIAGNOSTIC TROUBLE CODES (DTC)
ENGINE MISFIRE:	READY	OXYGEN SENSOR:	READY	DTCs: NONE
FUEL SYSTEM:	READY	SECONDARY AIR SYSTEM:	NOT SUPPORTED	
COMPREHENSIVE COMP:	READY	A/C REFRIGERANT:	NOT SUPPORTED	
CATALYTIC CONVERTER:	READY	EGR FLOW:	READY	
EVAPORATIVE SYSTEM:	NOT SUPPORTED	HEATED CATALYST:	NOT SUPPORTED	
HEATED O2 SENSOR:	READY	KOEO:	PASS	
MIL STATUS:	OFF	KOER:	PASS	

Your vehicle passed the I/M test. Thank you! If you have questions regarding the I/M program call (385) 468-3837. For current air pollution information call (385) 468-3837 in Salt Lake County. Your efforts to control air pollution help and are appreciated. If you have questions regarding your specific I/M inspection, please contact your Inspection Station listed below:

Figure 5: County I/M Results Document

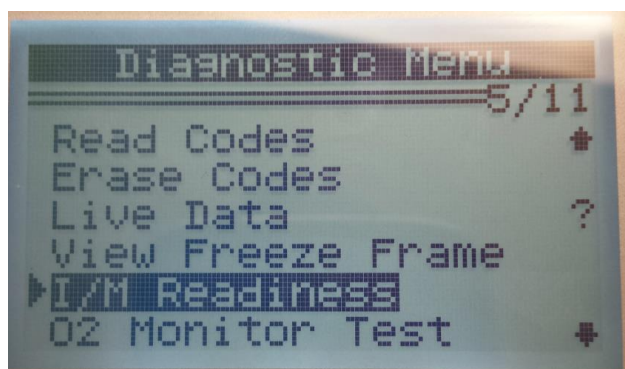


Figure 6: Scan tool menu

Since DTCs Cleared		Since DTCs Cleared	
MIL Status	OFF	Evap System Mon	N/A *
Misfire Monitor	OK	Sec Air System	N/A
Fuel System Mon	OK	A/C Refrig Mon	N/A
COMP. Component	OK	Oxygen Sens Mon	OK
Catalyst Mon	OK	Oxygen Sens Htr	OK
Htd Catalyst	N/A *	EGR System	OK

Figure 7: Scan tool emissions results, pages 1 and 2

Conclusion

Based on the testing procedures, the functionality and requirements of the OBD-II emulator have been met and verified.

Scope

There are many additional features which this product is capable of supporting, which were not implemented due to time constraints, budget, and the scope of this class. These include: live vehicle data (e.g. RPMs), freeze frames, Diagnostic Trouble Codes (DTCs), etc.

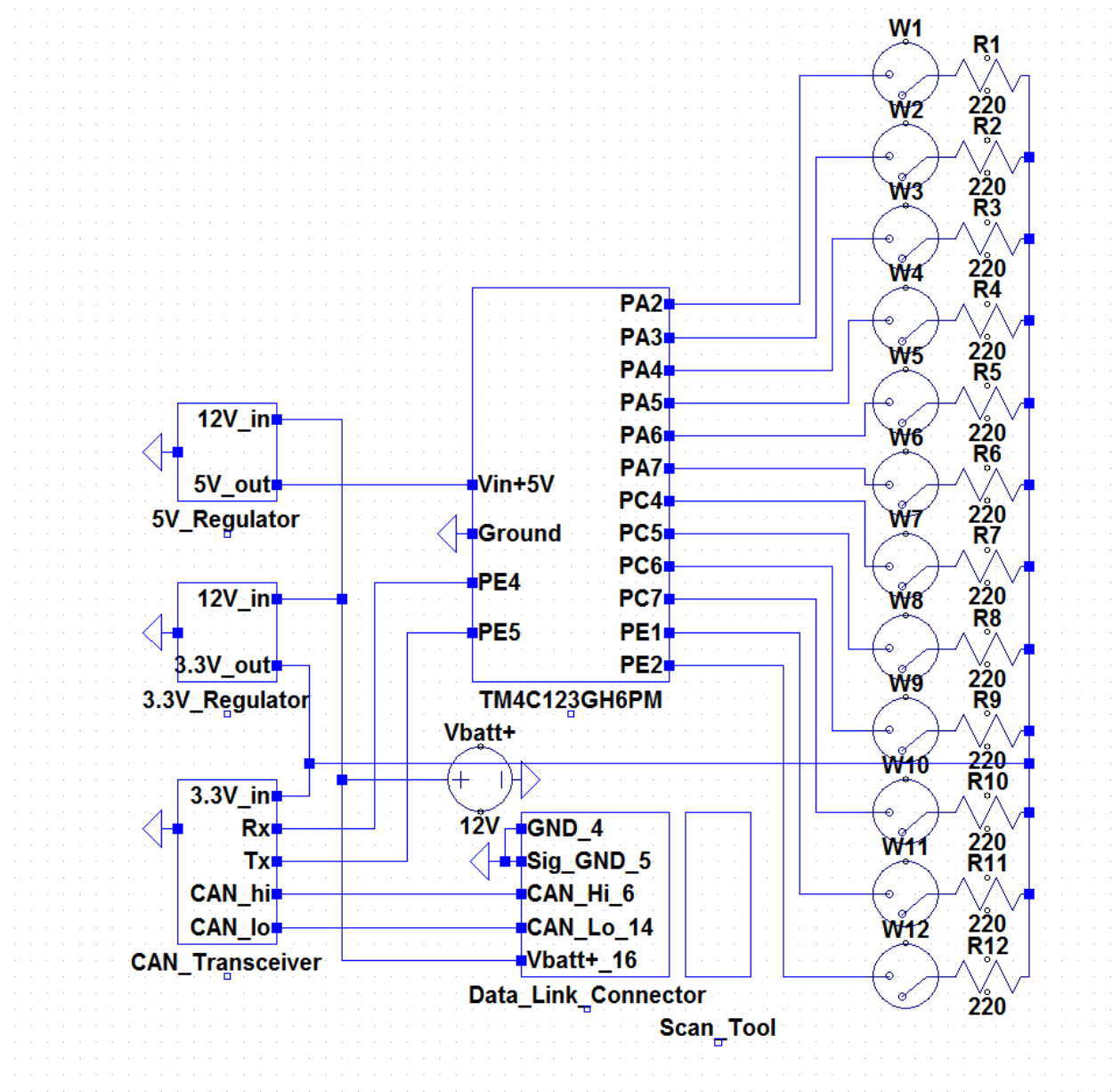
Optimality

From a power consumption standpoint, the design could be more optimal. For instance, the microcontroller could be put into deep sleep mode, and wake up only when CAN communication is required. Also, switching voltage regulators instead of linear voltage regulators would reduce idle power consumption. However, these approaches were not chosen because the recommended implementation of this product is only a temporary installation.

Disclaimer

Using this device in lieu of a motorized vehicle during an official emissions test constitutes emissions testing fraudulency, a criminal offense in the United States. The designers of this product in no way endorse this course of action and are not responsible for any illegal use of this product.

Appendix A: Hardware Schematic



Appendix B: Source Code

main.c

//main.c

```
#include "GPIO.H"
#include "math.h"
```

```
volatile unsigned char *SYSTCTL = (unsigned char *) 0x400FE000;
volatile unsigned char *CORE    = (unsigned char *) 0xE000E000;
volatile unsigned char *TM0     = (unsigned char *) 0x40030000;
volatile unsigned char *TM1     = (unsigned char *) 0x40031000;
```

```

volatile unsigned char *TM2 = (unsigned char *) 0x40032000;
volatile unsigned int *CANMODULE = (unsigned int *) 0x40040000;
volatile CAN *CAN0 = (CAN *) 0x40040000;
volatile unsigned char *WDT0 = (unsigned char *) 0x40000000;
GPIO *PA = (GPIO *) 0x40004000;
GPIO *PC = (GPIO *) 0x40006000;
GPIO *PE = (GPIO *) 0x40024000;
GPIO *PF = (GPIO *) 0x40025000;

unsigned char ByteA = 0x00, ByteB = 0x07, ByteC = 0x65, ByteD = 0x00;

void enableClock(void);
void enablePortA(void);
void enablePortC(void);
void enablePortE(void);
void enableCAN0(void);
void enableWDT0(void);
void configurePorts(void);

void CAN0_Handler(void)
{
    unsigned short DA2;
    unsigned char PID;
    unsigned char STS;
    unsigned short NEWDAT,temp;
    unsigned char i;
    STS = CAN0->STS.B._0;
    CAN0->STS.B._0 = STS & ~0x18;
    if((STS & 0x10) != 0x10){
        //if it's not RXOK, I don't care
        return;
    }

    NEWDAT = CANMODULE[0x120/4];
    for(i = 0; i < 16; i++){
        NEWDAT = NEWDAT >> 1;
        if(NEWDAT == 0){
            NEWDAT = i + 1;
            break;
        }
    }
    if(NEWDAT == 0x00){
        NEWDAT = CANMODULE[0x124/4];
        for(i = 0; i < 16; i++){
            temp = NEWDAT & (1 << i);
            if(temp != 0){
                NEWDAT = i + 16;
                break;
            }
        }
    }
    CAN0->CRQR.B._0 = NEWDAT;
    CAN0->MCTL.H._0 = 0x0000;
    CAN0->MCTLR.H._0 = 0x0000;

    DA2 = CAN0->DA2R.H._0;
    PID = (DA2 & 0xFF);

    CAN0->ARB2.H._0 = (0x7E8 << 2) | 0x8000 | 0x2000; //transmit
    CAN0->CRQ.B._0 = NEWDAT;
    CAN0->MCTL.H._0 = 0x9088;

    switch(PID){

```

```

        case 0x00:
            //0x 06 41 00 BE 1F A8 13 00
            CAN0->DA1.H._0 = 0x4106;
            CAN0->DA2.H._0 = 0xBE00;
            CAN0->DB1.H._0 = 0xA81F;
            CAN0->DB2.H._0 = 0x0013;
            break;
        case 0x20:
            //0x 06 41 20 90 05 E0 15 00
            CAN0->DA1.H._0 = 0x4106;
            CAN0->DA2.H._0 = 0x9020;
            CAN0->DB1.H._0 = 0xE005;
            CAN0->DB2.H._0 = 0x0015;
            break;
        case 0x40:
            //0x 06 41 40 FE 1C 20 00 00
            CAN0->DA1.H._0 = 0x4106;
            CAN0->DA2.H._0 = 0xFE40;
            CAN0->DB1.H._0 = 0x201C;
            CAN0->DB2.H._0 = 0x0000;
            break;
        case 0x01:
            //0x 06 41 01 00 07 65 00 00
            CAN0->DA1.H._0 = 0x4106;
            CAN0->DA2.H._0 = (ByteA << 8) | 0x01;
            CAN0->DB1.H._0 = (ByteC << 8) | ByteB;
            CAN0->DB2.H._0 = (0x00 << 8) | ByteD;
            break;
        default:
            //should probably reset uc here
            SYSTCTL[0x608] |= 0x20; //PF
            for(i = 0; i < 0xFE; i++);
            PF->DEN.b._1 = 0x1;
            PF->DIR.b._1 = 0x1;
            PF->DATA.b._1 = 0x1;

            //WDT0
            WDT0[0x008] |= 0x3;
            break;
    }
    CAN0->MCTL.H._0 |= 0x0100; //TXRQST
    CAN0->CRQ.B._0 = 0x01;

    CAN0->ARB2R.H._0 = (0x7DF << 2) | 0x8000; //receive
    CAN0->CRQR.B._0 = 0x01;
    CAN0->MCTL.H._0 = 0x1088;
}

int main(void)
{
    configurePorts();

    while(1);
}

void enableWDT0(void)
{
    *(int *)&WDT0[0x0] = 0xFFFF;
}

void enableCAN0(void)
{
    unsigned int i;
    CAN0->CTL.B._0 = 0x45; //INIT, SIE, CCE

```

```

while((CAN0->CRQ.b._15 & 0x1) == 0x1);          //wait for busy bit to clear

//WRNRD, ARB, CONTROL = 1
CAN0->CMSK.B._0 |= 0xB0;
CAN0->ARB2.W = 0x0;
CAN0->MCTL.W = 0x0;
CAN0->BIT.H._0 = 0x2383;
CAN0->CTL.B._0 = (CAN0->CTL.B._0 & ~0x40);          //clear CCE

for(i = 1; i <= 32; i++){
    while((CAN0->CRQ.b._15 & 0x1) == 0x1);          //wait for busy bit to
clear
    CAN0->CRQ.B._0 = i;
    CAN0->CRQR.B._0 = i;
}

CAN0->CMSK.B._0 = 0xF3;
CAN0->MSK1.H._0 = 0x0;
CAN0->MSK2.H._0 = (0x7DF << 2);
CAN0->ARB1.H._0 = 0x0;
CAN0->ARB2.H._0 = (0x7DF << 2) | 0x8000;

CAN0->CMSKR.B._0 = 0x73;
CAN0->MSK1R.H._0 = 0x0000;
CAN0->MSK2R.H._0 = (0x7DF << 2);
CAN0->ARB1R.H._0 = 0x0000;
CAN0->ARB2R.H._0 = (0x7DF << 2) | 0x8000;

for(i = 1; i <= 32; i++){
    while((CAN0->CRQ.b._15 & 0x1) == 0x1);          //wait for busy bit to
clear
    CAN0->CRQ.B._0 = i;
    CAN0->CRQR.B._0 = i;
}

CAN0->CTL.B._0 = (CAN0->CTL.B._0 & ~0x01);

CAN0->CRQ.B._0 = 0x1;
CAN0->CRQR.B._0 = 0x1;

CAN0->MCTL.H._0 |= 0x1088; //UMASK, DLC = 8, EOB
CAN0->MCTLR.H._0 |= 0x1088;

CAN0->DA1R.H._0 = 0x0000;
CAN0->DA2R.H._0 = 0x0000;
CAN0->DB1R.H._0 = 0x0000;
CAN0->DB2R.H._0 = 0x0000;

//interrupts
CAN0->CTL.b._1 = 0x1;          //CTL IE
CAN0->MCTLR.b._10 = 0x1;      //MCTL RXIE
CORE[0x104] |= 0x80;          //enable interrupts CAN0
}

void enableClock(void)
{
    unsigned int i;
    *(int*)&SYSCTL[0x060] = 0x8E3D40;          //sysclk 16MHz
    for(i = 0; i < 0xFF; i++){
        SYSCTL[0x608] |= 0x15;          //enable PE PC PA
        SYSCTL[0x103] |= 0x1;          //legacy enable CAN0
    }
}

```



```

        SYSCTL[0x600] |= 0x1;                //WDT0 clock enable
    for(i = 0; i < 0xFF; i++);
}

void enablePortA(void)
{
    PA->DEN.B._0 |= 0xFC;
    PA->DIR.B._0 &= ~0x03;
    PA->PDR.B._0 |= 0xFC;
}

void enablePortC(void)
{
    PC->DEN.B._0 |= 0xF0;
    PC->DIR.B._0 &= ~0xF0;
    PC->PDR.B._0 |= 0xF0;
}

void enablePortE(void)
{
    //PE4-5 CAN0
    PE->AFSEL.B._0 |= 0x30;
    PE->DEN.B._0 |= 0x3F;
    PE->PCTL.B._2 = 0x88;                //CAN
    PE->DIR.b._5 = 0x1;                 //PE5 output
    PE->DIR.b._4 = 0x0;                 //PE4 input
    PE->PDR.B._0 |= 0x10;               //PDR

    PE->DIR.B._0 &= ~0x0F;
    PE->PDR.B._0 |= 0x0F;
}

void configurePorts(void)
{
    enableClock();
    enablePortA();
    enablePortC();
    enablePortE();
    enableCAN0();
    enableWDT0();

    ByteA = (~PA->DATA.b._7 << 7);
    ByteB = ((PA->DATA.B._0 & 0x70) >> 4);
    ByteC = (((PA->DATA.B._0 & 0xC) << 4) | ((PC->DATA.B._0 & 0xF0) >> 2) | (PE->DATA.B._0 & 0x3));
}

```

GPIO.h

//GPIO.h

```

#ifndef GPIO_H
#define GPIO_H

```

```

typedef struct{
    volatile unsigned int _0:1,_1:1,_2:1,_3:1,_4:1,_5:1,_6:1,_7:1,
        _8:1,_9:1,_10:1,_11:1,_12:1,_13:1,_14:1,_15:1,
        _16:1,_17:1,_18:1,_19:1,_20:1,_21:1,_22:1,_23:1,
        _24:1,_25:1,_26:1,_27:1,_28:1,_29:1,_30:1,_31:1;
}bitReg;

```

```

typedef struct{
    volatile unsigned char _0,_1,_2,_3;
}byteReg;

```

```

typedef struct{
    volatile unsigned short _0,_1;
}halfwordReg;

//access register via bit, byte, halfword, and word
typedef union{
    bitReg b;
    byteReg B;
    halfwordReg H;
    volatile unsigned int W;
}REG;

typedef struct{
    volatile unsigned char DATA_[1019]; //TI bit addresses
    REG DATA; //0x3FC
    REG DIR; //0x400
    REG IS; //0x404
    REG IBE; //0x408
    REG IEV; //0x40C
    REG IM; //0x410
    REG RIS; //0x414
    REG MIS; //0x418
    REG ICR; //0x41C
    REG AFSEL; //0x420
    volatile unsigned char RES[220]; //reserved
    REG DR2R; //0x500
    REG DR4R; //0x504
    REG DR8R; //0x508
    REG ODR; //0x50C
    REG PUR; //0x510
    REG PDR; //0x514
    REG SLR; //0x518
    REG DEN; //0x51C
    REG LOCK; //0x520
    REG CR; //0x524
    REG AMSEL; //0x528
    REG PCTL; //0x52C
}GPIO;

typedef struct{
    REG CTL; //0x000
    REG STS; //0x004
    REG ERR; //0x008
    REG BIT; //0x00C
    REG INT; //0x010
    REG TST; //0x014
    REG BRPE; //0x018
    volatile unsigned char RES1[4];
    REG CRQ ; //0x020
    REG CMSK; //0x024
    REG MSK1; //0x028
    REG MSK2; //0x02C
    REG ARB1; //0x030
    REG ARB2; //0x034
    REG MCTL; //0x038
    REG DA1 ; //0x03C
    REG DA2 ; //0x040
    REG DB1 ; //0x044
    REG DB2 ; //0x048
    volatile unsigned char RES2[0x32];
    REG CRQR ; //0x080
    REG CMSKR; //0x084

```

```
REG MSK1R;    //0x088
REG MSK2R;    //0x08C
REG ARB1R;    //0x090
REG ARB2R;    //0x094
REG MCTLR;    //0x098
REG DA1R ;    //0x09C
REG DA2R ;    //0x0A0
REG DB1R ;    //0x0A4
REG DB2R ;    //0x0A8
}CAN;

#endif //GPIO.h
```