Gasoline Economy Management

G.E.M.

GROUP 8

PEDRO BETANCOURT – CPE

ALEXANDER PATINO – CPE

MOHHAMAD “JAKE” PULLIAM – EE

Overview and Motivation

Many newer cars are being sold with on board fuel efficiency monitoring systems. We wanted to bring this type of information system to a wider range of vehicles at a low cost. When drivers receive instant feedback they're more likely to adjust their habits.

What is GEM?

GEM stands for Gasoline Economy Management. A system to monitor driver activity to help drivers develop more fuel efficient driving habits.

Two components:

- A small device that connects to the vehicles on board diagnostic port
- An Android app that wirelessly connects to the device to monitor and display metrics

Specifications

Low Power Consumption
- Multiple weeks of use without drawing down the vehicle battery

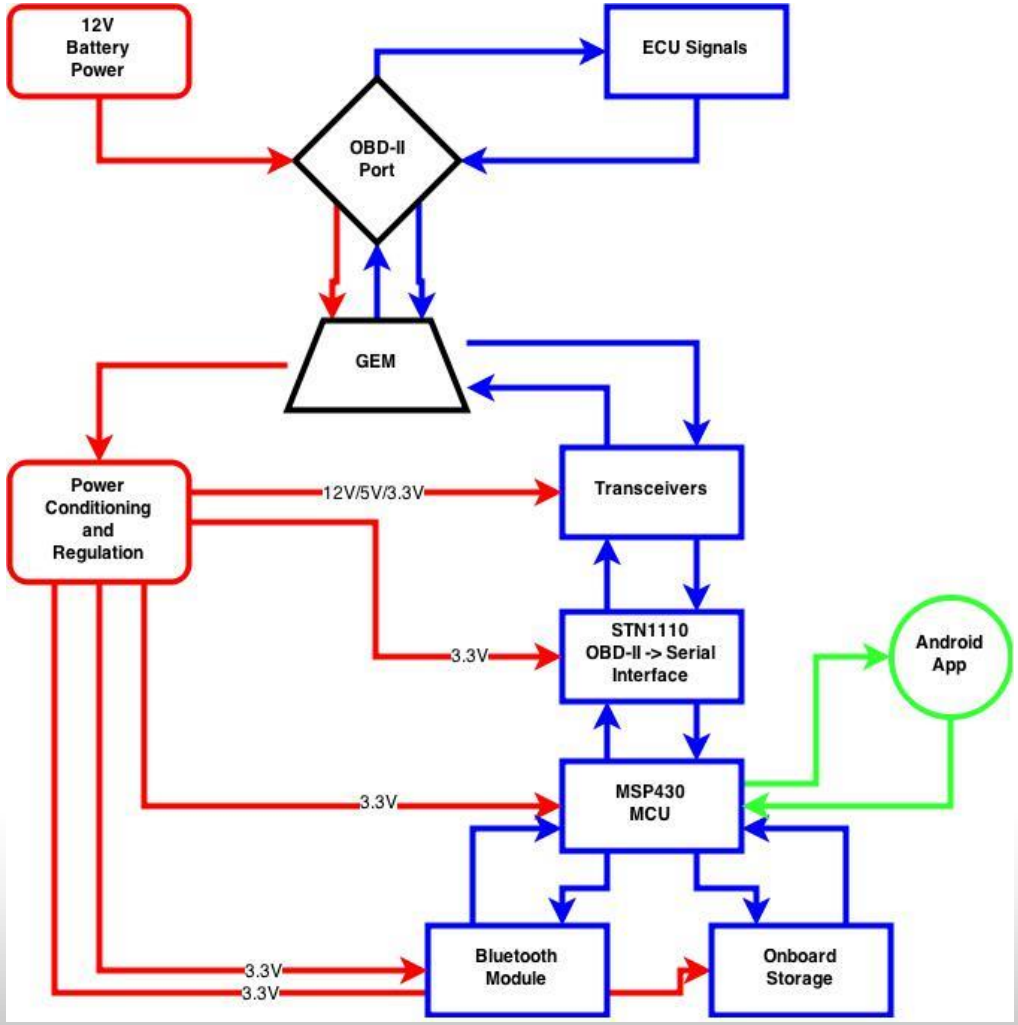
Interface with all vehicles manufactured after 1996 (OBD-II Spec. Mandated)
- SAE J1850 PWM/VPW
- ISO 9141-2
- KWP2000
- CAN

Bluetooth Connection
- BLE compliance

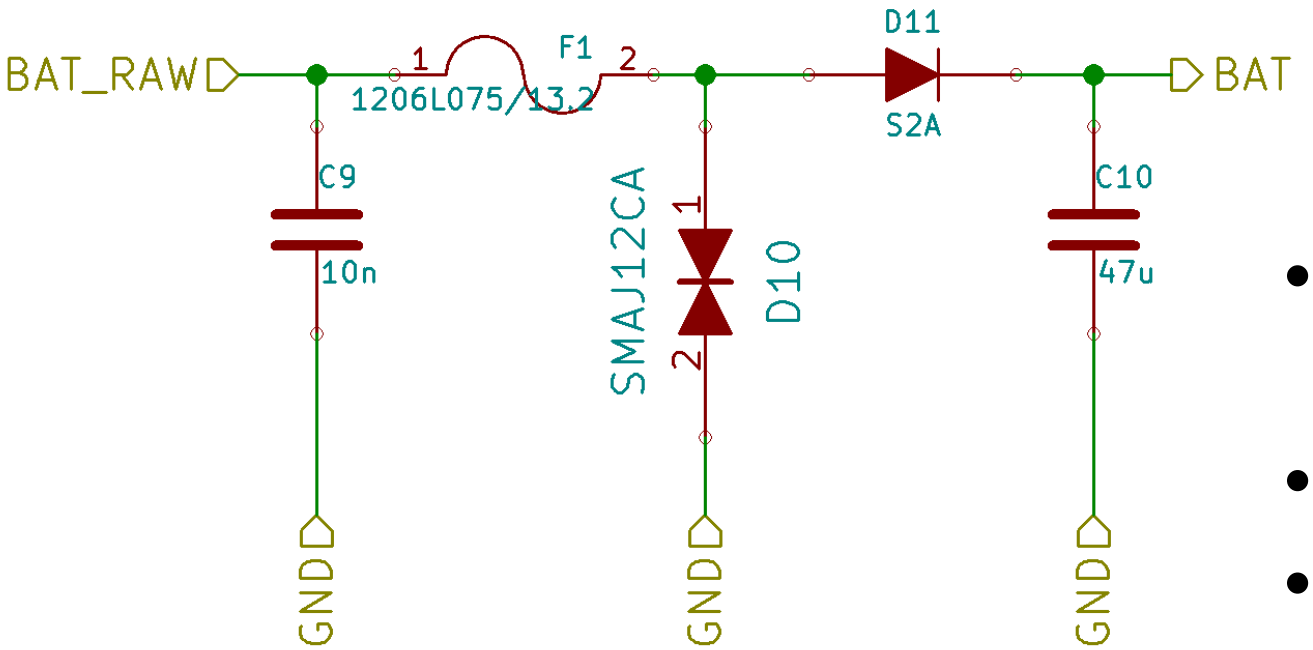
Local Storage
- Minimum 25 MB on-board storage

Portability
- Maximum of 5 vehicle profiles

Block Diagram

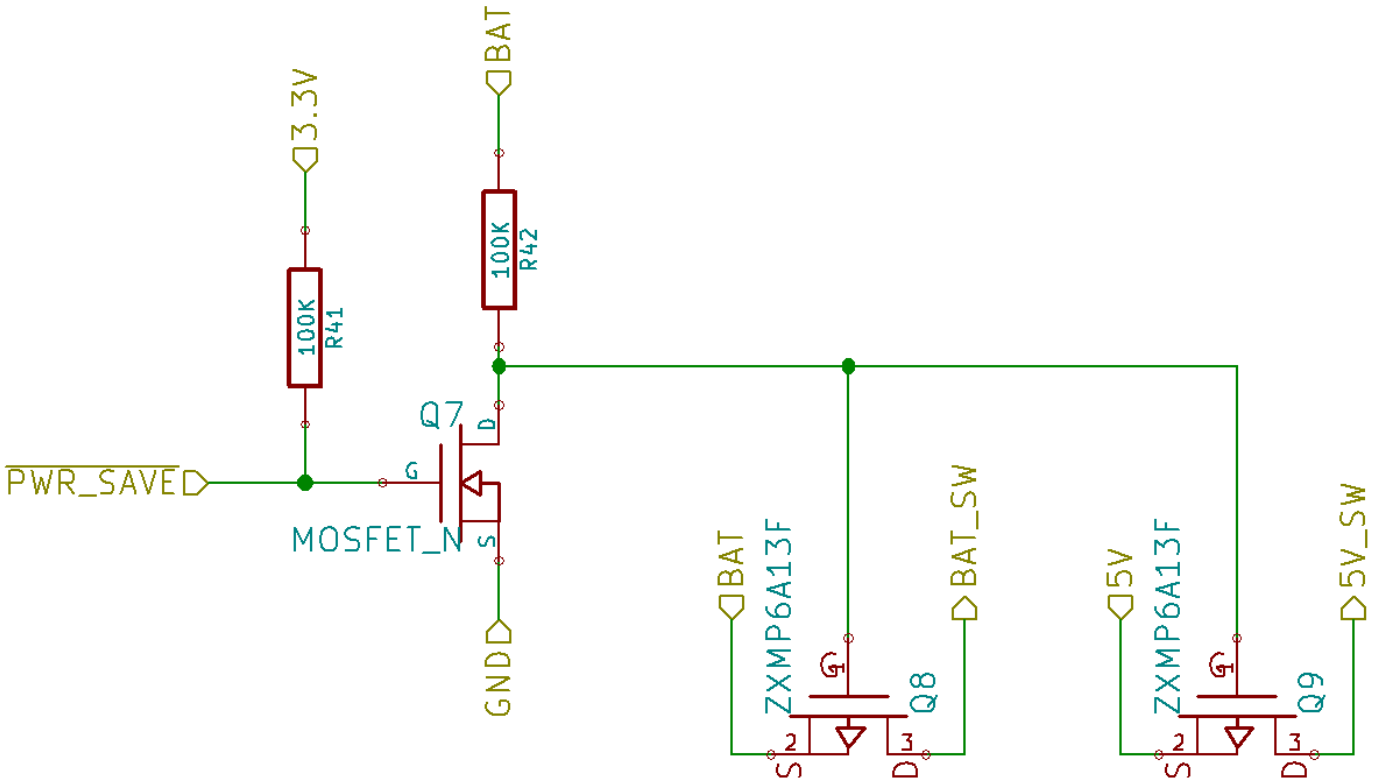


Power Conditioning



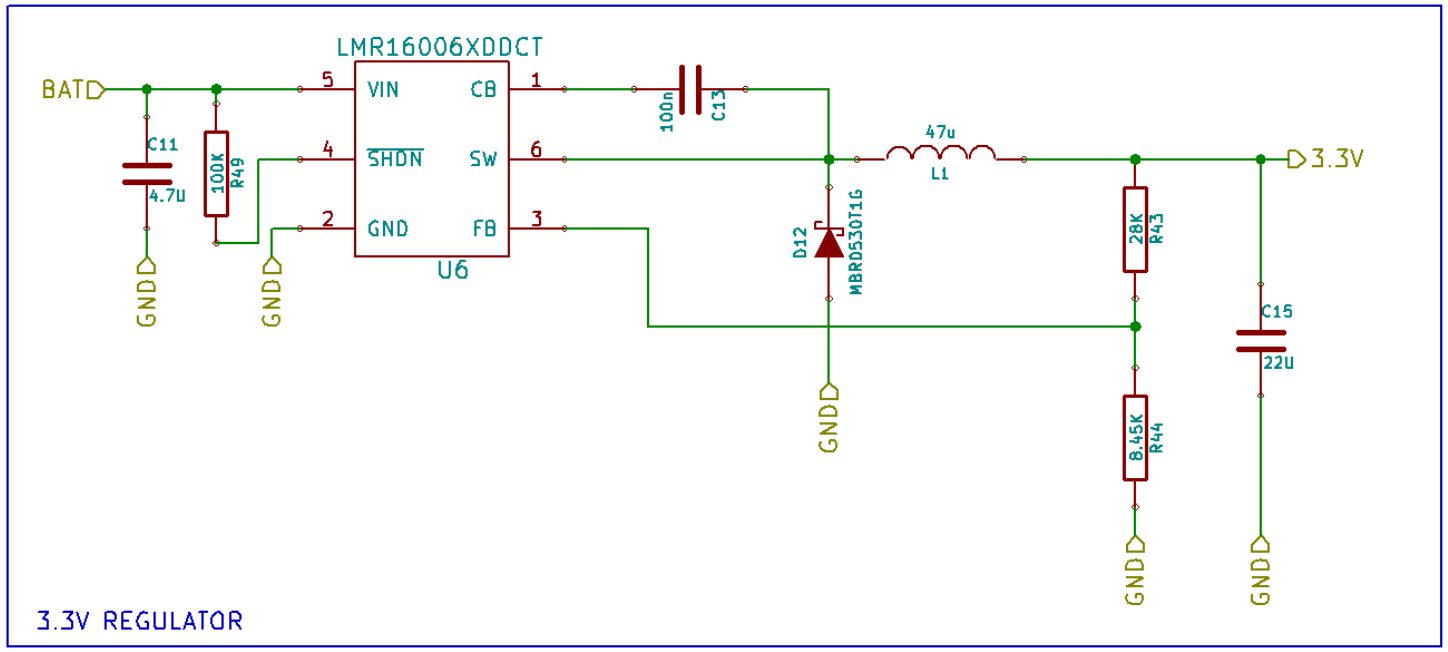
- Overcurrent and Overvoltage Protection
- Reverse Voltage Protection
- Noise Filtering

Power Management



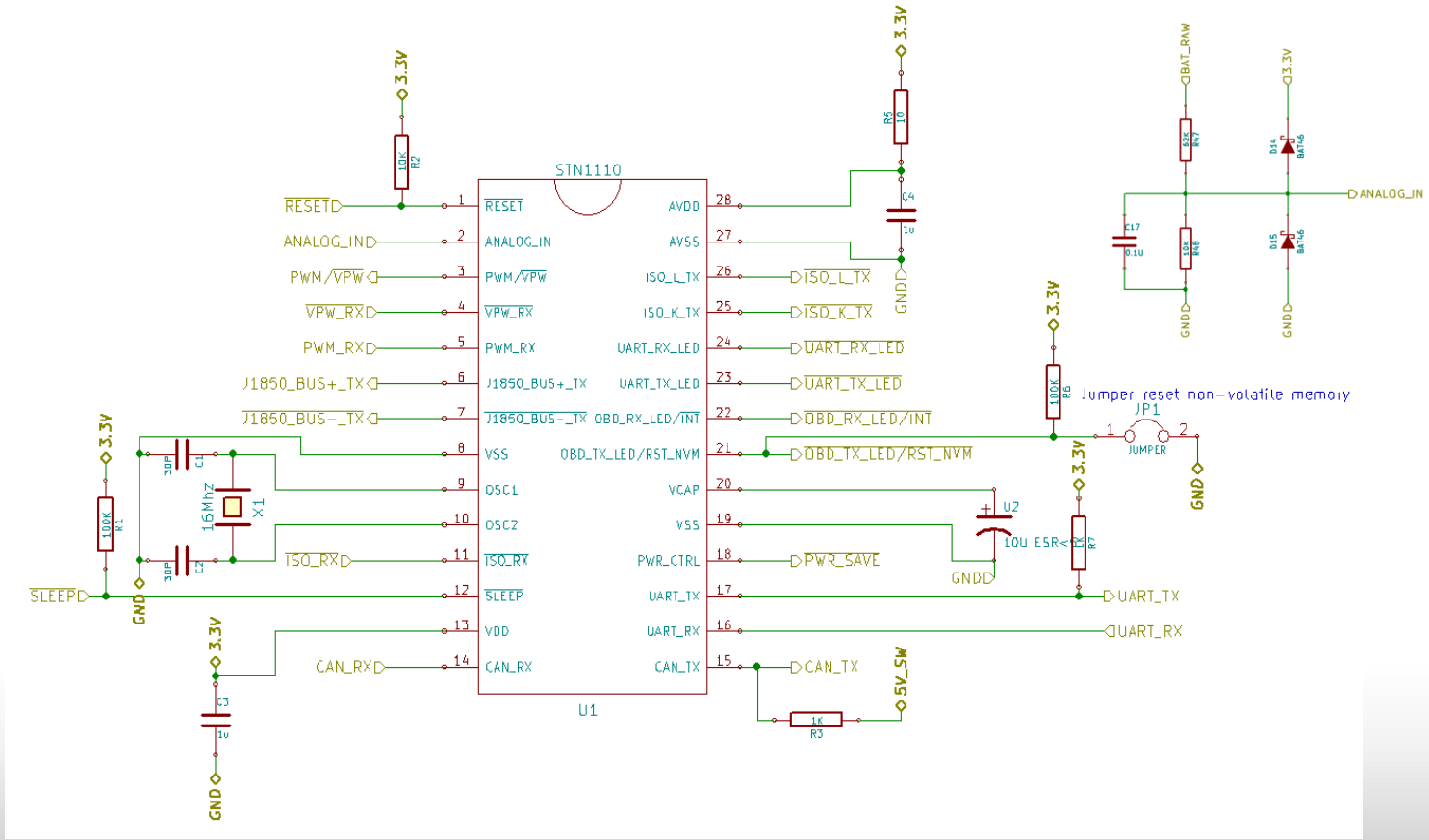
- 5V and BAT rail controlled by STN1110 Power Manager.
- 3.3V Systems manage their own power.

Voltage Regulation (3.3V & 5V)



- 3.3V and 5V use similar voltage regulators.
- LMR16006 Series are high frequency switchers.
- Provides current limit and thermal shutdown.
- Approaches 90% efficiency.

STN1110 OBD-II to Serial Interface

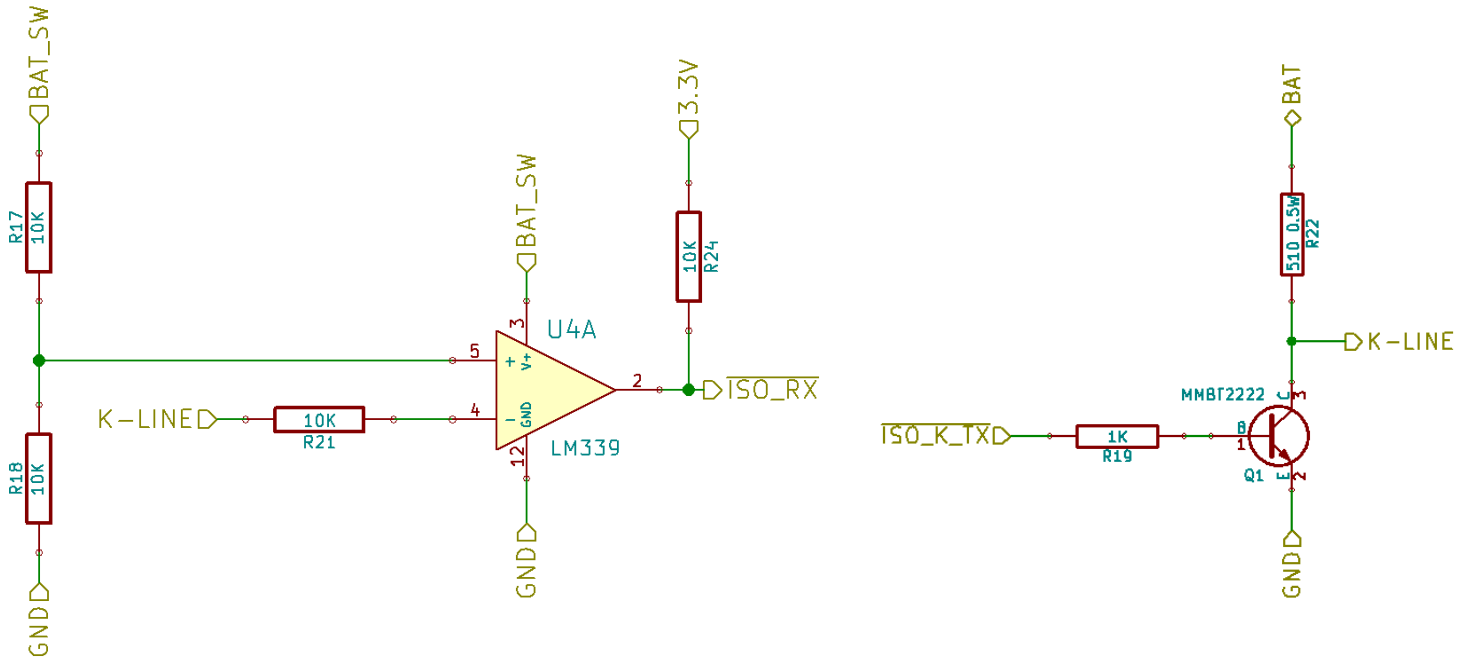


- Detects protocol and converts to serial data
- Accepts commands via ASCII
- Operates using the industry standard ELM327 protocol to simplify communications
- Has low power sleep mode
- Voltage sense can wake device from low power sleep mode

Signal Voltages

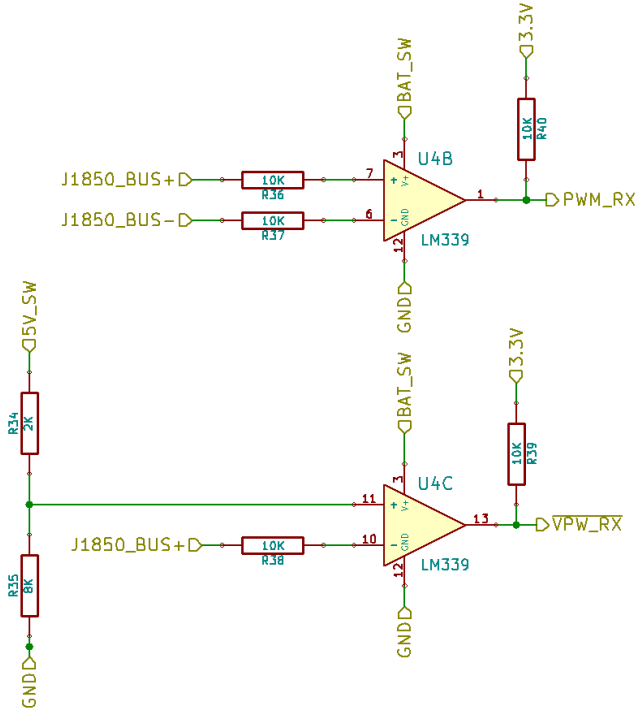
Name	Nominal High Voltage	Nominal Low Voltage
J1850 PWM	+5V	0V
J1850 VPW	+7V	0V
ISO 9141-2 & KWP2000	+12V	0V
CAN	+3.5V	+1.5V

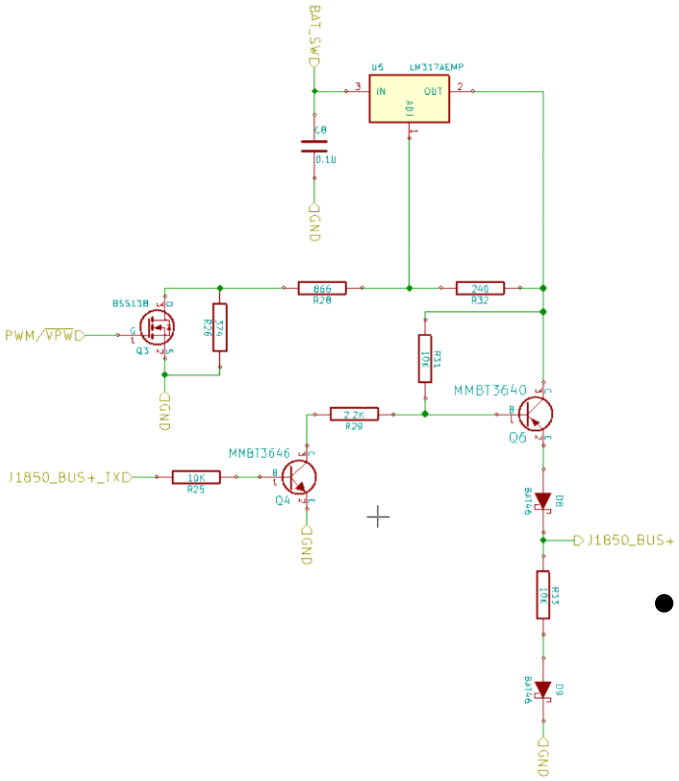
KWP2000 Transceiver

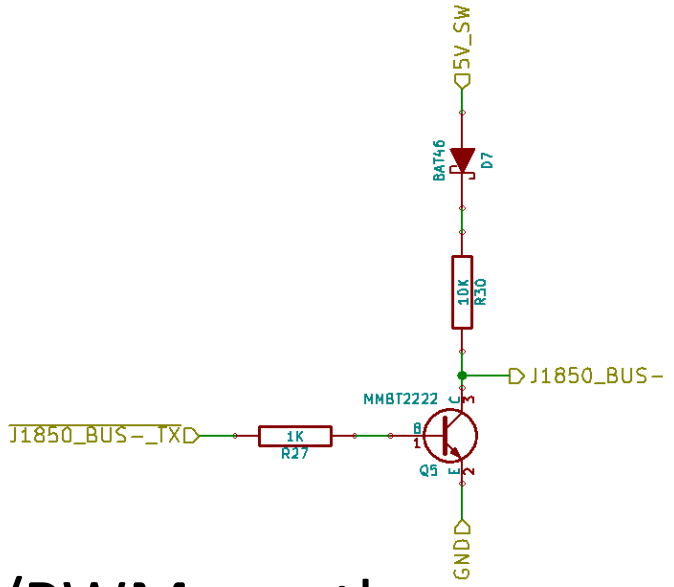


- Based on LM339 quad comparator
- Open collector output. When K-line > ½ BAT_SW (12V) the output is floating (high impedance)
- KWP2000 typically only uses K-line, L-line is used for wakeup when needed

J1950 PWM/VPW Transceiver

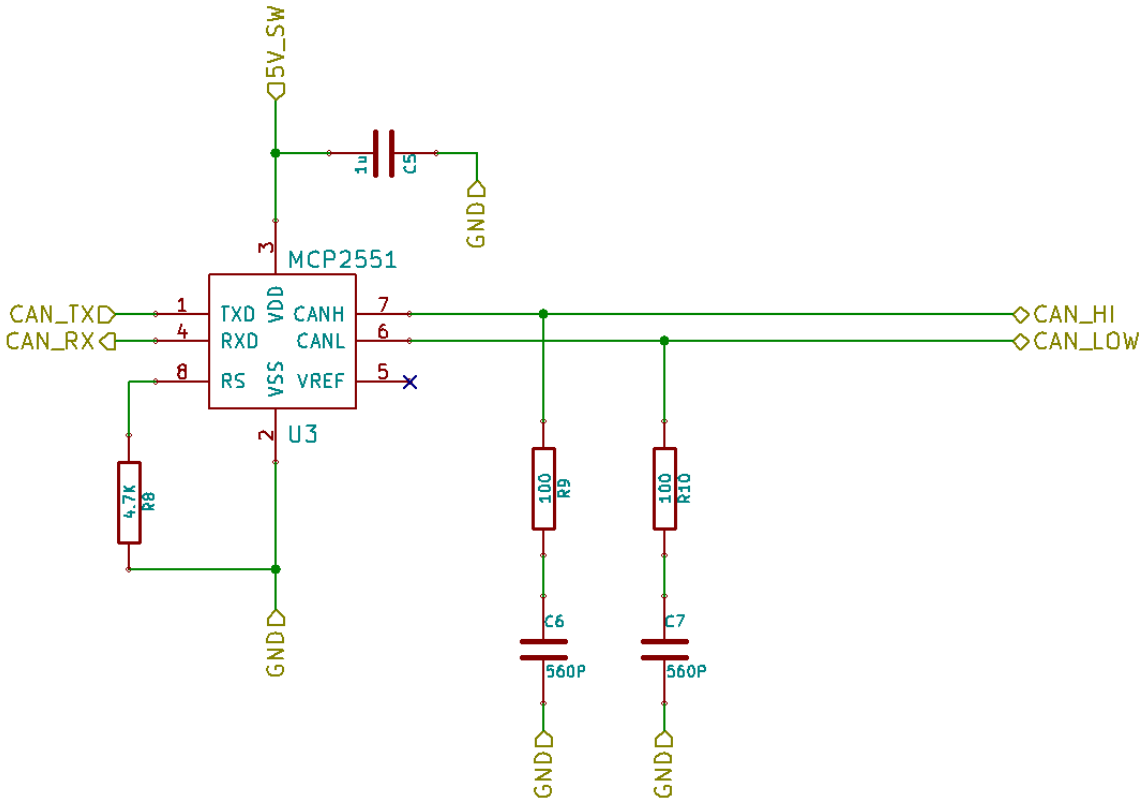






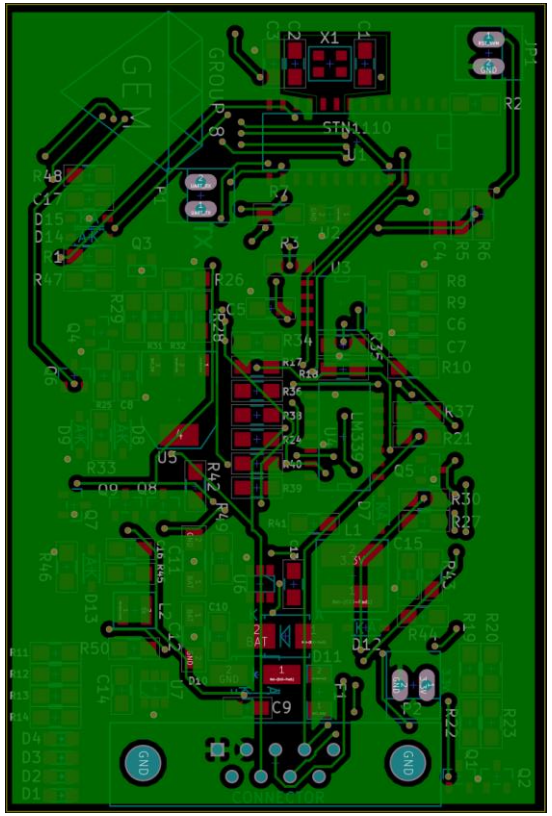
- VPW/PWM use the same line but not the same voltage.
- Linear regulator switches between the required voltages.

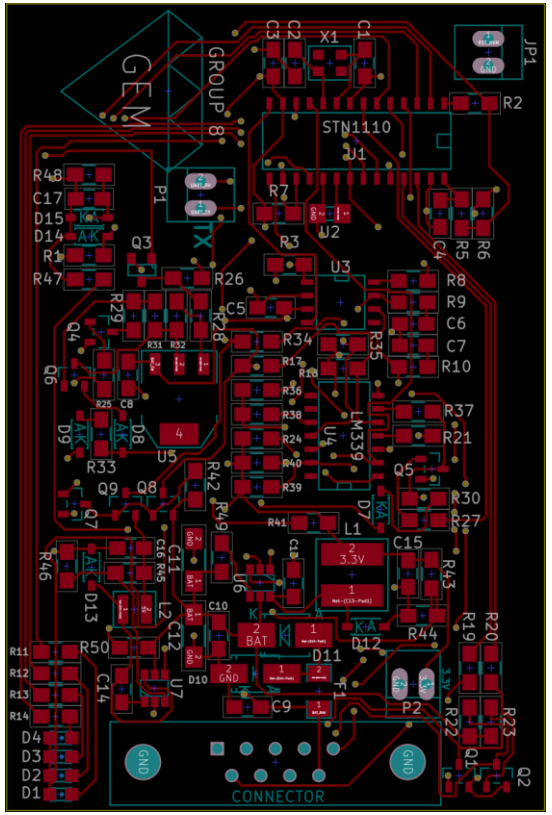
CAN Transceiver



- The MCP2551 is a high speed CAN transceiver.
- CAN is an industrial data standard.
- CAN has been adopted as the standard for vehicle networks.

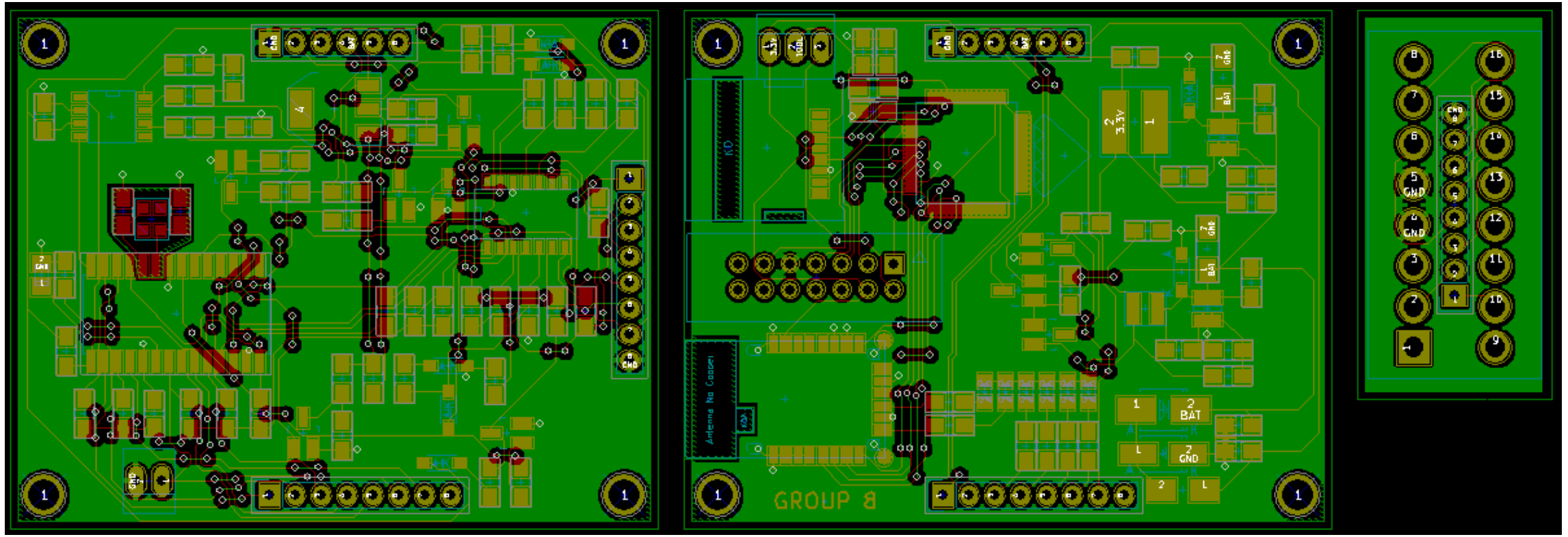
Initial Prototype





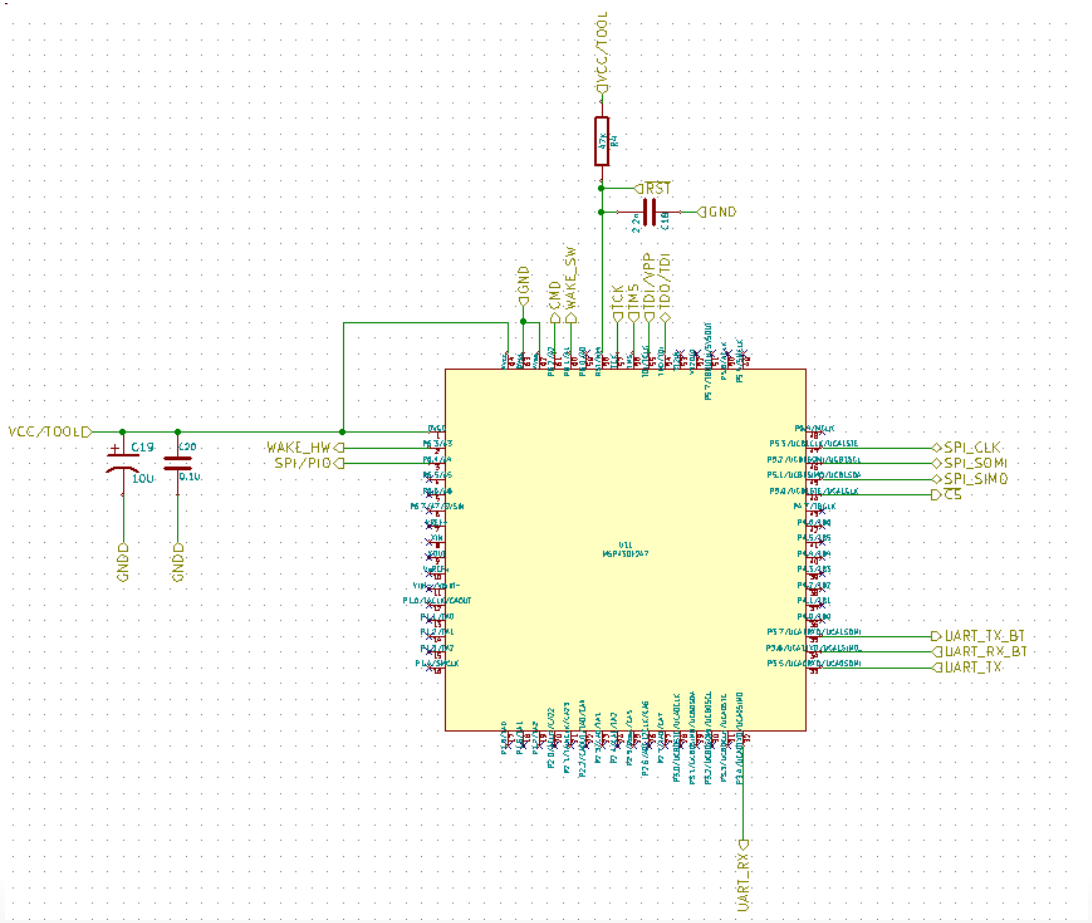
- First design for the frontend board.
- Does not integrate MCU and BLE.
- Small form factor.
- Lots of mistakes that we found and corrected in the final board.

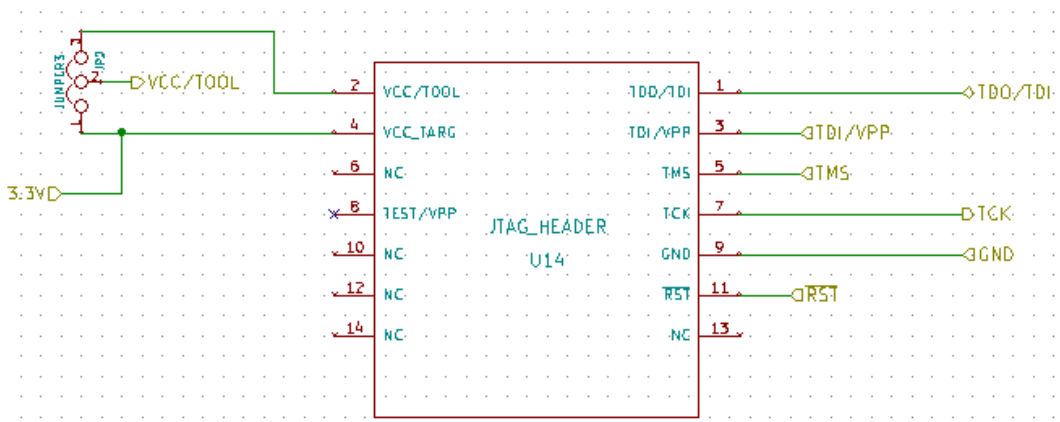
Final PCB



- Integrates stacking headers.
- Space saving design.
- Corrects for problems in prototype designs.

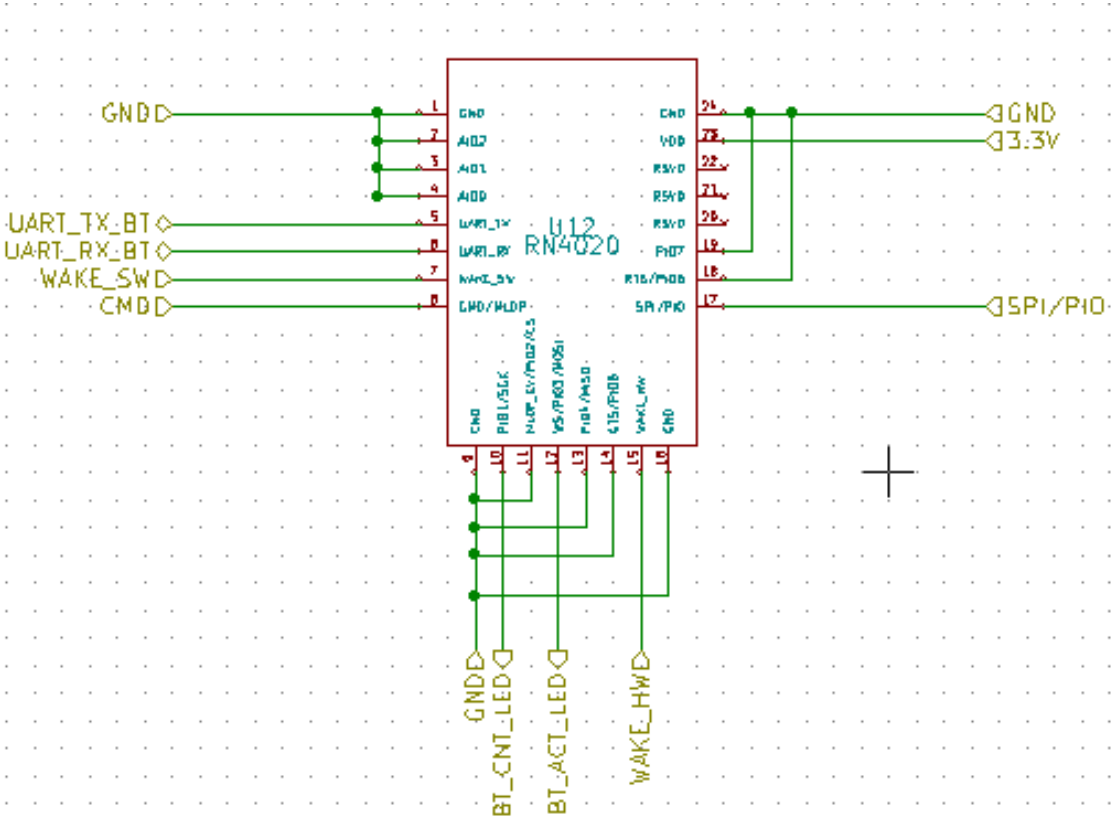
MSP430F247





- Ultra-low power 16 bit microcontroller
- 64-pin QFP package
- Entire team familiar with the msp430
- JTAG for on-board programming and debugging
- Provides up to four serial communication interfaces
- 32KB Flash memory, 4KB RAM
- Unused pins if additional hardware is needed
- Our project did not require the use of an OS

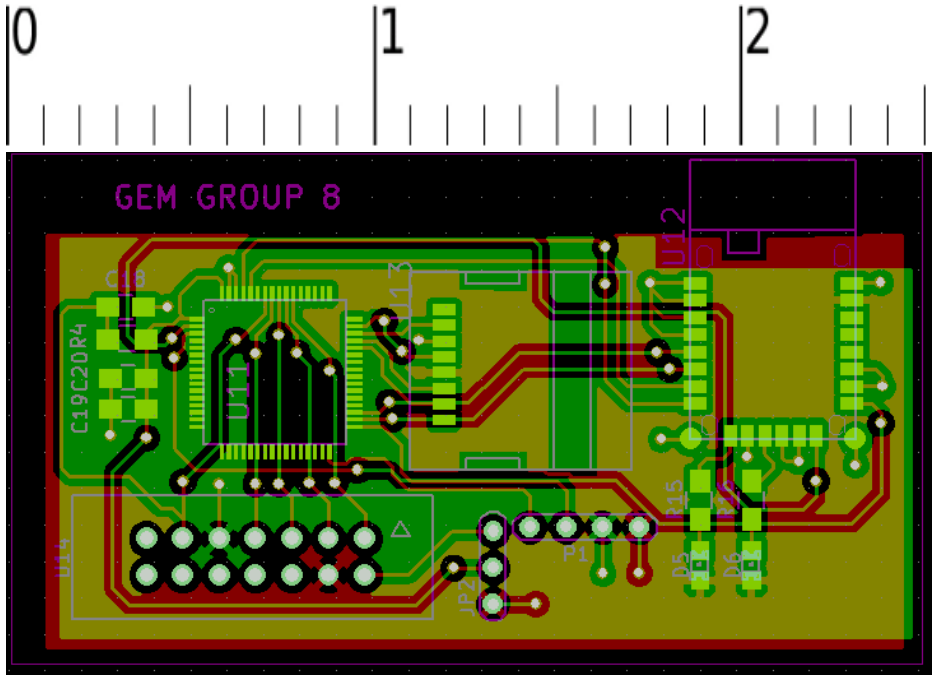
RN4020 Bluetooth Module



- The RN-42 (classic Bluetooth), RN4020 (BLE), and several stand along antennas were considered
- Bluetooth 4.1 (BLE/ Bluetooth Smart) best fit our energy specifications
- UART communication with it’s host controller
- ASCII command API
- Integrated antenna and fully implemented BT stack with several profiles included
- MLDP private profile which allows any data received via UART to be transmitted wirelessly
- Saved us from having to implement a complex BLE stack on our microcontroller

Digital Board First Revision

- Our first board revision isolated all low voltage (3.3V) hardware from the rest of the power system.
- Included MCU, Bluetooth, SD Card, and JTAG header
- 2 layer board with a ground and VSS copper pour on each
- 2.5” x 1.4”
- JTAG 14 pin breakout header for onboard programming/debugging
- 4 breakout pins for serial communication and power with the front end(power) board
- Allowed for development of firmware and debugging, independent of the power system



Initial Firmware Design

Design based on 5 subsystems run by a main subroutine

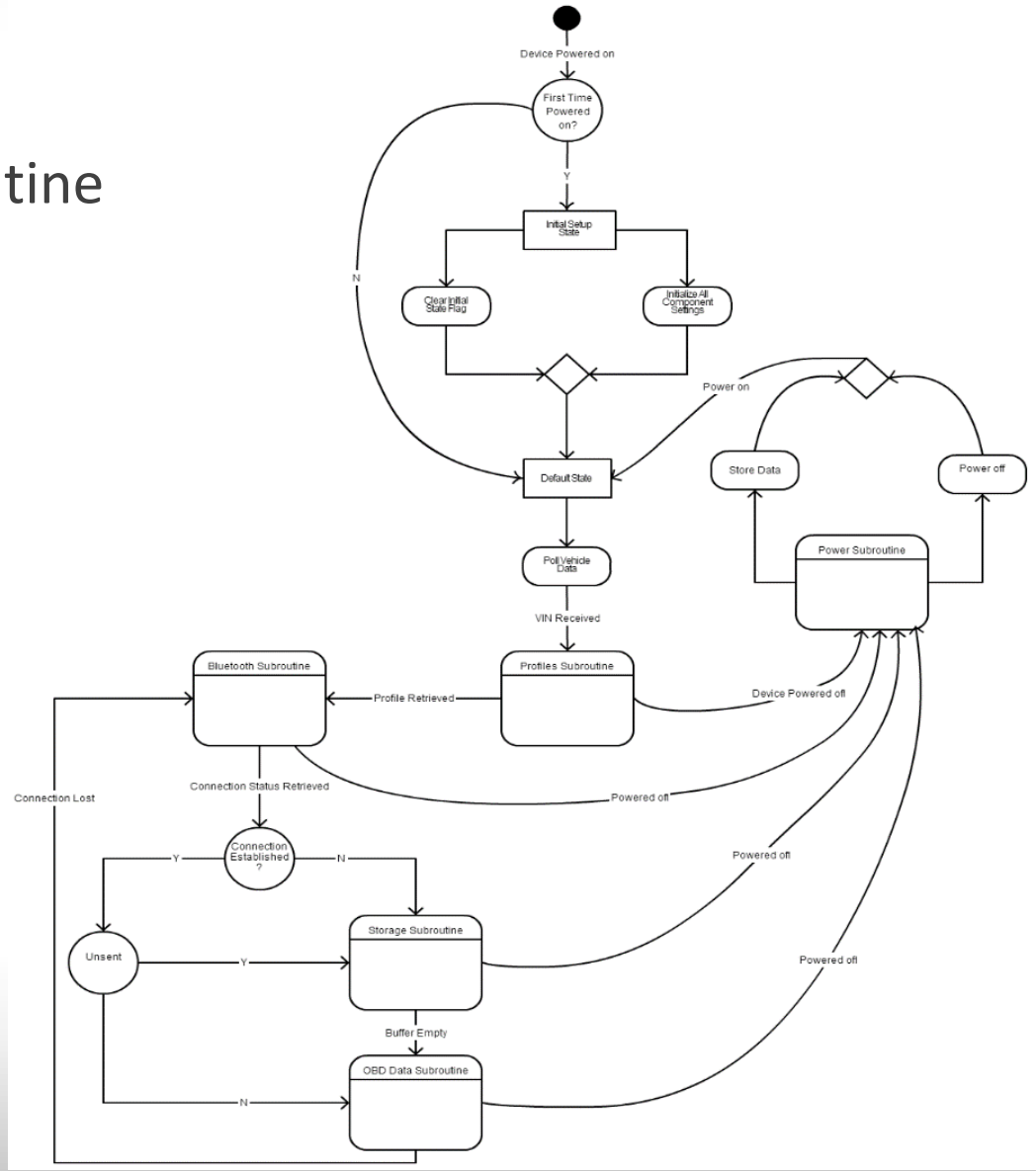
- Bluetooth
- Vehicle profiles
- Data logging
- OBD-II data pipelining
- Power

First time power-on state configures peripherals and enables core functionality.

VIN matching for identifying profiles.

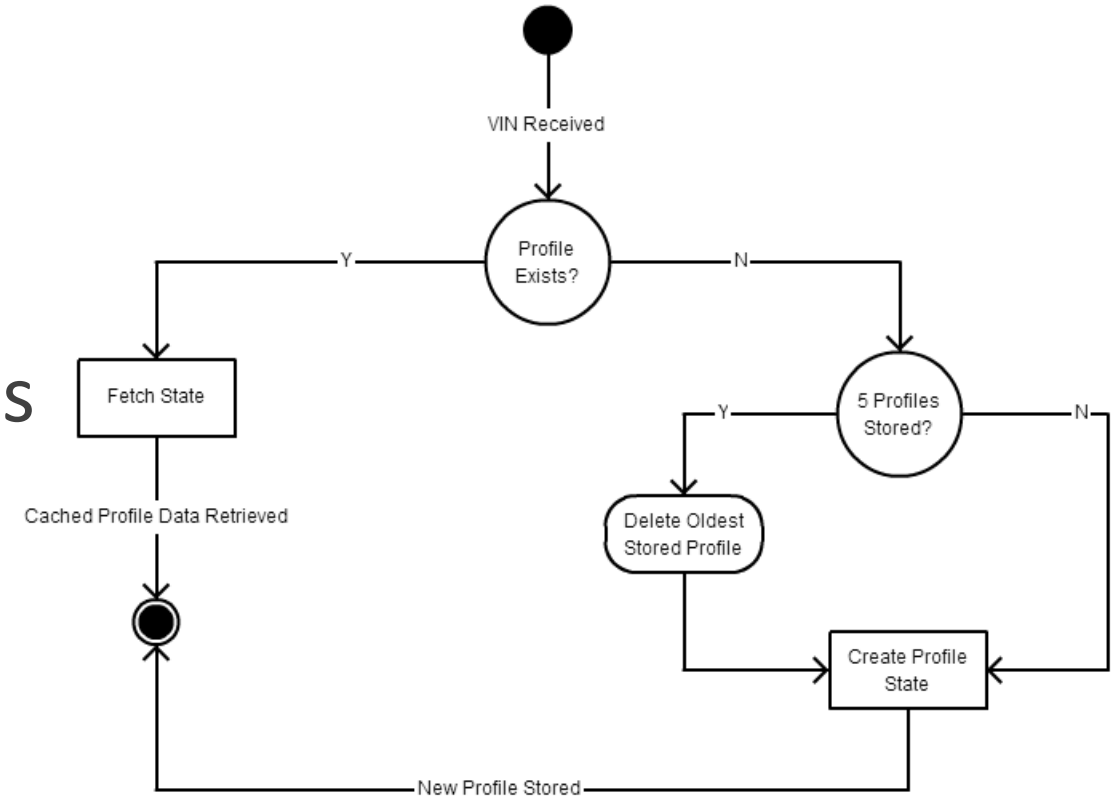
Log data when no connection is available.

Continuous OBD-II data stream over Bluetooth while connected.



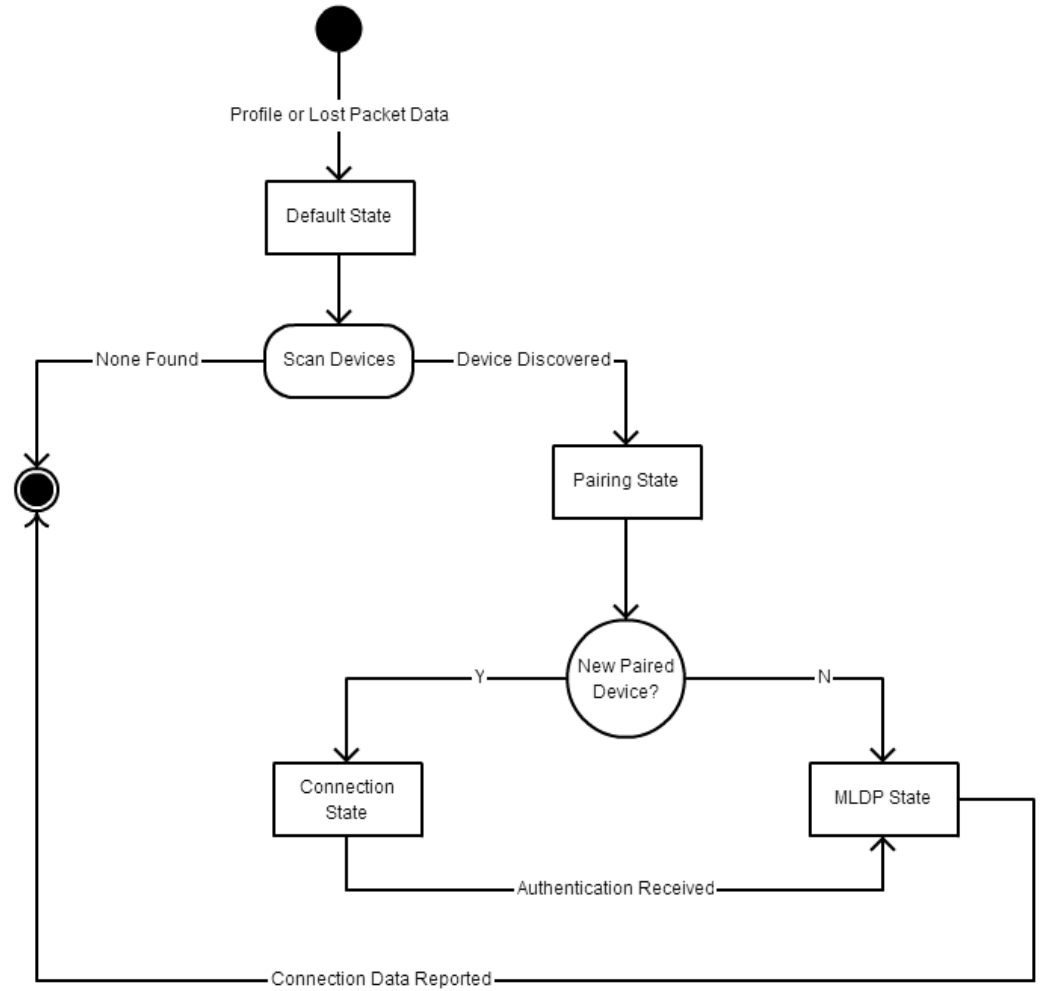
Initial Firmware Design – Vehicle Profiles

- Vehicle profile system allowing up to 5 vehicles
- Allows portability of GEM system
- Enables quicker set-up/connections if vehicle is recognized
- Save vehicle state upon power off
- Profile matching with unique VIN



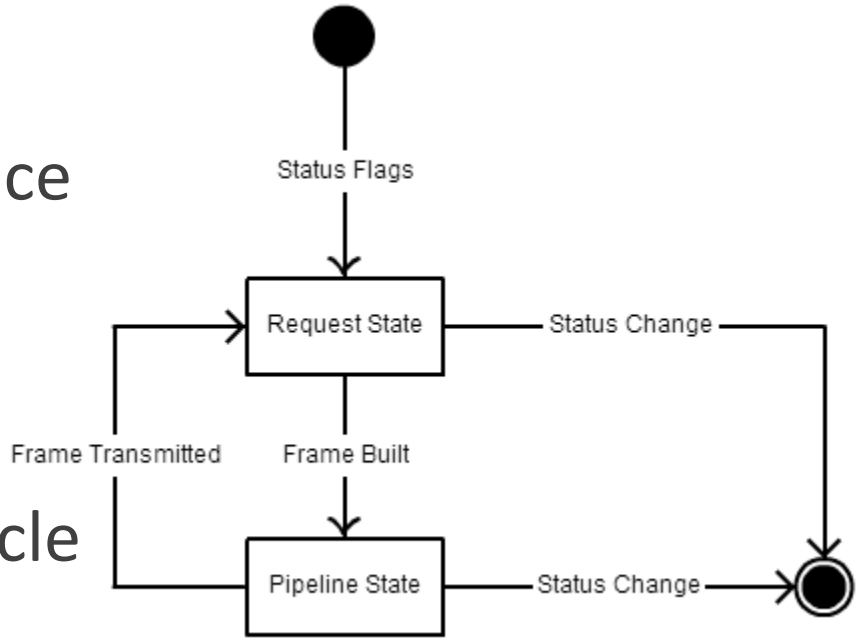
Initial Firmware Design – BLE

- Scanning, discovery, and pairing of devices through the RN4020
- Use profile state to establish quicker connections with previously paired devices
- Detect packet loss, and trigger the data logging subroutine
- Enter MLDP state if a stable connection is established, to prepare device to pipeline OBD-II data
- Full authentication and connection report frame used



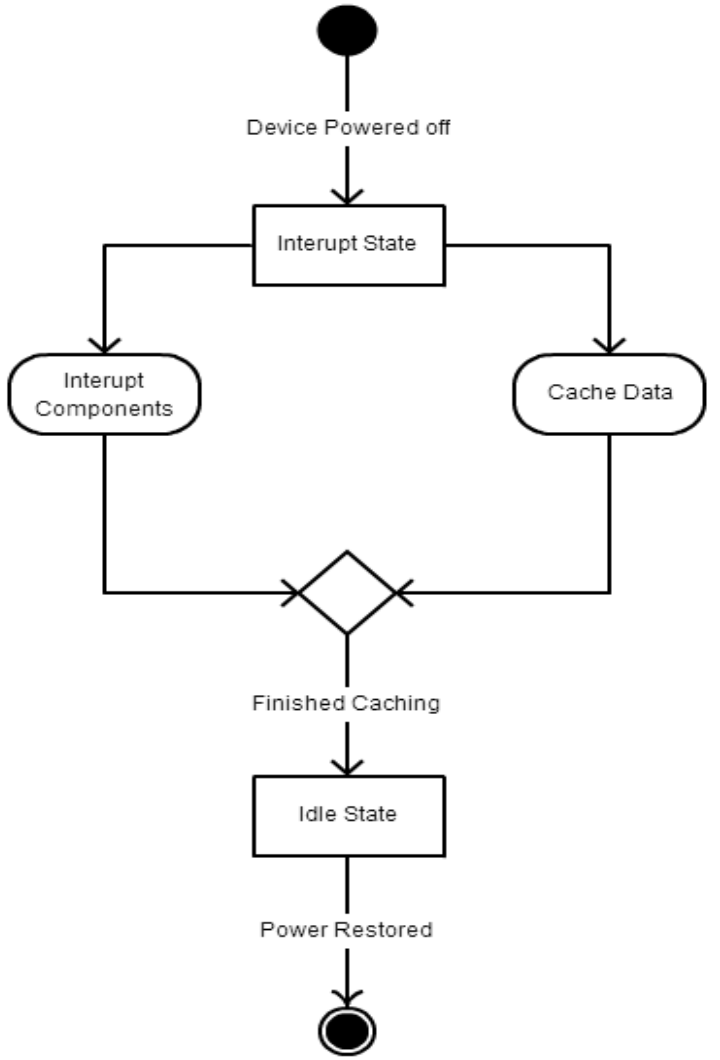
Initial Firmware Design – Data Transmission

- Triggered by a successful connection status
- System is set-up to pipeline data to a mobile device through the RN4020 Bluetooth module
- Alternates between a request and pipeline state
- The request state builds up a small frame by continuously polling the STN1110 for OBD-II vehicle data
- The pipeline state blasts completed packets to the RN4020 via UART, and the RN4020 transmits to the mobile device
- Execution can be interrupted by a system status change



Initial Firmware Design – Power

- The power subsystem handles all device sleep and idle states
- Can interrupt any other routine's execution
- Wakes up peripheral devices, or interrupts subroutines and saves the device state before a loss of power occurs
- Sends commands to the STN and RN4020 to control power consumption based on whether or not vehicle ignition is detected



Final Firmware

Several optimizations allowed for a simpler and significantly more efficient final firmware.

The use of profiles no longer a necessity since retrieving the current vehicle information from the OBD port only takes milliseconds.

Saving and retrieving states and flags into device memory wastes more time than it saves.

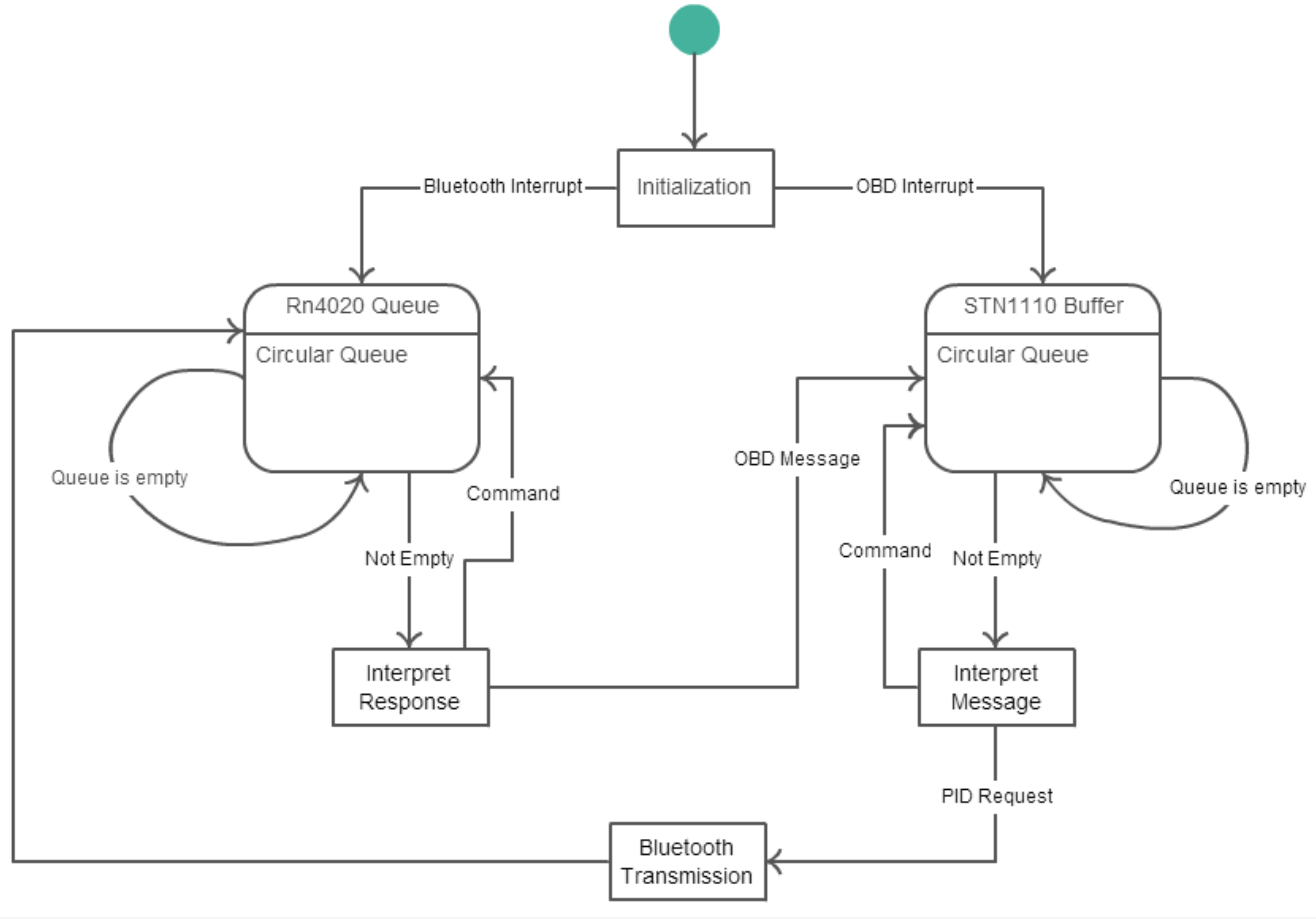
Sending a PID request from the mobile device wirelessly to the RN4020, passing it to the MSP430 through UART, interpreting the message, sending it to the STN1110, receiving a response, and finally transmitting it back it to the mobile device wirelessly only takes a few hundred milliseconds at most.

Tools used:

- Code Composer Studio
- TI Grace – Graphical Peripheral Configuration Tool
- MSP430 Flash Emulation Tool

Final Firmware

- Initialization sets the RN4020 to auto-advertise, changes device name to GEMBT, and enables the MLDP profile
- It also configures the STN
- Two circular queues used to keep a buffer of incoming UART messages from the STN1110 and RN4020
- Any Rx'd character triggers an interrupt, and that character is added to the queue
- Messages always processed in the order they were received



Why Android?

One of the earliest application design decisions was which platform(s) to develop the app for.

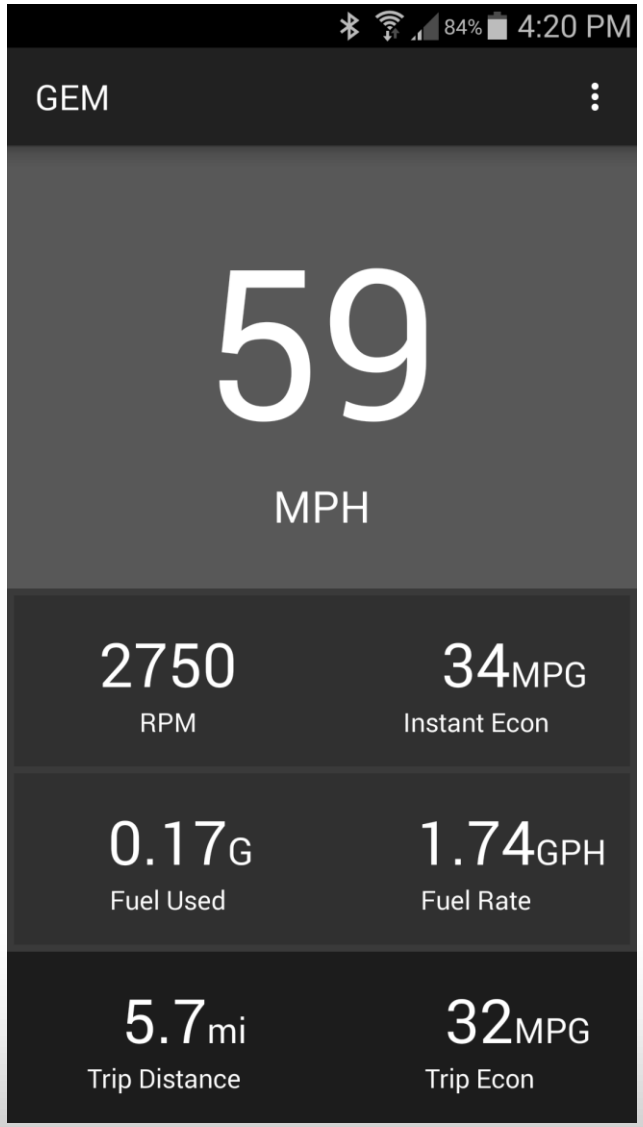
The following factors were taken into consideration:

- Java and C# programming experience.
- Eclipse and Android Studio IDEs are available on any OS.
- The team has multiple Android devices.
- The Android platform is open source.
- iOS OBD-II applications must use Wi-Fi.
- Desired Android development experience

Application Features

The OBD-II device transmits vehicle data to the GEM application enabling the following vehicle metrics to be tracked and displayed to the user:

- Vehicle speed (mph)
- Engine revolutions (rpm)
- Fuel consumed (g)
- Tripometer (mi)
- Instant fuel economy (mpg)
- Fuel consumption rate (gph)
- Trip fuel economy (mpg)



Software Development Tools

Android Studio – Android platform IDE.

SublimeText2 – Text and source code editor.

GitHub – Version control system.

Google Drive – Cloud file storage service.

Adobe Photoshop – Graphics editor.

OBD-II Emulator GUI – Vehicle data simulator.

Component Descriptions

CLASS DeviceScanActivity

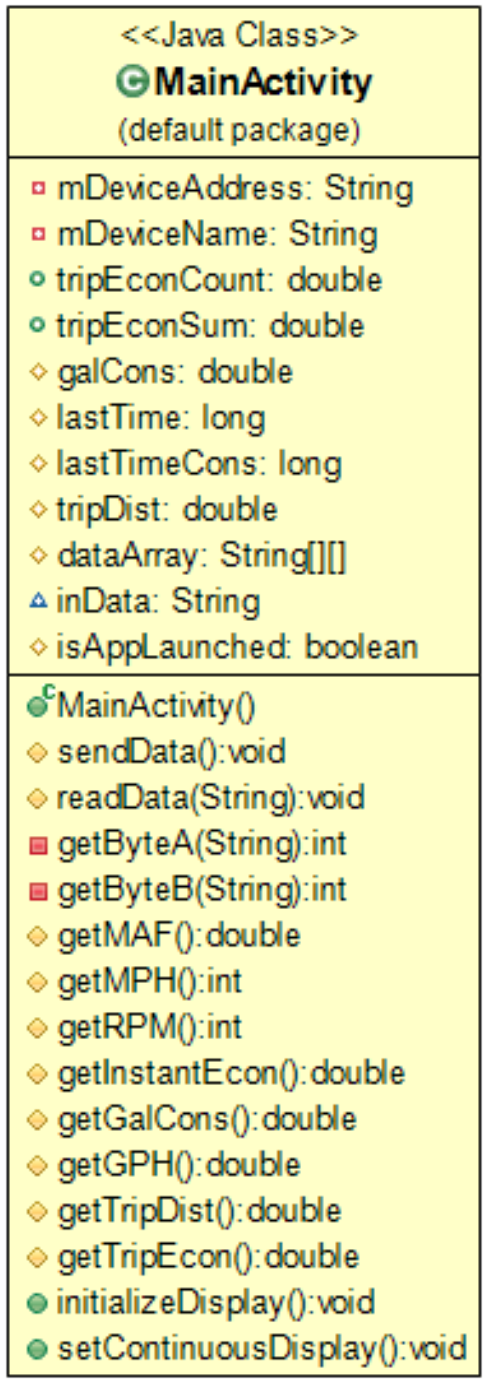
Launches the application, scans and displays Bluetooth LE devices.

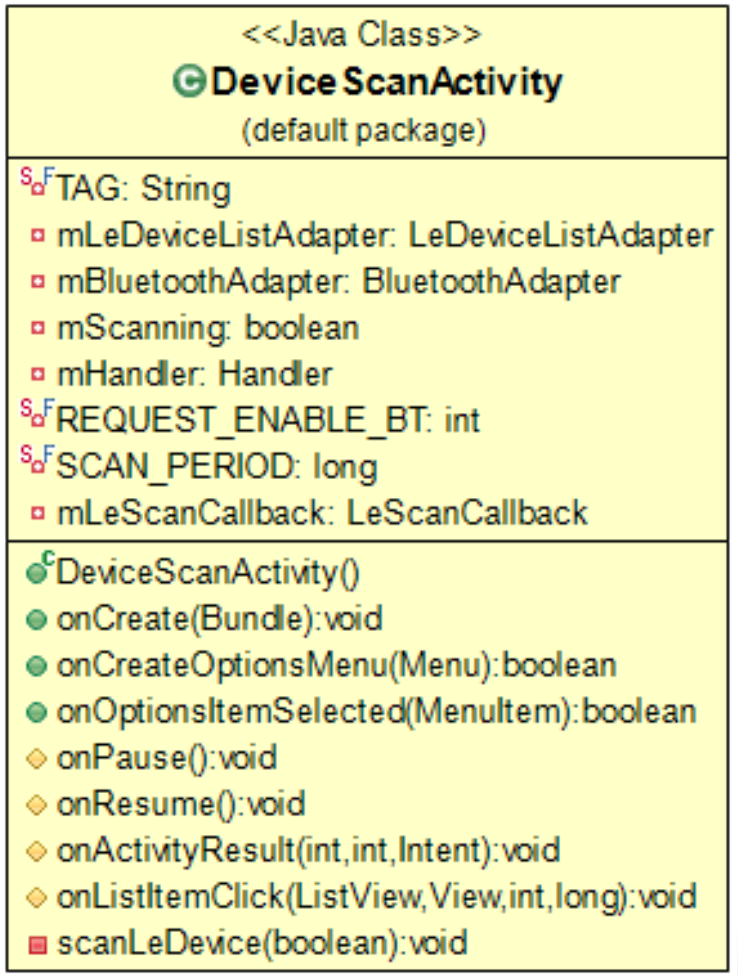
CLASS MainActivity

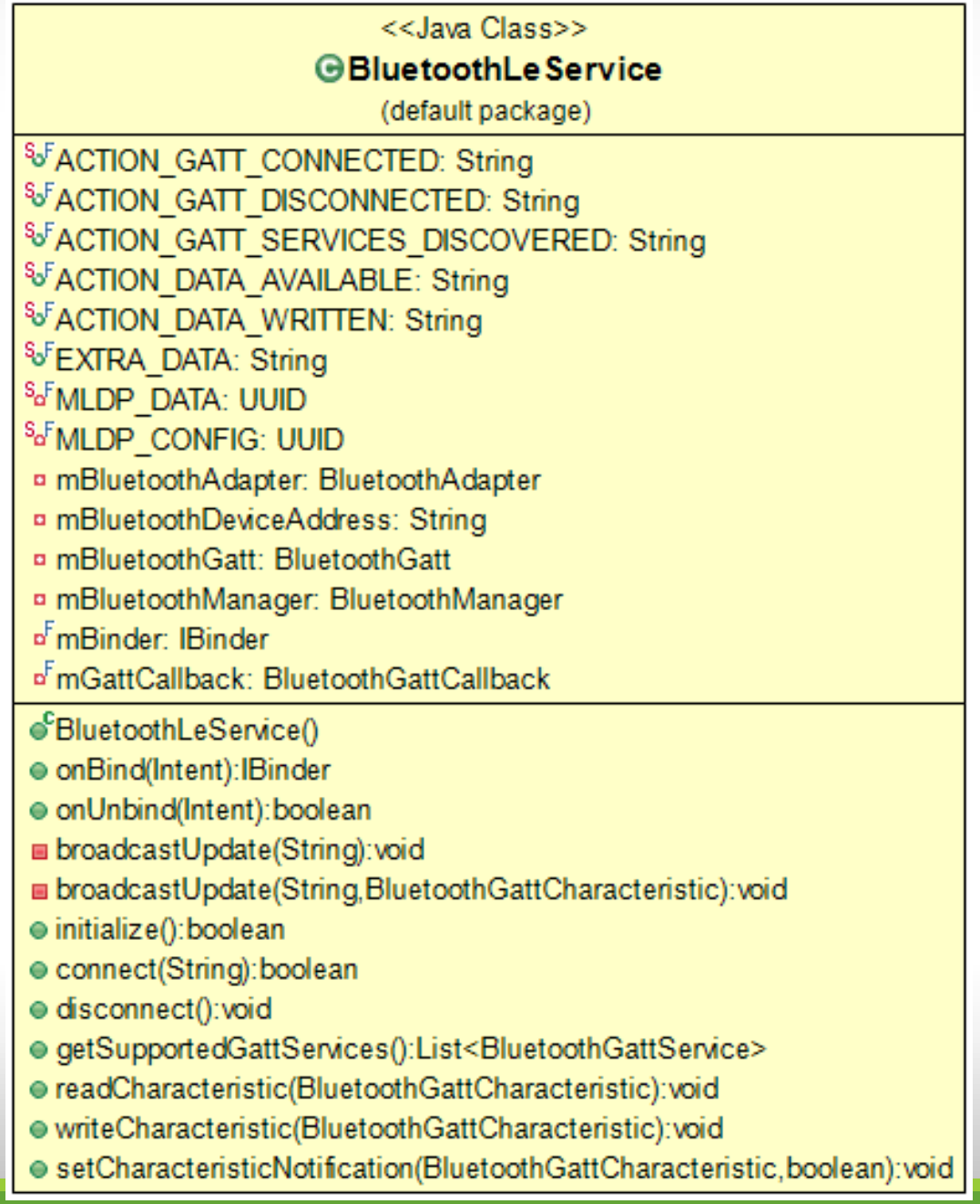
UI activity that communicates with the Bluetooth LE Service to manage and display vehicle data received from a Bluetooth LE device.

CLASS BluetoothLeService

Service that handles connections and data communication with a Generic Attribute (GATT) server hosted on a Bluetooth LE device.


<<Java Class>>
MainActivity
(default package)
mDeviceAddress: String
mDeviceName: String
tripEconCount: double
tripEconSum: double
galCons: double
lastTime: long
lastTimeCons: long
tripDist: double
dataArray: String[][]
inData: String
isAppLaunched: boolean
MainActivity()
sendData():void
readData(String):void
getByteA(String):int
getByteB(String):int
getMAF():double
getMPH():int
getRPM():int
getInstantEcon():double
getGalCons():double
getGPH():double
getTripDist():double
getTripEcon():double
initializeDisplay():void
setContinuousDisplay():void


<<Java Class>>
DeviceScanActivity
(default package)
TAG: String
mLeDeviceListAdapter: LeDeviceListAdapter
mBluetoothAdapter: BluetoothAdapter
mScanning: boolean
mHandler: Handler
REQUEST_ENABLE_BT: int
SCAN_PERIOD: long
mLeScanCallback: LeScanCallback
DeviceScanActivity()
onCreate(Bundle):void
onCreateOptionsMenu(Menu):boolean
onOptionsItemSelected(MenuItem):boolean
onPause():void
onResume():void
onActivityResult(int,int,Intent):void
onListItemClick(ListView,View,int,long):void
scanLeDevice(boolean):void


<<Java Class>>
BluetoothLeService
(default package)
ACTION_GATT_CONNECTED: String
ACTION_GATT_DISCONNECTED: String
ACTION_GATT_SERVICES_DISCOVERED: String
ACTION_DATA_AVAILABLE: String
ACTION_DATA_WRITTEN: String
EXTRA_DATA: String
MLDP_DATA: UUID
MLDP_CONFIG: UUID
mBluetoothAdapter: BluetoothAdapter
mBluetoothDeviceAddress: String
mBluetoothGatt: BluetoothGatt
mBluetoothManager: BluetoothManager
mBinder: IBinder
mGattCallback: BluetoothGattCallback
BluetoothLeService()
onBind(Intent):IBinder
onUnbind(Intent):boolean
broadcastUpdate(String):void
broadcastUpdate(String,BluetoothGattCharacteristic):void
initialize():boolean
connect(String):boolean
disconnect():void
getSupportedGattServices():List<BluetoothGattService>
readCharacteristic(BluetoothGattCharacteristic):void
writeCharacteristic(BluetoothGattCharacteristic):void
setCharacteristicNotification(BluetoothGattCharacteristic,boolean):void

Data Calculation

Up to two bytes of data is received from the GEM device.

Byte A								Byte B							
A7	A6	A5	A4	A3	A2	A1	A0	B7	B6	B5	B4	B3	B2	B1	B0

Each byte is received as two hexadecimal values.

Example: When an RPM value of `0FA0` (4000_{10}) is received, the methods getByteA and getByteB are called so that Byte A = `0F` and Byte B = `A0`.

The decimal representation of the RPM is calculated by calling getRPM:

(Byte A_{10} x 256 + Byte B_{10}) / 4 (Vehicle sends 4 times the RPM, divide by 4)

-> (15 x 256 + 160) / 4 = 1000 revolutions per minute

Test Data

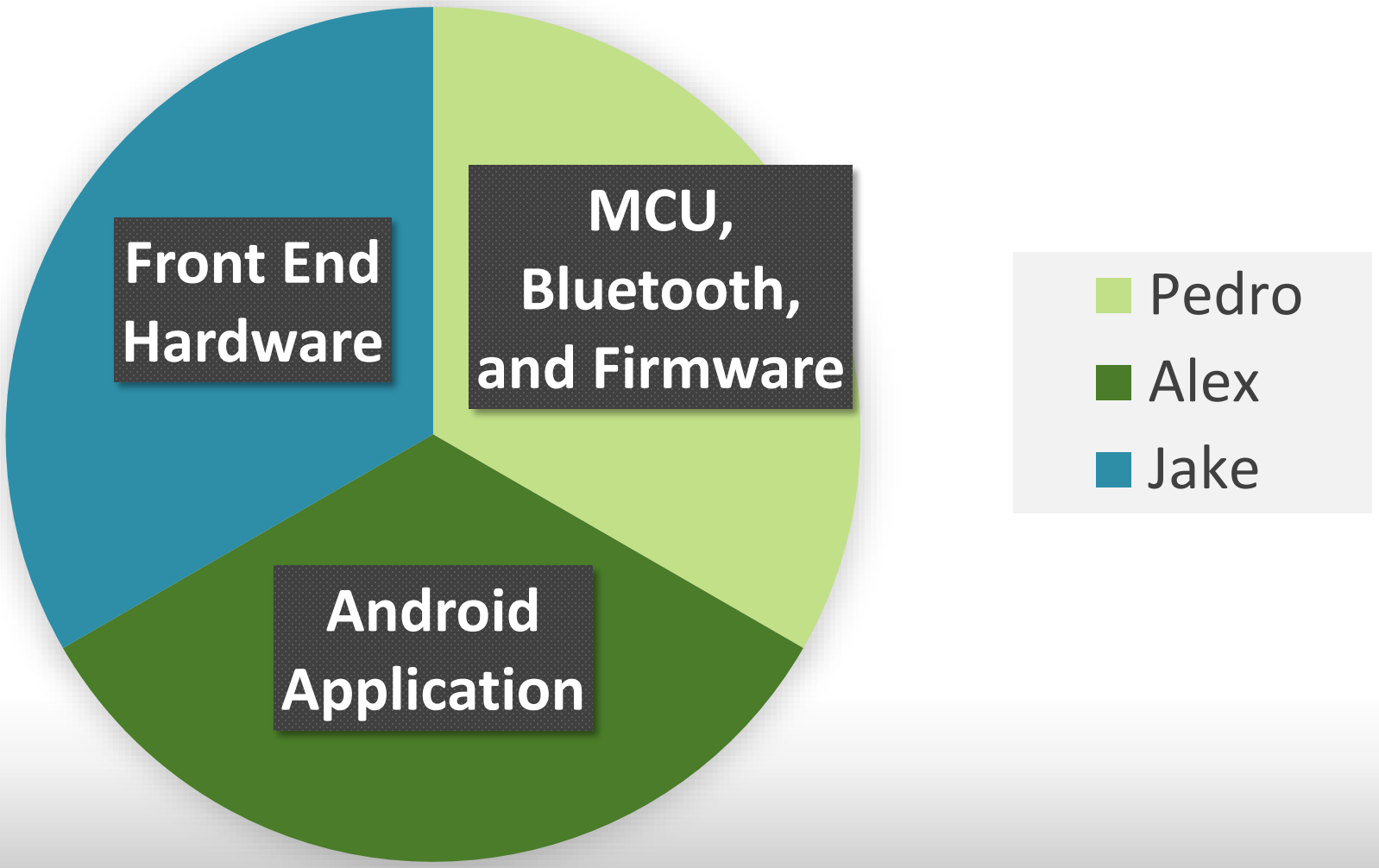
Test PID	Values set on emulator	Responses	Description	Notes
01 1F	{5,100,2000,5000,3000}	411F {0005,0064,07D0,1388,7530}	Run time since engine start (in seconds)	Can record time since engine start up to ~18 hours
01 0C	{1000,2000,3000,4000,5000}	410C {0FA0,1F40,2EE0,3E80,4E20}	Engine RPM in 1/4th value	0x2EEE0/4 = 3000 RPM. RPM needs to be divided by 4
01 0D	{10,25,50,100,125}	410D {0A,19,32,64,7D}	Speed in km/hr	Just convert from hex to decimal
01 04	{5 ,10, 20, 30, 40, 50, 60, 70, 80, 90, 95}	4104 {0C,19,33,4C,66,7F,99,B2,CC,E5 ,F2}	Engine load %	Take the response, (its in hex), and multiply by (100/255). All responses are within 3% error. Less than 10 gives inaccurate readings
01 11	{10,25,50,75,100}	4111 {19,3F,7F,BF,FE}	Throttle position, as a %	Take the response, (its in hex), and multiply by (100/255). < 3 % error
01 10	80	4110 1F40	MAF air flow rate %	1F40 = 8000. (80.00). The emulator reads a percentage, but It's supposed to be a value in g/sec with [((A*256)+B) / 100]
01 5E	{10,15,20,40,100,150}	415E {0087,008C,0091,00A5,00E1,0 BB5}	Engine fuel rate. L/H	((A*256)+B)*0.05 Where A = the first 2 hex values, and B=the next 2. Inaccurate for smaller values

Budget Overview

Item	Cost	Supplier
PCB's	$107.50	Oshpark
Stencils and Jigs	$43.07	Oshstencil
Circuit Components	$229.55	Mouser
OBD-II Emulator	$229.00	Freematics
Total	$609.12	

- Original funding request: $774.08
- Remaining budget: $164.96

Division of Labor



Successes

- Low power draw < 60 mA when operating
- Communicates via BLE
- Small form factor
- OBD responses on the order of milliseconds

Thanks

Thank you to Boeing for providing the funding for this project.

Thank you to our professors that provided insight and feedback on this project.

Questions