Wireless Communications for Remote Data Logging

Submitted to UMD Electrical Engineering Department

Final Project of ENEE408K Senior Capstone

Dylan Deslongchamp, Keonwoo Kim, Georgios Mastoras, Andrew Nixon, Vineeth Vajipey

University of Maryland, College Park

12/18/22

Table of Contents

I. Team Member Contributions/Signatures

Dylan Deslongchamp: Hardware
Contribution: I researched and provided recommendations for many of the components that made it to the final design. Performed calculations to check components and for PCB analysis through interpreting datasheets and applying circuit analysis.
Electronic Signature: Dylan Deslongchamp, I pledge on my honor that I have not given or received any unauthorized assistance on this report.

Andrew Nixon : Hardware
Contribution: I researched and provided recommendations for many of the components that made it to the final design. I worked on each version of the schematic and outlined the final PCB in Altium.
Electronic Signature: Andrew Nixon, I pledge on my honor that I have not given or received any unauthorized assistance on this report.

Keonwoo Kim : Software
Contribution: I was assigned to research, design, and test our software implementations. On top of that, I visited the TerpRacing team to learn that our semester goal was not what was given to us in the beginning of the semester, and that we would be forced to only develop a demo product. During the mid-semester, I was unavailable due to being sick and fell behind the group. I decided to spend more time on the report and the slides as the end of the semester was coming up.
Electronic Signature: Keonwoo Kim, I pledge on my honor that I have not given or received any unauthorized assistance on this report.

George Mastoras : Software
Contribution: I helped research, design and test all iterations of our software implementations. This involved learning to use new technologies and implementing the circuit design to get them working. I worked alongside Vineeth, to develop our understanding of all software related topics, including all softwares used as well as the technical aspects of each component. Vineeth and I spent several hours each week developing the demonstrations in the lab, in addition to investing extra time at home individually. Most of the work done was researching and code development.
Electronic Signature: I, Georgios Mastoras, pledge on my honor that I have not given or received any unauthorized assistance on this report.

Vineeth Vajipey : Software
Contribution: I helped research, design, and test all versions of our software demonstrations throughout the semester. I worked alongside George in the lab to develop a fundamental

understanding of the Teensy, XBee, and CAN IC modules and how they operate. I worked on building demos for XBee-XBee communication, CAN - CAN communication, and the final complete demonstration. This mostly involved writing code using the Teensyduino IDE and XCTU, but also a lot of research into the design and electrical and computer engineering theory behind the devices that we used.

Electronic Signature: Vineeth Vajipey, I pledge on my honor that I have not given or received any unauthorized assistance on this report.

II. Executive Summary

Our collective goal is to design and build a device that will communicate from a FSAE vehicle to a command center during a race. This would require two sub teams, a hardware team and software team, to individuals finish sub goals in tandem, printing and designing of a PCB and creating software to read CAN data and transmit said data to a control center, respectively. The two teams then would come together and implement the designed software into the final PCB and troubleshoot.

The device will be housed in a waterproof box behind the driver's head and will receive data from different sensors on the vehicle via Controller Area Network (CAN) Bus. It will then transmit that data 1-2 miles to a remote Terps Racing computer in order for the control team to monitor the driver and the vehicle while the race is in progress. The signals will come in on a CAN high and CAN low bus, and there will be about 15 different sensor data represented on this bus. Power to our device will come from a 12V battery located on the vehicle, which will be stepped down to different voltages using DC-DC converters. The device should be lightweight and compact in order to not add any additional unnecessary weight to the vehicle and fit behind the headrest of the driver.

The software and hardware teams worked together to create multiple demos and PCB designs. Teams began with very basic demos of sending just any signal from one transceiver to another and having a PCB with only major components. The teams developed these earlier demos to incorporate more of the desired functionality of the FSAE TerpsRacing team. This resulted in having a final demo that combines the first demo with CAN generated inputs and a much more compacted and properly designed PCB. In the end, our team was unable to achieve all of our goals like getting the PCB printed in time and implementing software onto it, but we were able to achieve major goals of sending and receiving data from a CAN bus "on the car" to a "control center" and in creating a PCB design in Altium.

If we were to continue forward we would have a few PCBs printed and soldered with all the components then fit and secured within the waterproof box. Then we would begin to test the software with our final PCB and see if any issues either with the hardware or software arised. Once we finished troubleshooting we would pass off our final device to the TerpsRacing team to be implemented onto their FSAE electric vehicle to be used in their future competitions.

III. Main Body

1. Hardware

1.1 Criteria

1.1.1 Relevant FSAE Rules

Rule IN11.2 (Rain Test Conduct) from the 2023 FSAE Rule Book [1] confirms that our device must be able to withstand contact with water. The rule is state below:

The water spray will be rain like, not a direct high pressure water jet

A. Water will be sprayed at the vehicle from any possible direction for 120 seconds.

B. The water spray will stop.

C. *The vehicle will be observed for 120 seconds.*

1.2.1 Terps Racing Requirements

After corresponding with the UMD Terps Racing team, they conveyed that they would like to see our device meet the following specifications:

A. The device should be able to transmit information at a maximum range of 1-2 miles in clear line of sight on a race track
B. The device should have 4 input lines: 12V, 0V, CAN high and CAN low
C. The device should fit within an area of 7”x6”x3” (Width x Height x Depth) behind the driver’s head.
D. The microcontroller should be easily accessible and removable from the box for programming.
E. External connectors to the car should be water resistant Molex MX150.
F. An expect 15 different sensors’ data will be sent, list of different sensors can be seen in Appendix (C)

1.2 Components

A purchase list with the cost of all components can be seen in the Appendix (F)

1.2.1 DC-DC Converters

Lead acid battery supplying 12V for the low voltage systems is stepped down to both 5V and 3.3V through two separate DC DC converters. 5V supplies the Teensy4.1 microcontroller and 3.3V supplies XBee transceivers and CAN module. The 12V to 5V part number is PDM2-S12-S5-5, which outputs a constant +5V. The 12V to 3.3V part number is PDM2-S12-S3-5, which outputs a constant +3.3V. This part is shown in figure 1, below.

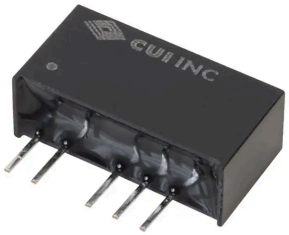


Figure 1: Both the 12V to 5V And 12V to 3.3V DC-DC converters have the same footprint and appearance. [2]

1.2.2 CAN Transceiver

Serves as a bridge between the Car CAN bus system and our microcontroller. We chose the CAN IC with part number MAX33040EAKA+T, shown in figure 2.

Figure 2: CAN IC operates at 3.3V with 5 Mpbs transmission rate. [3]

1.2.3 Coin cell

Gives Teensy access to a Real Time Clock (RTC), shown in figure 3, even when EV power is not supplied. The module will be able to accurately timestamp data points even if the car is shut off for days or weeks between sessions.

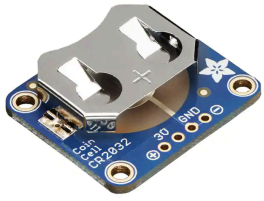


Figure 3: A simple commercial part to be mounted to our PCB that holds a 3V coin cell. [4]

1.2.4 Microcontroller

The Teensy, shown in figure 4, is an extremely fast board (600MHz) that outcompetes other microcontrollers in virtually every way. It has expandable ram, meaning the board can move from kilobytes to megabytes of RAM and flash. It has a huge user base where most common questions have been answered, and operates through object-oriented C++, which will make

programming easier and more organized. Arduino software is more familiar than Hercules software.

Figure 4: A Teensy 4.1 Microcontroller top view. [5]

1.2.5 XBee

Xbee, shown figure 5, was the recommended board, and the S2C Pro meets the maximum operating range (2 miles) for clear line of sight while operating at 2.4 GHz for maximum data transmission at 250 Kbps. The non-Pro S2C model does not have the necessary range of 1-2 miles and only has a range of about 0.7 miles.

Figure 5: XBee Pro S2C board with 2mm spaced pins. [6]

1.2.6 Additional Memory

The Teensy 4.1 has space allocated for two additional RAM chips to be soldered in. We added 2x8 MB RAM, shown in figure 6. Many other MCU's lack this option.

Figure 6: Supplemental memory. [7]

1.3 Power Budget

1.3.1 12V-5V Budget

The Teensy is expected to draw a maximum 100mA while not driving any additional significant loads from its pins. The DC-DC converter with a 5V output is a 2W part capable of outputting a maximum 400mA at 5V. This is over 4 times the expected current, so exceeding this is extremely unlikely.

1.3.2 12V-3.3V Budget

Two major components will accept power from the DC-DC converter with a 3.3V output. The CAN IC maximum current draw is 50mA [8]. The XBee maximum current draw is 120mA [9], making 170mA combined. The 3.3V converter can supply a maximum 400mA, which is well over twice the expected max.

1.4 Additional Sub Components

1.4.1 Coupling Capacitors For Converters

Decoupling capacitors are used in electronic circuits to shunt high frequency AC noise to ground and allow DC signals to pass unobstructed. The PDM2-S DC-DC converter datasheet recommends shunting a 2.2μF capacitor between the input terminals and 10μF between the output terminals for both the 5V and 3.3V models.

1.4.2 Load Resistors For Converters

Without any load current flowing at the output of the DC-DC converter, the output voltage can be deviated from the nominal value. Adding a resistor here allows the required current to flow so voltage can stay at the correct level. Both 5V and 3V models have a minimum output current of 40mA, so load resistors should be 5V/0.04A = 125 Ohms and 3.3V/0.04A = 82.5 Ohms for the 5V and 3.3V models respectively.

1.4.3 Filter Inductors For Converters

The DC-DC converter manufacturers recommend using source inductors to meet electromagnetic compatibility (EMC) standards. Their datasheet recommends 6.8 microHenry inductors in the 3.3V and 5V cases.

1.4.4 Power Intake Protection

Fuses can provide valuable protection against short circuits and can limit damage to PCB’s. Maximum power drawn from our board is expected to be 5V*0.1A + 3.3V*0.170A = 1 W (approximately). This equates to less than 100mA input current at 12V, so we selected a 500mA fuse. This should not endanger the circuit of breaking the fuse while operating at high power, but

will protect it in the event of a short. Our circuit also features a TVS diode at the input, which provides additional protection against transients and overvoltage threats.

These parts have yet to be selected and would need to be done moving forward.

1.4.5 XBee Components

It's unclear whether a high or low signal will silence the reset signal, so our circuit provides access to both 3.3V and GND just in case. The schematic presents this as a resistor, but it will actually be a jumper or a short straight between the two signals. Additional testing will be required to see which one is necessary. The XBee hardware reference manual recommends placing at least a 1.0 microFarad and a 47 picoFarad capacitor at Vin, and an additional 10 microFarad capacitor if using the programmable modules. We decided on 10 and 0.1 microFarad capacitors in parallel.

1.5 Circuit

Figure 7. shows our final schematic as designed in Altium. This schematic was used to generate the PCB in section 1.6. The inputs at B. and C. come from the electric vehicle through external Molex connectors. Regarding B, the fuse separates +12V from the input port and the TVS diode is shunted to the 0V input. +12V splits to each DC-DC converter and passes through input protection circuits before being stepped down to +5V and +3.3V. +5V powers only the Teensy4.1, while +3.3V powers the CAN IC and XBee. CANH and CANL proceed straight to the CAN IC module after being shunted by 60 Ohms. The CAN IC outputs CRX1 and CTX1 to the Teensy, which then passes these processed signals to XBee as RX and TX respectively. The coin cell is a standalone +3V power source whose only interaction is with the special VBAT pin located on the center rail of Teensy pins. This schematic depicts a device that will only transmit signals, and no effort has gone into receiving signals from the remote monitoring station. Because the XBee is a transceiver, it is possible that it will inadvertently pick up lingering signals, but these will be ignored. Certain other parts, such Micro-SD cards, additional RAM, the XBee antenna, Molex Connectors, and additional RF cable between the XBee and antenna, are not featured here because they don't impact the central power or signal pathways.

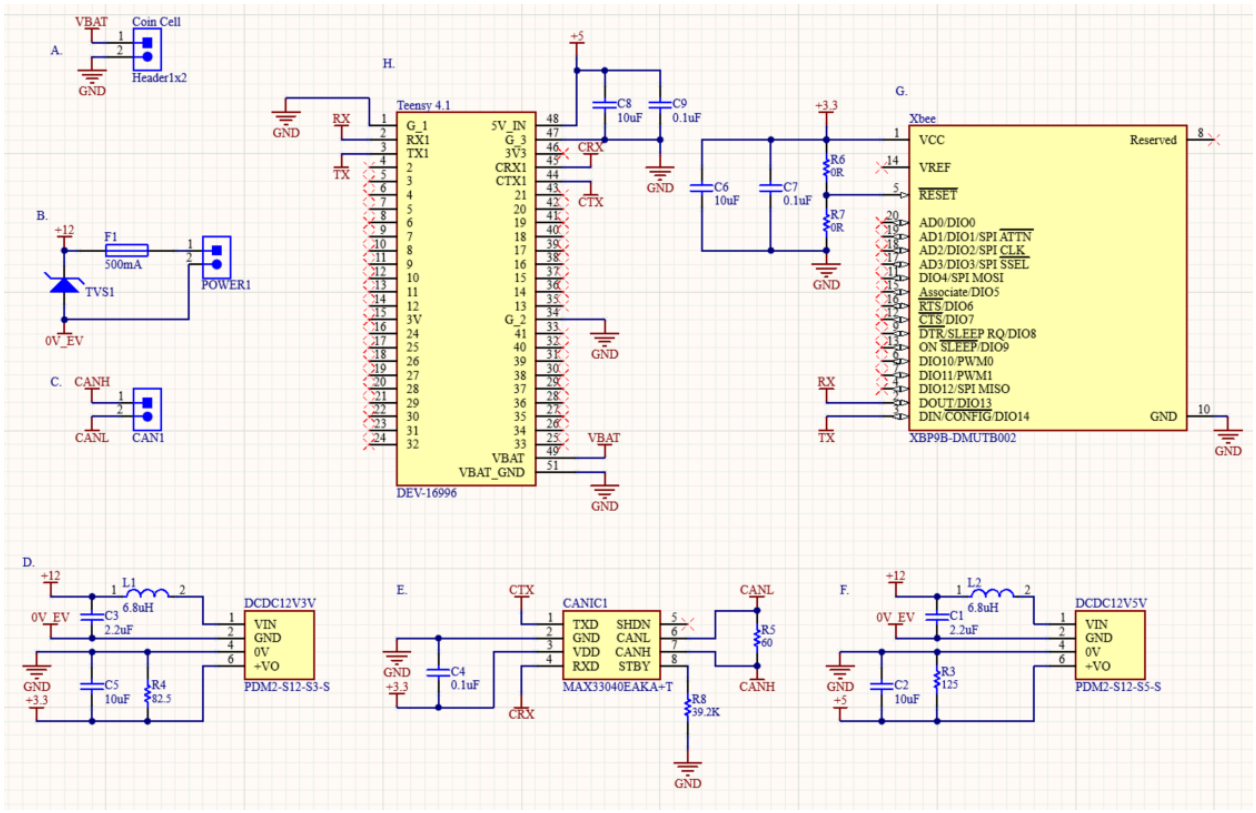


Figure 7: Final Schematic. Standalone regions from top left going counter clockwise: **A.** coin cell for keeping accurate timestamps between driving sessions; **B**. input power circuit with fuse and TVS diode; **C**. CAN High and CAN Low input port; **D**. 12V to 3.3V DC-DC converter with input protection and a designated load resistor at R4; **E**. CAN IC with input protection, recommended resistor at pin 8 for STBY, and recommended 60 Ohm resistor between CAN High and CAN Low; **F**. 12V to 5V DC-DC converter with input protection and load resistor at R3; **G**. XBee module with input protection at VCC, and pathways to VCC and GND from Not(RESET) at pin 5; **H**. Teensy 4.1 microcontroller with input protection identical to the XBee.

1.6 PCB Development

1.6.1 Design Considerations

1.6.1.1 Placement of Components

Decoupling capacitors were placed as close as possible to component I/O ports in order to minimize the area of conductive loops, thereby minimizing EMF interference from stray inductance. Our board, shown in figures 8 and 9, is two sided. The large red and blue portions are GND sections which came from creating a polygon pour on the top and bottom. The DC-DC converters are expected to be the major sources of heat that is generated while transmitting signals, so these two parts were distanced as much as possible to better handle thermal energy. The coin cell is placed underneath simply out of convenience. The pre-made part we selected makes it easy to swap 3V coin cells if one ever dies. Using a two sided board removed any danger of a short circuit and allowed us to shrink the size of the board.

1.6.1.2 Ease of Use Functions

The orientation of parts was carefully chosen to maximize ease of use by Terps Racing members. Regarding the red image, the Teensy is centrally located with no hindrance on the left for plugging in a micro-USB cable even while the board is still inside the PCB. There is no hindrance on the right for installing and uninstalling a micro-SD, again even while the Teensy is still installed in the PCB. This board was purposefully designed so that the Teensy is removable. Pins are soldered straight onto the Teensy, but the PCB will have all female ports to receive it, so if team members want to remove the Teensy for any reason, they don't need to remove the PCB from the waterproof enclosure or desolder anything. Most parts were placed on the top layer to make visual inspection easier. If an issue is suspected with the CAN IC or coin cell, the board will need to be removed from the enclosure.

1.6.3.2 Final PCB Design Remarks

It was not possible to print the PCB during the semester. Our team had only limited experience with Altium, so it took longer than expected to design the schematic and PCB. Additionally, our final review with the TA's and Dr. Resalayyan will not take place until after this final report is submitted, so any changes that happen as a result of that conversation cannot be reflected here. There are several things we are wary of in our design. We assume that the SMA connection on the XBee will meet a short RF cable (probably 1-3 inches) before reaching the antenna. Due to it not being possible to print the board or install it in any kind of enclosure, it was impossible to do any tests on how this cable might impact signal fidelity. The input ports labeled POWER and CAN are intentionally not facing the same direction. We want to make it as clear as possible that these ports are not interchangeable because mixing them up would almost certainly destroy the board, or at least the CAN IC.

1.6.2 Final PCB

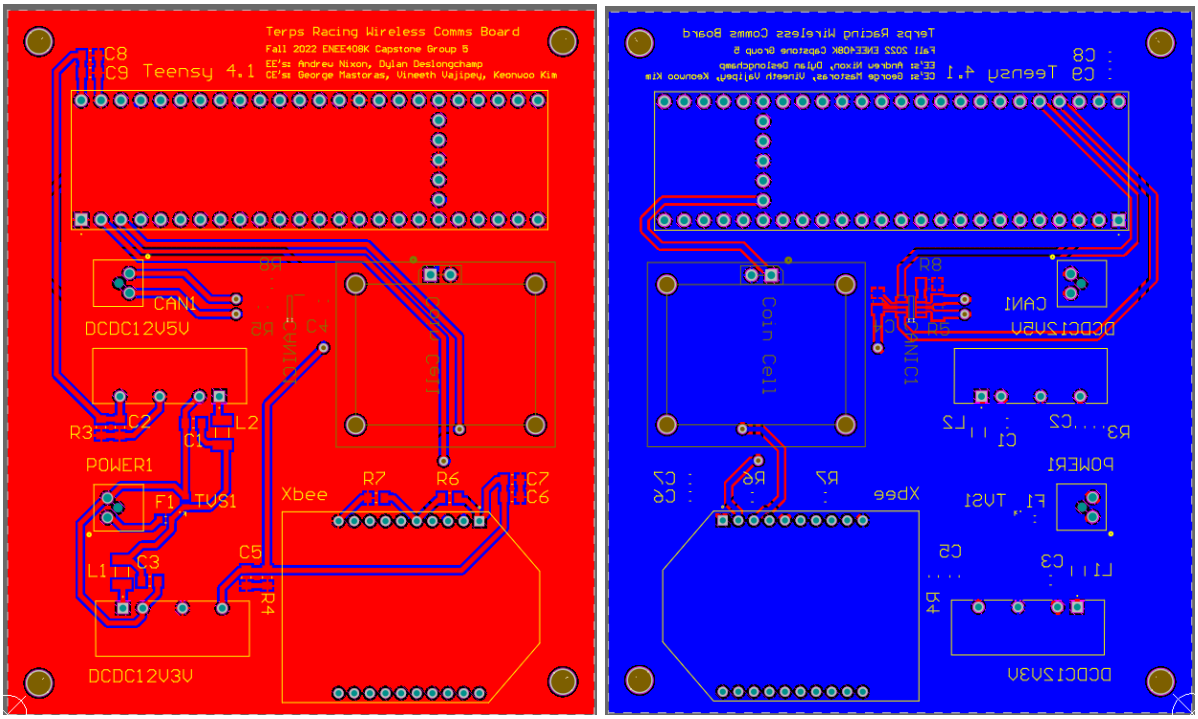


Figure 8. The front (red) and back (blue) of the final PCB in 2D layout mode.

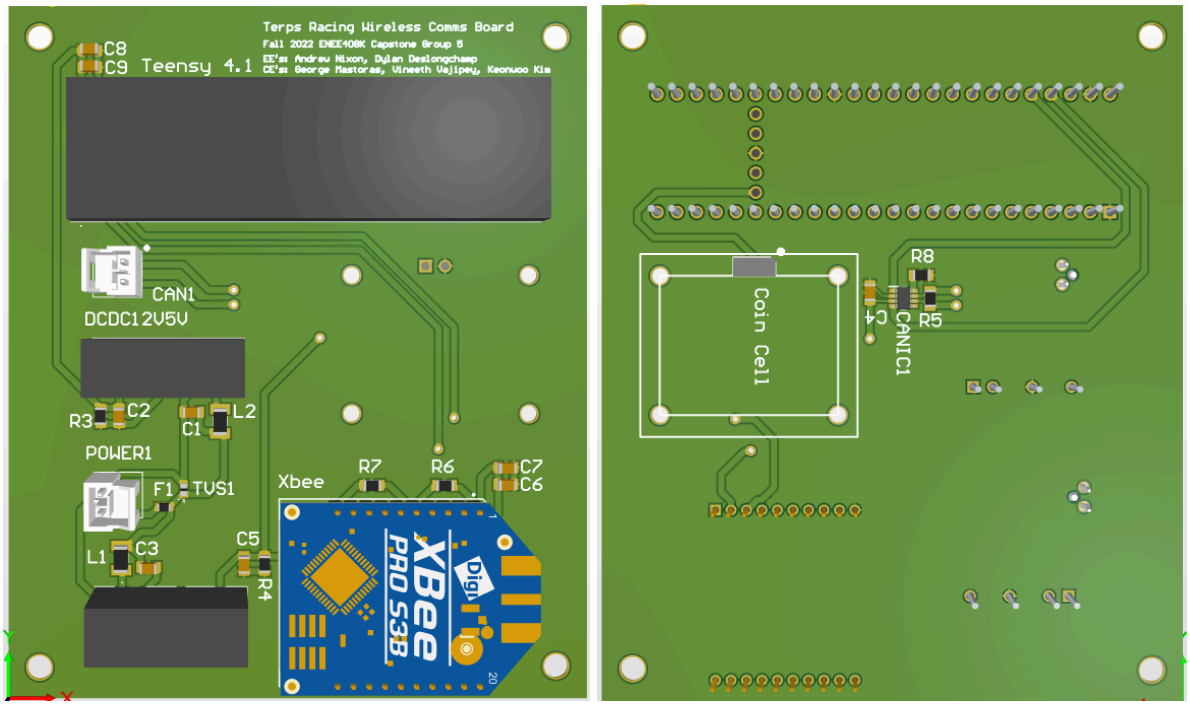


Figure 9. The front (left) and back (right) of the final PCB in 3D layout mode.

1.7 Tools

1.7.1 Soldering

- When two or more components are attached via melting a solder and then allowing it to harden locking components in place. This involves using solder, solder paste, solder iron, light, fan, and components to make it occur.
- As stated previously we were unable to print our PCB in time to allow us to solder our components on to it. We would have soldered almost all main and sub components onto the PCB besides the coin cell.

1.7.2 Altium

- Altium is a software that allows electronic design automation software to engineers who design printed circuit boards.
- Altium allows for schematics, 2D and 3D PCB design and has some footprints and 3D models available online
- We were given the TerpsRacing Altium library to allow us to not have to create every component from scratch.
- None of our team had any experience with Altium before this semester and so this was one of the greatest hurdles for us to overcome.

1.8 Hardware Demo

In Appendix (A) and (B) you can see our Final High Level Block Diagram and our demo Final High Level Block Diagram.

- The final demo hardware is slightly different from our final PCB hardware.
- We did not have the final PCB and we did not have any CAN input or voltage input from the car.
- The parts in red in (A) get changed to the green parts in (B) to allow for our software demo to run. This includes:
    - Taking power from computers and not 12V batteries to converters.
    - Using breadboards instead of a PCB for connections
    - Using a 3rd Teensy to create and send CAN data (Teensy is acting like the car)
    - Removal of RTC due to lack of time to implement

2. Software

2.1 Overview

2.1.1 Goal

The primary function of the software is to provide optimal transmission of CAN frames wirelessly between the TerpsRacing Car and the team's control center using RF radios. In theory, the EV system collects data from sensors via CAN modules and sends the frames to the control center, via point-to-point communication, using the XBee RF radios. The receiver then allows the control center to collect and interpret data, to be used in real time by the racing team.

2.1.2 TerpsRacing / FSAE Considerations

Our goal in selecting hardware was to ensure ease of design for the hardware team, but to also have sufficient processing power and transmission speed of the microcontroller and radio to be able to send at least ten points of data for fifteen different sensors per second.In addition to needing a certain speed of communication, for both radio module and microcontroller, we wanted to keep a small profile allowing the PCB to take up minimal space and contributing little to the weight of the vehicle.

2.1.3 Moving Objectives Throughout Semester

Our end goal and target for the class, specifically for the software team, changed throughout the semester due to different availability and needs of the Terps Racing team. While this susbsytem was originally intended to serve as the primary communication between the onboard CAN system and the racing team, a generous donor supplied the team with a similar product, which allowed the Terps Racing team to cease development of their CAN system. This impeded the software team's development of the originally desired software, since the originally considered testing environment was no longer available.

Parts of these sections will reference calculations and designs with the originally intended design, however, there will also be discussions concerning the replication of the CAN signal and the development of a CAN testing and development environment.

The original goal was to show the one hundred fifty points of transmission. We selected hardware to ensure this was possible, however, our end goal became to verify we were capable of transmitting a single frame within. This is a significant change, however it is justified. Showing one hundred fifty points of data transmission is still possible, and shown later.

2.1.4 Calculations and Max Transmission with Goals

As stated earlier, the microcontroller, CAN transceiver and RF module we decided upon were the Teensy 4.1, MAX33040EAKA+T CAN integrated circuit, and XBee S2C Pro. With our originally intended goal of one hundred fifty points of data transmission, these were the best, most affordable options.

The Teensy operates at a speed of 600MHz. The Teensy's serial ports along with its CAN Tx and Rx ports can read at a speed of up to 6Mb/sec. The final size of our CAN frame, which is slightly longer than an actual frame, was 24 bytes of 192 bits [5]. This number was based on the size of the CAN_message_t object from the FlexCan_T4 library. However, normal CAN frames are not of this size. Normally the entire length of a can frame is between 45 - 111 bits. Our communication, although not following the normal standards of communication, goes above and beyond the requirements as we have developed and tested communication of 24 bytes per frame. This is above the 45-111 bits that normal CAN frames are.

Technically, this 6Mb/sec maximum communication rate can be pushed even further if the maximum clock speed is bypassed. However, this was not necessary as Teensy is not the bottleneck for the maximum rate. The other two external devices, each at a rate of 1 Mb/sec, limited the maximum transmission rate.

After verifying the microcontroller can match the CAN IC speed, we had to decide upon a RF module. We chose the XBee brand because they had accompanying software, were cheap and fairly easy to learn to use. The XBee we selected was the XBee S2c Pro. It sends data at a max of 250Kbps and Serially 1MBps. It also operates at a frequency of 2.4 GHz , and considering our goal of sending 150 of these a second, the XBee's max of 300+ a second [6] is more than enough and allows for some delay which is inevitable in RF communications.

To determine the maximum speed of communication our system could potentially achieve, we must consider the maximum communication speeds of each individual component, and the minimum speed amongst them. For our demonstration, the Teensy's speed is more than enough for the XBeeand SN65HVH23's maximum transmission rate of 1 Mb/sec because the Teensy's maximum rate is 6Mb/sec. Thus, the maximum CAN transmission rate ends up being 1,302 CAN messages / sec.

This calculation can be repeated with the components of the PCB board. The XBee and Teensy maximum communication rate stays the same, but the CAN module is much faster, at a rate of 5 Mb/sec. Although this is faster, our final transmission rate remains the same. We can make the most of this advantage by updating the CAN information quicker. This means that we would have the most up to date information possible. Another idea that we did not implement was the

idea of packaging the CAN frames together and then sending them as a bundle. This would allow us to receive the data from the CAN modules at 5 Mb/sec and then transmit all that information over the XBee communication at a rate of 1Mb/sec. This would be 5,208 CAN frames/sec transmitted from the CAN bus to the Teensy.

2.2 Development Environment

2.2.1 Hardware for Demos

The Teensy board interfaces with the XBee module through 4 wires: Vcc (3.3V), Ground, Digital-In, and Digital-Out [5]. The communication occurs serially between the DigIn/Out pins of both boards. We used the Teensyduino IDE and Arduino bootloader to program the Teensy 4.1 board. We also used this IDE to import packages and the necessary libraries to interface with the Xbee and CAN modules.

There were several things done to make interfacing and using Teensy much easier.

The first thing was soldering pins into all of our Teensys. The hardware team got this done once Teensys arrived. This allowed us to plug the Teensy into our breadboard directly; this then allowed us to use male-female wires to connect to the radio modules as well as the CAN ICs.

The other thing done to Teensys was the soldering of additional RAM memory. The memory was added to the RAM to make testing easier to manage, memory-wise. Once the RAM was soldered onto the boards, a memory test from Paul Stoffregen on GitHub called *teensy41_psram_memtest* [10]. After running this test on the Teensys we were able to verify that the RAM was soldered correctly and it was functioning properly.

These two hardware changes were essential in the development of the software.

2.2.2 Software - XCTU

The XCTU software is a program designed by the creators of our chosen RF module, *Digi*, to help with programming the XBee radio module as well as provide a serial testing environment that simplifies testing and basic communication [11].

The software was used to configure the XBee to AT (transparent mode) and also set receiver and sender address. Our first milestone was to show communication between the two radios using the serial monitor provided within the XCTU software. This came with added functionality that would select the proper baud rate and format message, something that had to be manually done in the Arduino IDE. To communicate with XCTU, the radios were connected to the serial inputs and outputs of our microcontroller. XCTU, similar to writing in any serial monitor, provided us

with the ability to send packets, both individually and repeating on a given interval. This allowed us to test various distances and was helpful in early stages of testing.

The XCTU software was always intended to be a stepping stone to our final demonstration and for that reason is not used in the final demo, as we were able to get the same functionality with the Arduino IDE.

2.2.3 Software - Arduino

The Arduino IDE was a core factor in our decision to select the Teensy 4.1 as our microcontroller. Compared to other microcontrollers, Ardunio had a built-in bridge for Teensy functionality called Teensyduino [12]. Downloading and installation of this IDE extension was extremely simple and allowed for immediate use of the Teensy as a normal Arduino development board. In addition to Teensyduino, came the Teensy bootloader. This small application made programming the Teensy much more simple and prevented code that would cause infinite loops from running, making testing much less prone to errors.

In addition to ease of use with our microcontroller, the Arduino IDE also offered us a serial monitor with variable baud rates. This allowed us to choose the optimal baud rate for our system and still have the messages displayed. This was done automatically in XCTU, but the functionality was built into Arduino.

2.2.4 External Libraries

There were several external libraries used throughout the project that made testing much easier. One of these libraries was automatically loaded through the installation of Teensyduino. The other and most important for the full demonstration development was the FlexCAN_T4 library [13]. This allowed us to use the CAN IC's to both replicate CAN frame transmission as well as read it.

2.3 XBee Development (Mid Semester)

2.3.1 Goal

For the mid-semester report, our goal was to demonstrate wireless communication between two Teensy 4.1 boards via XBee. We used XCTU to send, read and display packets that were also made within the software.

2.3.2 XBee Mode - AT API

The XBee radio has two different modes of operation, API and AT (transparent). In API mode, you are allowed to make a web of radio modules. Any given module can send and receive from several other modules. There is an available XBee Teensy library that works with API mode,

however there is much more to using the radios in API mode than AT mode and for certain basic applications, this is not necessary. Transparent mode (AT), is a much simpler mode of communication that is used between only two radios. Additionally, the address of the sender and receiver is programmed using XCTU software, unlike API mode, where the receiver and sender addresses can be changed constantly. For our goals of only point-to-point communication between two radios, AT mode was the clear choice and the one we used to implement our software.

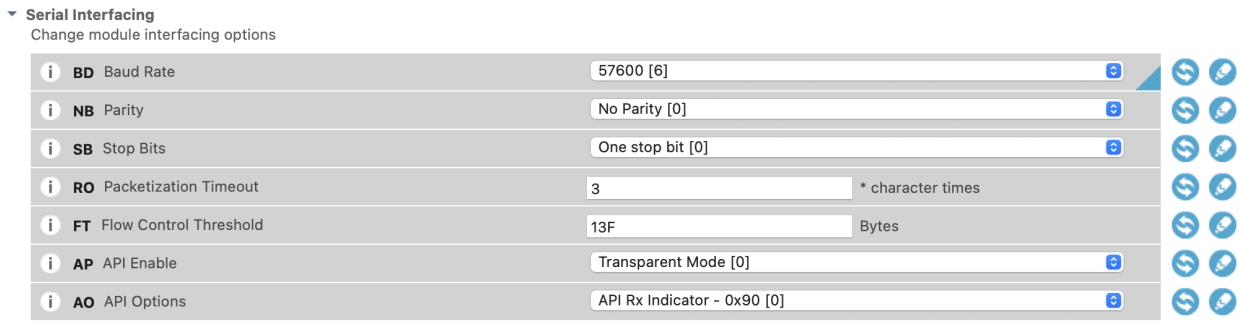


Figure 10: Shows the XCTU screen where API mode was disabled (AT enabled) [XCTU SS]

2.3.3 XBee Programming - XCTU Setup

The first step in programming the XBee modules were to locate them using XCTU software. Given our hardware connections, all we had to do was use the software to locate each radio given the port they were connected to. After getting our radios recognized on XCTU we were able to program them accordingly.

After deciding upon AT mode for our XBee, the only programming changes we had to make on the XBee's were to change the names to match the corresponding functions, (receiver and sender) and to change the destination address accordingly. When programming the receiver module, we changed the destination address to the same address that we programmed the sender to transmit to. In our case we chose FFFF, but this did not make any difference. After the names were changed and the writing and receiving addresses were changed appropriately, the radios were programmed (written to). After that anything written by our sender radio would go directly to the receiver.

2.3.4 Demonstration Setup

Once our radios were recognized in XCTU, the next step in our first demonstration (milestone) was to write the software that would allow the sender radio to get data from serial input and transmit said data to be read, again through serial input, by the receiver.

The code's purpose was to have the sender XBee waiting for serial input from the user in the form of XCTU packets. The packets were made in the software and allowed the user to send a

hardcoded byte array (String). Once there were bytes to be read from the Serial monitor, the microcontroller read those bits and wrote them to the XBee as the XBee became available for writing. In other words, the sender radio did not wait for there to be enough bytes to send the whole message, but rather sent it in little chunks. The code for the receiver, similar to sender, read the bytes in little chunks as they were received. Once all the bytes were received, the string was displayed on the serial monitor, granted the serial monitor was available.

Since the XBees were programmed for one to receive and the other to send, the same code was uploaded to each Teensy-RF system.

2.3.5 Code and Explanation

```
#define XBeeSerial Serial1

unsigned long baud = 19200;

const int led_pin = 13;

const int reset_pin = 4;

const int buffer_len = 80;

byte buffer[buffer_len];

void setup() {

  pinMode(led_pin, OUTPUT);

  digitalWrite(led_pin, LOW);

  digitalWrite(reset_pin, HIGH);

  pinMode(reset_pin, OUTPUT);

  // being serial communication

  Serial.begin(baud);

XBeeSerial.begin(baud);

}

void loop (){

  unsigned char dtr;

  unsigned char data_terminal_ready;

  int read_var, write_var, data;

  unsigned char prev_dtr = 0;
```

We first defined a Serial Object that functions as the connection between Teensy and XBee as Serial1. This object is used throughout the entire demo. You also can see variables for baud rate, led pin, reset pin, buffer length, and a character array for the actual buffer. The baud rate determines the speed at which the Teensy communicates with the computer.These baud rates were chosen for this demonstration because that is XCTU's native speed. After that, both the Serial and XBee serial object were begun with the selected baud rate, which allowed us to access them through ports on the computer. The led pin was used throughout the demonstration to assist

with testing visually, but is irrelevant for the actual demonstration. The reset pin is crucial as it allows us to keep the Teensy awake and avoid stalls.

There are additional variables (read_var, write_var) declared afterwards that are responsible for data validation and used in various helper methods explained below.

Sender Code

```
// first check if there is data coming from the Serial Monitor
(transmitter)

  read_var = Serial.available();

  if (read_var > 0) {

    write_var = XBeeSerial.availableForWrite();

    if (write_var > 0) {

      if (read_var > write_var) read_var = write_var;

      if (read_var > 80) read_var = 80;

      data = Serial.readBytes((char *)buffer, read_var);

      XBeeSerial.write(buffer, data);

    }

  }
```

The first two conditional checks are Serial.available and XBeeSerial.availableForWrite. Both of these methods must return a value greater than 0 (both methods return the number of bytes available to be read or written). If both of these conditions are met, the length is set to max if it is greater and then the Data is read from the Serial monitor. This was user input in the form of packets using the XCTU software. The last thing is to take that read data from the user (from the serial) and write it to the XBEE. Since we are using transparent mode, the XBee essentially serve as a tunnel. So writing to this XBee means that we can read from the other.

Receiver Code

```
// check if there is data going to receiver
read_var = XBeeSerial.available();
if (read_var > 0) {
  write_var = Serial.availableForWrite();
  if (write_var > 0) {
    if (read_var > write_var) read_var = write_var;
    if (read_var > 80) read_var = 80;
    data = XBeeSerial.readBytes((char *)buffer, read_var);
    Serial.write(buffer, data);
    digitalWrite(led_pin, HIGH);
  }
}
```

The receiver code is very similar to the sender with a few slight differences. There are the two of the same serial available checks, meaning the receiver will not try to receive unless the Serial is available to write to and the XBee is also available and receiving data. The next step is to do length validation again to ensure the data is not greater than the buffer size. The last step is to determine the size of the data to be read and displayed using .readBytes on the XBee and then display the code by writing to the Serial monitor using Serial.write.

```
// Helper Methods
// keeps teensy awake
  dtr = Serial.dtr();
  if (dtr && !prev_dtr) {
    digitalWrite(reset_pin, LOW);
    delayMicroseconds(250);
    digitalWrite(reset_pin, HIGH);
  }
  prev_dtr = dtr;
  // update baud if not correct
  // account for 2.5% error given by teensy
  // baud rate
  if (Serial.baud() != baud) {
    baud = Serial.baud();
    if (baud == 57600) {
      XBeeSerial.begin(58824);
    } else {
      XBeeSerial.begin(baud);
    }
  }
```

Both of these helper methods are crucial to this program and ensuring our two radios are capable of communicating consistently. The first method is a data terminal ready check. It essentially serves to keep the Teensy awake, by resetting it, if data was not recently received because if data was received the Teensy would not sleep.

The second method is a baud rate helper function. The Arduino bootloader, which is the software that allows us to upload a program to the Teensy, communicates at 58824 baud, which is a 2.1% error when using a baud rate of 57600. Additionally, teensyduino has its own error when configuring UART to closest baud rate, it uses 57143 which is 0.8% error [14]. Together this is over the 2.5% error serial communications can tolerate so it is too large. Therefore instead of setting to 57600, set to 58824 (same as bootloader) to account for that error and have sufficient baud rate.

2.3.6 Problems

There were a few problems with the development of the XBee demonstration. Some of these problems were hurdles in simply learning to use new software, such as learning the nuances of teensy bootloader, learning to use XCTU and understanding the differences between the Arduino IDE and XCTU in sending and receiving serial data. The most frustrating part of this demonstration development were difficulties in getting the XCTU software to recognize connected modules. For reasons still unknown to us, this was quite problematic. There were times where things would work just fine and seemingly break for no reason, only to randomly fix themselves later. XCTU's solution would be to reprogram and reset the radios constantly, however this was hardly ever the fix. We are still unaware of why these problems arose, however, they were easily dealt with, albeit quite frustrating.

2.4 CAN Development

2.4.1 Theoretical and Practical Goal

Our CAN subsystem's primary function was to retrieve data from the car via the CAN bus and communicate it with our car-side Teensy-XBee module. This subsystem constantly receives data from the car and updates global variables representing the sensor's current value only when necessary. The updated data would then be packaged and sent over the XBee, displaying the data for the control team.

As previously stated, the original (theoretical) goal of the CAN subsystem changed drastically. We did not have access to any EV CAN bus data, so our focus for this demonstration shifted from reading and interpreting CAN frames from an EV and doing some data manipulation, to using an additional Teensy to replicate CAN frames and successfully transmit them from one Teensy to another using two CAN transceivers. This was a drastic change, but a result of something outside of our power.

2.4.2 CAN Frame Theoretical

The Controller Area Network(CAN) system consists of several nodes, which are sensors attached to a CAN transceiver. These different nodes are connected together and offer immediate, interrupt style data sending . Only one node can update its data at any given time, and this is determined by the priority of the component. After the highest priority component has updated, the rest are given a chance to update as well [15].

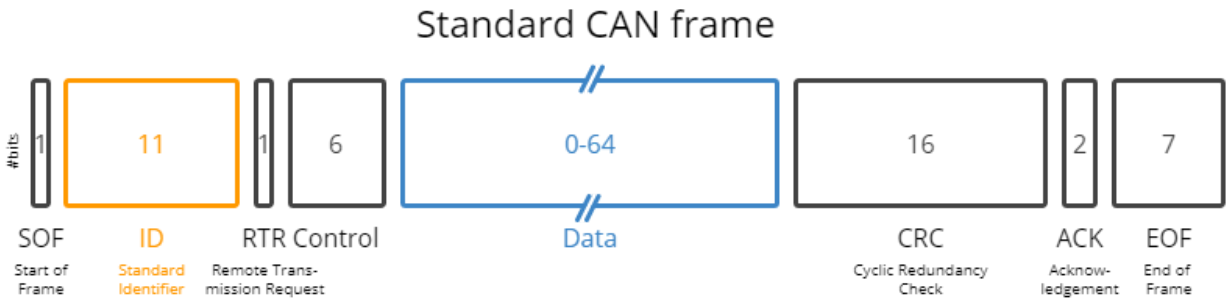


Figure 11, Graphical demonstration of a standard CAN frame [15]

A standard CAN message takes the form of the CAN Frame. This a 111 or 45 bit number (CAN FD vs CAN Standard) that contains several bits for control and optimizations. The majority of the bits are for the data transmission and ID. Based on the CAN frame data from the car can be extrapolated and viewed as desired. This is done by reading CAN L & H as a differential line using a CAN transceiver and some type of microcontroller. In our original design, the goal was to interpret these CAN frames within the car, using non blocking functions, and update them as frequently as ten times a second. The original data points were fifteen different sensor values that can be seen in appendix figure (C). In our final demonstration, we did not concern ourselves with these sensor values at all.

2.4.3 FLEXCAN_T4 - Practical CAN Frame

Following the changes in our goal for the CAN development, we had to make changes in the software and libraries used in our code implementation. Rather than receive CAN frames from an EV, we had to implement a third teensy that would send CAN frames to our original *sender* XBee.

To do this, we used a Github library called FlexCAN_T4. Using this library, we were able to replicate a CAN message, send it and receive it. These CAN frames are slightly larger than legitimate CAN frames, however, as previously shown in calculations in section 2.1.4, the XBee was more than capable of sending enough messages given these larger CAN frames.

Example CAN Frames can be seen in Appendix (J).

2.4.4 Demonstration Setup and Goal

Our CAN communication will utilize the SN65HVD230 CAN IC. This board interfaces with the Teensy module via four ports, Vcc (3.3 V), Ground, CAN Tx, and CAN Rx [16]. The Teensy sends and receives serial data across the CAN Tx/Rx pins at a rate of up to 6Mbp/s, however our selected demonstration CAN IC maxes out at 1Mbp/s. The communication between CAN modules occurs on two additional pins: CAN H & CAN L. The wiring diagram for the CAN demonstration is seen in figure (G) located in the appendix.

2.4.5 Code and Explanation

To demonstrate our working CAN system we constructed a system of two Teensy modules and two CAN boards, as pictured in the block diagram (G). One Teensy was designated as the transmitter and was loaded with a program that sent data through the CAN port. The other was loaded with a receiver program that waited for CAN packets and printed them to the serial monitor. Below are some annotated code snippets.

Setup Code

```
#include <FlexCAN_T4.h>

FlexCAN_T4<CAN1, RX_SIZE_256, TX_SIZE_16> can1;

CAN_message_t msg;

# in setup()

  can1.begin();

  can1.setBaudRate(250000);
```

The setup code simply requires initializing the CAN bus and creating a blank CAN_message object to be used for the CAN frame. Additionally, the CAN transceiver begins serial communication and sets BaudRate to a high enough speed to handle the CAN bus' update speed.

Transmitter Code

```
# in setup()

msg.id = 0x100;

msg.len = 8;

msg.buf[0] = 10;

msg.buf[1] = 20;

msg.buf[2] = 0;

msg.buf[3] = 100;

msg.buf[4] = 128;

msg.buf[5] = 64;

msg.buf[6] = 32;

msg.buf[7] = 16;

# in loop()

can1.write(msg);
```

A CAN frame is hardcoded as seen above and then the write method is called to send the CAN frame through the teensy to the CAN transceiver.

Receiver Code

```
#In loop()
if ( can1.read(msg) ) {
    Serial.print("CAN1 ");
    Serial.print("MB: "); Serial.print(msg.mb);
    Serial.print("  ID: 0x"); Serial.print(msg.id, HEX );
    Serial.print("  EXT: "); Serial.print(msg.flags.extended );
    Serial.print("  LEN: "); Serial.print(msg.len);
    Serial.print(" DATA: ");
    for ( uint8_t i = 0; i < 8; i++ ) {
      Serial.print(msg.buf[i]); Serial.print(" ");
    }
    Serial.print("  TS: "); Serial.println(msg.timestamp);
  }
```

With the first CAN transceiver containing the frame, this section of code takes the Frame through another transceiver and interprets the data. There is a data validation check, to first check if the CAN transceiver has a message. If there is a message, it is read and the individual components of the CAN frame are displayed on the Serial monitor.

2.5 XBee with CAN Development

2.5.1 Goal

The final demonstration combines the two previous ones to successfully transmit a CAN frame (or multiple frames) from one point to another wirelessly, using XBee's to send and a microcontroller to read and display. The final data will not come from the actual EV as that CAN system is not ready or readily available to us for testing. Therefore, we will use Teensy and hardcode the data, either in the code directly, or load it via a text file located on the onboard SD card. These hard coded frames will then be serialized and sent to the control center to be read and displayed.

2.5.2 Demonstration Setup

The setup for our demonstration involves three Teensy 4.1 microcontrollers, two XBee modules, and two CAN interfaces. Our first sender Teensy only populated a CAN frame and communicated it to the second Teensy. This second module receives that information and converts it to a buffer. This buffer is then communicated via XBee to the final Teensy. The Teensy microcontrollers are connected to the CAN ICs with Vcc, Ground, and CANTx/Rx, while the modules communicate with themselves via CAN_High and CAN_Low. The Teensy and XBee modules are connected similarly, with Vcc, Ground, DigHigh and DigLow. These connections can be referenced as part of the final block diagram in Appendix section E.

2.5.3 Code and Explanation

CAN_Sender: Initialize CAN frame and send.

```
#include <FlexCAN_T4.h>

FlexCAN_T4<CAN1, RX_SIZE_256, TX_SIZE_16> Can1;
CAN_message_t msg;
int t, delta;

void setup() {
  Serial.begin(9600); delay(400);
  Can1.begin();
  Can1.setBaudRate(250000);
  msg.id = 0x100;
  msg.len = 8;
  msg.buf[0] = 10;
  msg.buf[1] = 20;
  msg.buf[2] = 0;
  msg.buf[3] = 100;
  msg.buf[4] = 128;
  msg.buf[5] = 64;
  msg.buf[6] = 32;
  msg.buf[7] = 16;
  delta = 0;
}

void loop() {
  Can1.write(msg);
}
```

This code is very simple and has one primary function: to initialize a singular CAN frame and continually send that frame using Can1.write(); In the setup, we initialize our serial and CAN

communication channel, and then we set the values for the buffer, id, len. The loop just writes the CAN frame to the previously opened communication channel.

CAN_receiver and XBee_sender: Essential code within loop() function.

```
Can1.read(msg);

  if (read_var > 0) {
    flag = true;
  }

  if (flag == true) {

    write_var = XBeeSerial.availableForWrite();

    if (write_var > message.length() + 1) {

        delay(1000);

        Serial.flush();
        if (read_var > write_var) read_var = write_var;
        if (read_var > buffer_len) read_var = buffer_len;

        message = "MB: " + String(0) +  " ID: " + String(msg.id) + "
EXT: 0 LEN: ";
        message += String(msg.len) + " DATA: " + msg.buf[0] + " " +
msg.buf[1] + " ";
        message += msg.buf[2] + " "  + msg.buf[3] + " "  + msg.buf[4]
+ " ";
        message += msg.buf[5] + " "  + msg.buf[6] + " "  +
msg.buf[7];
        byte plain[message.length()];
        message.getBytes(plain, message.length());
        plain[message.length()] = 0x0a;

        XBeeSerial.write(plain, message.length() + 1);

        count = 0;
      }
      Serial.clear();
```

The CAN receiver and XBee sender file is the backbone of our demo. This receives the CAN frame information from the first sender, decodes and converts the frame to a buffer, and then communicates that data over XBee. The code block with flag waits to receive a serial monitor

input before beginning the XBee communications. This is to make sure that initial garbage data is not sent, and to provide the user and demo with more control. Once any message is sent to the Teensyduino serial monitor, the Xbee communication begins sending messages. The CAN message is first read, and the msg variable is updated. Then the message String message variable is updated using the data inside the CAN message variable. This string contains all the data that we received from the CAN frame. This string is then converted to a byte array and the XBeeSerial.write() function is used to send the byte array buffer, with the given length. Another piece of important information is the if statement that checks if the XBee is ready to transmit a message that is greater than the size of the message. Otherwise the XBee would send the message even if the buffer was not completely written.

XBee Receiver: Essential Code for receive and decode buffer to CAN message.

```
#include <FlexCAN_T4.h>
#define XBeeSerial Serial1

CAN_message_t msg;
unsigned long baud = 57600;
const int led_pin = 13;
const int reset_pin = 4;
const int buffer_len = 24;
byte buffer[buffer_len];
unsigned char dtr;
int read_var, write_var, data;
unsigned char prev_dtr;
int count;

void print_buf(){
  for(int i=0; i < buffer_len; i++){
    Serial.print(buffer[i]);
  }
  Serial.println();
}

void loop() {
  unsigned char prev_dtr = 0;

  // check if there is data going to receiver
  read_var = XBeeSerial.available();
  if (read_var > 0) {
    write_var = Serial.availableForWrite();

    if (write_var > 0) {

      if (read_var > write_var) read_var = write_var;
      if (read_var > buffer_len) read_var = buffer_len;

      data = XBeeSerial.readBytes((char *)buffer, read_var);
      Serial.write(buffer,data);
      digitalWrite(led_pin, HIGH);
```

```
    }
  }
  //method that keeps teensy awake
  dtr_method();
  // shown above
  baud_update();
}
```

This code receives the data from the XBee communication. It uses the XBeeSerial.available() function to check if there is any data that can be received via serial communication and then prints the data to the current serial monitor. There is no converting of the buffer into a CAN_message object because that has already been done and transmitted. The data we have sent is the properly formatted data, so all we need to do is to decode it as usual. We use XBeeSerial.readBytes to read and decode the communicated data. We see this through XCTU, but can also be seen through the Arduino IDE.

2.5.5 Results and Problems

Figure 12: Screenshot from XCTU to show the data from our final demonstration.

Throughout the development process for the final demonstration, we ran into many roadblocks. Although we were able to develop the communication demos for XBee - XBee and CAN - CAN relatively easily, the integration of the two communications proved to be difficult. This was primarily due to the different rates of communication and the need to synchronize the availability of the different communication channels. In many iterations of the demo, our project would output incoherent data caused by baud rate and data transmission inconsistencies. This would occur when the XBee communication occurred before there was enough space or if the indexes of the packet for out of bounds. Using additional checks and conditional statements, we were able to keep most of these inconsistencies in check. We believe more work needs to be put

into understanding exactly how the two communication methods are occurring simultaneously to achieve the maximum CAN data transmission rate, which we are not close to achieving.

IV.Conclusion

Collectively our goal was to design and build a device that will communicate sensor data from a FSAE vehicle to a command center during a race over a distance of 1-2 miles. Each of our two subteams worked to achieve their individual goals this semester while maintaining communication and understanding across both teams. While designing the device we had to consider FSAE regulations and desired functionality from the TerpsRacing team, as in the end we wanted the device to be used by TerpsRacing on their FSAE electric vehicle.

We designed the device to be housed in a waterproof box behind the driver's head. This led us to want the box/device to be as compact as possible without giving up functionality and performance of the device. The design is powered via a 12V battery located on the vehicle, which will be stepped down to 5V and 3.3V using DC-DC converters on the PCB. The device gets about 15 different sensor data to send via a CAN bus from the electric vehicle so a CAN IC is included on the PCB. Additionally a Xbee was used as our transceivers as its functionality most directly fit with our desired goals. We decided to use a Teensy 4.1 as our microcontroller as our software team had more familiarity using Arduino coding that had a plug in for the Teensy 4.1 and its functionality was standard to what we needed. All of these components were designed on our PCB and schematic that was created using Altium.

Over the course of the semester we developed and implemented code that would allow us to communicate wirelessly between several teensys. Along the way our goals shifted, pushing us to further develop the software to incorporate a third teensy that replicated the CAN signals originally intended to come from the Terps Racing EV. By the end of the semester, we were able to successfully transmit a replicated CAN frame between two XBee radios. If we were to this project again or further develop it, there would be a significant interest in exploring API mode, as well as actually developing the software to work with a legitimate CAN system. Over the semester we iterated through the engineering and design process to develop different demonstrations that incrementally built up to our final project. We first developed a stable XBee to XBee communication system, and then did the same for CAN - CAN communication. The last step was integrating the two. We took a heavily research based approach to our project as we were not experienced in wireless communications nor familiar with the specific hardware that was used. Ultimately, the software team learned a great deal about developing products for real life applications including the challenges that come with the process.

The software and hardware teams over the semester worked to create multiple demos and PCB designs that achieved the given goal. This progress evolved into a more properly designed PCB in a compacted method, and the final software that combines the earlier demos with CAN bus to transmit sensor values "from the car". By the end of the semester we were able

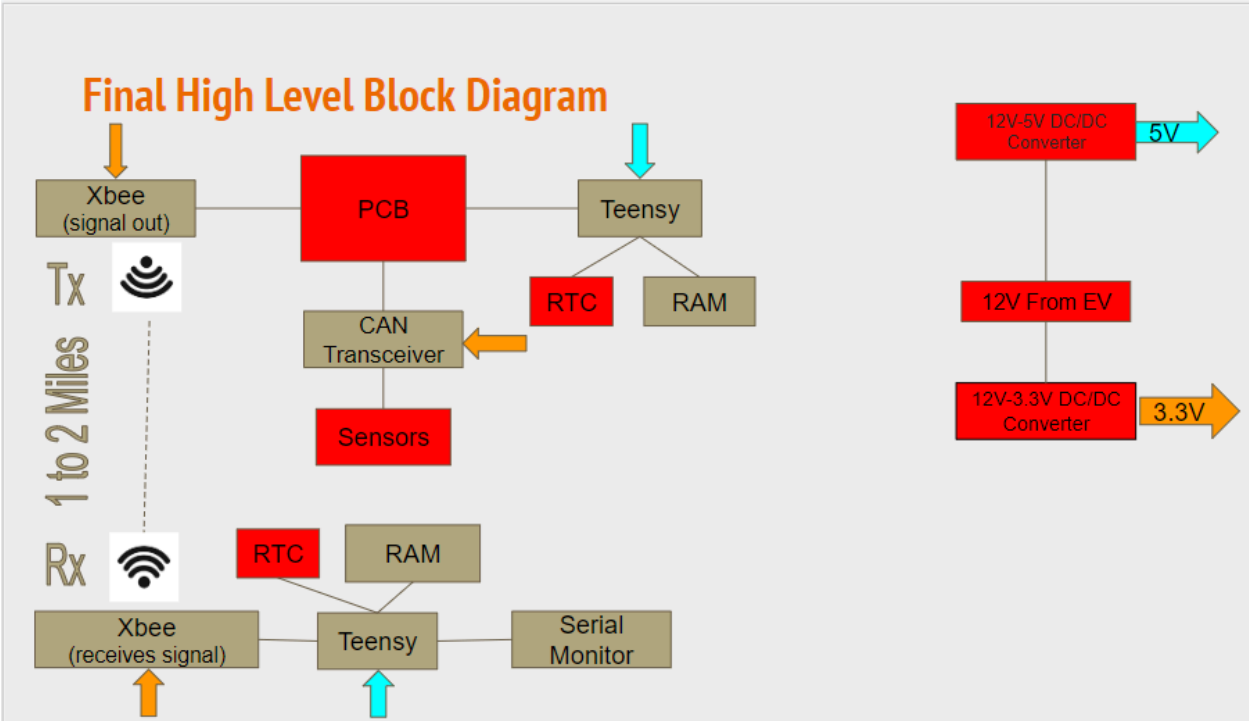
to have a final PCB designed in Altium with schematic and a final software demo showing sending and receiving data from a CAN bus “on the car” to a “control center”. We were unable to print our PCB in time, so we never got the chance to solder components to the PCB or implement and troubleshoot our software onto the final PCB. Moving forward we would send our PCB design off to be printed and then solder and put together our final PCB. As we wait for the PCB to be printed we would pick out a waterproof housing unit for our PCB. Also in this time we would pick out a TVS diode and fuse to use. Once the PCB is finished and in hand then we would begin to implement and test the software before finally handing the device off to TerpsRacing to be installed on the electric vehicle.

V. References

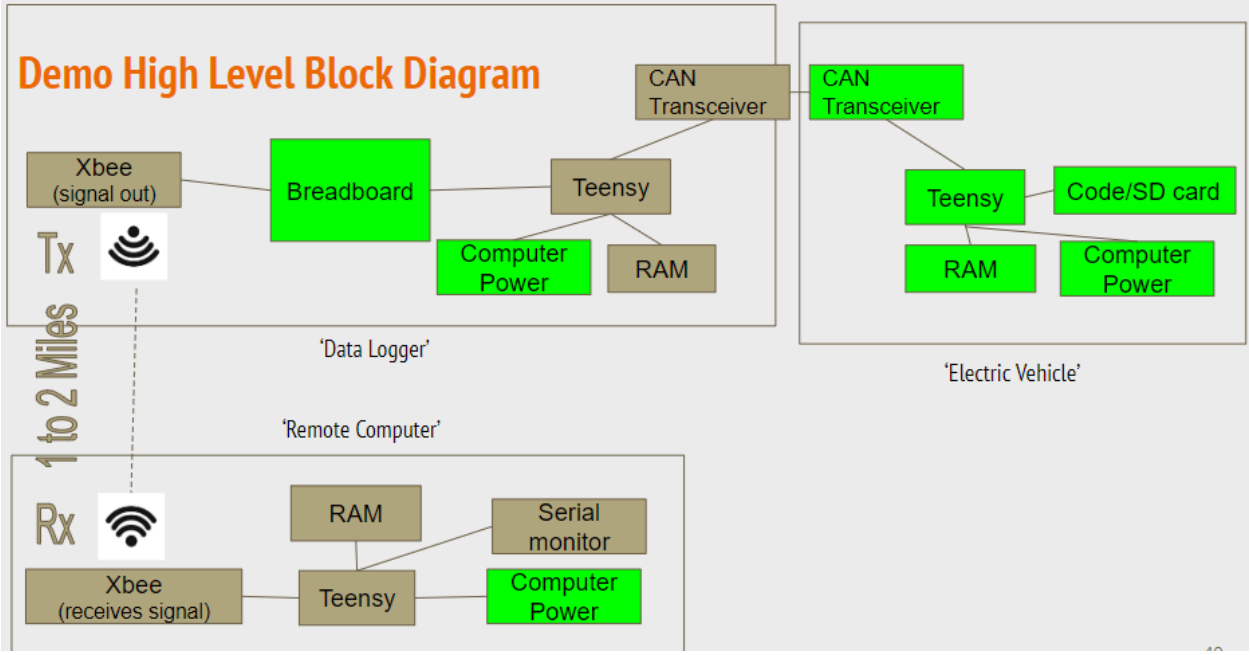
1. “Formula SAE Rules 2023.” SAE International, September 1, 2022. https://www.fsaeonline.com/cdsweb/gen/DownloadDocument.aspx?DocumentID=d2404288-c121-4242-9944-90aa0b1d6a5f.
2. “PDM2-S12-D5-S.” Digikey. Accessed December 18, 2022. https://www.digikey.com/en/products/detail/cui-inc./PDM2-S12-D5-S/4006969?utm_adgroup=DC%20DC%20Converters&utm_source=google&utm_medium=cpc&utm_campaign=Shopping_Product_Power%20Supplies%20-%20Board%20Mount_NEW&utm_term=&utm_content=DC%20DC%20Converters&gclid=CjwKCAiAkfucBhBBEiwAFjbkrx5nC-ubsophFOs6PoPulkSCDoKejbZOSC3b5fZUKtwfsSLASc3m6RoC5xEQAvD_BwE.
3. “Maxim Integrated MAX33040EAKA+T.” Mouser. Accessed December 18, 2022. https://www.mouser.com/ProductDetail/Maxim-Integrated/MAX33040EAKA%2BT?qs=IS%252B4QmGtzzrp4y4r9nEH%252Bg%3D%3D.
4. “1870 Coin Cell Connector.” Digikey. Accessed December 18, 2022. https://www.digikey.com/en/products/detail/adafruit-industries-llc/1870/6827145?utm_adgroup=Adapter%2C%20Breakout%20Boards&utm_source=google&utm_medium=cpc&utm_campaign=Shopping_Product_Prototyping%2C%20Fabrication%20Products_NEW&utm_term=&utm_content=Adapter%2C%20Breakout%20Boards&gclid=CjwKCAjw4c-ZBhAEEiwAZ105RVdCYCfWT4RVa5MtIsnGjfG_CNuxaFEC-8WMTkvF0-0UiS1XatoIUhoCCQMQAvD_BwE.
5. “Teensy 4.1 Development Board.” PJRC. Accessed December 18, 2022. https://www.pjrc.com/store/teensy41.html.
6. “Zigbee Modules - 802.15.4 XBee ZB S2C TH RPSMA Antenna.” Mouser. Accessed December 18, 2022. https://www.mouser.com/ProductDetail/DIGI/XB24CZ7SIT-004?qs=3VJ0tGt%252Bi1w3hGvWrESPVw%3D%3D.
7. “DRAM IoT RAM 64Mb QSPI (X1,X4) SDR 133/84MHz, RBX, 3V, Ind. Temp., SOP8.” Mouser. Accessed December 18, 2022. https://www.mouser.com/ProductDetail/AP-Memory/APS6404L-3SQR-SN?qs=IS%252B4QmGtzzqsn3S5xo%2FEEg%3D%3D&gclid=Cj0KCQjw1vSZBhDuARIsAKZlijQCeCwjCCO4bkf99xsl1ORbNIRVXZ2BXWbEGfUGn9mLy1oDmKZCH8IaAvbUEALw_wcB.
8. “CAN IC Datasheet: MAX33040E/MAX33041E.” Mouser, n.d. https://www.mouser.com/datasheet/2/256/MAX33040E_MAX33041E-1927821.pdf.

9. “Digi XBee ZigBee Datasheet.” Digi, n.d. https://www.digi.com/resources/library/data-sheets/ds_xbee_zigbee?view=fullscreen.
10. “teensy41_psram_memtest” Stoffregen, n.d. https://github.com/PaulStoffregen/teensy41_psram_memtest/blob/master/teensy41_psram_memtest.ino
11. “XCTU User Guide.” Digi, n.d. https://www.digi.com/resources/documentation/digidocs/90001458-13/default.htm
12. “Basic Teensyduino Usage.” PJRC, n.d. https://www.pjrc.com/teensy/td_usage.html
13. “FlexCan Library for Teensy 4.” tonton81, n.d. https://github.com/tonton81/FlexCAN_T4
14. “Using the Hardware Serial Ports.” PJRC, n.d. https://www.pjrc.com/teensy/td_uart.html
15. “Can Bus Explained - A Simple Intro [2022].” CSS Electronics, n.d. https://www.pjrc.com/teensy/td_uart.html
16. “3.3V CAN Transceiver with Standby Mode” Texas Instruments, n.d. https://www.ti.com/product/SN65HVD230

VI.Appendices



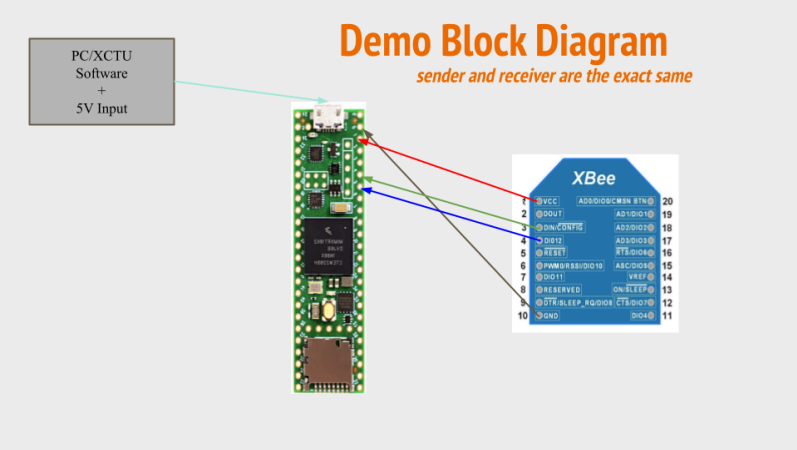
(A) Final High Level Block Diagram



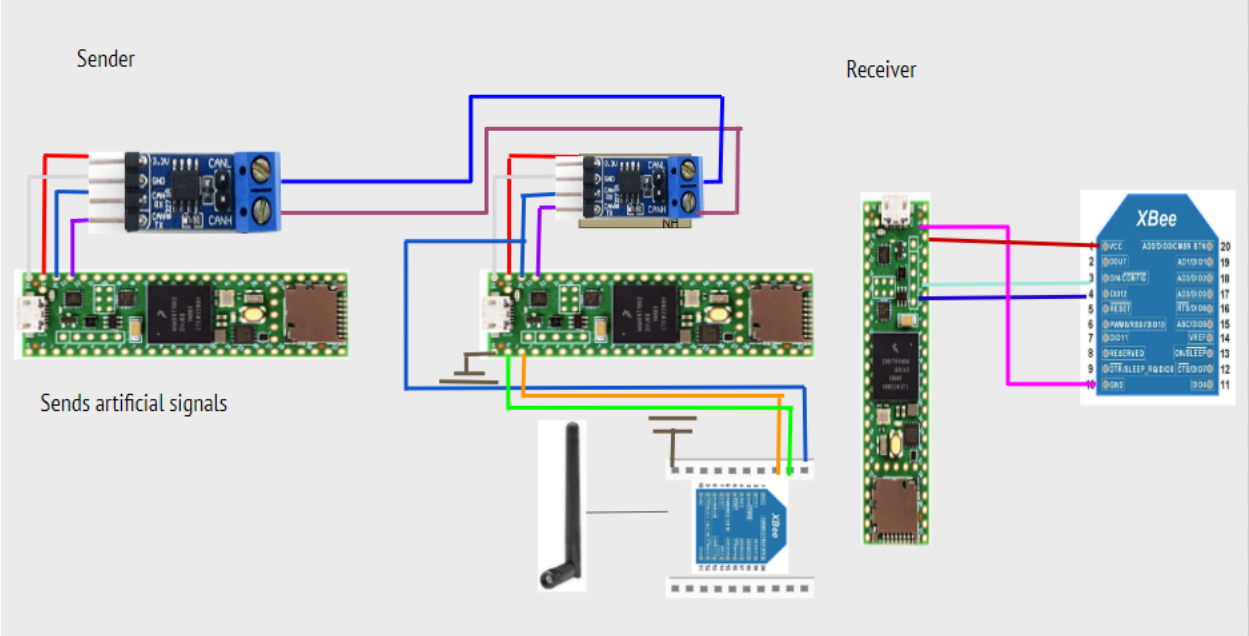
(B) Final Demo High Level Block Diagram

Final Data To Be Collected (One CAN Frame per Updated Sensor value)
Wheel speed
Steering angle
Suspension over travel
Brake Pressure
Battery Temperature
Battery Voltage
Torque
Torque Requested
Apps accelerating pedal position sense
Fault signals
Error Logs
Shut down circuit faults
SDC faults

(C) Table of Data Collected



(D) Demo Block Diagram for mid-semester

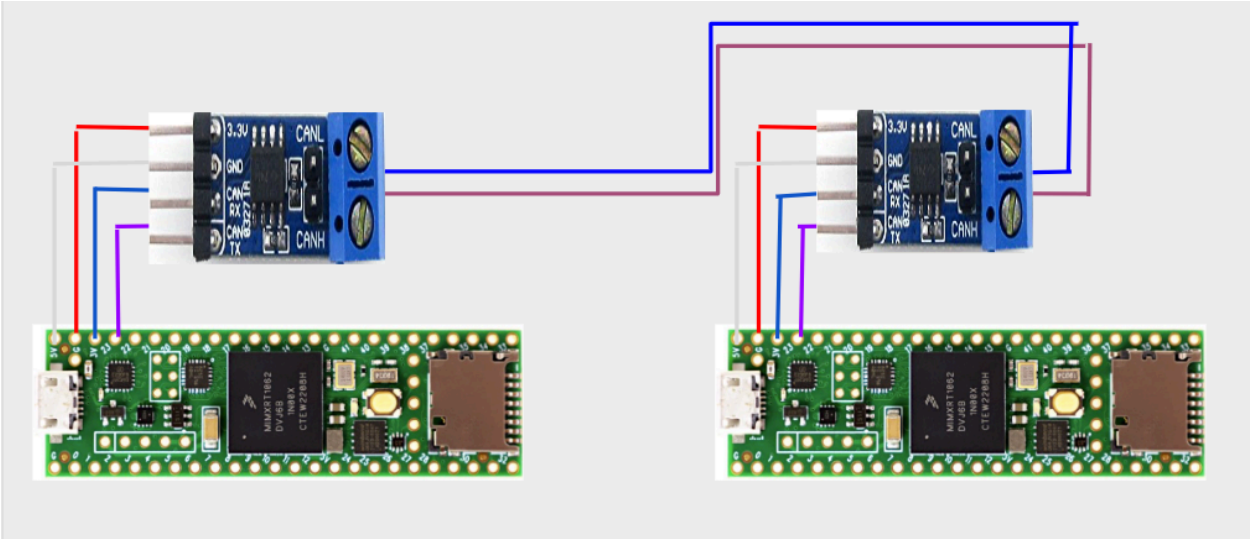


(E) Final Demo Block Diagram

Part	Link	Number Purchased	Notes	Cost (Total)
2.5mm Female Pins	https://www.mouser.com/ProductDetail/TE-Connectivity/1-535541-0?qs=VKrXD49J9ErfIT2bXrgBeA%3D%3D	12	NOT YET PURCHASED, Need to order 12	$2.55 ($30.60)
2mm Female Pins	https://www.mouser.com/ProductDetail/Harwin/M22-7131042?qs=WS5Jv%252B%252Bx1qUfXs5lYp1U1g%3D%3D	12	NOT YET PURCHASED, Need to order 12	$1.58 ($18.96)
CAN IC	https://www.mouser.com/ProductDetail/Maxim-Integrated/MAX33040EAKA%2bT?qs=IS%252B4QmGtzzrp4y4r9nEH%252Bg%3D%3D	3	NOT YET PURCHASED, May need to order 12	$3.64 ($10.92)
Coin Cell	https://www.amazon.com/Energizer-Batteries-Volts-2Pack-Packaging/dp/B0002DSVS8/ref=sr_1_2?gclid=CjwKCAiAkfucBhBBEiwAFjbkr9Z8hjTaI5U-cxw_YBNor4g9KYARpazPzEYi6j-o66Kpkg3WfrsuBhoCW6gQAvD_BwE&hvadid=616931741932&hvdev=c&hvlocphy=1018608&hvnetw=g&hvqmt=e&hvrand=6075658892394344047&hvtargid=kwd-1979921681&hydadcr=24661_13611822&keywords=battery+cr2032&qid=1671414960&sr=8-2	6	3V cells	$0.50 ($3.12)
Coin Cell Connector	https://www.digikey.com/en/products/detail/adafruit-industries-llc/1870/6827145?utm_adgroup=Adapter%2C%20Breakout%20Boards&utm_source=google&utm_medium=cpc&utm_campaign=Shopping_Product_Prototyping%2C%20Fabrication%20Products_NEW&utm_term=&utm_content=Adapter%2C%20Breakout%20	6	Need to verify the size of the mounting bolts	$2.50 ($15.00)

	Boards&gclid=CjwKCAjw4c-ZBhAEEiwAZ105RVdCYCfWT4RVa5MtIsnGjfG_CNuxaFEC-8WMTkvF0-0UiS1XatoIUhoCCQMQAvD_BwE			
DC-DC Converter: 12V to 3.3V	https://www.digikey.com/en/products/detail/cui-inc/PDM2-S12-S3-S/5138447	6	NOT YET PURCHASED, Need to order 6	$6.25 ($37.49)
DC-DC Converter: 12V to 5V	https://www.digikey.com/en/products/detail/cui-inc/PDM2-S12-S5-S/4009648?s=N4IgTCBcDaIAoBECyYC0BlAjG9BWDIAugL5A	--	NOT YET PURCHASED, need to order 6	$6.25 ($37.49)
Fuse	--	--	NOT YET PURCHASED, Needs to be selected	
MicroSD Cards	https://www.amazon.com/dp/B09X7BK27V/ref=redir_mobile_desktop?_encoding=UTF8&aaxitk=5288b902ddf0db4ee131e81e89e03de6&content-id=amzn1.sym.552bcbb2-81a1-4e8b-b868-3fba7d5af42a%3Aamzn1.sym.552bcbb2-81a1-4e8b-b868-3fba7d5af42a&hsa_cr_id=8438103560601&pd_rd_plhdr=t&pd_rd_r=e6266d43-7809-4fd9-a0ee-169711c0375b&pd_rd_w=t386o&pd_rd_wg=xWbP1&qid=1671414816&ref_=sbx_be_s_sparkle_lsi4d_asin_1_title&sr=1-2-9e67e56a-6f64-441f-a281-df67fc737124&th=1	--	NOT YET PURCHASED, mistakenly ordered SD, need to order MICROSD.	$7.99 ($47.94)
RAM	https://www.mouser.com/ProductDetail/AP-Memory/APS6404L-3SQR-SN?qs=IS%252B4QmGtzzqsn3S5xo%2FEEg%3D%3D&gclid=Cj0KCQjw1vSZBhDuARIsAKZlijQCeCwjCCO4bkf99xsl1ORbNIRVXZ2BXWbEGfUGn9mLy1oDmKZCH8IaAvbUEALw_wcB	12	Already soldered onto the Teensy	$1.74 ($20.88)
Resistors/ Capacitors/ Inductors	--	--	We assume these will be provided by the ECE department	--
Teensy 4.1 Board	https://www.mouser.com/ProductDetail/SparkFun/DEV-16771?qs=vmHwEFxEFR%2FenEKd%2FmWpTA%3D%3D	6	Pins/Ram already soldered on	$38.74 ($232.40)
TVS Diode	--	--	NOT YET PURCHASED, Needs to be selected	--
XBee	https://www.mouser.com/ProductDetail/DIGI/XB24CZ7SIT-004?qs=3VJ0tGt%252Bi1w3hGvWrESPVw%3D%3D	6	S2C, not S2C Pro	$32.13 ($192.78)
XBee antennas	https://www.mouser.com/ProductDetail/DIGI/A24-HASM-450?qs=YPg7lQ8MWSfg2uAoe%2F49%2FQ%3D%3D	6	All in one box	$6.60 ($39.60)

(F) Parts list. This contains a list with links of all the parts that have either been purchased already or will need to be purchased in the future. Highlighted in green means this item has already been purchased.



(G) CAN IC Demonstration Wiring Diagram