Controller Area Network Implementation:

Technical Details & Case Study

Rensselaer Polytechnic Institute

Executive Summary

Controller Area Network (CAN) is a protocol which enables microcontrollers to communicate with devices on the CAN bus without using a host computer. The project sponsor, Partners for the Advancement of Collaborative Engineering Education (PACE), would like to see engineering students emerge from college with an understanding of basic CAN concepts; therefore one aspect of this project is to provide those opportunities to students at Rensselaer Polytechnic Institute (RPI).

The uses of CAN are numerous, but one of the largest areas of application is the automotive industry. The Formula Hybrid Team at RPI employed CAN in their competition vehicle for the first time this year, and this design project was aimed at assisting them with their implementation of CAN. The LabVIEW programs from the previous design team were enhanced to suit the needs of Formula Hybrid. The previous team also created a lab exercise for Rensselaer's Microprocessor Systems course that taught CAN in the context of automobiles using an RC car [1]. However, this lab has not been utilized in recent semesters, so the goal was to create functioning lab material from which students could gain experience.

One of the major accomplishments in regards to Formula Hybrid was developing a serial protocol for use with radio modules to send data wirelessly from the car to a laptop. Although not used to its fullest extent during the 2012 competition, it laid the groundwork for future Formula Hybrid teams to improve upon in later cars. On the courseware side, working code and LabVIEW VIs were developed to be used with the RC car. One recommendation to future design teams is to improve communication with Formula Hybrid and to specify the division of tasks with them at the outset of the project, so as to avoid misunderstandings and to leave more time for other semester objectives.

Contents

Glossary

- CAN – Controller Area Network
- PACE – Partners for the Advancement of Collaborative Engineering Education
- MPS – Microprocessors Systems
- FHYB – Formula Hybrid
- VI – Virtual Instrument
- UI – User Interface
- GUI – Graphical User Interface
- RC – Radio Controlled
- EKG – Electrocardiograph
- RF – Radio Frequency
- USB – Universal Serial Bus
- CSV – Comma Separate Values
- RTR – Receive, Transmit, Receive
- µC – Microcontroller
- EEPROM – Electrically Erasable Programmable Read-Only Memory

1. Introduction

The capstone CAN project is an educational program intended to promote undergraduate learning about CAN systems. It is sponsored by PACE, an organization dedicated to furthering education in engineering and technology. PACE works with various companies that utilize CAN systems in products and systems, one such company is General Motors. PACE has a motivation to help undergraduate engineering students learn the fundamentals of CAN systems, in order to produce more knowledgeable graduates.

In previous semesters, the PACE CAN team was tasked with developing educational tools, as well as a small-scale demonstration of the technology. The previous group had the primary goal of debugging an RC car that implements a CAN system. They also added additional hardware modules to the car to simulate real-world automotive equipment [1]. This semester, the PACE CAN team was tasked with assisting the RPI Formula Hybrid Team in the development of the CAN system on their car. This year is also the first year that the Formula Hybrid team implemented a CAN system, which was introduced to improve their overall car design by consolidating much of the electrical wiring necessary for control and communication. Members of the Formula Hybrid team had been working on the CAN system since the fall 2011 semester, fashioning their own custom PCBs for the CAN nodes [2]. The Formula Hybrid Team's main project for the semester was to implement a functional CAN network to wirelessly transmit sensor data and provide basic control for the formula car, however the Capstone team specifically assisted with the data acquisition tools and provided custom Vis to display the data visually. The expectation of the Formula Hybrid team was that the PACE CAN team will be able to assist in the current implementation of CAN as well as provide improved design options for future Formula Hybrid teams.

In addition to providing assistance to the Formula Hybrid team, knowledge of CAN systems was also required to provide useful educational tools for Professor Kraft and the MPS lab class. In previous semesters, capstone teams have written up LabVIEW based labs for the MPS Class, but little has made a substantial impact in the Class. The goal for the PACE CAN team this semester is to integrate a small-scale implementation of a CAN network with the CAN-bus debugging LabVIEW program.

The long-term objectives for this project focus on creating specific tools in academia to provide students with the opportunity to learn, utilize, and implement CAN. Through the implementation of CAN within the Classroom and support and consulting for the Formula Hybrid team, this purpose will be fulfilled by the successful completion of this project. This report details the objectives and deliverables of the project, existing technologies and previous work, the development, selection, testing, and specifications of the design, the impact on the engineering community and society, as well as recommendations for future work. Accompanying this report are all instruction manuals associated with the deliverables for Formula Hybrid and MPS.

2. Project Objectives & Scope

Long Term Objectives

The long-term objectives for this project focus on creating specific tools in academia to provide students with the opportunity to learn, utilize, and implement CAN. Guided by this objective, the Capstone team emphasized the development of two specific applications of CAN: One for a Rensselaer Club, the Formula Hybrid Team, and another for a Class at Rensselaer, the Microprocessor Systems Class. The Capstone team will create one functional lab for the MPS Class that students can use to learn the CAN bus and its associated protocols in semesters to come. Also, the FHYB Team will have a working and well-supported CAN system in their 2012 car, as well as a good foundation to work on this bus system for coming years. The main objective was to provide both with CAN solutions in the form of a prebuilt board that will interface a microcontroller with the CAN bus. The customer payoff will be represented in providing a base to achieve the primary long-term objective, as stated above. Specifically, the customer payoff will be the working labs for the MPS class, and improved functionality and base for future iterations for the Formula Hybrid team car, the knowledge that students, both on the Formula Hybrid Team and the MPS Students, gain from the CAN bus with hands-on experience.

Semester Objectives

The Semester Objectives for the Controller Area Network Capstone Team are broken into two specific applications: Courseware Development for the MPS Lab and Consulting and Support for the Formula Hybrid Team Support.

Courseware Development

1. Expanding functionality of LabVIEW educational tools
2. Creating user manuals for students to learn the CAN interface
3. Design small-scale laboratory set-up
4. Create Labs based on learning process

Formula Hybrid Support

1. Utilize LabVIEW educational tools for data acquisition and debugging of formula hybrid CAN bus
2. Develop Network Architecture for CAN bus
3. Design small-scale proof-of-concept system

Approach

Courseware Development: Learn the CAN bus and document progress via LabVIEW exercises. Inspect previous labs for additions and improvements. Utilize existing framework to create new labs.

Formula Hybrid Support: Implement custom CAN bus system for Formula Hybrid team race car. Create best practice implementation of CAN bus for future use. Provide

LabVIEW tools to collect and manipulate car sensor data. Develop control algorithms to improve handling of Formula Hybrid car.

Project Scope

It is important to specify the scope of the Capstone team for both applications of the CAN Bus. For the Courseware Development for the MPS Class, Prof. Russell Kraft and members of the Capstone team outlined a document describing the objectives of the project development for this semester and future iterations of this project. Similarly, to define these specifications with the Formula Hybrid Team, there was a grievous discussion to outline the Capstone groups' specifications. These specifications are both contained within Appendix C, both documents define the scope of for each application.

Project Deliverables

Below are the project deliverables for this semester's Capstone Team. Each deliverable corresponds to at least one semester objective as stated above (CD refers to Course Development and FH stands for Formula Hybrid).

1. LabVIEW UI External Specification for Formula Hybrid Team CAN Bus (FH 1)
2. Working CAN Bus for Formula Hybrid Car (FH 2)
3. User Interface Manual and Video for Formula Hybrid Application (FH 1)
4. LabVIEW Software Specification Documentation (CD 2)
5. Lab Setup and Procedure Documentation and CAN Interface User Manual (CD 2) (CD 4)
6. Working Proof-of-Concept for CAN Bus (Hardware & Software) (CD 3)
7. Design Specification Documentation for Formula Hybrid CAN Bus (version 2) (FH 3)
8. Custom LabVIEW UI for Educational Labs (CD 1)

3. Assessment of Relevant Existing Technologies

Automotive Applications of CAN

CAN technology began to appear in automobiles in 2003 and virtually every car since 2008 utilizes CAN. [3] It decreases the complexity of wiring, provides improvements to the weight of the car – which is significant in the design formula cars – and makes diagnostics less cumbersome. [4] The average car nowadays has dozens of electronic modules for tasks such as cruise control or engine RPM monitoring. There are four main classes of serial data bus speeds (A, B, C, and D), with D being the fastest. The architecture of a car usually has multiple buses operating at different speeds, with the fastest bus being dedicated to the most critical components. [5] Figure 1 shows an example of a typical CAN automobile layout in a car displaying control, data acquisition tools, and even more advanced systems such as a Global Positioning System.

In addition to basic control and monitoring functions, there are numerous other aspects of automobiles where CAN is applied, especially in the way of safety features. This includes optimization of fuel consumption by switching between gas and/or electric power and in what proportions, as well as tire pressure monitoring or electric seat belt tensioners. [6] The uses of CAN even extend to cutting edge concepts such as regenerative braking, driver assistance or predictive driving, and even driver EKG monitoring. There are countless other uses in use and in development, and it will only continue to proliferate. [6]

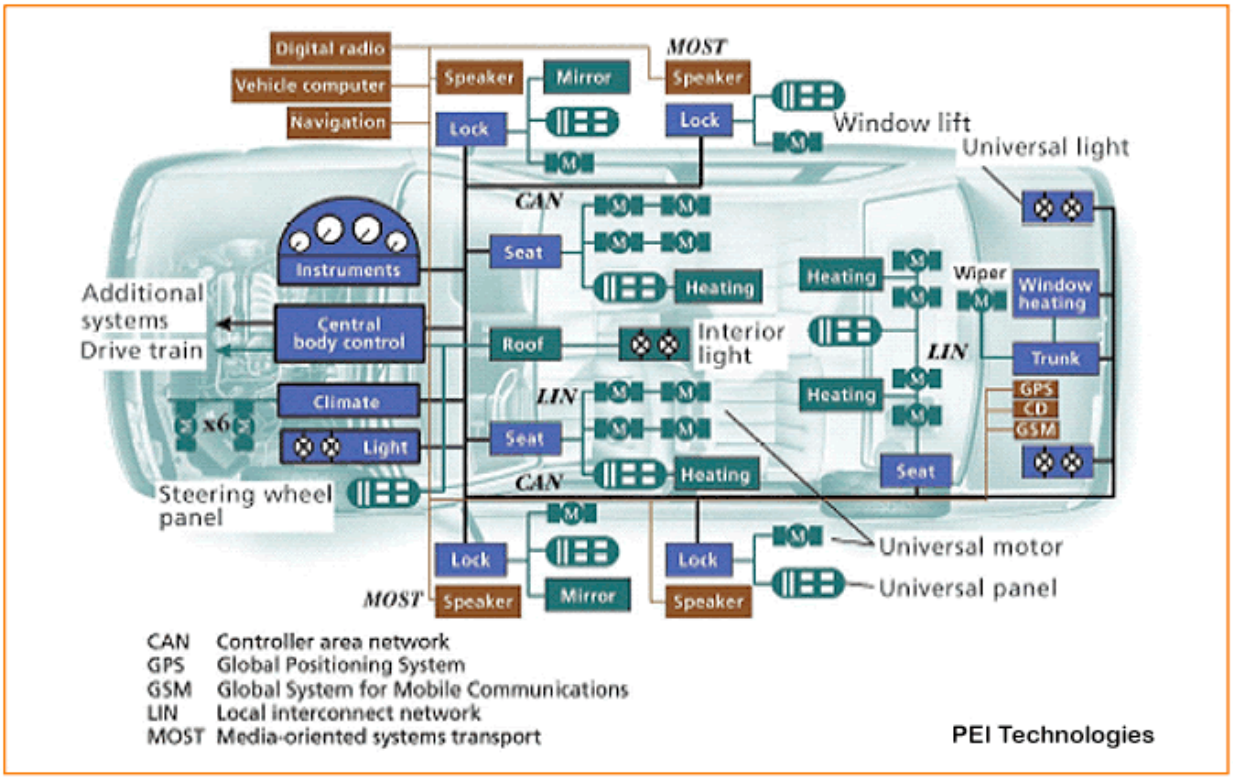


Figure 1: Example of the Controller Area Network Automobile Layout [3]

Non-Automotive Applications of CAN

CAN also has numerous applications outside the automotive industry, including maritime and avionics, industrial control and factory automation, and even medical equipment. One particularly well-defined area where CAN is used is elevator design. Some of the functionality includes requesting the current location of an elevator car, opening and closing the doors, and displaying to the user which direction they are traveling and to what floor, all over a bidirectional communication channel. As with automobiles, safety is a huge benefit when talking about CAN is industrial and other commercial applications as it can be a reliable and robust system. [7]

Past Years Applications of CAN

The spring 2009 design team developed LabVIEW VIs for snooping and controlling the CAN bus on the RC car in the Microprocessor Systems Lab, which were meant to receive and transmit CAN messages [1]. To use LabVIEW with CAN, National Instruments has many hardware solutions. To connect the CAN bus to the laptop that is running the VI, the NI CAN Demo box, shown in Figure 2, will be used. Also pictured is NI's USB-8473s, which will be used to connect CAN devices to the laptop.

Figure 2: NI CAN Demo Box and USB-8473s Connector [8] [9]

This semester's team adapted those VIs to fit the needs of the RPI Formula Hybrid team. The front panel that the user sees reflects the specific sensors that Formula Hybrid is using for their car. The Formula Hybrid team was originally attempting to use Python scripts to create a GUI to view the sensors readouts, and therefore openly welcomed the option to use a higher-level programming language. The graphical nature of LabVIEW, its numerous pre-made and customizable controls and indicators, and the available library of CAN functions, make it simpler and more aesthetically pleasing than their original approach. However, the Formula Hybrid team wanted to be able to use LabVIEW to monitor the car during the competition, as opposed to just using LabVIEW when testing the car while it's up on the blocks. This was accomplished using XBee RF modules transmitting data wirelessly from the car to a laptop running LabVIEW. In order to receive the CAN messages, the LabVIEW program needed to be altered to accept serial data.

The Spring 2009 group also wrote documentation for a lab exercise for the RPI Microprocessor Systems course, ECSE 4790. There were separate versions of the controlling VI for the students and the TAs, each with different comments added to the block diagram [1]. The lab exercise seems relatively simplistic for a senior-level course, and most likely would not take very long to complete in a lab session, so the current design team expanded upon the learning modules involving LabVIEW. Additionally, the RC car in the lab setup needed to be fixed as it was not operational at the start of the semester [1]. It will eventually be able to communicate with the PIC and 8051 microcontrollers as well as send and receive data to and from the custom LabVIEW interface.

4. Professional and Societal Considerations

The CAN bus project may have a far reach beyond the immediate solution of the engineering problems encountered, with social, professional and environmental effects. In the realm of the environment and sustainability, the Formula Hybrid team's current and future efforts were greatly assisted by the development of the CAN-based data bus with its complementary 802.15.4 XBee-based wireless system and the LabVIEW Vis for real-time interpretation and logging of the telemetry and other data. This will help the team to make faster and longer strides in the future as the Formula Hybrid car is further developed, allowing the engineers on the team to grow more skilled in the design and fabrication of hybrid cars with more advanced features. When these engineers then go into industry, they will be able to more quickly advance the state of the art of hybrid cars and thus quicken the replacement of less fuel efficient non-hybrid gas powered vehicles. The end result is that overall carbon emission from automobiles will be reduced as the fleet is replaced with lower emission vehicles.

Professionally, the overall goal of the microprocessor systems lab's half of the project is to familiarize engineering students with the CAN bus in particular and networking in embedded systems generally. The lab produced for this class along with the LabVIEW debugging interface developed and the working car system to run the lab on satisfy this goal, and thus meet this need of industry for skilled engineers with experience with these technologies while simultaneously helping engineering students to secure gainful employment in their field at graduation.

The social effects of the project stem primarily from the software used for the Atmel/Arduino-based nodes of the Formula Hybrid CAN bus system. Open-source libraries were leveraged extensively for this part of the project, and several bugs were identified in the libraries used. Since the source was available, the bugs were able to identified and fixed rather than worked around. To fulfill the terms of the particular open-source license used for the Arduino compatible libraries and also out of a spirit of solidarity with hobbyist developers, these bug fixes were submitted to the 'upstream' developers of the libraries. This illustrates a growing trend in software development, where software engineers even in professional contexts are often willing to share the code they have developed that is not proprietary or a core business competency with others for mutual benefit: those who receive the code are able to leverage work already done rather than reinventing the wheel themselves, and those who release the code receive bug fixes and other improvements from their users that they would not have received had the code not been released.

5. System Concept Development

Hardware Concept

The final concept that the team decided on for the Formula Hybrid CAN bus system, based on the Atmel 328P microcontroller, was chosen due to its advantages in a variety of areas. The choice of the PCB design that was used was not subject to debate, as the customer had already developed that subcomponent and was highly resistant to change in that regard. This allowed the design team to leverage several months of work that had been done on this part of the system. For the software that ran on this system, the team chose a design where the 'slave' sensor nodes push updates to the 'master' collector node, rather than an earlier concept where the master would have sent requests for updates to the slaves, which then would have responded. This choice reduced the bandwidth required and simplified the design of the system, as the slave nodes don't have to look for input at all.

The team chose a PIC18F25K80 microcontroller for the final design due to its processing speed, memory capacity, input/output configuration, and especially due to its integrated CAN module, which would eliminate the need for an external CAN controller. The MCP2551 CAN Transceiver was chosen for the design because it is a popular, inexpensive, and simple IC to translate CMOS/TTL logic levels to CAN format. Building upon previous teams' work, the other components used in the design were largely due to constraints from the MPS car system – the SPI DAC used to control the motor and the 7405 inverter to control the lights, as well as the MCP9700 temperature transducer and the other monitor circuitry built into the car.

Software Concept

The software concept that the team developed not only focused on delivering proper data, but also on security, reliability, and user interface design. The final software concept has several components.

1. **LabVIEW VI**: A LabVIEW VI that contains the instruments necessary for the Formula Hybrid team's car.
2. **CAN/Serial switch**: The user is able to easily switch between direct CAN input from the NI USB-to-CAN adapter, or the serial XBee input. This is to allow for direct-wired connection in case of interference or other conditions that limit the use of wireless or serial.
3. **Data Logging**: Graphs show the data being collected and can be saved into a comma separated value format. This was added after request of the Formula Hybrid team.
4. **Serial data transfer protocol**: A new protocol was developed to transfer data over a serial link. This was done because of the XBee wireless modules, which act as serial connections and cannot be interfaced directly with the CAN libraries. This protocol went through several revisions; see Design Analysis.
5. **LabVIEW DLL plugin:** A C++ program to implement and control the serial protocol was developed and plugged into the LabVIEW program via a

replaceable DLL. All the interaction that the LabVIEW program needs to do with the protocol is handled by the DLL. This makes the protocol easy to change and requires no modification of the LabVIEW VI itself.

Aside from the main components of the concept, some design considerations were made. The first of these is security. On a wired connection, interception of the data is not a concern, but with the introduction of a wireless connection, it is possible data may be intercepted and read. To combat this, the XBee utilizes IEEE 802.15.4 security measures and AES encryption [10]. This makes it extremely difficult to decipher the data. Also, another major part of the concept is robust and graceful handling of line interruption. Both wireless interference and physical line disconnection are hazards that could occur. The system properly pauses and resumes data display upon disconnection and reconnection.

There were some considerations to be made in terms of the best way to perform data logging as well as overall user interface design. Initially, all of the values from the front panel indicators were being written to a two dimensional array and then written further to a spreadsheet file. This was not a modular solution and was not the best way of creating a log file. It was later decided that LabVIEW's built-in charting functionality would be used to show a live view of the data graphed on the front panel. With that, however, it was decided that having as many graphs as indicators took up a lot of space on the screen. In the final version, tabs were created to hold groups of similar sensors, thus allowing a user to export the graphed data to a spreadsheet file all while maximizing the screen space.

For MPS, some development decisions were made based on the team's time constraints. For example, there is both a "sender" and a "viewer" LabVIEW VI to transmit and receive data from the RC car in the lab. While it would be technically possible to create a producer/consumer structure to show CAN messages as they are being sent in a loop, it was determined that this may not be the best solution. Instead, the two separate Vis can be run from two computers, which is possible since the lab has desktops and students will be expected to have laptops in class as well.

6. Design Analysis

XBee Protocol

Development of the serial protocol that was used for retransmitting CAN messages over the XBee wireless link was a joint effort between the PACE CAN team and the Formula Hybrid team. The original design of the protocol can be seen in Figure 3.

```
FLAG ID MESSAGE… PARITY

FLAG = 0x01

ID and MESSAGE are 7-bit values that use the top 7 bits of the byte.
The last bit of each byte is always 0 so that they cannot be confused
for the FLAG.

ID is a simple integer value ranging from 0 to 127

The meaning of MESSAGE is determined from ID. If the message needs
more than 7 bits, more MESSAGE bytes can be added as long as they
follow the rule of only using the top 7 bits.
PARITY uses the high-order bit as parity of the ID and MESSAGE
```

Figure 3: Serial Protocol v1

Upon discussing this protocol with the Formula Hybrid team, it was determined that despite the simplicity and speed of using parity for error checking, the reliability would be very low. The protocol was revised to include a CRC-8 checksum in the last two bytes.

The Formula Hybrid team also expressed concern with the number of available IDs if it was limited to 7 bits. The CAN specification allows for IDs up to 11 bits long. They also changed their mind about how the length of the message would be determined. Instead of looking up the ID in a table, the length of the data would be transmitted in the frame. The specification was modified to include these three changes as seen in Figure 4.

```
0b0000 0001 (FLAG)
0bIIII III0 (First 7 bits of ID)
0bIIII LLL0 (Last 4 bits of ID, first 3 bits of DATA_LENGTH)
0bLDDD DDD0 (Last 1 bit of DATA_LENGTH, first 6 bits of DATA)
0bDDDD DDD0 (Next 7 bits of DATA)
0bDDDC CCC0 (Last 3 bits of DATA, first 4 bits of CRC)
0bCCCC 0000 (Last 4 bits of CRC)
```

Figure 4: Serial Protocol v2

The final design of the protocol was determined by the Hardware sub-team. Upon inspection of the protocol, they determined that to complete the implementation within their time constraints, it would need to be simplified. The ID was shortened back to 7 bits, since there were not nearly enough sensors to exhaust the available IDs, and the layout of the protocol was simplified. The final protocol can be seen in Figure 5.

```
0b0000 0001 (FLAG)
0bIIII III0 (ID)
0b000L LLL0 (DATA_LENGTH)
0bDDDD DDD0 (First 7 bits of DATA)
0bDDDD DDD0 (Next 7 bits of DATA)
0bDD00 0000 (Last 2 bits of DATA)
0bCCCC CCC0 (First 7 bits of CRC)
```

Figure 5: Serial Protocol v3

Formula Hybrid LabVIEW

The first revision of the Formula Hybrid LabVIEW program was a mockup (Figure 6) based on the specifications that the Formula Hybrid team provided. It contained all of the sensors that the Formula Hybrid team thought that they needed at that time and was able to read off of the CAN system by using code from the Spring 2009 team's final deliverables.

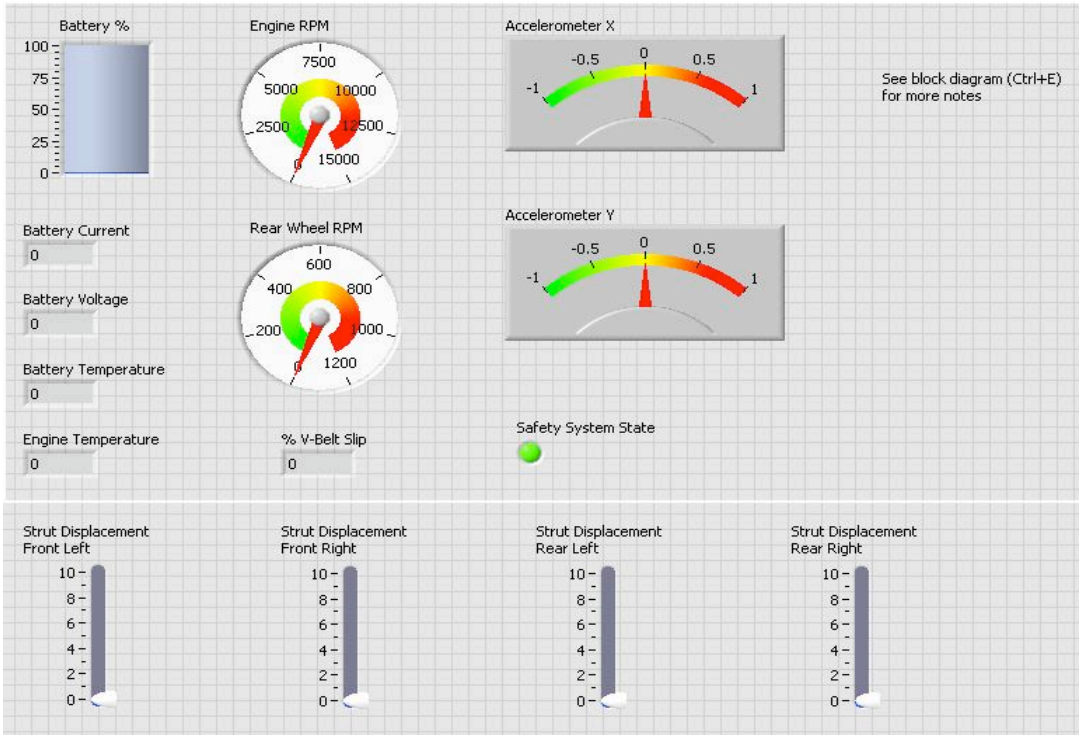


Figure 6: Formula Hybrid LabVIEW Mockup

After discussing the mockup with the Formula Hybrid team, the mockup was modified with to contain the correct sensors for them at that point. At this point they also

requested that they be able to monitor their car wirelessly. They requested that an XBee wireless module be used, which meant that it was necessary to be able to read serial data in the LabVIEW program. A copy of the LabVIEW program was made that used serial instead of CAN.

Upon further discussion with the Formula Hybrid team as they finalized their vehicle, the final list of sensors that would be visible on the LabVIEW panel, and the ranges of values for each sensor were determined. Formula Hybrid also requested warnings when parameters went out of range and a mechanism for logging sensor data with respect to time. It was also decided to merge the serial and CAN LabVIEW programs, using a toggle switch to change between the two protocols. The final result is visible in Figure 7.

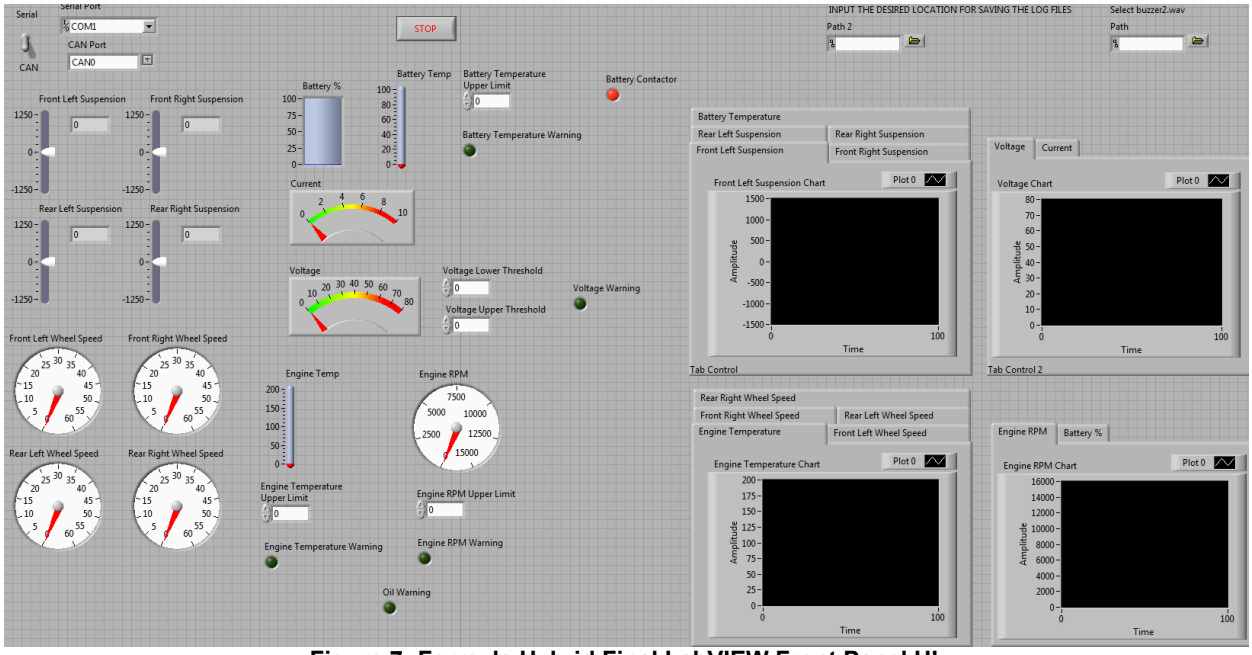


Figure 7: Formula Hybrid Final LabVIEW Front Panel UI

Formula Hybrid CAN Node

The Formula Hybrid team already had a functioning PCB (Figure 8) when we came in to help them. Upon inspection of this board it was found that it had design issues that needed to be fixed.

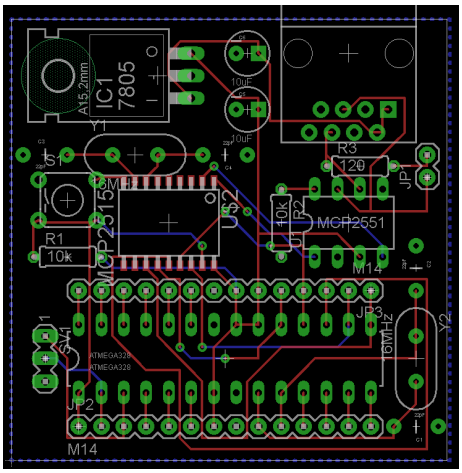


Figure 8: Formula Hybrid CAN Node v1 [2]

In the version 2 design, a few components were replaced to improve the speed and reliability of the node. The processor was changed to a PIC microcontroller since the Arduino that the node was using is a hobbyist chip that is not suitable for use in the CAN nodes. The RJ45 connector was replaced with a screw-in DE-9 connector because the RJ45 connector is not strong enough to stay in place when it is mounted in a vehicle. The vibrations would easily knock it out. It was found that there was an unnecessary jumper on the board that simply added to the cost, so it was removed.

There was noise noticed on the bus that made the CAN signal difficult to read, so the version 2 design contains a bypass capacitor that will insulate the line against switching noise. An external crystal oscillator was added to provide a more accurate clock than the internal clock of the PIC which will also make reading the CAN signal more reliable. Finally, as a convenience, an in-circuit serial programmer was added so that the nodes can be reprogrammed and repurposed without needing to use a socket or desoldering the microcontroller.

Microprocessor Systems LabVIEW

Testing the LabVIEW VIs for MPS was straightforward as the RC car in the lab was functional for the most part. The lab setup was used to receive and send commands to the car. The headlights/tail lights, turn signals, horn, and motor current and temperature, and wheel RPM were all working. Reading the steering position, however, was not completely working. Prior to using the car itself, two laptops were used to send randomly-generated test data to debug using the LabVIEW sending and receiving VIs (this was the case for Formula Hybrid as well).

A state machine architecture was created for the sending VI, which may not have been necessary to implement. The primary purpose was to only have the software send data to the car when a button was pressed and to just wait for user input while it was not pressed. The user could just as easily have entered the ID and data values and pressed the “enter” key for the same results without making things overly complicated.

7. Final Design and Engineering Specifications

Three final systems were developed and improved for this project – two for the Formula Hybrid team's car and one for the Microprocessor Systems courseware. For schematics and images of the final product, please refer to Appendix C.

Formula Hybrid CAN Node Initial Design

The first system for Formula Hybrid was developed mainly by one of their members, Nathaniel Quillen. The custom PCB he designed consisted of an Atmel ATMega328P microcontroller using the Arduino bootloader, a Microchip MCP2515 CAN controller IC, a Microchip MCP2551 CAN Transceiver IC, crystal oscillators, a 7805 voltage regulator, a custom pin-out RJ-45 connecter, and various loading resistances and capacitances. Each board has ample ports for general-purpose digital and analog input and output to wire to the various sensors and lights throughout the car. Integrated into this network, although not used in the final working version of their car, was an XBee wireless communication system to relay data collected on the CAN bus back to a computer in the pit. To support the XBee module, software libraries and a specialized serial communication protocol were developed.

Formula Hybrid CAN Node Revised Design

The second system developed for Formula Hybrid was a revised design of their CAN node PCB as a proposal for use in the future. This design is implemented with a PIC18F25K80 microcontroller. While similar to the ATMega328P, the PIC chip in this design is slightly more robust and features an integrated CAN module, eliminating the need for the external MCP2515 controller as with the previous design. The new design still incorporates the use of a 7805 voltage regulator, MCP 2551 CAN Transceiver IC, and a crystal oscillator. This new board, however, uses a DB-9 connector with the standard CAN pin-out.

The CAN network designed for the car consisted of six nodes – one at each wheel, one in the dashboard, and one in the battery box. The inputs to these nodes were a variety of analog and digital signals. On the digital side were pulse-train inputs for wheel and engine RPM and outputs for the brake lights. On the analog side were inputs from a linear potentiometer on each wheel indicating strut travel, engine temperature sensor, oil level sensor, and throttle level, and brake level.

Microprocessor Systems CAN Bus System Design

The system designed for the Microprocessor Systems courseware was largely based on the original setup from 2008. The main components of this system are the car, which is controlled by a PIC microcontroller, the control and meter boxes, which are read and controlled respectively by a 8051F040 microcontroller (Silicon Labs 8051F040 development kit, as used in LITEC and MPS), and LabVIEW VIs for both control and monitoring of the car. The three components all communicate via the CAN protocol. The PIC subsystem is based on a PIC18F25K80 microcontroller. Connected to this microcontroller are a MCP2551 CAN transceiver, a LM-741 SPI DAC IC, 7405 Inverter

IC, and various analog and digital inputs and outputs. The digital input in the system is from the RPM sensor. The digital outputs in the system are the headlights and taillights, left turn signal, right turn signal, horn, and servo output (PWM). The analog output is for the motor speed, which the DAC IC is used for. The analog inputs are current measurement and temperature measurement. The 8051 connected to the meter box has 2 analog outputs for speed and current, and temperature is controlled by a PWM output. The digital outputs for this module are left and right turn signal indicators. The 8051 connected to the control box has two analog inputs, one for speed and one for direction. There are four digital inputs – horn, left signal, right signal, and headlights/taillights.

Formula Hybrid LabVIEW System Design

The LabVIEW subsystem for Formula Hybrid consists of a VI that contains graphic sensor readouts for the various values monitored on the car. In addition to the numeric indicators for each sensor, the incoming CAN data from the XBee is displayed in charts on the front panel. The sensors that are the most crucial to safety and performance are displayed by default, but the hidden charts can be viewed by clicking on their corresponding tab. After each run the data from the charts can be exported to Excel for analysis. In addition, the sensor values with their IDs will be saved to a CSV (comma separated values) file as the program runs. A unique file name is generated each time by appending a time stamp to the file path chosen. For safety reasons, the most critical sensors are constantly evaluated against user-inputted thresholds. If these bounds are surpassed, the user is alerted by red LEDs and a loud beeping noise.

Microprocessor Systems LabVIEW System Design

The LabVIEW subsystem for Microprocessor Systems consists of two laptops, each with a separate VI loaded, and each with a NI-USB-8473 USB High Speed CAN interface. These components all use a DB-9 connector with the standard CAN pin-out to communicate. Once fully assembled, the PIC receives control signals from the CAN bus, and upon reception of a Remote Transmission Request, it replies with the appropriate data. The 8051 connected to the meter box sends RTR messages to the PIC, and upon reception of a response it updates its output to the meter box accordingly. The 8051 connected to the control box reads the signals and sends appropriately addressed messages to the PIC to control its various functionalities. The LabVIEW VIs have the capability of monitoring all CAN network traffic, as well as a customized dashboard corresponding to the appropriate CAN addresses on the network. It also has the functionality to send a message with any data to any address allowable within the CAN protocol. For the controlling VI, the messages are pushed rather than pulled, and so a message is only sent to an ID when the user decides to change a value, thus minimizing the number of messages being sent since none are sent until a value is changed by the user. The team used an event structure was used that monitored the position of the "Write to CAN" button. After typing an ID and a value into the numeric controls, the user presses the button, which tells the system to write that value to the desired ID.

8. System Evaluation

LabVIEW Software System Evaluation

Testing for the LabVIEW software was completed using a C++ program that simulated serial data coming from the Formula Hybrid Car. This program used the XBee adapter before the hardware was ready to simulate input data as it would be seen from the CAN bus. This allowed the protocol to be implemented in LabVIEW before the Arduino and associated hardware was finished.

Bandwidth Testing

After the CAN bus system for the Formula Hybrid team was working correctly at the sensor update rates specified in the data dictionary, additional bandwidth testing was done to find the update rates at which the system began to fail. This was done in order that the Formula Hybrid team would know the limits of the system if higher update rates for various sensors were desired in the future. The testing was conducted to detect packet loss on the CAN bus itself by sending unitless mock auto-incrementing data for three sensors (two at a given update rate and a third at half that rate) from one slave node to master, simulating the traffic from one wheel node, with the master printing output to serial to simulate sending the data to the XBee wireless module through its normal serial interface. Since the mock data was auto-incrementing, the master could tell easily if a packet had been lost on the CAN bus by checking whether the integer data received from the CAN id corresponding to each mocked sensor was greater than the last data received from that sensor by one; if this was not the case, a packet had been dropped.

Initially we encountered packet loss at update rates as low as 35 Hz for the sensors of this single simulated slave node, with the onset of packet loss being consistent for multiple trials when the data from one of the slave nodes reached the value of 216. Knowing that the size of each packet to be 8 bytes, that 10 bits including the start and stop bits were sent over serial for each byte, and that 3 packets were sent in a burst in the worst case and 2.5 in the average case, we knew that the traffic over the bus at this 20Hz update rate averaged only 4000 bits/second and that thus we were not exceeding the 500 kbit/s bandwidth of the CAN bus itself nor the 9600 bit/sec bandwidth of the serial connection. Some study, however, indicated that the calls to write to serial on the Arduino were blocking – that is to say that the function Serial.print() does not return until all the data it is passed has actually been sent. Based on this and the knowledge that our MCP2515 CAN controller has a receive buffer large enough to hold 3 packets, we hypothesized that the time for which the call to write to serial was blocking was longer than the time between bursts of packets being sent, and that the receive buffer of the controller was thus overflowing leading to packet loss. Some calculations suggested this was likely: (3 packets * 8 bytes/packet * 10 bits sent/byte) / (9600 bits / second) yields a minimum time to send of 25ms before accounting for overhead, meaning that update rates greater than 40Hz would never be possible with this setup.

We increased the serial rate from 9600 baud to 115,200 baud, and immediately were able to achieve update rates for the sensors of 460 Hz before the onset of packet loss

on the CAN bus, corresponding to a time between packet bursts of 2174 microseconds. This seemed to confirm our hypothesis, as the minimum time to send 3 data packets in a burst via serial at this rate is 2083 microseconds. Based on this, we recommend maximizing the serial data rate at which the Arduino communicates with the XBee. Unfortunately, testing with the Arduino and XBee revealed that although each device is independently capable of serial data rates of 115,200 bits/s, communication from the Arduino to the XBee at rates greater than 57,600 bits/s is often garbled. This imposes a practical limit on the system in its current configuration that the master node may not receive updates at a rate greater than approximately 230 Hz from 3 sensors; given the actual system has 14 sensors, this places a limit on each node that it may not send updates from its sensors at rates greater than approximately 49 Hz.

Given these constraints we recommend that if greater update rates are desired, either most of the data received by the master note be logged to some storage device such as an SD card with only some fraction being sent via serial to the XBee for transmission in real time, or that multiple master nodes be used. Given that the data rate at an update rate of 230 Hz is merely 46 kbit/s while the CAN bus is capable of data rates of 500 kbit/s, it is likely that multiple master nodes could allow for up to 10 times greater utilization of the CAN bus before further bandwidth limitations would be encountered.

Formula Hybrid System Evaluation

The performance of the Formula Hybrid CAN bus system exceeded the needs of the current iteration of the car, and should meet the needs of future iterations. Bandwidth and congestion testing revealed that the system as it stands can serve overall update rates up to approximately 500Hz without losing packets, whereas the current update rate is on the order of 24Hz, with the limit here being the speed of sending data to the XBee wireless transmitter via a serial bus with a maximum speed of 115,200 bits/s. While the reliability of the system has been nearly flawless on the lab bench, the robustness of the system to the vibration and electrical noise on the formula hybrid car is largely untested; however, there are several improvements to the version 1.1 and 2.0 specifications that address worries in this area, including shielded cable and more solid connectors. The primary failure mode of the system is packet loss. Internally, due to the ring architecture of the bus this mode is all-or-nothing, although the likelihood of loss here is minimal due to the extent of physical damage to the car that it would take to sever both of the data lines on the bus (and indeed if this happened there would be other more pressing issues with the car as well). Externally, if the wireless system goes out of range of the receiver or if intervening obstacles block the signal, there will be momentary packet loss. Since the system is connectionless, data will simply resume being received once the transmitter and receiver are back in range of each other. Several schemes for buffering lost data were devised to ameliorate this, but in the end they were not selected since adding a necessary acknowledgement of receipt to the system would lead to an unacceptable waste of bandwidth.

9. Significant Accomplishments and Open Issues

The Spring 2012 semester CAN team was able to accomplish many of the objectives we set at the beginning of the semester, including extra goals set during the semester. From the perspective of our customers, the RPI Formula Hybrid team and Professor Kraft, the CAN team provided useful deliverables for their different needs. The CAN team also devised additional ideas to improve the systems, which could not be explored this semester, and remain open for pursuit by the next CAN team.

The Formula Hybrid section of the CAN team produced a few major deliverables for the RPI Formula Hybrid team. This first of these is the working Arduino code implementation of a CAN communication system. This code has helped the FHYB team with their first attempt at a CAN system, and will contribute to the progress of next year's team. Accompanying the functional code is the testing code and testing procedures for the individual 'version 1' CAN nodes designed by Formula Hybrid. These testing procedures provide a method of debugging individual nodes for all Arduino-based CAN implementation. We also developed a protocol to facilitate XBee/CAN compatibility, which provides future Formula Hybrid teams with the ability to transparently and wirelessly acquire data from the CAN bus. To utilize this XBee functionality, a LabVIEW VI was developed as a user interface for monitoring the CAN bus, and will definitely be a useful tool for future teams. In addition to all of these deliverables, the team wrote the accompanying documentation, which will ensure that future Formula Hybrid teams will be able to quickly grasp the knowledge required to effectively use the provided tools. Taking into account all we have learned about CAN systems this semester, the CAN team also used that experience to propose an alternate (version 2) design for Formula Hybrid's CAN system. However, there are some issues that the team discovered and did not have the time to address during the semester. One of these issues that the next team should take a look at is the implementation of the Arduino bootloader as the method of running a CAN network. This software platform is adequate, but not ideal. There were multiple issues with the libraries that needed to be debugged. Another issue is the bandwidth limitation of the XBee radios. CAN is designed to operate at much higher data rates than Serial communication, so the amount of messages that could be sent to the LabVIEW VI was significantly limited by this technology.

The CAN team was just as successful in developing the MPS lab reports for Professor Kraft, producing deliverables that will help future MPS students to better understand CAN systems and more accurately test their knowledge in that area. Specifically, the team created an improved version of the LabVIEW VI based on the last CAN team's work. This VI was developed in conjunction with the VI intended for use as a debugging tool by RPI Formula Hybrid, and will provide education in both LabVIEW VI creation and CAN systems to MPS students. To accompany this VI, the team also greatly improved upon the previous team's lab procedure and manual. The improved lab procedures will enable MPS students to more quickly learn the required information, and will test students on a wider range of CAN material. The team made sure that the VI solution and testing materials were well documented as well. The only open issue that had not

been addressed with regards to the LabVIEW VI is that currently, two VIs are necessary: one for reading from the CAN bus, and one for writing to it. Ideally, these VIs should be integrated for simplicity's sake.

10. Conclusions and Recommendations

The Spring 2012 CAN capstone team project was an overall success on all accounts. We succeeded in our main goals for the semester and tackled new problems as they presented themselves throughout the semester. Most of the objectives for the MPS lab were set based on the previous team's work in CAN systems; with regards to this goal, the further development of the educational tools while building upon the last team's work, the team exceeded expectations. The RPI Formula Hybrid section of the project, however, created some new challenges that the last team did not face. The objectives that relate to this part of the project mainly arose from the needs of the Formula Hybrid team, as well as their current capabilities. The capstone team's project was to be based on the implementation of the CAN system already in place by the Formula Hybrid team. The capstone team was given this job in order to take some of the work load off of the electrical component of the Formula Hybrid team, as well as to provide accurate continuity and other design suggestions, based on the previous capstone work and research into CAN.

The Capstone team learned a lot this semester about CAN system functionality and creating LabVIEW VIs. After gaining this experience, the team came up with many suggestions and recommendations for both the Formula Hybrid team and the next CAN capstone team. One item that we cannot stress enough is to read over the documentation that the capstone team created in the repository and archives. It is good practice to first understand how the system is supposed to work at a basic level, and then proceed to try the system tests to observe them in action. The major recommendation, with regards to the Formula Hybrid side of the project, is to establish specific and thorough requirements for what part of the CAN system the capstone team will be working on. Most of this part of the project involved troubleshooting and debugging, and establishing these specific goals early on will free up the necessary time to get the CAN system working properly. Another suggestion we have is to dedicate any time not used developing the main system for RPI Formula Hybrid to working on 'version 2' CAN implementation designs, and to persistently pursue these alternative options with Formula Hybrid. The Spring 2012 capstone team worked on these alternate design options to provide a basis for future research into improved CAN solutions.

In the end the team provided a working CAN implementation with LabVIEW tools to the Formula Hybrid team, as well as lab documentation and procedures for the MPS class. The team achieved these objectives by first building an understanding of CAN systems, followed by carefully examining the requirements of our customers and establishing specific deliverables that would satisfy those requirements. Any future capstone team should be able to be just as successful, so long as they study our accomplishments and learn from our mistakes.

References

[1] E. e. a. Pathyil, "ECSE Design Final Report: Controller Area Network (CAN) Version 1.6.," Rensselaer Polytechnic Institute, Troy, NY, 2009.

[2] N. Quillin, 8 December 2011. [Online]. Available: http://key-basher.blogspot.com/search/label/CAN-bus. [Accessed 2012].

[3] AA1Car, "Controller Area Network (CAN) Diagnostics," [Online]. Available: http://www.aa1car.com/library/can_systems.htm. [Accessed 28 January 2012].

[4] N. Murphy, "Introduction to Controller Area Network (CAN)," Netrino, 8 July 2008. [Online]. Available: http://www.netrino.com/Embedded-Systems/How-To/Controller-Area-Network-CAN-Robustness. [Accessed 5 February 2012].

[5] M. Guardigli, "Hacking Your Car," Only Dead Fish Go¬ With the Flow, 4 October 2010. [Online]. Available: http://marco.guardigli.it/2010/10/hacking-your-car.html. [Accessed 28 January 2012].

[6] Can in Automation (CiA), "Passenger Cars: Application Examples," 2011. [Online]. Available: http://www.can-cia.org/index.php?id=187. [Accessed 5 February 2012].

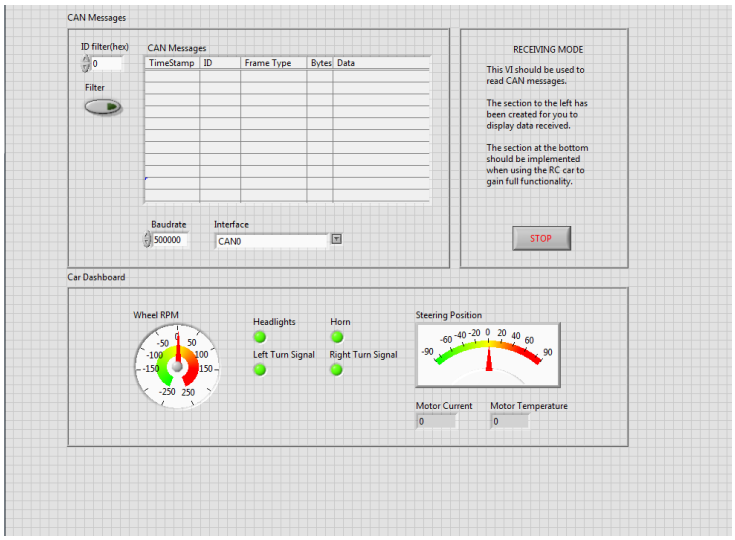
[7] M. Passemard, "Microcontrollers for Controller Area Network (CAN)," Atmel, 2011. [Online]. Available: www.scribd.com/doc/80457440/Can. [Accessed 17 May 2012].

[8] National Instruments, "NI CAN Device Simulator," [Online]. Available: sine.ni.com/nips/cds/view/p/lang/en/nid/201709. [Accessed 17 May 2012].

[9] National Instruments, "NI USB-8437s," [Online]. Available: sine.ni.com/nips/cds/view/p/lang/en/nid/203385. [Accessed 17 May 2012].

[10] Digi, "XBee-PRO® 802.15.4," 16 May 2012. [Online]. Available: http://www.digi.com/products/wireless-wired-embedded-solutions/zigbee-rf-modules/point-multipoint-rfmodules/xbee-series1-module#overview. [Accessed 16 May 2012].

Microprocessor Systems User Interface Layout



Appendix A: User Manuals and Other Documents

Microprocessor Systems CAN Bus Hardware & Software Specification

Introduction
The Controller Area Network (CAN) Team is delivering an improved version of the existing final laboratory for the Microprocessor Systems (MPS) class at Rensselaer as part of the ECSE Design Capstone Course. The purpose of implementing a CAN Bus in the MPS class is to have students leave college with practical CAN Bus experience that can be applied to real world applications.

Requirements
The primary requirements are to improve upon the already existing laboratory setup for CAN and provide documentation for both the lab Hardware & Software, and provide students with a Laboratory Procedure to follow to perform the lab. The enhanced procedure will implement a 3-stage approach to teach how CAN implementations work. The first stage introduces students to the National Instruments LabVIEW tools for CAN and their support hardware. This also covers the physical requirements for the bus and nodes while introducing the communication protocol. The second stage adds a modified RC car as a new device on the bus and the list of commands to which it will respond. Students are required to add to a simple VI program so that all the implemented functions on the car can be activated through their VI panel. The final stage will require the students to write code for the C8051F040 microcontroller to directly implement the features in C that were previously implements in the VI. They will be expected to use their newly gained LabVIEW skill to aid in the troubleshooting and debugging of the CAN bus commands during C code development.

In order to facilitate easy grading and student learning, two separate LabVIEW VI will be developed. One will be the TA's version, which is the complete, final solution to the laboratory assignment. The TA version will have instruments wired correctly with proper IDs to receive data from the PIC microcontroller, as well as a text debug output to see all CAN data coming in on a CAN port. The student version will be missing the code to receive data for any instruments present on the car.

Functional Description
The Microprocessor Systems class implementation of CAN Bus is focused on creating a lab to control and receive data from a remote controlled (RC) car via a CAN Bus. The Bus network will employ the use of two SI 8051 Controllers, one for Control and one for Data Acquisition. The 8051 microcontrollers are connected via a CAN Bus to a PIC microcontroller on a small RC car. The CAN Bus will employ the use of the LabVIEW CAN Demo Box, which simulates CAN data over a serial connection. This Demo Box will be utilized on the CAN bus as a random CAN Data generator and later as a snooping tool to display the CAN Data from the RC car. In Stage 3 the LabVIEW Demo Box will be eliminated and data will instead be retrieved through the 8051 microcontrollers.

The Laboratory will be designed for implementation in three stages, each building on the other to increase the students' understanding of the CAN protocol. Students will learn how the CAN protocol works by first looking at the protocol itself and seeing sample data. Then, they will experiment with data control and acquisition through the bus. Finally, they will develop code to send and receive CAN-formatted data on the 8051 microcontroller as well as in LabVIEW.

PART I

This stage displays CAN message data for students to further their understanding of the CAN message protocol. In this stage of the lab, the LabVIEW CAN Demo Box is connected to the student's computer via a serial connection. The Demo Box's simulated CAN data is displayed in LabVIEW via the LabVIEW-provided and custom designed VIs. The CAN messages will be read on the user's computer via a GUI to display and understand the CAN messages being sent. The ultimate goal of this stage is to understand how CAN messages are sent and received and the physical layout of the network. This will be reflected in the provided lab documentation.

The provided custom LabVIEW front panels CAN-sender-MPS.vi & CAN-viewer-MPS.vi for the last part of this exercise will be the same one used for the next steps of the lab, but the RC car will not be used. Only the text field will update and show what data is coming in. Since this section of the lab is only for discovery of the CAN protocol the instruments are not needed.

PART II

This stage expands upon many of the features in PART I by adding in the RC car with the CAN Demo Box on the CAN Serial bus. Rather than providing random data via the CAN Demo box, CAN message data is displayed from the RC car. Students can execute commands using different control mechanisms and will see that they are executed on the LabVIEW GUI. Both control and data acquisition utilize the CAN Bus connection between the RC car and the LabVIEW Demo Box, while the viewer VI acts as a snooping tool. Students are asked to copy CAN-viewer-MPS.vi to another file and add new control panel indicators to it to display all the data available from the RC car.

Part III

This part provides the same functionality as PART II, but instead utilizes the 8051 microcontroller to send and acquire the CAN data; the same functions as the LabVIEW VIs. In this implementation, two C8051F040s generate the same CAN commands (one for sending commands and one for receiving status data) and display information from the PIC microcontroller residing in the RC car. As in previous stages the students will be able to view the status of different values on the car via the LabVIEW GUI as a troubleshooting tool.

During this stage of the lab, students will be required to implement code to receive CAN data and output to the instruments on the front panel. An example of one instrument will be provided, and students should complete any others required for the car. The full set will be present in the TA version.

Expandability
This project can be later expanded by other Capstone teams by providing improved laboratory procedures for more complex CAN Bus implementations and control schemes. The expandability of this project will be largely determined based off of the needs of the Microprocessor Systems class and Professor Kraft

Resources
The resources that will be used in tandem with the deliverables for this lab are:
The course webpage (includes Course Handouts, Manuals, and Lab Exercises) – MPS Webpage
For the specific Controller Area Network Lab - MPS Handouts

Overview of the Controller Area Network

The Controller Area Network bus, or CAN bus, is a vehicle bus standard designed to allow devices and microcontrollers to communicate with each other within a vehicle without a host computer. This document will guide a user through understanding the physical and data link protocols for the CAN bus as well as specific information about the Microcontrollers and installation and bring up of the CAN bus.

Controller Area Network Physical Layer Description

CAN Bus Physical Layer Protocols

The CAN Bus Physical Layer Protocol is described in the ISO 11898. The most common protocol is ISO 11898-2, which defines the use of a two-wire balanced signaling scheme, commonly referred to as high-speed CAN. ISO 11898-3 defines another two-wire balanced signaling scheme for lower bus speeds and provides fault tolerance capabilities; it is often referred to as low-speed CAN.

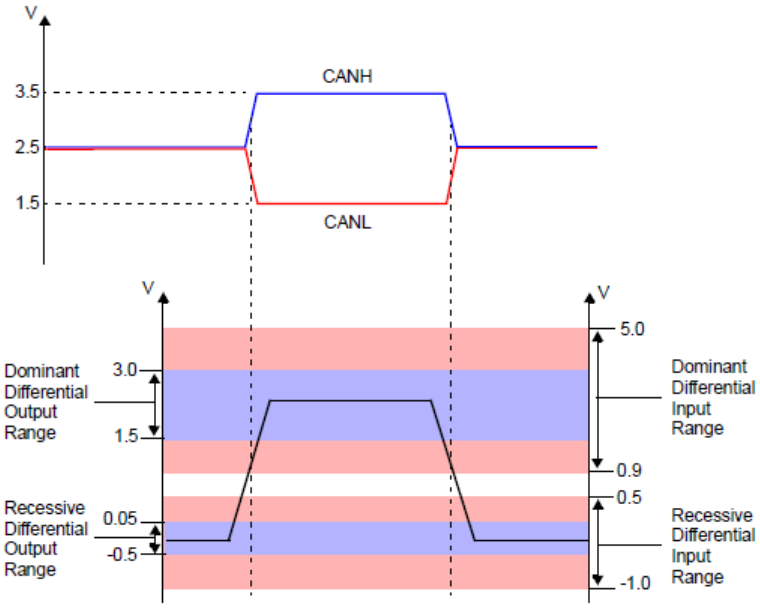


Figure 9: CAN Signaling and Differential range

The most common media is a twisted pair 5 Volt differential signal which will allow operations in high noise environments. Figure 9 shows the ranges for the CAN high and CAN low signal. For a short description of the different Physical Layer Standards and Protocols visit this webpage. Below is Table 1, which describes the major CAN Bus Physical Layer Protocols:

Table 1: CAN bus protocols

Standard	Common Name	Bit Rate	Max nodes (ECU)	Max Length
ISO 11783	ISOBUS	250 KBit/s	30	40 m
ISO 11898-2	High speed-CAN	max 1 MBit/s	110	6500 m
ISO 11898-3	Fault Tolerant CAN	max 125 KBit/s	32	500 m
SAE J1939	J1939	250 KBit/s	30	40 m

Bus Speed

The maximum bus speed for the CAN bus, according to the standard (ISO 11898-5), is 1Mbit per second. Low-speed CAN only handles bitrates of up to 125kbit/s. Single wire CAN handles up to around 50kbits/s in its standard mode and, using a special high-speed mode used for ECU programming, up to around 100kbit/s. There are many different Controllers and Transceivers that can handle high speed CAN, the data sheets will contain more information on the maximum and minimum Bus Speed.

CAN Bus Connectors

There is no standard for CAN bus connectors. However, higher layer protocol defines preferred connector types, which includes:

- 9-pin DSUB (CiA)
- 5-pin Mini-C and/or Micro-C (DeviceNet and SDS)
- 6-pin Deutch connector (CANHUG)

Though an industry standard has not been officially defined, many use the 9-pin DSUB cabling. Below is Table 2 which describes the Pin out description for the 9-pin DSUB connector layout with Figure 10 a picture of the DB-9 connector:

Table 2: CAN Pin out description for db-9 connector

Pin #	Pin name	Pin Description
1	-	Reserved
2	CAN_L	CAN_L bus line (dominant low)
3	CAN_GND	CAN Ground
4	-	Reserved
5	(CAN_SHLD)	CAN Shield (Optional)
6	(GND)	CAN ground (Optional)
7	CAN_H	CAN_H bus line (dominant high)
8	-	Reserved (error line)
9	CAN_V+	Power (Optional)

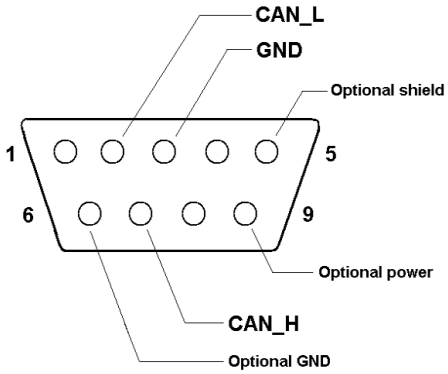


Figure 10: DB-9 connector

CAN Bus Cable Length Properties

The cable length of the CAN bus is limited by the Arbitration scheme utilized by the messaging system. The scheme requires that the wave front of the signal must be able to propagate to the most remote node and back again before that bit is sampled. Thus the CAN bus length is dependent upon the speed of the CAN bus system. Table 3 below contains the relationship between cable lengths and the bit-rate:

Table 3: CAN maximum allowable bit-rate

Bus Cable Length	Maximum Allowable Bit-Rate
40 meters	1 Mbit/s
100 meters	500 kbit/s
200 meters	250 kbit/s
500 meters	125 kbit/s
6 kilometers	10 kbit/s

CAN Bus Propagation Timing and Delay

Signal propagation plays an important role in bus arbitration at the data link level. The size of the propagation delay segment in the CAN bus is dependent upon the bit-rate and the length of the bus. The signal propagation delay is defined as the time that it takes a signal to travel from one node to the one farthest away on the bus (taking into account the delay caused by the transmitting and receiving node), synchronization and the signal from the second node to travel back to the first one. The maximum signal propagation delay can be determined using the two nodes within the system that are the farthest apart from each other. The first node can determine whether its own signal level (recessive in this case) is the actual level on the bus or whether it has been replaced by the dominant level of another node. To mathematically describe the propagation time, we use the following equation:

$$t_{propagation} = 2(t_{table} + t_{controller} + t_{optocoupler} + t_{transceiver})$$

CAN Bus Termination

When using the high speed CAN, the CAN bus must be properly terminated with two 120 Ω resistors at each end of the bus regardless of the speed of the bus. This ensures that the bus gets the correct DC levels for proper operation and that all signal reflections

are completely removed without any reflections. Below is Figure 11 displaying the proper termination of the CAN Bus:

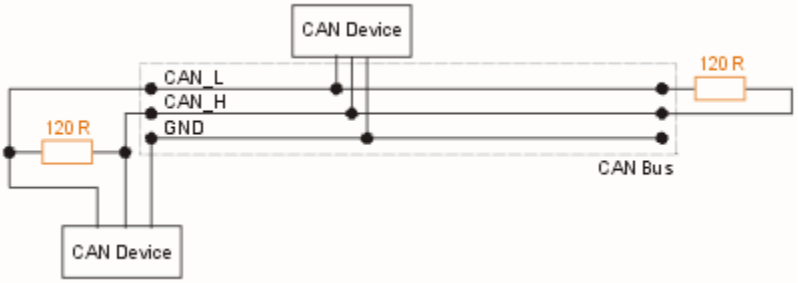


Figure 11: Proper Termination of CAN bus

There are many different termination methodologies that exist for the CAN Bus system. Standard Termination employs the use of two 120 Ω resistors as stated above. The utilization of Split Termination enables for easily achieved emission reduction. Along similar lines, Biased Split Termination is a method used to maintain the common mode recessive voltage at a constant value, thereby increasing EMC performance. Split Termination uses four 60 Ω resistors with a bypass capacitor tied between the resistor and the ground. See below for Figure 12 outlining the three different termination concepts:

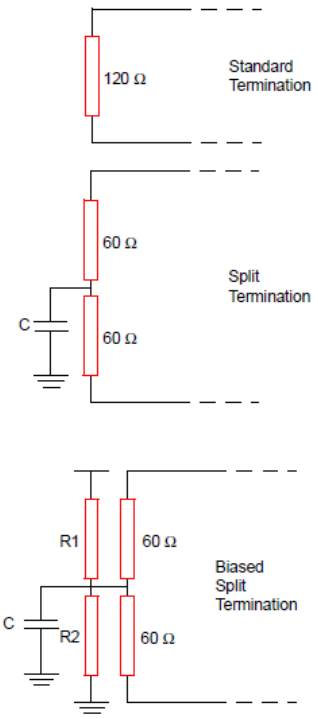


Figure 12: Termination Schemes for CAN Bus

Termination of the CAN Bus may only be necessary at high bit rates. Below is Table 4 describing the results of a previous CAN Bus Termination Test at different bit rates:

Table 4: Port Communication according to bit-rate

Bit Rate	Cable Properly Terminated?	Port to Port Communication Successful?
1 Mbit/s	Yes	Yes
1 Mbit/s	No	No
125 kbit/s	Yes	Yes
125 kbit/s	No	Sometimes
40 kbit/s	Yes	Yes
40 kbit/s	No	Yes

Controller Area Network Data Link Layer Description

Bit Encoding

The CAN bus utilizes Non Return to Zero (NRZ) bit coding. In NRZ bit coding the signal level remains constant over the bit time and thus just one time slot is required for the representation of a bit (as opposed to multiple for Manchester or PWM).
The signal level can remain constant over a longer period of time; therefore measures must be taken to ensure that the maximum permissible interval between two signal edges is not exceeded. This is important for synchronization purposes. Bit stuffing is applied by inserting a complementary bit after five bits of equal value. The receiver must then un-stuff the stuff-bits so that the original data content is processed.

Bit Timing and Synchronization

On the bit-level CAN uses synchronous bit transmission as displayed in Figure 13 below. This enhances the transmitting capacity but also means that a sophisticated method of bit synchronization is required. In this type of synchronous transmission protocol there is just one start bit available at the beginning of a frame, thus the receiver must be continuously resynchronized to correctly read the messages. Phase buffer segments are inserted before and after the nominal sample point within a bit interval.
The CAN protocol regulates bus access by bit-wise arbitration. The signal propagation from sender to receiver and back to the sender must be completed within one bit-time. For synchronization purposes a further time segment, the propagation delay segment, is needed in addition to the time reserved for synchronization, the phase buffer segments. The propagation delay segment takes into account the signal propagation on the bus as well as signal delays caused by transmitting and receiving nodes.

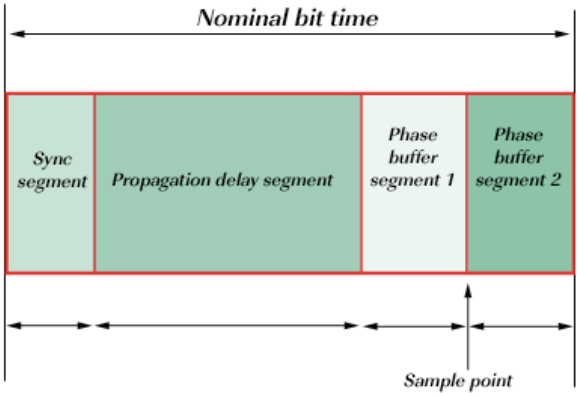


Figure 13: CAN Nominal bit-time description

Two types of synchronization are distinguished: hard synchronization at the start of a frame and resynchronization within a frame. After a hard synchronization the bit time is restarted at the end of the sync segment. Therefore the edge, which caused the hard synchronization, lies within the sync segment of the restarted bit time. Resynchronization shortens or lengthens the bit time so that the sample point is shifted according to the detected edge. The device designer may program the bit-timing parameters in the CAN controller by means of the appropriate registers.

CAN Bus Arbitration and Message Priority

The CAN bus arbitration is an important feature of the CAN bus system, and is the reason for the available transmission bandwidth. A CAN controller may start a transmission when it has detected an idle bus. This may result in two or more controllers starting a message (almost) at the same time. The conflict is resolved in the following way. The transmitting nodes monitor the bus while they are sending. If a node detects a dominant level when it is sending a recessive level itself, it will immediately quit the arbitration process and become a receiver instead. The arbitration is performed over the whole Arbitration Field and when that field has been sent, exactly one transmitter is left on the bus. This node continues the transmission as if nothing had happened. The other potential transmitters will try to retransmit their messages when the bus becomes available next time. No time is lost in the arbitration process.

The node will, of course, win the arbitration and happily proceeds with the message transmission. However, no node will send a dominant bit during the ACK slot to acknowledge that the message was properly received. The transmitter will sense an ACK error, send an error flag, increase its transmit error counter by 8 and start a retransmission. This will happen 16 times; then the transmitter will go error passive. By a special rule in the error confinement algorithm, the transmit error counter is not further increased if the node is error passive and the error is an ACK error. So the node will continue to transmit forever, at least until someone acknowledges the message.

Message Scheduling

The CAN specification (ISO 11898) defines a collision resistant messaging protocol. Every CAN message begins with an 11 bit identifier. This identifier is unique to each device on the CAN network and is used to determine the priority of each device. The unique identifier also allows receiving devices to determine the source of a message.

The CAN bus is physically designed so that if any stations transmit a ‘0’, all devices listening on the bus will see a ‘0’. The only way for a ‘1’ to be read on the bus is if all transmitting devices are sending a ‘1’. This characteristic is used along with the unique identifier to prevent collisions. The first line of defense is to listen to the CAN bus and not transmit if any other device is currently transmitting. The second line of defense is to handle the case where two devices start transmitting at the same time. The transmitting devices listen to the bus while transmitting to be sure that the signal on the bus is the same as the signal that they are transmitting. If a device transmits a ‘1’, but it reads a ‘0’ it means that another device on the bus is transmitting its identifier as well. The device that has its signal drowned out stops transmitting because the device that transmitted the ‘0’ has a higher priority. By the time the entire identifier is transmitted, all devices on the bus that are not the highest priority will have backed off. The device that didn’t need to back off can simply transmit the rest of the message and be sure that it didn’t collide. Figure 14 describes visually how this process occurs.

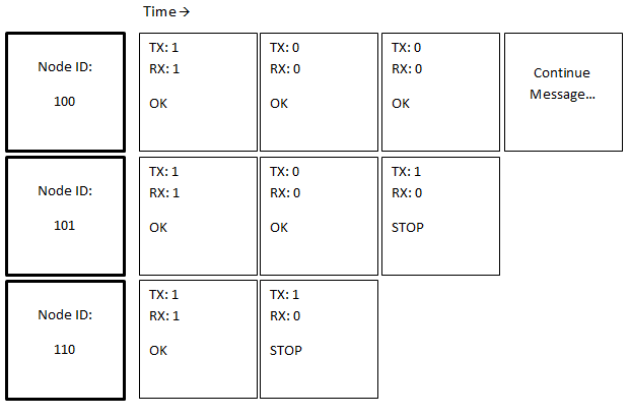


Figure 14: Example of Multiple Nodes Attempting to Transmit Simultaneously

Error Correction

To handle errors that occur in transmission, the CAN protocol defines error detection mechanisms. These mechanisms include transmitter-based-monitoring, bit stuffing, cyclic redundancy checks, and message frame checks. To ensure that errors are found by all nodes, any node that finds an error will set an error flag to inform all other nodes that the message was corrupt. All nodes then discard the corrupt message and the original sender node retransmits the original message. This retransmission is subject to the same arbitration on the bus as any other message.

The CRC enforces error checking by appending 16 bits to the end of the message. 1 bit is used as a delimiter while the remaining 15 bits contain the checksum. The CRC cannot detect every bit error and does not allow for message reconstruction like a set of parity bits could, but it detects errors to a satisfactory level.

The message frame (Figure 15) check finds errors by looking for bits that should always be recessive that have been read as dominant. These bits are the SOF, EOF, ACK delimiter, and CRC delimiter bits.

Figure 15: CAN Message frame with 11 bit identifier

Because the transmitting node is always listening for its own signal, the transmitting node can detect errors in the line itself.

CAN also uses bit stuffing to detect errors. Messages with 5 consecutive bits must have a complementary bit stuffed in. The stuffing allows there to be a rising edge for synchronization purposes. It also keeps the message for being mistaken for an error frame or the 7 bit interframe space that indicates the end of the message. If a message is read with more than 5 consecutive bits, there must have been a transmission error.

Computer Area Network Microcontrollers and Transceivers

CAN Controller MCP2515

One of the options for CAN Controller was the MCP 2515 Chip, which is dedicated to performing only the role of the CAN controller. This is the simplest solution for the CAN network, and includes all of the necessary features to operate full CAN V2.0B. The components and features, as outlined in Figure 16, of this chip can be classified into three categories:

- The CAN System
- Control Logic
- Serial Peripheral Interface (SPI)

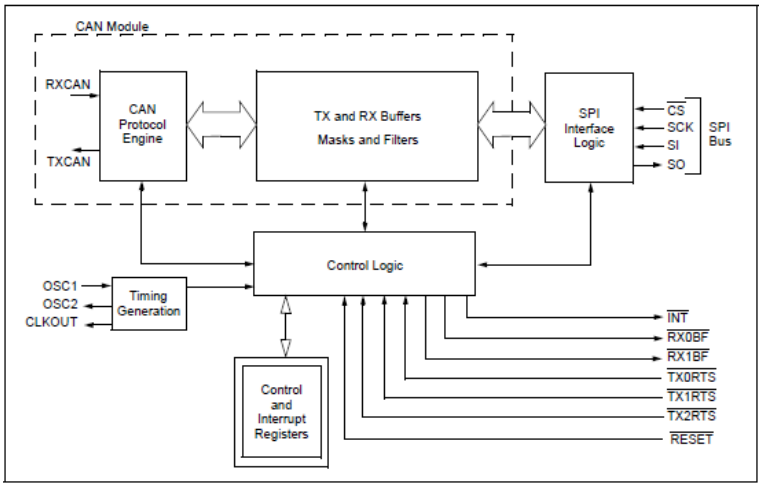


Figure 16: MCP2515 Block Diagram

The first important function the MCP2515 controls is to interface with the CAN bus/network in order to facilitate communication with other CAN nodes. It is important to note that the CAN architecture only concerns the transmission and reception (Tx/Rx) of data in the form of CAN messages.

The role in Tx/Rx that the controller plays is largely through a series of buffers that handle the data coming from and going to the CAN network. Transmission is controlled on the Tx buffers via Tx pins, or alternatively, writing to the memory registers with the SPI. Reception is configured with various masks and filters that determine whether a valid message from the CAN bus is passed to one of the Rx buffers.

The second module in the architecture is the control logic portion. This system is implemented in order to control the CAN system and Tx/Rx buffers. It includes several control pins, including a general-purpose interrupt pin for each Rx buffer, and three Tx pins. The use of the Tx pins for control is optional because the SPI can be used to set the control registers to handle transmission.

The third important part of the CAN controller is the SPI, which provides a user interface via a standard DB-9 serial port. The MCP2515 includes a SPI bus to allow a user to read and write to the registers on the chip. The SPI accepts standard commands, along with specialized commands for CAN implementation. These commands are necessary for allowing manipulation of the Tx/Rx properties of the MCP2515.

CAN Controller C8051

The C8051F044-GQ Integrated Circuit (Figure 17) is a multi-purpose microcontroller with a very wide range of applications. When considering the CAN capabilities of the C8051 processor, no major practical differences from the MCP2515 can be found. It has a dedicated core CAN processor, RAM for use as a Tx/Rx buffer, and control bit registers. The Bosch CAN processor handles the transmission of data and filtering; the RAM stores incoming data, message objects, and identifier masks; and the control registers are used with a serial interface for the same purpose as the SPI bus on the MCP2515.

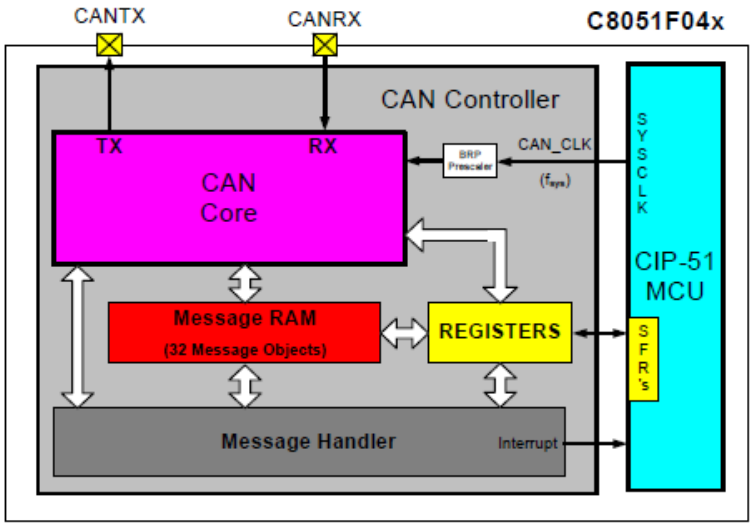


Figure 17: CAN Architecture in C8051 IC

CAN Transceiver MCP2551

The CAN-based system that we will implement will also require a chip to handle the signal transmission, and act as an intermediary between the controllers and the bus. The MCP2551 chip uses differential Tx/Rx and provides an interface between the CAN controller and bus, as needed. The MCP2551 offers many different protection features including:

- Battery short protection
- Electrical transient protection
- Thermal shutdown protection

It operates as either ‘dominant’ or ‘recessive’, depending on the voltage of the TXD pin. During operation, it can be in the high-speed mode when a fast Tx time is needed, slope-control mode when a slower speed is needed, and standby if Tx is not being used. Slope-control is a useful feature for avoiding electrical interference when the full Tx bandwidth is not needed.

CAN Node V2.0 Technical Features

In response to the many issues with the current node, the Capstone group has redesign a new version of the CAN Node. Below is a list of the changes from the CAN Node V1.0 and improvements to be made to the CAN Node V2.0:

I/O Screw Terminals – The CAN Node V2.0 will use screw terminals for the I/O connectors instead of soldering the connections directly to the board. This is better because it is a less permanent type of connection and it is also much more durable. This provides a better guard against the typical vibrations that occur on a Formula Hybrid Car.

Bypass Capacitors – The addition of a bypass capacitor on the CAN Bus will insulate the line against switching noise. This will improve the overall signal integrity of the CAN Bus.
Hardwire Terminator – Rather than using a Jumper for the Terminator, hardwiring it directly to the CAN Bus will remove a redundant connector and the removal of a menial part makes the Node more cost effective.

PIC Microcontroller – The PIC microcontroller is the biggest and most significant change to the CAN Node V2.0 as it offers many improvements over the Arduino. The most apparent is the built in CAN controller which consolidates the component into the Microcontroller reducing the effective size and cost of the Node. PIC also offers industry standard hardware performance, dealing with Jitter and other SI issues very well, with easy to access libraries for CAN applications.

External Crystal Oscillator – The External Crystal Oscillator provides a more stable clock signal than the internal RC clock on the PIC microcontroller.

DB-9 Connector – The CAN Bus connector will be changed to a DB-9 connector primarily to provide a more robust and consistent connection between the CAN Nodes. This will provide improved reliability to the message transmission. The line will contain CAN_H, CAN_L, power, and ground connections.

In-Circuit Serial Programmer – The In-Circuit Serial Programmer provides the capability to easily connect to the microcontroller to update and change the firmware.

Figure 18 is the schematic of the version 2.0 CAN Node using the PIC microcontroller.

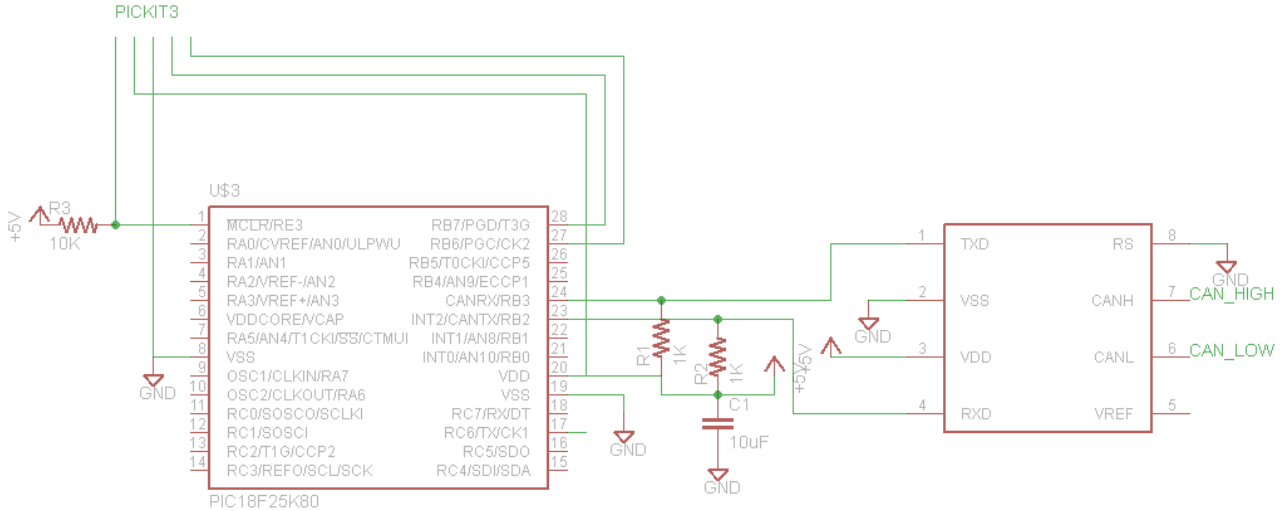


Figure 18: CAN Node V1.0 Schematic

Formula Hybrid Recommendations for Future Teams

The RPI Formula Hybrid team attracts many talented and motivated young engineers who are eager to both learn in a hands-on fashion and to design and build a competitive hybrid car. Unfortunately, due to the organization of the team as a club, external time constraints and some endogenous cultural defects, there are some issues that tend to hinder the ability of a capstone team to effectively work for or with the formula hybrid team. Since the formula team is organized as a club, there will often be uneven time and effort dedicated on their part to working on the car outside of 'crunch time' near external deadlines set by the competition. This issue is compounded by a prevailing attitude within some segments of the formula team that the tight schedule precludes rigorous engineering analysis from being done as "there is no time for it"; as one might expect, this attitude is entirely counterproductive and leads to more rather than less time being taken to complete most tasks in addition to limiting the quality of the final product.

There are a number of strategies to effectively eliminate or mitigate the negative effects of these problems. First of all, it is important to establish as early as possible the scope of work to be done by the capstone team, both in terms of the technical requirements and specifications and the division of labor such that the capstone team will be certain of the extent of their authority and responsibilities. Secondly, the schedule for the capstone team and the electrical sub-team of the formula team to complete their goals for the semester should be developed and agreed upon at the earliest opportunity. This will prevent issues where the capstone team is relying on a decision or working system to interact with that the formula team is reluctant to finalize due to misunderstandings or disagreements with regards to said schedule. Finally, it is crucial for the capstone team to be as proactive as possible with the decisions that are within their authority to make and should essentially never try to make a joint decision with the formula team on such matters, as they are extraordinarily reticent to make some decisions and for others the expertise of the capstone is needed for the decision to be made. The best path to take for decisions entirely within the purview of the capstone team is to simply make them and explain the rationale to the formula team afterwards at the next formula hybrid meeting, with calculations as necessary. For decisions that affect things that the formula team will have to interact with or which are not entirely within the domain of the capstone team, the best course of action is for the capstone team to analyze the problem and distill the possible solutions into two or at most three well-defined options and to present and explain these options to the formula team for a quick decision.

Formula Hybrid LabVIEW User Interface Manual

How to Use the VI

1. The first step is to install the latest version of LabVIEW on the laptop to be used, if it isn't installed already. Follow the steps in the section: "Installing NI-CAN and LabVIEW." Then follow the instructions in the section: "Setting Up and Using the XBee Wireless Serial Connection."

2. The LabVIEW program **serial can viewer final.vi** will be used to acquire CAN messages wirelessly over the XBee and view the data from each sensor as indicators on the front panel. Be sure that the laptop also contains the accompanying files that the LabVIEW program will call, which are listed in the Appendix. Open the program on the laptop. Connect the XBee receiver to the computer, and then select the corresponding port from the drop down menu on the front panel labeled "Serial Port." Be sure that the toggle switch on the front panel is set to serial.

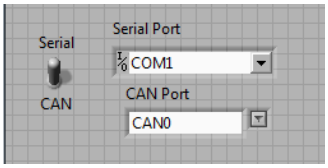


Figure 19: Serial Connection Setup

3. Each time an indicator is updated, its new value, along with its ID number and the elapsed time, will be written to an Excel file. To log the incoming data to a spreadsheet, the desired folder must be selected on the front panel control labeled "Directory." LabVIEW will append a time stamp to the chosen file path so that a unique file name is created for each run. Also, each of the indicators will be graphed in a chart on the front panel. To view different charts than the ones shown, simply click on the corresponding tab of one of the chart tab control boxes on the front panel. Furthermore, by right clicking on a chart after the run is complete and selecting "Export," and then "Export to Excel," the data will be opened up in a blank spreadsheet, allowing for additional analysis. Note: the default value in LabVIEW for the chart history length is 1024 data points. Right click on each of the charts and select "Chart History Length…" to double check that they are set to a sufficiently high value, such as 1,048,576 (the limit on the number of rows in Excel 2007 and 2010 – older versions are limited to 65,536 rows).

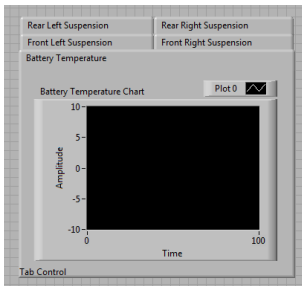


Figure 20: Tab Control Box with Charts

4. Certain indicators have critical thresholds that must be specified. To assign a critical threshold for the applicable sensors, type a value into the numeric control next to the corresponding indicator. These are the bounds that if surpassed, will alert the team that the current conditions are hazardous. Whenever a value goes outside of that range, LabVIEW will sound an audible beep and an LED will turn red. In order to hear this, select the buzzer2.wav file in the file path, ensure that the laptop's volume is set appropriately, and/or use headphones. If at any time the car is deemed unsafe by any of the sensors, stop the car immediately and proceed to diagnose the issue.

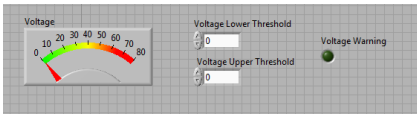


Figure 21: Indicator with Threshold Controls and Warning LED

5. Before starting the car, click the run arrow or type Ctrl+R to start the program. If at any time the program must be stopped, click the "STOP" button on the front panel. It is better practice to click on the stop control button on the front panel rather than the stop icon on the LabVIEW menu at the top of the screen.

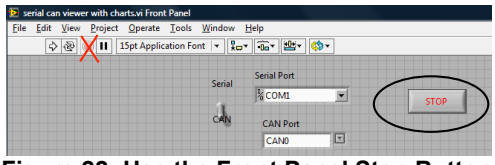


Figure 22: Use the Front Panel Stop Button

6. If future work requires that other sensors be added or changed, refer to the section: "Editing Serial LabVIEW VI." Type Ctrl+E to quickly switch between the front panel and the block diagram.

Editing Serial LabVIEW VI

This section identifies the process of changing the serial CAN viewer VI to alter CAN IDs and add/remove instruments from the front panel.

All other parts of the VI except for the highlighted portions shown in Figure 23 are protocol related and should not be changed.

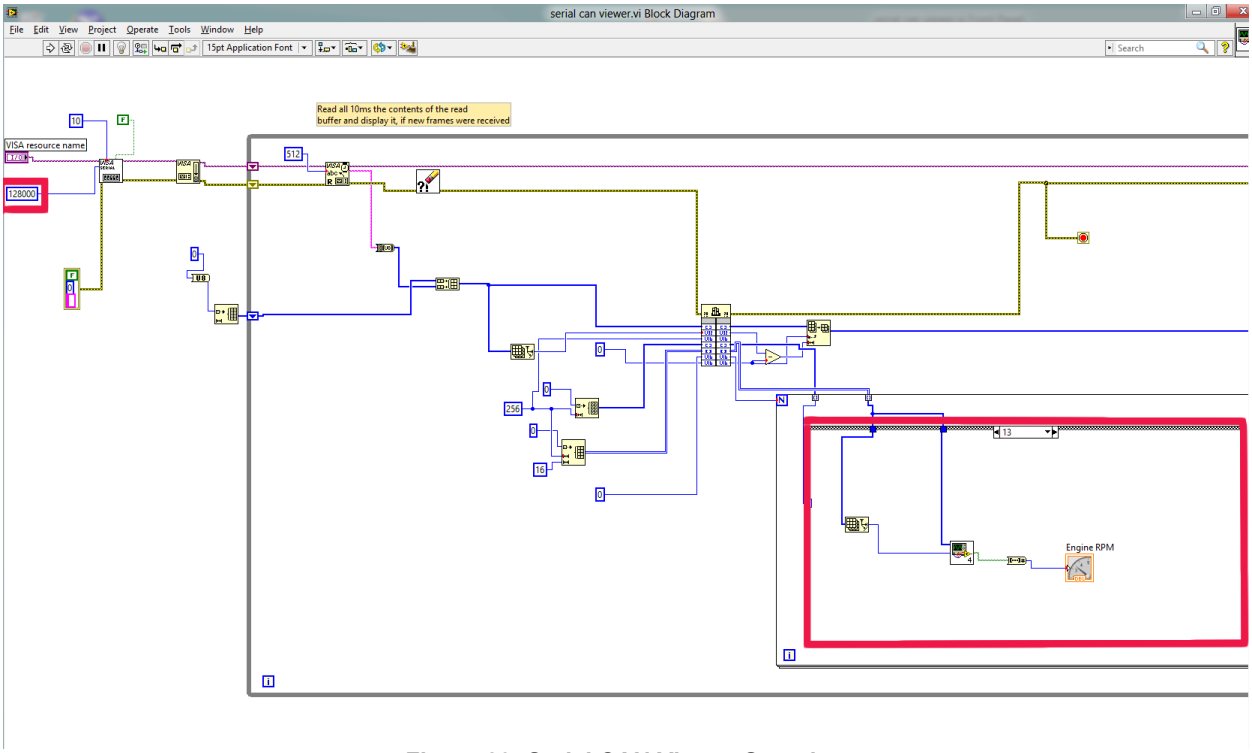


Figure 23: Serial CAN Viewer Overview

Baud Rate

The upper left most highlighted portion of Figure 23 is the baud rate. If necessary, change this value to the baud rate of the serial connection.

Instrument and CAN IDs

The other highlighted portion of Figure 23 is where data from the DLL (dynamic link library) block is identified by its CAN ID and displayed on the front panel.

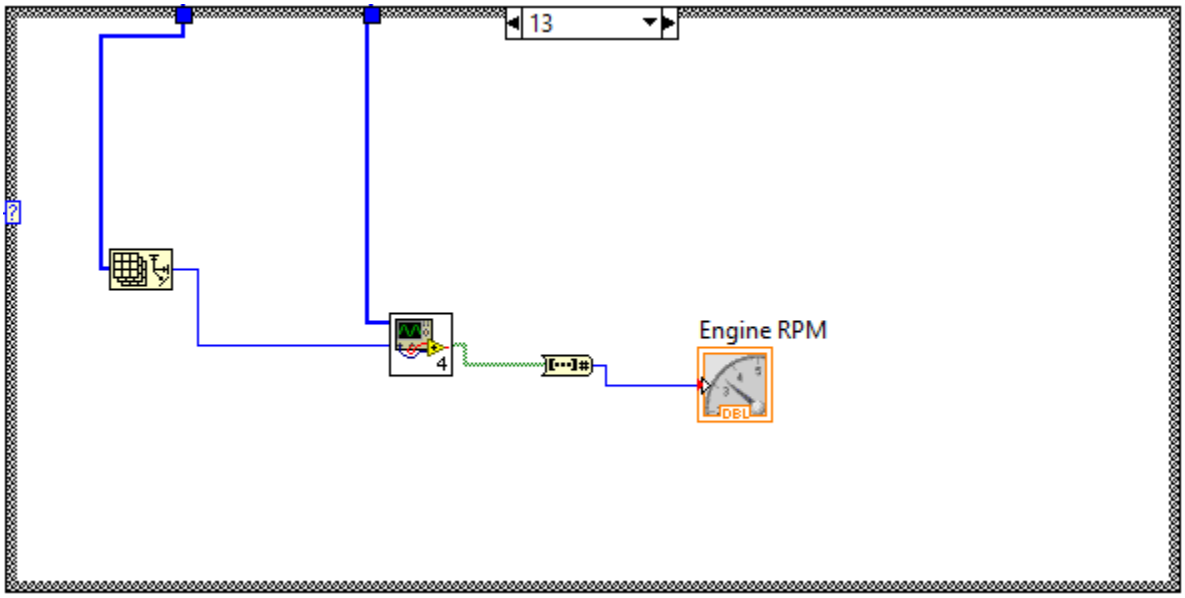


Figure 24: CAN ID and Instrument

A particular instrument on the front panel is showing the data for the given CAN ID. In this example the CAN ID is 13 (0x00D). To change the CAN ID for this sensor, change the value shown in the case statement.

Some processing is done on the data to change it from a byte to a bit array to a number. These are the three leftmost blocks in Figure 24. This is necessary if the instrument shows a numeric value. For Boolean values, refer to the Oil Warning sensor (ID 15) for data transformation.

To change the instrument, replace the existing instrument with a new one of your choice.

To add an ID, add a case to the case statement and copy over a similar case to modify with the procedure above.

Microprocessor Systems Final Design Specifications

Below is the schematic for the PIC microcontroller for the Microprocessor Systems application of the project. This is the microcontroller schematic (partial) for the RC car.