ECE 499 - Senior Project Report

Komatsu Sponsored - ECU Communication and Networking

Team Members: Zach Oakes, Christian West

Project Advisor: Aleksander Malinowski

Department of Electrical and Computer Engineering

March 29th, 2018

Abstract

Controller Area Network (CAN) communication is standard throughout the automotive and construction equipment industries. This is due to its low-cost, durability, and communication structure that enables it to broadcast messages to each device on the network in real time with a given priority. Testbenches in these industries commonly use CAN bus technology to detect faults in products that are both existing and under development. These testbenches provide the environment that is required to validate proper operation and ensure quality. This project aims to create a low-cost and easily-repurposable testing and development platform for academic projects relating to CAN communication. To achieve this, we mounted six Raspberry Pi 3b single-board computers with commercially available CAN shields, along with an Ethernet switch, and power supply. This topology also allows for direct access to each device, enabling a developer to easily control them over Ethernet. The result of this projects yields a viable testing and development bench for CAN network communication at a low cost due to the use of common off-the-shelf components. The accessibility of this test bench will make for smoother development of future projects related to networking or automotive applications.

I. Introduction

The automotive industry uses an Electronic Control Unit (ECU) to control electrical systems or subsystems in a vehicle. Networks of these units are capable of controlling and monitoring entire machines such as those found in the construction and mining industries. As the number of ECUs in a network continue to increase they become more complex. Managing this increase in sophistication has become an important challenge for companies to overcome. This is where testing and validation becomes useful. This is a step in the development cycle where manufacturers perform FMEAs (failure mode and effect analysis) to discover bugs and failures early on in the development process. These tests are typically performed on testbenches: hardware (or virtual) environments that enable the testing of other hardware and software features.

It would be advantageous for automotive companies producing heavy machine equipment to implement a robust ECU network that is “smart” enough to recognize each connected module along with important information. Such information could include serial numbers, ECU identifiers, and the truck frame number[i] the network is on. Additionally, such a network might have the ability to handle memory retainment for the swapping of ECUs or detecting the addition of unauthorized modules. This is very appealing, as this type of network can serve as a platform to monitor where and when a machine is located and to collect data such as diagnostics and prognostics that could potentially be sent to a central database for monitoring. This will also increase security ensuring that paid features will only be enabled on the intended machines.

This project should deliver a prototype of a simulated ECU network that encompasses these features along with proposing alternative methods in order for these features to exist.

II. Problem Statement

Komatsu is interested in exploring new designs to modify the current ECU networks on their ultra-class mining trucks. The goal would be to increase security and introduce more functionality. This research and development will serve as an early step to enable Komatsu to uniquely identify and track every ECU in the field. The ability to track every ECU individually will create accountability for each Komatsu truck in service as well as allowing for autonomy in servicing these ECU networks.

The following are the required functionality that will need to be implemented into the test network:

1. Simulate a CAN Network of ECU’s utilizing Master-Slave topology
2. Develop software to obtain each module’s ID number along with the truck frame number (a unique identifier for each truck chassis, like a VIN for on-road vehicles)
3. Information retainment in case of network powering off
4. Develop a method to recognize, authorize, and update new modules replaced or added to each system
5. Create a protocol to allow for a secure transfer of ECU information

Completing this work will allow for current and future ECU networks to become "smart" while saving time and money, with the potential for increased autonomy.

Currently, if a unit were to fail, an agent is sent to the site with equipment to manually program the new ECU with its installation information. If this information was already capable of being automatically obtained, then a smart network would eliminate the labor cost for programming and the potential for operator error.

III. Background

i. CAN Communication

CAN protocol is a communication method that allows different devices to communicate without a host computer. It accomplishes this by reducing the amount of harnessing to a single network, known as the CAN Backbone, that typically consists of a twisted pair of wire. This simple design allows for an inexpensive, durable network. CAN communication enables peer-to-peer or broadcast data transmissions meaning that all modules can transmit and receive data on the bus. Each module decides whether a message is relevant or if it should be filtered. With the intelligence within each module and not the network itself, modules may be added or modified with minimal impact. Another advantage to using CAN bus is message prioritization. This allows for non-interrupted transmission of higher priority messages and for the network to be deterministic. Finally CAN specification includes error detecting code. This is otherwise known as Cyclic Redundancy Code (CRC), that performs an error check on each frame. Errored frames are ignored.

In order to satisfy the physical layer of our testbench, the requirements of CAN communication were referenced. CAN High-Speed is defined by ISO11898-2 as a differential data bus. This behavior can be seen in the following image:

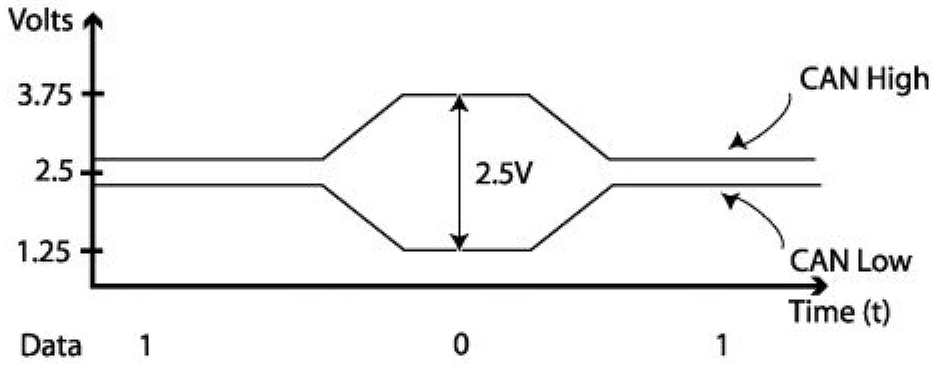

Figure 5.1 - CAN Traffic

From an idle voltage of 2.5 V, CAN_H and CAN_L pins are driven apart for logical 0 (dominant state) and back together for logical 1 (recessive state). If the bus is recessive, any device on the bus can assert dominant. This behavior is how messages are acknowledged: the sending unit writes recessive for one cycle and the receiving unit sends back dominant. If the bus still reads recessive at this time then the transmitting unit thinks that transmit failed since it hasn't been acknowledged.

Each node on the network are connected to a two wire bus that serves as the backbone of the network. This backbone supports all the communication done throughout the network. The wires used are a twisted pair at a nominal impedance of 120 Ω.

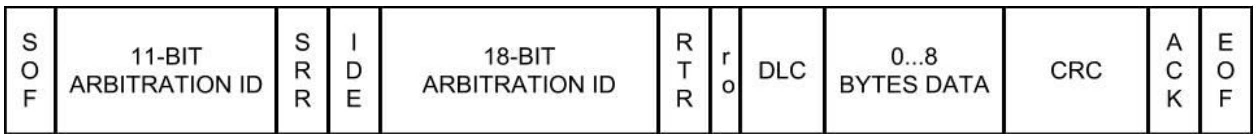

Figure 6.1 - CAN FRAME

The following defines the CAN frame in figure 6.1:

SOF (start-of-frame) bit – Start a message with a dominant (logic 0) bit.

Arbitration ID – identification and priority for the message. Indicates a standard 11-bit arbitration ID or extended 29-bit arbitration ID.

IDE (identifier extension) bit – differentiates between standard and extended frames.

RTR (remote transmission request) bit – bit used to request ID from a module that did not transfer within the time period. Normally the RTR bit is set to a dominant (logic 0) but set to a recessive (logic 1) RTR bit to indicate a remote frame.

DLC (data length code) – indicates the number of bytes the data field contains (0 - 8 bytes).

CRC (cyclic redundancy check) – 15-bits and a recessive delimiter bit for error detection.

ACK (ACKnowledgement) slot – single bit that is generated by any receiving module to indicate proper CRC. Transmission will reattempt no ACK is detected.

EOF (end of frame) – seven bit recessive sequence to indicate the end of the frame

ii. ISO Layers

The International Standard Organization (ISO) has developed a conceptual model to standardize terminology for talking about communications over a network. This model is called the Open Systems Interconnection model (OSI model). This model defines the seven layers that can be seen below in figure 5.1. The scope of this project only includes the first three layers: the Physical, Data Link, and Network Layers (bordered in blue in figure 5.1).

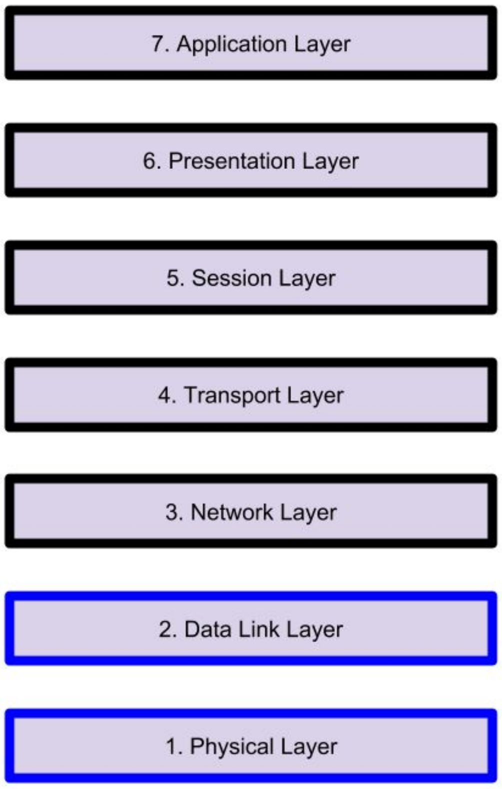

Figure 5.1 - OSI Model Layers

The Physical Layer of a network is the lowest layer. It addresses the physical characteristics of the network. These details include the electrical, mechanical, and procedural properties of the transmission media; anything that describes the physical data link connecting network nodes that will transmit the raw bits. For example: the types of connectors used, twist level of wires used, length of cable, etc.

The Data Link Layer is second layer of the model that is the first to assign meaning to the bits being transmitted over a network. Details include size of each packet, timing, basic error detection and correction, etc. Each device on the network at this level possesses its own unique identifier, commonly assigned during production.

Considered as the backbone of the OSI model, the Network Layer will decide what path data transfer will follow between nodes. These protocols, commonly Internet Protocol or Netware IPX/SPX, exist in every host and routers. This layer accepts requests from the Transport Layer above is, and sends requests to the Data Link Layer.

iii. Comparison of Encryption Methods

There are two schemes for encryption. These involve using either symmetric-key or asymmetric-key methods. The symmetric-key scheme, also known as private-key cryptography, uses the same key for encryption and decryption. This means that the key must remain secure. The asymmetric-key scheme, also known as public-key cryptography, uses two keys separately for encryption and decryption; a private key and a public key. The public key is used to encrypt messages and is made available, while only the holder may use their private key to decrypt the message. The following further describes a few different data encryption algorithms:

Blowfish -a symmetric cipher that is much faster than the methods Data Encryption Standard (DES) and the International Data Encryption (IDEA). It works by splitting messages into 64-bit blocks, encrypting each piece separately. This method is unpatented and royalty-free making it widely used in the public domain. Due to these reason, and that our goal is simply to encrypt CAN messages, we decided to use this method in our implementation.

Twofish -the successor to the blowfish algorithm and another symmetric cipher. It uses block sizes of 128-bits with keys that may be up to 256 bits. Twofish is also unpatented and freely available. This algorithm has proven to perform well where little RAM and ROM is available and where keys are changed frequently.

AES - another symmetric block cipher that is held as the standard by the U.S. government to protect classified information. It is capable at handling 128-bit blocks with keys up to 256 bits.

IV. Implementation

i. Hardware

Although Master-Slave topology is not required for CAN protocols such as J1939, we followed this topology to mirror our software implementation. Note that this has no effect on the build physically but it further illustrates our implementation- see figure 8.1. Each ECU is simulated using a Raspberry Pi; one serves as the Master ECU and the rest as slave ECUs. The CAN network will connect in parallel to each ECU with the ends terminated by 120 Ohm resistors.

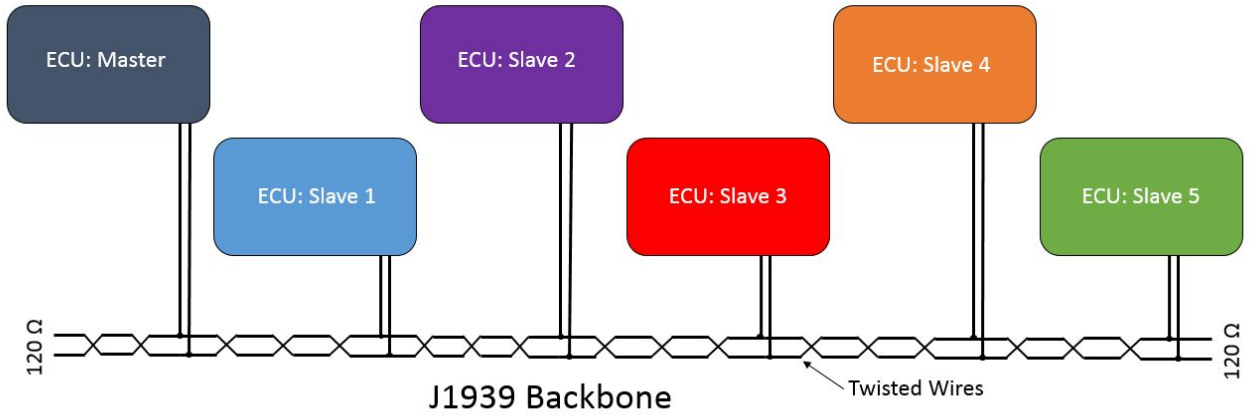

Figure 8.1 - System Diagram of ECU Network

a. The Physical Layer

The above topology is realized in the physical build as seen in figure 9.1. This encompasses the CAN backbone that is created by using twisted pairs of wires. The blue wire is designated as CAN_H where as the green is CAN_L. This backbone connects to each of the ECUs in parallel, which are being simulated with raspberry pis paired with a CAN shield called piCAN2. This shield, developed by Copperhill technologies, enables the raspberry pi to easily interface with the CAN network with its onboard DB9 connector, in our case, the screw terminal. Lastly, the red circles highlight the 120 Ω resistors used that yield the 60 Ω impedance between the CAN_H and CAN_L lines.

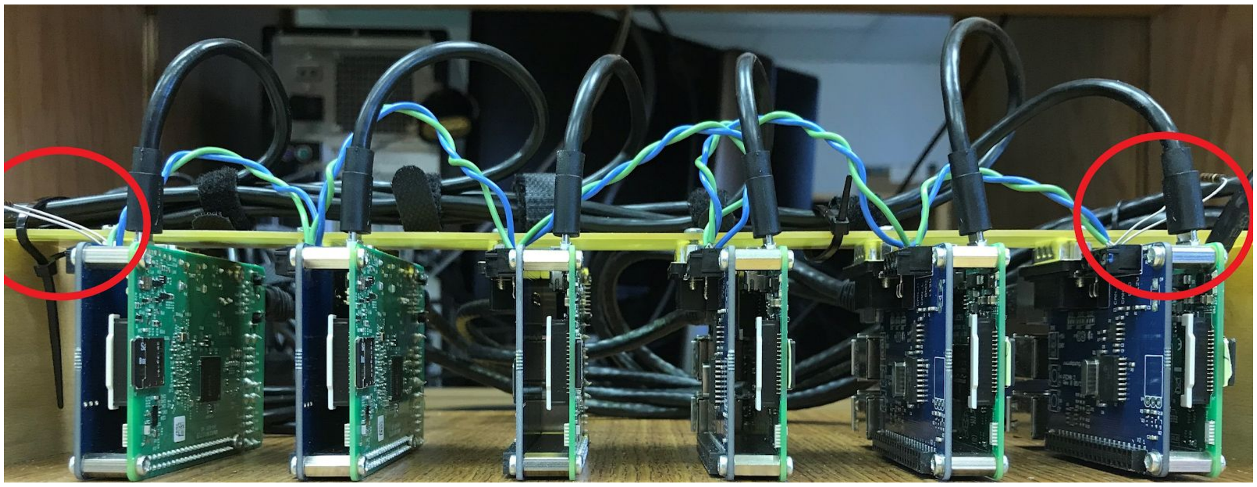

Figure 9.1 - ECU Network Implementation

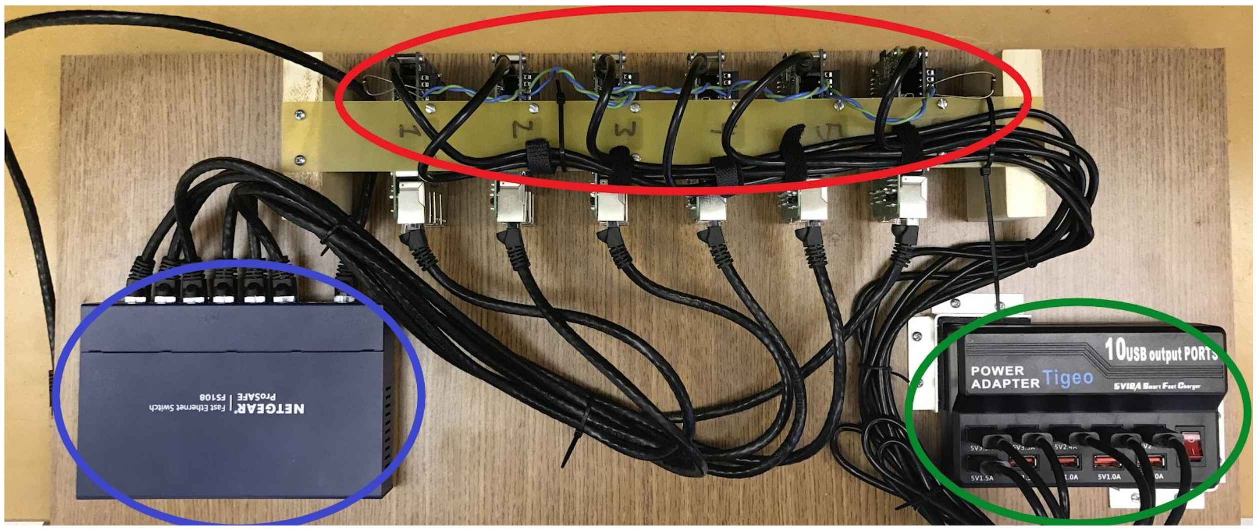

Figure 9.2 - Platform Design

b. Data Link

The platform design can be seen in figure 9.2 which includes the previously described ECU network and is highlighted by the red circle. The power is supplied by the Tigeo Power Adapter (highlighted by the green circle) which can supply up to 5V at 12 Amps. Since the Pi's are only running scripts and sending messages back and forth, this is sufficient. In terms of the data link layer, there is a network switch (highlighted by the blue circle) that is used as a network bridge. This allows for a user to directly access each module via ethernet. Each module is configured with a static IP address.

The full setup for development consists the entire platform connected to a laptop via the ethernet switch. Using other tools such as SSH, via putty, and a shared internet connection through the laptop, higher levels of the OSI model were utilized. This enabled us to install necessary python libraries, Coperhill's drivers, and full web access when developing.

ii. Software

All the code for our demos lives in one directory on the Raspberry Pi, this allows for easy deployment of changed code with only one `scp` command. The directory tree of the source code is as follows:

```
/home/pi/ecu
├── blowfish
│   └── __init__.py
├── ecu_comms
│   ├── ecu_info.xml
│   └── __init__.py
└── __init__.py
```

The top level `__init__.py` file is empty, but is required for using Python imports.

When the device is initialized by a call to `python ~/ecu/ecu_comms/__init__.py` under Python 2.7.x, it will make a new `ECU` object. This object will then read from the `~/ecu/ecu_comms/ecu_info.xml` file to find out the password for encryption and information about itself along with the rest of the devices that it thinks are on the network. Once this information is loaded, the master device iterates over the list of ECUs that it expects to find and confirms that they all still exist and have the parameters that it expected to see.

VI. Results And Future Work

Overall this project was pretty successful, our client is happy and we won the Dean's award for our work.

i. Results

Recall our functional requirements for this project:

1. Simulate a CAN Network of ECU's utilizing Master-Slave topology
2. Develop software to obtain each module's ID number along with the truck frame number (a unique identifier for each truck chassis, like a VIN for on-road vehicles)
3. Information retainment in case of network powering off

4. Develop a method to recognize, authorize, and update new modules replaced or added to each system
5. Create a protocol to allow for a secure transfer of ECU information

Most of the items here have been accomplished. Number 2 was decided to be out of scope, and a solid groundwork for number 4 is in place. The network is simulated in hardware, and information can be passed along it securely due to the blowfish encryption. The system is protected from power down by storing the network information in an XML file in nonvolatile memory.

ii. Future Work

Our project, while complete, has some optimization left to do.

It was pointed out to us at IAB presentations that our network doesn't do a good job of simulating the conditions found on an actual machine, because of the shortness of the wiring harness and the lack of ambient electrical noise in the lab. A good extension would be to try and add some realism by adding an electrical noise generator to the CAN bus.

Upon reviewing the XML file used to store network information it was brought up that XML might not be the best tool for the task of storing this information. A future implementation of the same functionality might instead use a JSON file, which is less verbose and maps better onto a Python dictionary than an XML file does.

Heartbeat functionality would also be good. Right now the only time that the network is scanned and verified is on boot. Ideally this would be a regular action ever couple of minutes, to make sure that the network hasn't been tampered with since boot.

References

http://copperhilltech.com/pican2-controller-area-network-can-interface-for-raspberry-pi/
http://sgframework.readthedocs.io/en/latest/cantutorial.html
http://copperhilltech.com/content/CIA_article.pdf
http://www.axiomatic.com/whatiscan.pdf ----C CAN traffic picture

https://www.schneier.com/academic/archives/1998/12/the_twofish_encrypti.html
https://www.schneier.com/academic/blowfish/
List data sheets
http://ww1.microchip.com/downloads/en/AppNotes/00228a.pdf

Appendix A - Bill Of Materials

Quantity	Description	Price	Ext. Price
6	Raspberry Pi3 Model B	$35.10	$ 210.60
12	8GB micro SD cards	$7.11	$ 85.32
6	CAN Interface for Raspberry Pi	$47.95	$ 287.70
1	USB HUB for Power	$23.99	$ 23.99
1	3 foot USB 2.0 (6 Pack)	$8.99	$ 8.99
1	Ethernet Switch	$23.28	$ 23.28
1	Ethernet Cables 2 Feet (10 pack)	$13.99	$ 13.99
		Total	$ 653.87

Table 1: Parts List ECU Network

Appendix B - Source Code

The entire source for this project can be cloned from Bitbucket

```
git clone git@bitbucket.org:km_ecu/ecm_software.git
```

This is a private repository as per the terms of our non-disclosure agreement with Komatsu, contact either of the authors to ask for access.

A. Python Code

The pure-python blowfish library used in this project was written by Larry Bugbee, and is available from his website (http://www.seanet.com/~bugbee/crypto/blowfish/blowfish.py), renamed to `blowfish/__init__.py` so it could be imported as a module.

The remainder of the code was contained the `~/ecu/ecu_comms/__init__.py` file, included below:

```
import xml.etree.ElementTree as ET
import can
from textwrap import wrap
import re
import sys
sys.path.append("/home/pi/ecu/")
from blowfish import blowfishCTR


class ECU:
    def __init__(self):
        """Read from disk the demographic info about this device and then set the
parameters appropriately """
```

```
        # Read in the info from the disk, if we have any
        self.bus = can.interface.Bus(channel='can0', bustype='socketcan')
        try:
            file_ = './ecu/ecu_comms/ecu_info.xml'
            root = ET.parse(file_).getroot()
            for child in root.find('self'):
                setattr(self, child.tag, child.text)
            self.network = list()
            for ecu in root.findall('ecu'):
                child_dict = dict()
                for child in ecu:
                    child_dict[child.tag] = child.text
                self.network.append(child_dict)
            self.password = root.find('password').text
            self.build_list()
        # This will be thrown if the file doesn't exist yet or hasn't set the right
values
        except (IOError, AttributeError) as e:
            print 'File for the attributes is corrupted or missing'
            raise e

    def poll(self):
        """Go down the list of ECUs and comfirm that we have all of them. Only the
master should do this"""
        for device in self.network:
            self.send_data(dest=device['serial'], data='{}?{}'.format(device['nick'],
self.serial))
            # todo recieve response and store it to a new array of devices
            print self.recv_data()

    def send_data(self, data, dest=0x0):
        """Send data to dest over the CAN network.
        For our purposes this will just mean that dest will be the CAN header
        and if data is bigger than one frame it'll be wrapped. This should work to
        send strings as bytearrays, 8 characters at a time"""
        print "enter send action, data is {}".format(data)
        secure = self.encode(data)
        print "encrypted message is {}".format(secure)
        secure = wrap(str(secure), width=8, replace_whitespace=True)
        for chunk in secure:
            print "trying to send {}".format(chunk)
            self.bus.send(can.Message(arbitration_id=int(dest), data=map(ord, chunk)))
        # send a blank message to show that it's the end
        self.bus.send(can.Message(arbitration_id=int(dest), data=[]))
        print 'Sent message: {} to {}'.format(data, dest)

    def recv_data(self):
        """Listen for messages to us and reassemble them back into strings
        todo make this have a timeout
        """
        data = ''
        for msg in self.bus:
            if msg.arbitration_id == int(self.serial):
                data += msg.data
                if msg.data == '':
                    break
        if data:
            return self.decode(data)
        else:
            return 'No Response'
    def build_list(self):
        """Build a list of devices on the network. If we're a slave we'll do this by
listening
        and if we're a master then we'll do this by polling for devices on the network
        """
        if self.MasterSlave == 'Master':
            self.poll()
        else:
            msg = self.recv_data()
            # if the message looks like an ID request
            if re.match('.+\?\d+'.format(self.nick), msg) is not None:
                # If it's asking about us specifically
                if self.nick in msg:
                    self.send_data("{}\n{}\n".format(self.MasterSlave, self.role),
dest=int(re.findall('\d+', msg)[-1]))

    def encode(self, message):
        encrypted_msg = blowfishCTR('e', self.password, message)
        return encrypted_msg

    def decode(self, secure_text):
        decrypted_msg = blowfishCTR('d', self.password, str(secure_text))
        return decrypted_msg


if __name__ == '__main__':
    device = ECU()
```

B. Example ecu_info.xml file

The `ecu_info.xml` file contains information about the current device (under the `self` tag) and the rest of the devices that are expected to be on the network (under the `ecu` tags). Additionally this file contains the password used for Blowfish encryption (under `password`). The file used in demos is below; note that only five ECUs are defined, as the sixth was used as an activity monitor.

```
<ecu_list>
    <self>
        <MasterSlave>Master</MasterSlave>
        <serial>001</serial>
        <role>Everything</role>
        <nick>Alice</nick>
    </self>
    <ecu>
        <MasterSlave>Slave</MasterSlave>
        <serial>002</serial>
        <role>Wheels</role>
        <nick>Bob</nick>
    </ecu>
    <ecu>
        <MasterSlave>Slave</MasterSlave>
        <serial>003</serial>
        <role>Brakes</role>
        <nick>Carol</nick>
    </ecu>
    <ecu>
        <MasterSlave>Slave</MasterSlave>
        <serial>004</serial>
        <role>Hydraulics</role>
        <nick>Dave</nick>
    </ecu>
    <ecu>
        <MasterSlave>Slave</MasterSlave>
        <serial>005</serial>
        <role>Suspension</role>
        <nick>Eve</nick>
    </ecu>
    <password>thisisthepasswordtobeusedforkomatsu123456</password>
</ecu_list>
```

Appendix C - Raspberry Pi Initial Setup

For the initial setup, the Raspberry Pi will need to have internet access. First, install the PiCAN2 shield onto the Raspberry Pi. Next, install any updates to Raspian that have been published since the last version.

```
sudo apt-get update
sudo apt-get upgrade
```

Then you'll need to get the setup for SPI communication added to the `/boot/config.txt` file. The easiest way to do this is with an elevated `nano` editor.

```
sudo nano /boot/config.txt
```

Add these lines to the bottom of the file:

```
dtparam=spi=on
dtoverlay=mcp2515-can0,oscillator=16000000,interrupt=25
dtoverlay=spi-bcm2835-overlay
```

Save the file and reboot the Pi.
If you'd like some debugging tools for the Pis, then you can get the CopperHillTech recommended ones at http://www.skpang.co.uk/dl/can-test_pi2.zip, just unzip that somewhere convenient on the Pi.

The next thing that you'll need to do is install Hardbyte's Python-CAN library. It can be found at https://github.com/hardbyte/python-can. Either clone the repo or download as a zip and unzip it somewhere convenient. Install the library proper with a setup.py command, except that Raspian doesn't usually ship with the needed library for this, so install the `python-setuptools` library first

```
sudo apt-get install python-setuptools
sudo python3 setup.py install
```

At this point everything that you'll need is installed on the Pi, so it's probably easier to make copies of that SD card than it would be to run through this whole process again.

Before you use the CAN bus you'll need to bring the interface up:

```
sudo /sbin/ip link set can0 up type can bitrate 500000
```

Once that is all done you should be able to run our code, so pull down our code as described in Appendix B and run by calling

```
python ~/ecu/ecu_comms/__init__.py
```