Cook Engineering Design Center

Final Paper

Submitted in partial fulfillment of the requirements for

ENGS 90: Engineering Design Methodology

Hot-Wiring of the Future: Exploring Car CAN Buses

March 6, 2013

Sponsored by

Siege Technologies

Project Team 9

Chris Hoder

Theodore Sumers

Grayson Zulauf

THAYER SCHOOL OF ENGINEERING AT DARTMOUTH

Hot-wiring of the Future: Exploring Car CAN Busses

Christopher Hoder
Grayson Zulauf
Theodore Sumers

Thayer School of Engineering
Dartmouth College
Hanover, NH 03755
March 2013

Contents

Acknowledgements

We would like to thank Professors Lotko and Collier, Professor Sergey Bratus, Professor Vincent Berk, Professor Doug Fraser, technical contacts Travis Goodspeed and Andrew Righter, our sponsor, Daniel Bilar and Siege Technologies.

Executive Summary

Over the past decade, automobile electrical systems have evolved into complex networks of sensors and microprocessors, generally referred to as Electronic Control Units (ECUs). A typical, modern car contains between 50 and 70 ECUs, which are responsible for virtually every aspect of normal operation, from anti-lock braking to engine timing. These ECUs communicate with each other over the car's Controller Area Network (CAN) bus using the mandated CAN protocol standard. Siege Technologies believes malicious parties could exploit these unauthenticated intra-vehicle networks, and has thus sponsored our team as an early-stage research project into the security flaws present in cars' CAN busses. The group was given three primary deliverables:

1. A proof-of-concept demonstration of vulnerabilities in CAN networks, by compromising one or more safety-critical car systems.

2. A methodology for uncovering and exploiting security weaknesses in the CAN bus of any vehicle.

3. A software package to implement that methodology on any vehicle.

We used our limited budget to purchase a 2004 Ford Taurus and successfully reverse-engineered the manufacturer-specific protocols used on its CAN bus. We developed and demonstrated hacks on safety-critical ECUs, most notably a complete denial-of-view attack in which we systematically manipulated every component on the dashboard, including faking speedometer readings.

The team documented its approach to reverse-engineering the test vehicle's higher-level protocols and produced a generalized methodology for characterizing and exploiting a vehicle's CAN bus. This methodology outlines a series of experiments to map out a vehicles CAN bus and begin decoding the messages on it.

As we developed our methodology, we built a software package to implement it. This was an iterative process; additional knowledge gleaned from experimentation allowed us to improve our software, and vice versa. The team first developed basic CAN communication capabilities for Siege's GoodThoper10 hardware (sniffing network data and injecting packets), then more advanced functionality for experimentation and hacks.

The final result is an intuitive interface allowing the user to view, store and analyze raw CAN data, quickly implement our experimental methodology, and use our code base to implement their own vehicle-specific hacks. We produced extensive user guides and documentation for each deliverable.

The project sponsor, Daniel Bilar of Siege Technologies, declared the project a success. In concert with Daniel and our project advisor, Sergey Bratus, the group is currently preparing to apply to present the project at REcon 2013, a conference with a focus on advanced exploitation techniques.

Future research into CAN vulnerabilities will require a modern test vehicle. Our ability to implement even more pernicious attacks, such as disabling the brakes, was limited only by the extent of the CAN bus in the test vehicle. Based on published academic research, we believe that more modern vehicles, which contain far more ECUs as well as wireless communication such as Bluetooth, will prove even more vulnerable to attack than our test vehicle.

Finally, the software package we developed compares favorably with state of the art commercially available analyzers; if Siege re-wrote the firmware on its GoodThopter10 board, the package could be marketed directly as a tool to allow hobbyists to reverse-engineer their own vehicles.

1 Overview

1.1 Project Background

Do you trust your speedometer?

A modern car contains a sophisticated array of sensors and computers, responsible for virtually every aspect of normal operation, from anti-lock braking to engine timing to the infotainment system. These Electronic Control Units (ECUs) are networked, silently passing vast amounts of information to each other over a common wire running through the car. Starting in 2008, new vehicles in the U.S. are required to use the Controller Area Network (CAN) communication protocol.

1.2 Problem Statement

Due to the isolated nature of individual cars' networks, the CAN protocol does not have any built-in security measures. As a result, car CAN buses implement unencrypted and unauthenticated communication between safety-critical components. Siege Technologies recognized this as an under-explored potential threat, both in the transportation sector and in emerging uses of the CAN protocol such as manufacturing and medical practice. They sponsored our team to conduct an early-stage research project into the security flaws present on cars' CAN buses to determine if the area was worth exploring further.

We developed a software package for interacting with and analyzing a CAN bus, and reverse-engineered the internal communication in our experimental car, a 2004 Ford Taurus. We were able to reliably gain control over the networked ECUs — including the speedometer, odometer, and door locks — and delivered our extensively documented methodology and code to Siege Technologies to facilitate further research. Our sponsors declared the project a success.

1.3 CAN Protocol Overview

The CAN protocol has three primary fields: an Arbitration ID (ArbID) to uniquely specify the transmitting ECU and encode the priority of the message, a 64-bit data field containing the information, and a robust error checking (CRC) field. Each data field utilizes a higher-level protocol (HLP) that must be known to decipher the actual information being transmitted. Figure 1 shows the different parts of a sample CAN message.

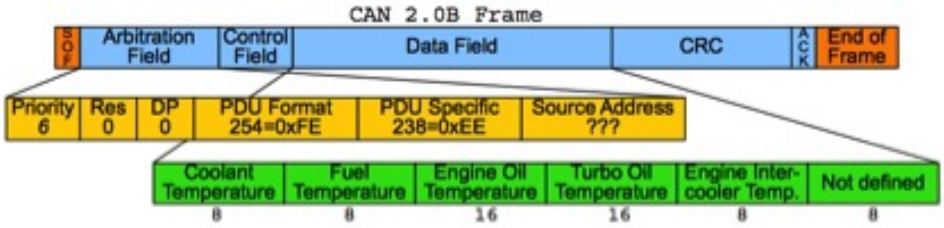


Figure 1: A sample CAN 2.0B frame, showing an ArbID associated with the engine diagnostics [1]. The green dissection shows the higher-level protocol encoded in the transmitted information.

2 State of the Art

2.1 Commercial Tools

A variety of commercially available tools connect to the CAN bus via the on-board diagnostics port (OBD-II). Most are capable of reading the vehicle's status but have very limited injection capabilities and cannot be used for reverse-engineering or hijacking ECUs. For a list of commercial tools, please see Table B.1 in Appendix B.

Licensed dealers and garages have access to more advanced diagnostic tools, which provide full access to the network. These allow trained users to reprogram and reset ECUs as desired for firmware update purposes. However, these tools are manufacturer specific, require a license to purchase, and are extremely expensive [2]. A scan tool for Ford's Integrated Diagnostic System costs $5,575 [3].

The group acquired two tools to help us explore packet communication and the state of the art. The first was the Microchip CAN Analyzer Tool, a low-cost monitor tool for debugging high speed CAN networks. The second was the Actron Elite AutoScanner, a mid-range diagnostic scan tool that contains car specific protocol information.

The Microchip CAN Analyzer Tool is intended for debugging systems, providing useful insight into error signals and direct access to raw CAN messages. Critically, however, it lacks capabilities for complex interactions with the CAN bus, as shown in Table 4.

The Actron Elite AutoScanner allowed us to explore how commercial sensor readers work. By listening to the network while using the AutoScanner's diagnostic functions, we were able to observe its interactions with the vehicle's ECUs. We were able to see that the diagnostic communication is a simple request-response communication. Mechanics use a similar scanner to check emissions, which are a requirement to pass inspection, and obtain the vehicle's VIN number. As with the rest of the CAN bus, these interactions are implicitly

trusted; we were able to spoof a system response and trick the scanner. Additional work on faking scan-tool interactions could easily allow owners to cheat inspections.

2.2 Academic Research

Prior work on the security of intra-vehicle networks has focused on three concerns: *why* an attacker would devote resources to accessing a CAN bus, *how* an attacker might gain access to the system, and *what* the attacker could do after achieving access.

This work has demonstrated sufficient incentives for adversaries to hack a CAN bus, including the ability to easily counterfeit electronic components, steal intellectual property from the manufacturer, and gain control over exploitable ECUs such as microphones, GPS devices, and safety systems [4].

Other research has examined the various attack surfaces on the CAN bus, including indirect physical access, short-range wireless access, and long-range wireless access. The authors were able to access any component on a moderately priced "late model" sedan from each attack surface [5].

Finally, researchers have explored CAN vulnerabilities after the network has been breached. One group was able to control two 2009 vehicles via a software program running a laptop connected to the OBD-II port [6]. The authors developed a software analysis package, reverse-engineered higher-level protocols, and successfully impacted operations in three settings: the laboratory, a stationary car, and a moving car. The authors were able to disrupt or disable a wide range of components, including safety-critical features.

However, in all of the academic work examined, the authors refused to release precise methodology, results, or computer code that would allow replication of the experiment, for fear of malicious use. Despite the vivid proofs of concept in these studies, car companies have not, to our knowledge, pivoted their focus from safety (inadvertent malfunction) to security (malfunction due to malicious intent). In fact, "car companies are still trying to gauge if these issues present a meaningful risk to ordinary drivers [7]." Siege is interested in understanding risks posed by security flaws; since they do not detail the steps necessary to hack a CAN bus, the end results of the academic studies are not directly useful to Siege.

3 Deliverables

Siege has provided the motivation and designated a toehold to the network (the OBD-II port) and have asked us to develop a methodology for hacking the CAN bus, and subsequently demonstrate the possibilities once control is achieved. Thus, the scope of this project is limited to the third topic — what an attacker could do with access to the CAN

bus.

Siege requested three key deliverables: a software package enabling interaction with an automobile CAN bus, an extensible methodology for reverse-engineering the network, and a proof-of-concept hack that demonstrates control over some component of the automobile. All deliverables were successfully completed before the conclusion of ENGS 90 to the satisfaction of our sponsor. These are briefly addressed in turn here, and revisited in greater depth in Section 5, *Specifications* and Section 6, *Work Accomplished*.

3.1 Software Package

The software package allows the user to interact with the CAN bus via the provided GoodThopter10 board. Siege specified that we use open-source software and requested the following functionality: the ability to read data and store it in an accessible way, the ability to inject packets onto the bus, the ability to analyze the data, and integration with the Wireshark network protocol analyzer. More detailed specifications were developed based on the experiments defined in our methodology.

3.2 Reverse-Engineering Methodology

For our project to have any value, it must be relevant beyond the test vehicle used. Thus, Siege requested we develop a detailed yet extensible methodology for reverse-engineering the higher-level protocols used on an automobile. We designed, implemented, and documented a series of experiments which took us from the initial black-box state, in which nothing was known about the network, through the development of sophisticated hacks demonstrating a high level of control over the network.

3.3 Proof of Concept Hack

To demonstrate the capabilities of the software package and the ability to decode the communication on our experimental car, Siege asked to conclude the project with a proof-of-concept hack. We developed a number of example hacks leveraging our understanding of and control over the network, including locking the doors, falsifying speedometer readings, and incrementing the odometer.

4 Project Strategy

Our work comprises the early stages of an R&D project for Siege Technologies, a highly atypical ENGS 89/90 project. As a result, we were given only high-level deliverables, without any specifications. We began ENGS 89 with limited knowledge of the problem space and no clear path to completion.

Our first approach was to directly obtain information on the higher-level protocols

by any means possible. We contacted manufacturers directly, spoke with auto mechanics and licensed dealerships, posted on auto enthusiast forums, and reached out to Dartmouth alumni in the industry, with little success. We learned that the protocols were manufacturer-specific, carefully guarded proprietary information. After being unable to directly obtain the protocols, we moved to reverse-engineer the network.

4.1 Reverse-Engineering the CAN Bus

We began by researching generic network attack patterns. Network attack typically consists of reconnaissance (obtaining information about the system without interaction), passive listening (observing traffic on the network without interaction), and active probing (injecting packets and observing network responses).

We used this framework to develop a CAN-specific approach, including a detailed experimental methodology for both passive listening and active probing (see Section 6.3, *Work Accomplished: Methods and Experiments)*. We then determined specifications for our software toolkit based on the functionality necessary to implement our experiments.

5 Developing Specifications

5.1 Initial and Final Specifications

At the start of the project, the CAN bus was essentially a black box. Because of the lack of public information available about CAN buses, we did not and could not know what components were on the bus or how they interacted. We knew we would need to read data from the bus and inject packets onto it, but did not how to do so in a useful way. Thus our initial specifications, shown in Table 1, focused mostly on basic read/write functionality:

Table 1: Original specifications for the project before experimentation and refinement began. Due to the black-box nature of the CAN bus and lack of available information, our initial specifications were high-level and very basic.

Deliverable	Objective	Spec	Test
Protocol Knowledge	Understand Arbitration IDs	> 30	Predict packet responses
	Map Higher level protocols	> 5	Write data packet, predict response
Software	Parse Data Packet	Y/N	Display parsed components of input data packet
	Inject data packet to car	Y/N	Component change due to data packet injection
	Integrate with Wireshark	Y/N	Load parsed data into Wireshark
Both	Predictability	$> 99\%$	Repeatedly predict result of injection
	Reliability	$> 99\%$	Same response to same packets

As we designed and implemented our experimental methodology, we continuously updated our specifications. This was a cyclical, iterative process. Adding software capabilities

enabled more sophisticated experiments, which in turn gave us a better understanding of the network; that improved understanding helped us develop even more advanced software. We began with basic read-and-write functionality and ended with sophisticated, automated hacks leveraging our knowledge of the test vehicles higher-level protocols.

By the end of the project, we had acquired a far greater knowledge of the CAN bus and the specifications necessary to hack it. The complete, final list of specifications for our project can be found in Appendix C. The most critical specifications are listed below in Table 2; they are discussed in greater detail in Section 6, *Work Accomplished*.

Table 2: A table with the key final specifications for our project. A complete version of this table, including justifications, quantifications, and testing strategy where appropriate, is included in Appendix C. The contrast between this table and our original specifications highlights the depth of knowledge regarding the capabilities required to hack a CAN bus gained during the project.

	Explanation	**Current Status**
Software Package		
Read Capability		
Basic Read	Intercept packets off the CAN bus reliably	Complete
Valid Packet Confirmation	Confirm packet was a valid message	Complete
Frequency Identification	Determine CAN bus frequency	Complete
Speed	Able to intercept all packets on bus	Read in 115 packets/second
ArbID Filtering	Use hardware to positively filter	Complete
Write Capability		
Basic Write	Inject packets to the CAN bus	Complete
Speed	Inject packets faster than ArbIDs send	Complete
Randomized Write	Inject a series of randomized data	Complete
Displaying Data		
Output to SQL database	Store packets in SQL	Complete
Output in .pcap format	Required Wireshark format	Complete
Commenting on data	Tag packets for ease of analysis	Complete
Intuitive Real-Time Display	Display traffic in a helpful manner	Complete
Injecting Packets Data		
Directly Input Packet	Directly enter desired ArbID and data bytes	Complete
Inject List of Packets	Inject a pre-determined packet sequence	Complete
Experimental Methodology		
Generalizable	Methods are generalizable to other vehicles	Complete
End-to-End Process	Methodology from black box through hack	Complete
Interpretable	Implementable by an inexperienced user	Complete
Proof of Concept		
Replicability	Able to reliably cause effect	Complete
Control	Able to systematically manipulate components	Complete
Affect Safety	Cause a direct safety hazard through hacking	Complete

5.2 Testing Strategies

Software Functionality

We built our software package from the ground up, testing and debugging each individual component as we went. We started with basic read capabilities, allowing us to sniff the network; we debugged by checking error flags on Siege's hardware and verifying the messages were properly formatted in the CAN protocol by hand. Once we had tested and functional reading capabilities, we were able to use them to test and debug the rest of our code. We used a splitter cable to hook two GoodThopter10 boards up to the CAN bus simultaneously, then used one board to inject and the other to listen for packets. This allowed us to observe the network as we perturbed it to verify we were having the intended effects.

User Interface

Throughout the course of our project, we did all of our experimentation from the user interface we developed. Thus we served as our own test subjects, continuously modifying and refining the interface according to our needs as we went. We derived all our specifications from our understanding of the network and the functionality required to hijack it.

Experimental Methodology

Halfway through ENGS 90, we acquired the wiring diagram for our test vehicle and learned there was a second, medium-speed CAN network also accessible via the OBD-II port. This entirely new network provided an excellent opportunity to test our methodology; using our testing strategy and software toolkit, we were able to map out the ArbIDs present on the network, decode its HLPs, and implement a hack in a matter of hours.

Finally, to test the extensibility of our methodology to more modern cars, we used our software to listen to traffic on a 2011 Subaru Outback. We did not implement our full methodology for fear of damaging the vehicle, but were able to sniff and characterize the background traffic. Unsurprisingly, there were more ArbIDs present on the network than for our 2004 test vehicle; however, we are confident that while the additional complexity might result in a lengthier reverse-engineering process, it also provides the opportunity for even more damaging attacks.

5.3 The "Speed" Specification

The reading and writing capabilities of our software package were severely limited by the firmware on Siege Technologies' GoodThopter10 board. The firmware is extremely slow

(see Section 8.1, *Future Work: Hardware*, for details) which prevented us from observing all the traffic on the bus. We considered re-writing the firmware, but Siege discouraged us from making this a project goal. We were able to circumvent this limitation by implementing hardware filtering (see Section 6.2, *Work Accomplished: Hardware*).

6 Work Accomplished

6.1 Overview

At the beginning of the project, the group was provided with four GoodThopter10 boards and the firmware for reading individual packets off of the CAN bus. Over the course of the project, the team developed an experimental methodology, designed and implemented a software package enabling the user to interact with the CAN bus, and proved the efficacy of the two by hacking the CAN bus on the test vehicle. We gained a high level of control over safety critical components and developed advanced hacks leveraging our knowledge of the test vehicle's higher-level protocols.

For example, our `speedometerHack` method listens for a message from ArbID 513, which encodes information about the engine status. It intercepts the first packet sent by 513 and decodes the fifth data byte, which indicates the vehicle's speed. The method then alters that byte based on user input, builds a new packet from the captured one, and repeatedly injects it. As a result, the user can input a single integer, and the speedometer will read that amount over the true speed. Running `speedometerHack(15)` causes the speedometer to display a value 15 MPH over the true speed of the car. The effective Stuxnet hack of Iran's nuclear reactors used this same "denial of view" to convey normal operation during the hack [8]; because the car only has one external viewpoint, hacking it constitutes a complete denial of view.

Our sponsors deemed this, as well as the other hacks outlined below, as a successful culmination of the project. We pushed further to identify more components on the CAN bus as well as begin background characterization on our own, more modern vehicles.

6.2 Software Package

Our software package allows the user to read data off the CAN bus, work with the data to reverse-engineer the network, and then inject data back onto the bus to gain control over it. As seen in Figure 2 below, our package provides an interface which abstracts away the GoodThopter10; the user need not know anything about the board itself to work with the CAN bus.

We identified, implemented, and tested key functionalities for the package, which are

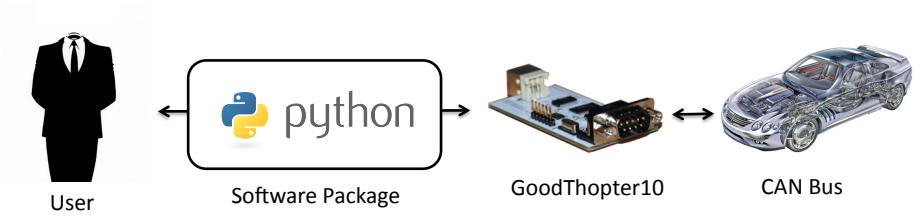


Figure 2: Our software package is the interface between the user and raw CAN data, allowing them to read, analyze, and re-inject packets to and from the bus, without needing to know anything about Siege's GoodThopter10 board.

outlined below:

1. Basic configuration and error checking for Siege's GoodThopter10 board
2. Basic reading capabilities to pull data off the CAN bus
3. Basic packet injection to write data to the CAN bus
4. More sophisticated read-in and injection to enable experimentation
5. Car-specific hacks and diagnostics
6. Data management to store and access the read-in packets
7. A flexible user interface to integrate and manage all existing functionality

Our implementation of this functionality is illustrated in Figure 3.

While these components are tightly integrated, they are also loosely coupled, meaning that any individual component can be run independently of the others. This structure gives entry-level users immediate access to the CAN bus through our interface, without requiring knowledge of the GoodThopter10 board or our code base, while allowing more experienced users to add additional functionality to the project, including vehicle-specific hacks.

This paper contains a high-level discussion of the functionality implemented; a detailed description of software included in our project can be found in the documentation delivered to Siege Technologies and appended to the paper.

Hardware

The GoodThopter10 board provided by Siege Technologies uses a MCP2515 chip to interface with the CAN bus. At the start of ENGS 89, Siege provided us with the board

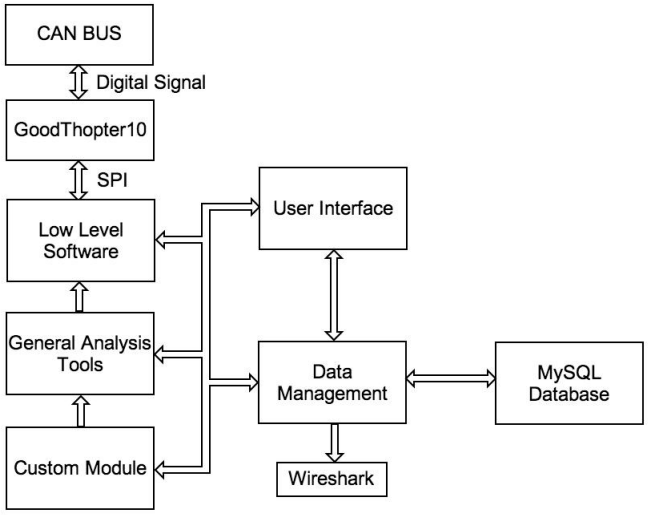


Figure 3: High level block diagram of the complete software package. Each block represents a separate class of code and their interactions. Custom Module is a sub class of the General Analysis tools which is again a subclass of the Low Level Software class. The data management class controls data formatting from the bus, from files, from the database and to Wireshark's file format. The user interface allows for easy coordination and data flow between these capabilities.

itself and basic code for SPI data transmission, allowing us to read and write data to the MCP2515s registers, but not directly onto the bus.

We wrote code to read in and inject packets, configure the MCP2515, and check the chip's hardware flags to provide additional information about the packets being read in. Finally, once we had implemented and tested read/inject capabilities, we wrote more advanced functions combining both to implement more complicated experiments. Detailed specifications for the hardware code can be found in Appendix C. All of the basic read/inject methods are contained in the `GoodFETMCPCANCommunication` and `GoodFETMCPCAN` files.

Reading in Packets

Siege provided us with a basic `sniff` function, which allowed us to read data off the MCP2515 but did not give any indication whether it was valid. Our first step was thus to add error checking, which allowed us to determine if the received packet was properly formatted. We were then able to experiment with different hardware configurations and

configure the MCP for our test vehicle. After doing so, we then wrote a series of functions allowing future users to test various frequencies and quickly configure the chip for a new car, including `frequencyTest` and a debug option for `sniff` which adds error-checking.

Once we had tested and verified our `sniff` function and configured the chip properly, we added additional status checks to see if packets were being missed; we discovered we were missing an unknown - but likely quite large - number of packets. Speed thus became a key specification; we needed to see most, if not all, of the traffic on the bus in order to reverse-engineer it.

The MCP2515 automatically reads in packets for retrieval by the computer, but has extremely limited on-chip storage. As a result, if packets are received faster than the host microcontroller (in this case, an MSP430 on the GoodThopter10 board) retrieves them, they cannot be read in, and are lost.

The code initially provided by Siege read in 9.5 packets/second. We pared down our code and switched to a Linux machine with faster USB data transmission, improving our read-in speed to 115 packets/second. At this point, we implemented hardware filtering (the `addFilter` and `filterForPacket` methods) which allowed us to block unwanted messages from being read in. This significantly reduced the number of packets received by the chip; when we filter for individual ArbIDs, we miss virtually no packets. We were thus able to analyze individual ArbIDs higher-level protocols. Using hardware filtering, we estimated a network speed of approximately 450 packets/second on the bus.

Injecting Packets

Once we had completed and tested our reading capability, we began to work on our injection capabilities. We first tested Siege′s basic write code against the MCP2515′s data sheet and verified that it worked as expected.

We wrote an injection function, `spit`, to automatically build a properly formatted CAN packet out of user data, write it to the MCP2515, and inject it to the network. We were able to use our reading capability to test this injection. We connected two GoodThopter10 boards to the test vehicle, used one to listen to the network, and the other to inject packets, and verified that we could successfully inject properly formatted CAN messages.

We then wrote a more flexible method, `spitMultiplePackets`, to load multiple packets and quickly send pre-loaded packets without needing to input the entire message each time.

Experiments

We created an `Experiments` class which builds on top of the basic hardware interaction code. This class implements vehicle-agnostic experiments such as packet fuzzing, packet send-response, ID sweeps, and remote transmission request sweeps. It is implemented as a sub-class, retaining all the functionality of our `GoodFETMCPCANCommunication` and `GoodFETMCPCAN` files.

Vehicle-Specific Functionality

Once we had a good understanding of the ArbIDs on our test vehicle and their higher-level protocols, we were able to write car-specific hacks, such as the `speedometerHack` mentioned earlier, in the `FordExperiments` module. These hacks leveraged our understanding of the CAN bus and give the user complete control over the network without needing to know any implementation details whatsoever.

We structured our software package to allow users to create their own custom modules as sub-classes of our `Experiments` class. This design is highly extensible, allowing future users to write their own car-specific sub-classes from our generic code base.

Data Management

Recording CAN traffic rapidly generates vast amounts of data. We expect the average user to sniff several hundred thousand packets over the course of a project, making proper data management and storage critical. This is particularly true given that the most effective means of learning is through pattern recognition and visualization. After considering various alternatives and conducting an trade study during ENGS 89 (see Figure D.1), we integrated our software with a MySQL database. A MySQL database provides an efficient and easy to use storage medium for large data sets. Table 3, below, shows the the data stored for each packet in our database. We deemed these to be the important components needed for proper analysis and reverse-engineering of the protocols.

Although our code base provides basic functionality for interacting with the MySQL database, we made extensive use of the free software Sequel Pro, which provides database management and visualization.

Additionally, as per our sponsor's request, we implemented software to format our data for the open-source Wireshark Protocol Analyzer. This section of the project required rewriting Wireshark's included `socketCAN` dissector, which incorrectly masked packet components and thus dissected them incorrectly. A checkbox in the user interface allows the user to export a selection of stored packets to Wireshark (for more extensive analysis) by

Table 3: Data Fields for MySQL database, showing all of the data stored for each CAN packet recorded. These are searchable, giving the user easy access to all previously observed packets.

Field	Type	Length	Unsigned	Allow Null	Purpose
id	int	11	yes	no	Unique identifier
time	double	20,5	yes	no	Timestamp for the packet
stdID	int	5	yes	no	Standard ID
exID	int	5	yes	yes	Extended ID (optional)
length	int	1	yes	no	Data length
error	bit	1	-	no	Error boolean for packet
remoteFrame	bit	1	-	no	Remote Transmission Request
db0	int	3	yes	yes	Data Byte 0
db1	int	3	yes	yes	Data Byte 1
db2	int	3	yes	yes	Data Byte 2
db3	int	3	yes	yes	Data Byte 3
db4	int	3	yes	yes	Data Byte 4
db5	int	3	yes	yes	Data Byte 5
db6	int	3	yes	yes	Data Byte 6
db7	int	3	yes	yes	Data Byte 7
msg	varchar	30	-	no	Raw Message
comment	varchar	500	-	yes	Comment Tag
filter	bit	1	-	no	1 if Filtering
readTime	int	11	yes	no	Length of sniff when read

converting the SQL packets into the .pcap format required by Wireshark.

User Interface

The graphical user interface (GUI) coordinates the various components of our project and abstracts away all lower level software, allowing the user to implement our methodology and reverse-engineer the CAN bus without knowing the details of the GoodThopter10 board. We had initially set the GUI as a low priority, but during experimentation, it became clear that a flexible and intuitive user interface would greatly facilitate analysis.

Most importantly, we needed to be able to observe packet traffic changes in real time and act accordingly. We developed an interface allowing us to observe fluctuations in the network traffic, develop a hypothesis about potential correlations, and quickly inject packets to test that hypothesis. We produced a complete User Guide for the GUI, attached to this paper; what follows is a summary of the design philosophy rather than an exhaustive description of functionality.

We designed the GUI on the fundamental principles of simplicity, usability, extensibility, and abstraction. Operating as a wrapper, the GUI coordinates data flow between components. Therefore, a user can access all of our methods and code outside the GUI if they so choose. We determined the key GUI functionalities to be (see the attached User Guide for a complete list):

1. Visualize incoming traffic
2. Inject data onto bus
3. Run all experimental methods
4. Documentation and display tools
5. Access to MySQL database
6. Automated file management to csv, pcap and MySQL formats

The final GUI provides all of these functionalities in a simple, intuitive window. The GUI was implemented using the Tkinter package in Python, a binding to the Tcl/Tk GUI toolkit[9].

Figure 4, below shows a screen shot of the main window for the GUI. The different software modules are accessible via the tabs at the top of the window. The left side frame displays sniffed packets in real time, while the right side allows the user to interact with the CAN bus by reading or injecting data; it calls the methods described in Section 6.2 such as: `spitMultiplePackets`, `addFilter`, and `sniff`.

The experiments tab, shown in Figure 5 allows the user to call methods from the `Experiments` class described above. Most notably, this tab also allows the user to inject randomly formatted, fuzzed data packets as well as re-inject previously fuzzed packets. This is a critical component because fuzzing is a powerful yet unpredictable tool; being able to re-inject sequences of random packets allows the user to determine which packets caused a specific effect. This allows for rapid isolation of relevant, impactful packets from other randomly generated ones.

The *ID information* tab allows the user to record information or theories as they experiment on the CAN bus. We provide three sub-tabs to record information about packets: General Info, Byte Info, and Packets, shown below in Figure 6. These features the user to record notes about observed correlations for the ArbID as a whole, specific data bytes of the ID, and important packets, respectively. All information is saved as a JSON file. JSON provides a simple, efficient data structure that can easily be parsed and loaded. JSON provides several advantages over other mark up languages such as XML and HTML [10], as shown in our trade study included in Figure E.1.

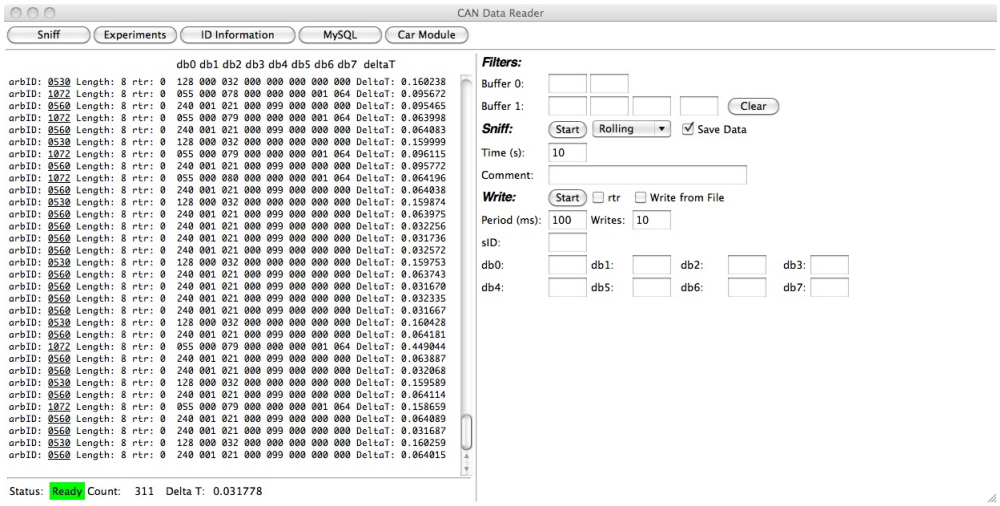


Figure 4: Screenshot of the the CAN Reader GUI with data displayed on the left Frame and user options on the right. The right side consists of five separate tabs which can be changed via the menu bar included at the top.

The MySQL tab, shown below in Figure 7 allows the user to upload sniffed packets to their MySQL database. Since file management is completely automated, the user only has to make sure that they have internet access to reach the server.

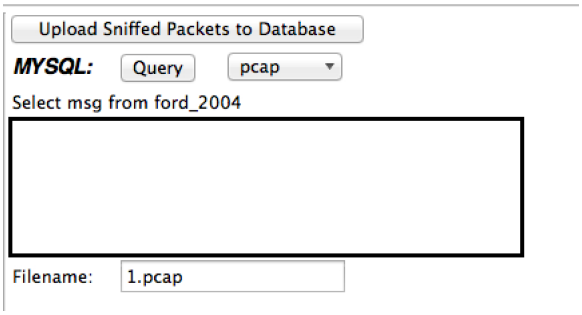


Figure 7: Screenshot of the Database and MySQL tab on the CAN Reader

Finally, the GUI was designed to be highly extensible. The user can easily specify database information, packet information, and module extensions through a settings dialog box, with the settings saved in a INI file. Module extensions allow for user customized code that can be linked to the GUI. Figure 8 shows an example of our car specific extension. Instructions on building GUI extensions are included in the User Guide.

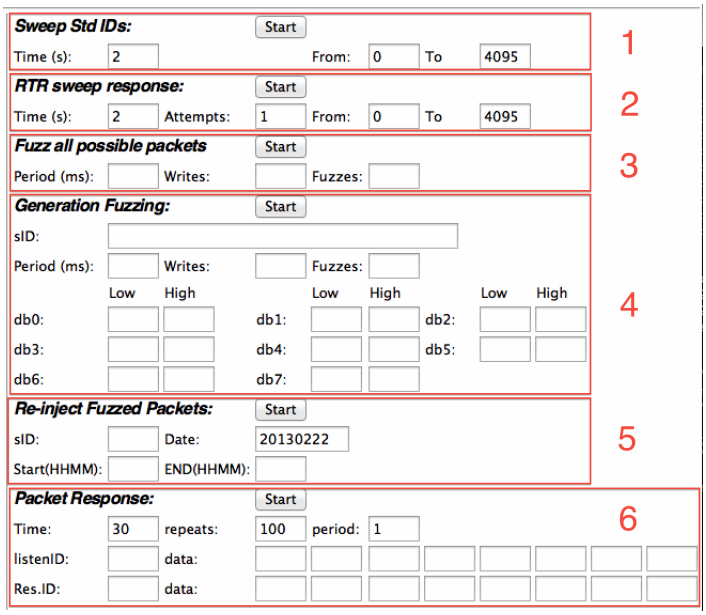


Figure 5: Screenshot of the the CAN Reader GUI experiments tab. This tab allows the user to run our more advanced experiments and methodologies. The different experiments available are numbered above in red. 1 and 2 allow for background characterization, 3 and 4 can be used to inject random packets, 5 allows the user to retest previous fuzzed packets. Experiment 6 allows the user to respond to a given message.

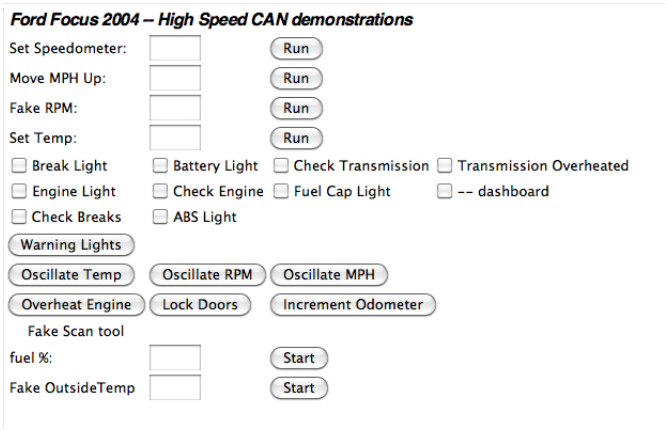


Figure 8: Screenshot of the the CAN Reader GUI custom car module. This tab can be created by a user without modifying any of our existing code base.

Overall, the GUI provides an intuitive way for users to access all of the functionality we developed, satisfying the needs of both entry-level and technically savvy users.

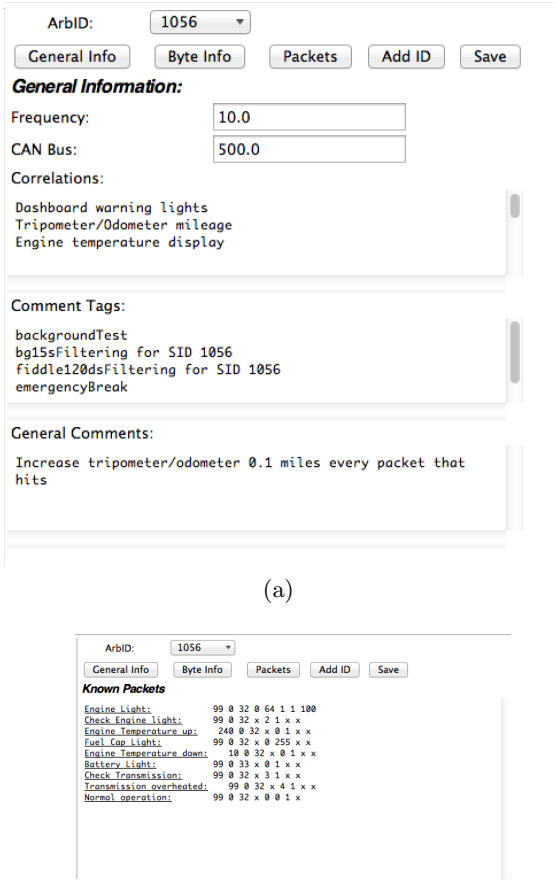


Figure 6: Screenshots of the three sub-tabs for the ID Information tab. These tabs allow the user to document their knowledge and document as they run experiments with our program.

Our package is a more flexible and powerful tool for reverse-engineering a CAN bus than commercially available products. The only comparable commercial product is the Microchip CAN Analyzer; our tool provides a more intuitive user interface as well as additional, more advanced packet reception and injection capabilities. A complete comparison is below, in Table 4.

Table 4: A comparison of our GUI to the commercially available Microchip CAN Analyzer

	Microchip	Our GUI
View types	Hex or Decimal	Decimal
Read Speed	1000 packet/sec	115 packets/sec
Read Packet Types	Standard and Extended	Standard and Extended
Read Packet Lengths	Varying	Varying
Sniff Views	Fixed, Rolling	Fixed, Rolling
Write speed	10 packets/sec	300 packets/sec
Write from screen	✓	✓
Write from file	✗	✓
Write packet types	Standard and Extended	Standard
Write packet lengths	Varying	8 bytes
Error checking	✓	✗
Data logging	✓	✓
MySQL Integration	✗	✓
Save to Wireshark	✗	✓
Intuitive User interface	✓	✓
Notation Section	✗	✓
Cost	$100.00	$27.50

6.3 Methods and Experiments

In order to hack the car, we needed extensive knowledge of the communications and data encoding employed by each Arbitration ID. This information, however, is vehicle-specific, and lacks value when exploiting the security vulnerabilities on other vehicles. Instead, Siege emphasized the development of an extensible reverse-engineering *methodology* as a key project deliverable. This methodology can then be employed to hack *any* vehicle utilizing the CAN protocol. This methodology was developed in tandem with our software package and refined over the duration of the course.

In Section 6.3, *Methodology Development*, the methodology itself is discussed and the experiments are summarized. An in-depth description of each experiment, including control variables, inputs, outputs, analysis and methodology, is included in the Methodology guide appended to this paper. Section 6.3, *A Case Study* examines an application of the methodology to a black-box scenario, verifying its extensibility and usefulness. The

experimental setup on the test vehicle is shown in Appendix **??**.

Methodology Development

At the highest level, the methodology follows a familiar process for system characterization. Any system can be characterized by its response to input signals, and a thorough understanding of our system (i.e. the electrical network present on our experimental car) would allow the a) prediction of the packets output in response to a physical input and b) prediction of the physical response ("output") in response to input packets.

Our methodology first requires the characterization of the unperturbed, background state, before developing the transfer function to predict output responses based on various inputs, as shown in Figure 9. This method allows the user to quickly move from the 'black-box' state of an unknown car to the 'white-box' state where the user has the ability to implement an impactful hack.

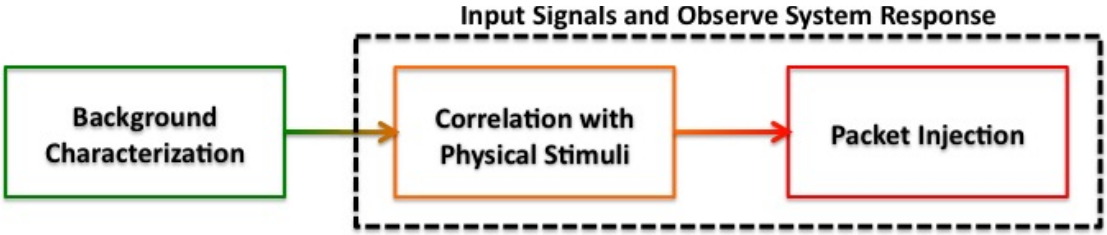


Figure 9: The highest level of the methodology flowchart. The project will first seek to characterize the background state of the car, before moving onto mapping input signals to output responses. The car has two potential inputs: physical stimuli and packet injections.

Background Characterization

The background state of the car can be defined using four experiments, all of which can be directly implemented by our software via the GUI. The *Arbitration ID Sweep* allows the user to listen on each possible Arbitration ID, categorizing every node that transmits on a regular basis. Next, the *Remote Transmission Request (RTR)* sweep requests a response from each possible identifier, adding those IDs that do not transmit during normal operation (but are present on the network) to the catalogue. Finally, the user will passively sniff the network, without filtering to capture the relative packet transmission (and any correlations) and with on-chip filtering to characterize the behavior of each ArbID. This process is shown in Figure 10.

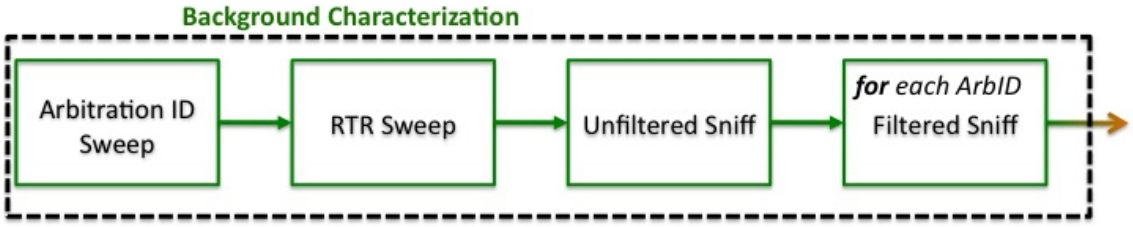


Figure 10: The four experiments required to characterize the background state of the car. The ArbID and RTR Sweeps ensure that the user captures all ECUs on the network, while the sniffs characterize the background packet transmissions and frequencies.

Correlation with Physical Stimuli

After characterizing the background, the user moves to begin defining the transfer function of the system: the correlation of inputs to outputs. In this section, physical stimuli are the inputs, while the observed packet response are the measured outputs. This portion requires the user to develop a list of perturbations to introduce - a number of physical stimuli with ECUs suspected to be on the CAN bus.We purchased a commercially available car-specific wiring diagram to develop a comprehensive list for testing, but possession of such a document is helpful and not necessary. After identifying the perturbations that produce a change for each ID, each associated perturbation is explored in detail during the *Characterize Change with Stimuli* step, down to associating individual packets with their exact corresponding physical state. These steps are shown in Figure 11.

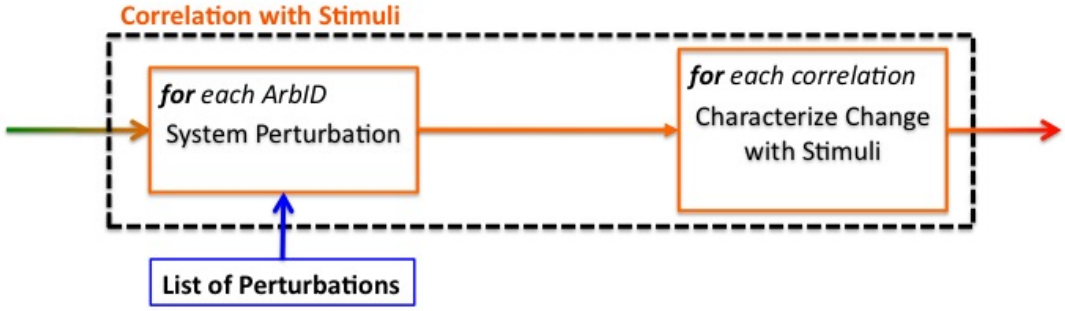


Figure 11: The two experiments used to predict packets from a physical stimulus. All perturbations will be run for each Arbitration ID on the network, and then each ArbID-Perturbation combination must be characterized.

Packet Injection

Injecting packets into the system flips the input-output relationship. Injecting a packet (or series of packets) can lead to observed physical responses associated with components listening to the injected packet ID. In the confirm inferences step, packet injection is used to verify any ArbID-packet correlations speculated in the prior step and to assess whether these correlations are responsive to injection. The *Boundary Analysis* and *Generative Fuzzing* experiments test the responsiveness of each ArbID to injected values at the edge of their range and to randomly generated values, respectively. This allows the user to experiment with packet values unachievable through physical stimuli. The inferences generated in these two steps are then confirmed in the final step of packet injection.

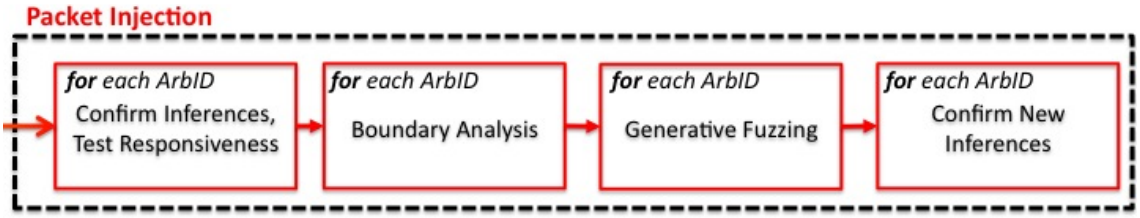


Figure 12: The four experiments used in the packet injection portion of the experimental methodology. The first and last step confirm inferences via packet injection, while the

Case Study

Because the methodology was developed *while* the group was hacking the CAN bus on the experimental car, this bus could not offer a complete test case for the methodology. Instead, we sought another black-box network to test the full methodology from conception through a successful hack. While looking at the wiring diagram for our 2004 Ford Taurus, we noticed a vehicle-specific, 125 kHz CAN bus - the perfect test for our methodology.

During the background characterization stage, we identified 12 Arbitration IDs transmitting on the network, with varying data packet lengths. We noted that the majority of these IDs were not dynamic during steady state transmission and that the entire bus transmits while the car is off, a divergence from the previously analyzed CAN busses.

We drew from our perturbation list to map ArbID and physical stimulus correlations, quickly mapping IDs to various physical actions such as door position, light position, the emergency brake, the gear of the car, and the brake pedal.

Some of these inferences were confirmed during the first stage of packet injection, and other ID-stimulus mappings were noted as non-listening, meaning that the stimulus pro-

duced the network noise, but not the other way around. During the fuzzing and boundary analysis stages, we noticed fuzzing packets from ID 1032 caused the doors to lock. We were able to deduce the correct packet combination to make this a repeatable injection.

We completed the methodology in four hours of work, yielding a number of hacks. For example, we can consistently and repeatedly trigger a constant 'door open' alarm, lock the doors, dim and brighten the dashboard lighting, and send a 'low brake fluid' error message.

6.4 Proof of Concept Demonstration

The successful completion of the software package and reverse-engineering methodology allowed the group to develop a number of hacks to demonstrate to our sponsor. The hacks demonstrated on the high-speed CAN bus included locking the doors, increasing the odometer mileage, triggering a variety of warning lights on the dashboard, altering the speedometer reading, altering the tachometer reading, and altering the engine temperature reading. Medium-speed bus injections were able to convince the car that any combination of the doors were open, dim and brighten the dashboard lights, trigger a brake system warning message, and tell the dashboard the incorrect state of the emergency brake.

We believe our ability to hack the car was limited only by the extent of implementation of CAN bus in our experimental vehicle. Experiments with fuzzing and grounding the CAN bus produced no discernible effect on the engine, despite it's apparent presence on the bus; thus we suspect that the engine in our test vehicle transmits but does not listen to the bus. More tightly integrated modern vehicles, with larger sensor networks and with more data passed between components, present even more enticing targets to potential attackers. Unfortunately, we were limited by our budget; with additional funding, one could purchase a more modern test vehicle to confirm these hypotheses.

For the proof of concept video used in the demonstration, we decided, after discussing the options with our sponsor, to demonstrate precise control over the speedometer, tachometer, and engine temperature gauge on the dashboard. This hack demonstrated profundity of knowledge as well as control over a number of safety-critical components on the dashboard. Our sponsor contact, Daniel Bilar, deemed this a successful proof of concept of the vulnerabilities present on automobiles' CAN buses.

7 Economic Analysis

7.1 Project Costs

The group had two primary budgets for this project: the budget for purchasing an automobile, set loosely between $2000.00 and $3000.00, and the discretionary budget,

Table of Project Expenditures			
Item	**Purpose**	**Quantity**	**Total Cost**
ScanTool Scanner	State of the Art	1	$ 212.49
Microchip CAN Bus Analyzer Tool	State of the Art	1	$ 99.99
GoodThopter10	Hardware	3	$ 52.65
OBD-II Connector Cables	Hardware	1	$ 55.57
OBD-II Male Connectors	Hardware	10	$ 2.01
OBD-II Male Housing	Hardware	1	$ 5.38
2004 Taurus Wiring Diagram	Analysis	1	$ 77.58
Gas	Analysis	26 gal	$ 95.94
Gas Canister	Analysis	1	$ 4.99
Connection to Med-Speed CAN	Analysis	1	$ 3.00
Linux Battery	Analysis	1	$ 23.20
		Discretionary Total	**$632.80**
2003 Ford Focus	Analysis	1	$ 2000.00
Focus Resale	Analysis	1	$ 1000.00
2004 Ford Taurus	Analysis	1	$ 2000.00
		Car Total	**$3000.00**

Table 5: Project Expenditures, with the exception of labor costs. A more detailed breakdown of expenditures for the GoodThopter10 board is included in Appendix A.1.

capped at $1000.00. Despite the initial setback of purchasing a car without the CAN protocol during ENGS 89, we were able to recover sufficient value from the first car to purchase a new, CAN-enabled car while staying within the $3000.00 limit on our budget as shown in Table 5. The group remained within the discretionary budget for the project.

To assess the labor costs for the project, we first defined the hourly cost of $125/hour, the median charged by professional engineering consultants [12]. Combined, the group worked for an average of approximately 50 hours weekly. Because we are students without an engineering accreditation, we assumed our productivity was between 10% and 60% of a professional engineer, and calculated the labor costs based on this assumed productivity. Depending on the productivity assumption, the total labor costs for the project vary between $12,500 and $75,000, as shown in Appendix Table G.1.

7.2 Potential Market

Siege sponsored our work as early-stage research, without a developed plan for the results. However, as we progressed through the project, we identified ways in which Siege could monetize a refined version of our product. Siege is principally a cyber technology

research and development firm, yet attacks on car networks are not yet a recognized threat; as such, we outlined a number of potential venues for monetizing our work, ranging from immediately viable to potentially marketable should CAN attacks become a realized threat.

The unit cost for the GoodThopter10 board, together with the open-source software we wrote and the necessary connectors, is quite low, at $27.50. Future costs will primarily lie in the labor associated with continued software development and analysis.

CAN Bus Development

We see immediate commercial potential in an end-to-end, listening and injection system for anyone interested in reverse-engineering or building their own CAN bus. This is generalizable across industries, including automotive, industrial, and medical applications. Siege did not initially envision this as a path to market, but is now actively exploring it. This would require minimal development beyond our code base - in fact, the functionality developed in ENGS 90 is competitive with the current state of the art, a CAN bus analyzer sold by Microchip. This market remains relatively small, primarily consisting of frugal hackers and hobbyists.

The Microchip CAN Bus Analyzer Tool, at $99.99, is designed for building a network rather than reverse-engineering one. Our GoodThopter10 code surpasses this analyzer in several key areas, including an improved user interface and experimental functionality aimed at reverse-engineering a network. The primary weak point of the GoodThopter10 is speed (the Microchip supports speeds up to 1 Mbit/s [13]), which could be improved by optimizing the GoodThopter10 firmware as outlined in Section 8, *Future Work*.

Cars Insurers

Further investment by Siege could allow them to develop a software product for car insurers, a $166 billion market in the United States alone [14]. Siege could add value in two primary ways: first, a software package could be implemented via the OBD-II port to log data and attacks against the car, known as Telematic Insurance. These insurance packages, pioneered by Progressive and now implemented nationwide, are designed to track stolen vehicles and reward consumers for good driving behavior [15]. Siege could work with insurers to expand these packages into the preventative sphere, as well as to track and report any intrusions or atypical bus behavior (i.e. using the bus to alter the odometer). Secondly, Siege could enable companies to further differentiate rates between cars by providing greater information to insurance companies about the risk of a hack for each type of car.

Car Owners

With further development, Siege could produce an "antivirus for cars" package, designed to protect the consumer from safety-compromising hacks and privacy intrusions. The computer security market is a useful analogy to this market - largely unexplored until hackers exploited weaknesses to cause widespread damage, and today an $8.2 billion market with 8% annual growth [16]. The potential damage from large-scale, transportation infrastructure hacks is becoming more evident to the average car user in the United States through the publicity generated by the recent sophisticated hacks from China [17] and the proclamations by Defense Secretary Panetta [18]. This heightened awareness, along with the increasing development of autonomous cars, indicates a trend towards a potential market for a direct-to-consumer package.

Car Manufacturers

At the highest threshold of additional development from Siege, they could contract with car manufacturers directly to license a security approach or security technology for the CAN bus. This market opportunity is enormous, with 13 million new vehicles sold in the U.S. alone in 2011 [19] and 68% of eligible Americans driving daily [20]. Siege envisions a software and hardware add-on to assure the integrity of the signals transmitted on the CAN bus. This could be targeted at high-end, highly computerized vehicles, which are both more vulnerable and enticing targets for such attacks. This add-on would be analogous to the difference between Microsoft Windows and Microsoft Windows CE: Windows retails for hundreds of dollars, but is not guaranteed or recommended for use in real-time systems, while CE is guaranteed for these systems but is priced in the thousands of dollars [21].

8 Future Work

8.1 Hardware

Future work on the hardware components of the project should focus on improving the GoodThopter10 firmware.

The only major specifications that were not met or exceeded were the speed of our read and write capabilities. The code provided by Siege for the GoodThopter10 allowed us to read in 9.5 packets/second, which was far below the 450 packets/second on the bus.

We were able to increase the speed to 115 packets/second by optimizing our code, allowing us to read in approximately 25% of the unfiltered traffic on the network. Implementing hardware filtering, allowed us to read in all of the packets sent from any individual ArbID. Further improvements would require re-writing the GoodThopter10 firmware.

Currently, to read in a packet, the computer must ask the board if a packet is available. If the board says yes, the computer then asks for the packet, which is then sent to the computer. This requires multiple round-trips from the computer to the MSP430 and to the MCP2515. These trips can be reduced by using the MCP2515 and MSP430's hardware interrupt capabilities. One could automatically load received packets into the MSP430, which has far more storage than the MCP2515. The computer could then occasionally query the MSP430 for all available packets. Travis Goodspeed, the board's architect, chose the to poll the MCP chip to allow portability to other hardware (such as BlueTooth).

If Siege Technologies intends to pursue further experimentation on CAN buses via the OBD-II port, we would strongly recommend re-writing the GoodThopter10 firmware to allow interrupt-driven message reading from the MCP2515 to the MSP430 and subsequent buffer-flushing from the MSP430 to the computer.

8.2 Methodology

In keeping with the open-source ethos of this project, the methodology designed, implemented, and documented by the team should be considered a living document. Any future users should reference and update the methodology with experiments they find useful in their work.

A newer experimental vehicle would provide the opportunity to observe and experiment with more complicated interactions between ECUs. This could require augmenting our methodology with methods for learning more complex protocols. Future projects utilizing more modern vehicles could also explore the potential for gaining access to the bus by means other than the OBD-II port. Many modern cars have Bluetooth and wireless capabilities linked to the CAN bus, like GM's OnStar or Ford's MyKey.

8.3 Software

As we found through ENGS 90, more advanced experimental methodology requires developing the toolkit to implement that methodology. Any updates to our final methodology will require corresponding, supporting updates to the software and user interface.

As covered in Section 7, *Economic Analysis*, a moderate amount of work could make our software package a fully functional, commercially viable CAN analysis tool. Effort would need to focus on portability, reliability testing, and error checking.

Most notably, some of the underlying code was written for only Standard ID packets with a data packet length of eight, since all packets on our test vehicle contained eight data bytes. A commercial product would need to be portable to systems with Extended

IDs and variable length data packets; however, this would not require too much effort.

8.4 General

We have provided our sponsor with a detailed list of known issues and suggestions for future work. Since this project has always been intended to support future work, we made a major effort to document all of our work throughout. We produced extensive documentation for our software package, outlining all the classes and methods written for our experiments and user interface. We also wrote a detailed user guide describing how to use our GUI and the project at a high level. These documents should provide sufficient information for a future group to easily pick up where we left off.

Finally, work will continue next term as we prepare our code base and methodology for an application and hopeful presentation at REcon 2013, a conference with a focus on advanced exploitation techniques.

Bibliography

[1] "CAN Datasheet." *Bosch Semiconductors*. N.p.. Web. 3 Mar 2013. http://www.bosch-semiconductors.de.

[2] Motor Craft Service. Web. 3 Mar 2013. http://motorcraftservice.com

[3] OEM Tools. OEM Level Tools. Web. 3 Mar 2013. http://www.oemtools.com/homeproducts/ford.html.

[4] Wolf, Marko, Andr Weimerskirch, and Thomas Wollinger. "State of the art: Embedding security in vehicles." *EURASIP Journal on Embedded Systems 2007* (2007).

[5] Checkoway, Stephen, et al. "Comprehensive experimental analyses of automotive attack surfaces." *Proceedings of USENIX Security*. 2011.

[6] Koscher, Karl, et al. "Experimental security analysis of a modern automobile." *Security and Privacy (SP)*, 2010 IEEE Symposium on. IEEE, 2010.

[7] Wright, Alex. "Hacking cars." *Communications of the ACM 54.11* (2011): 18-19.

[8] Gregory, Derek. "The everywhere war." The Geographical Journal 177.3 (2011): 238-250.

[9] Grayson, John E. *Python and Tkinter Programing*. Greenwich, CT: Manning, 2000. Print.

[10] "JSON: The Fat-Free Alternative to XML." *JSON.org*. N.p.. Web. 3 Mar 2013. http://www.json.org/xml.html

[11] Montgomery, Douglas C. *Design and analysis of experiments*. Wiley, 2008.

[12] Stelluto, Georgia, ed. "IEEE-USA Consultants Fee Survey Report." IEE-USA. IEE, n.d. Web. 3 Mar 2013. http://www.exality.com/files/ConsultantFeeSurvey2011.pdf.

[13] "CAN Bus Analyzer." *Microchip*. MicroChip. Web. 3 Mar 2013. http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en546534.

[14] "Non-Standard Auto Insurance Market Overview & M&A Trends." *StoneRidge Advisors*. StoneRidge Advisors, n.d. Web. 3 Mar 2013. http://www.stoneridgeadvisors.com/Content/View_From_The_Ridge_August_2012.pdf.

[15] *Compare the Box Insurance*. Gocompare.com. Web. 3 Mar 2013. http://www.comparethebox.com/telematics-insurance.

[16] Messmer, Ellen. "Good times projected for network security market in 2011." *NetworkWorld*. N.p., 4 Jan 2011. Web. 3 Mar 2013. http://www.networkworld.com/news/2011/010411-network-security.html.

[17] Perlroth, Nicole. "Some Victims of Online Hacking Edge Into the Light." *The New York Times. New York Times*, 20 Feb 2013. Web. 3 Mar 2013. http://www.nytimes.com/2013/02/21/technology/hacking-victims-edge-into-light.html.

[18] Baldor, Lolita C.. "Panetta: Cybersecurity focus of next NATO meeting." *San Francisco Chronicle*. Hearst Newspapers, 02 Feb 2013. Web. 3 Mar 2013. http://www.sfgate.com/news/politics/article/Panetta-Cybersecurity-focus-of-next-NATO-meeting-4300617.php.

[19] Nick Bunkley, The New York Times. *U.S. Car Sales Keep Up Their Firm Growth.* April 3, 2012. http://www.nytimes.com/2012/04/04/business/car-sales-keep-up-their-firm-growth.html

[20] United States of America. Bureau of Labor Statistics. *American Time Use Survey.* 2010. Web. http://www.bls.gov/tus/

[21] "Real-Time Systems with Microsoft Windows CE 2.1." Microsoft. Microsoft, n.d. Web. 3 Mar 2013. ¡http://msdn.microsoft.com/en-us/library/ms834197

[22] Richards, Pat. "A CAN Physical Layer Discussion." *Microchip*. Microchip, 16 Sep 2005. Web. 3 Mar 2013. http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en012057.

[23] Keskin, Ugur. "In-vehicle communication networks: a literature survey." *Computer Science Report 09-10* (2009).

[24] Barry, Keith. "Can Your Car Be Hacked?." *Car and Driver*. Car and Driver, n.d. Web. 3 Mar 2013. ¡http://www.caranddriver.com/features/can-your-car-be-hacked-feature.

[25] "GoodFET." Sourceforge. N.p.. Web. 3 Mar 2013. http://goodfet.sourceforge.net/

[26] Bratus, Sergey. "What hackers learn that the rest of us don't: Notes on hacker curriculum." *Security & Privacy, IEEE* 5.4 (2007): 72-75.

Appendix A

GoodThopter10 Costs

GoodThopter10 Costs		
Item	**Quantity**	**Total Cost**
Software	1	$ 0.00
FT232RL	1	$ 3.95
MSP430F2274	1	$ 3.50
MCP2515	1	$ 2.08
MCP2551	1	$ 1.18
20MHz SMD Crystal, 15pF	1	$ 0.99
0.1μ F Decoupling Capacitors	4	$ 1.00
15pF Capacitors	2	$ 0.30
0603 LED	5	$ 0.30
330R LED Series Resistors	4	$ 1.00
10k Pull-up Resistor	4	$ 1.00
USB Type B	1	$ 1.25
D-SUB 9 Position Mountable Connection	1	$ 1.00
	Total	*$ 17.55*

Table A.1: A complete breakdown of the component costs for the GoodThopter 10 board.

Appendix B

OBD-II Scanner Comparison

Table B.1: A brief list of available OBD-II scanners and writers available on the market. Tools that allow for writing onto the CAN bus are expensive and still contain limited capabilities. The exceptions are the Tactrix Openport 2.0 which is only compatible with a select number of cars and the Microchip CAN bus analyzer which is primarily designed for debugging networks.

Analyzer	**Read Ability**	**Write Ability**	**Interpret Ability**	**Cost**
OTC 3417 Diagnostic Tool	✓	✗	✓	$1,850.00
DRB III Scann Tool Kit	✓	✓	✓	$9,258.00
Nexiq Pro-Link IQ Diagnostic Scan Tool	✓	✓	✓	$24,950.00
Tactrix Openport 2.0	✓	✓	✓	$169.00
Ford's Integrated Diagnostic Tool	✓	✓	✓	$5,575.00
Auterra DashDyno SPD	✓	✗	✓	$299.99
Microchip Analyzer	✓	✓	✗	$99.00
Actron AutoScanner	✗	✗	✗	$240

Appendix C

Specifications

Figure C.1: Table with a complete list of project specifications, including priority, explanation, justification, quantification, testing criteria, and whether the specification was fulfilled.

	Priority	Explanation	Justification	Quantification	Test	Current Status	Implementation or Alternative
Read Capability							
Basic Read	Highest	Intercept 1 or more packets off the CAN bus reliably and accurately	Necessary for all experimentation	>99% reliability and accuracy for reading in packets	Use valid packet confirmation	Complete	client.rxpacket
Valid Packet Confirmation	Highest	Confirm packet was valid CAN message	Necessary to verify hardware is configured correctly and packets are not being garbled	Catch and report 100% of invalid messages received	Check MCP2515 error flags	Complete	sniff (debug)
Frequency Identificaiton	Highest	Determine the frequency of the CAN bus by re-configuring hardware and listening to network traffic	Necessary for all experimentation	Test all common CAN frequencies (100, 125, 250, 500 kHz) and report if messages are valid	Compared results against wiring diagram	Complete	frequencyTest
Speed	High	Able to intercept all packets sent on bus	Desirable to allow for better characterization of CAN bus	Read in 450 packets/second	See paper	Read in 115 packets/s	Would require firmware upgrades to Siege's GoodTHOPTER10 board
ArbID filtering	High	Use hardware filters to ignore all but designated ArbIDs	Mitigates inability to receive all packets sent over wire	Receive 0 messages from other ArbIDs on network	Verify fewer than maximum number of packets received	Complete	addFilter
Data filtering	Low	Use hardware filters to selectively listen for specific data contents	Mitigates inability to receive all packets sent over wire	Receive 0 messages that do not contain the specified data contents	Verify fewer than maximum number of packets received	Complete	filterForPacket
Message Identification	Low	Determine and report the message type (standard/extended, data/remote frame)	Helpful for characterizing CAN bus traffic	Identify and report message type for 100% of received packets	Compare results to packet contents	Complete	sniff (debug)
Write Capability							
Basic Write	Highest	Inject 1 or more packets to the CAN bus	Necessary to gain control over bus	Reliable, error-free packet injection (>99% accuracy)	Use *sniff* with a second GoodTHOPTER	Complete	spit
Speed	High	Inject packets faster than ArbIDs send them	Necessary to "talk over" normal traffic to gain control	Varies by ArbID; between 1 and 115 packets/second	Use *sniff* with a second GoodTHOPTER	Complete	spitMultiplePackets
Randomized Write	High	Inject a series of packets with targeted, randomized data	Used for "fuzzing" experiments	Packet injection with specified, randomized data bytes (>99%)	Use *sniff* with a second GoodTHOPTER	Complete	experiments.generationFuzz
Packet Response	Medium	Listen for a specific packet and inject with a response	Used to fake scan tool and hacks	Able to receive and respond to >99% of messages	Use *sniff* with a second GoodTHOPTER	Complete	experiments.packetRespond
Flexible Write	Low	Write and/or send multiple packets with a single method call	Simplifies code base and speeds up packet injection	None	Use *sniff* with a second GoodTHOPTER	Complete	spitMultiplePackets
Reading and Displaying Data							

Output to SQL database	Highest	Store received packets in SQL database	Necessary to organize experimentation	>99% reliability and accuracy sending and storing data in SQL database	Verify packets are properly passed to SQL database	Complete	DataManage.py
Output in PCAP format	Highest	Convert received data to PCAP format, which can then be loaded into Wireshark for analysis	Deliverable from Siege Technologies	None	Verify packets are read into Wireshark with correct formatting	Complete	
Commenting on Data	High	Tag groups of received packets with comments regarding experimental methodology	Necessary to organize experimentation	Able to intuitively associate groups of packets with comments describing them	Verify comments are properly passed to SQL database	Complete	"Comment" field in "Sniff" method; uploaded to SQL database
Intuitive Real-Time Display	High	Display network traffic in a helpful manner	Allows observation of real-time fluctuations in data values across multiple ArbIDs to associate network response with inputs to car	None	User feedback	Complete	"Streaming" or "Fixed" display
Automatic Packet Parsing	Medium	Break packets down into CAN message components (ArbID, DLC, data bytes)	Helpful for abstracting away details of CAN protocol	None	User feedback	Complete	
Injecting Packets							
Directly Input Packet	Highest	Directly enter desired ArbID and data byte values	Most basic form of injection	>99% reliability and accuracy	Use *sniff* with a second GoodTHOPTER	Complete	"Inject packet" field in GUI
Inject List of Packets	High	Inject a pre-determined sequence of packets	Necessary for more complicated hacks	Inject an unlimited number of pre-formatted packets in sequence	Use *sniff* with a second GoodTHOPTER	Complete	"Inject packets from file" field in GUI
Experimental Methodology							
Generalizable	Highest	All methods and experiments are applicable to other vehicles	Deliverable from Siege; necessary for the project to have any value	None	Used background characterization on a more modern car	Complete	All experimental methodology kept as generable as possible
End-to-End Process	Highest	Methodology takes the user from the black box conception of the car through an impactful hack	Deliverable from Siege; necessary for the project to have any value	None	Medium-speed CAN bus hack	Complete	Experimental methodology complete
Intepretable	Highest	A technical user with limited CAN experience can implement the methodology with little to no bring-up time.	Necessary to maximize usefulness for Siege.	None	Sponsor feedback.	Complete	Experimental methodology complete
Robust	Low	Methodology does not rely on particular protocol implementations to be successful. Accounts for more complex systems.	Necessary for use on more modern vehicles and more complex implementations.	None	Medium-speed CAN bus hack	Complete	Experimental methodology complete
Expedient	Low	Methodology should rapidly result in a successfully hack.	Shorter time maximizes demonstration of vulnerabilities.	< 1 day from black-box to hack	Medium-speed CAN bus hack	Complete	Experimental methodology complete

Proof of Concept								
Replicability	Highest	Able to repeatedly and reliably cause effect	Demonstrates true control over system			Complete		
Control	Highest	Able to systematically manipulate one or more car components	Demonstrates true control over system			Complete		
Affect Safety	High	Affect one or more components which could directly cause a safety hazard to the driver, passengers, or others	Demonstrates potentially catastrophic security flaw			Complete		
Failure	Medium	Able to cause clear failure of one or more components	Demonstrates some level of control over system			Complete		
Chronic Control	Low	Effects persist after packet injection ends	Demonstrates ability to permanently affect network			Not Complete		

Appendix D

SQL Trade Study

Specification:	Storage Method	Extensibility	Load time	Adding	Access	multi trial comparisons	sorting	filtering	Ease of use	complexity to implement	Sum
weight		**2**	**1**	**1**	**2**	**1**	**1**	**2**	**1**	**2**	
SQL	SQL database	4	5	5	5	5	5	5	2	1	**52**
Hash table	csv files	4	3	5	4	4	2	4	3	3	**47**
Double Linked List	csv files	2	2	5	2	4	2	4	5	5	**44**
List	csv files	3	2	1	2	4	2	1	2	5	**33**
No central packet storage	-	3	1	5	2	3	1	1	1	5	**33**

Figure D.1: A trade study showing the design decisions and weights that led to the selection of the SQL Database for the project data storage system.

Appendix E

JSON Trade Study

	Simple to implement	Interpertable	Human Readable	Import Software	Simple format	Compact layout	Efficient	Data heirarchy	Total
JSON	4	4	4	4	4	3	4	4	31
HTML	2	2	4	2	2	2	2	4	20
XML	3	3	4	3	3	2	3	4	25
CSV	1	3	4	2	4	4	1	1	20

Figure E.1: A trade study showing the design decisions and weights that led to the selection of the JSON storage system for storing the knowledge about a CAN bus.

Appendix F

Experimental Setup

Below are figures outlining the experimental setup used in order to run experiments on our test vehicle, a 2004 Ford Taurus. OBD-II to db9 connector cables are necessary to connect the GoodThopter10 board to the port. Figure F.1 shows the OBD-II connector port on a 2004 Ford Taurus which is located underneath the steering wheel.

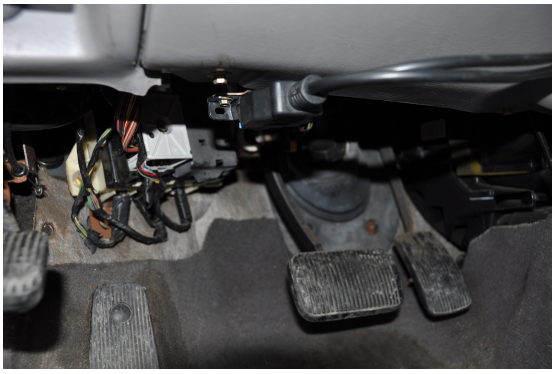
To connect directly to both the high speed CAN and the medium speed present on the Ford Taurus custom connectors were created and can be shown below in Figure F.2. Here we demonstrate two types of connectors that have been made.

These components can all be hooked up together to create the experimental setup shown in Figure F.1a. An additional image of the cables connecting from the OBD-II cables to the GoodThopter10 board can be seen below in Figure F.3.

When creating costume connectors for the OBD-II to GoodThopter10 board, the user should be aware that many different wirings exist for OBD-II to db9 cables. The pinouts for both a standard OBD-II port and for the GoodThopter10 board can be seen below in Figure F.4.

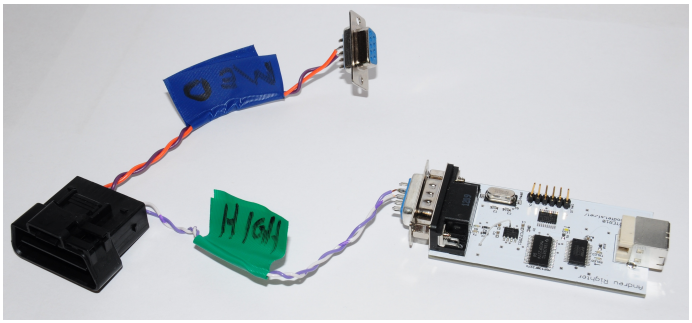
(a)

(b)



(c)

Figure F.1: Figure (a) shows an experimental setup on a 2004 Ford Taurus where the GoodThopter10 boards are connected to the car's CAN bus through the OBD-II port. The boards are then connected to the laptop shown on the right side of the picture. Figure (b) is an image of the OBD-II port on the car, located below the steering wheel and above the peddles. Figure (c) shows this OBD-II port with our connector cables attached

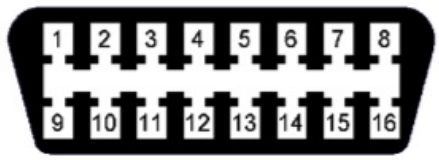


(a)

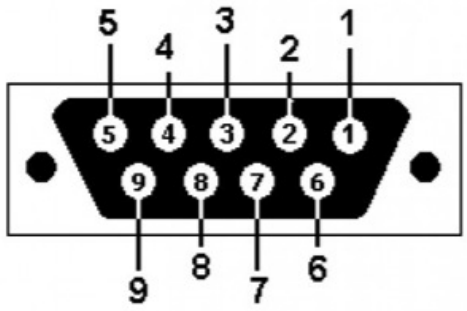
(b)

Figure F.2: Pictures of the custom cables created to connect the GoodThopter10 board to the OBD-II port of a vehicle. Figure (a) shows a custom OBD-II connector that was made which allows for connection to the medium speed CAN and high speed CAN on the 2004 Ford Taurus through one connector. Figure (c) shows the connector made which makes use of a commercially available OBD-II to db9 cable.

Figure F.3: Photo of the connection of our GoodThopter10 boards to the vehicle next to the gear shifter. These boards are connected to the OBD-II port via the db9 connection port on the GoodThopter10 and our computers through the USB port on the board.



1 – Vehicle specific
2 – J1850
3 – blank
4 – Chassis Ground
5 – Signal Ground
6 – CAN High
7 – ISO 9141 2K
8 – Vehicle specific
9 – Vehicle specific
10 – J1850 bus
11 – Vehicle specific
12 – Vehicle specific
13 – Signal Ground
14 – CAN Low
15 – ISO 19141 2L
16 – Battery Power



1. unused
2. CAN Low
3. unused
4. unused
5. unused
6. unused
7. CAN High
8. unused
9. unused

Figure F.4: The image on the left shows the pinout for the OBD-II port on a commercial vehicle. These are the standard pins but several have been left for vehicle specific features. For example, the 2004 Ford Taurus used pins 3 and 11 for a medium speed CAN High and Low respectively. The image on the right shows the db9 connections for the GoodThopter10 board. Pins 7 and 2 are used for CAN High and Low respectively.

Appendix G

Economic Analysis

	Assumed Productivity Fraction					
	10%	**20%**	**30%**	**40%**	**50%**	**60%**
Project Labor Cost	$ 12,500	$ 25,000	$ 37,500	$ 50,000	$ 62,500	$ 75,000

Table G.1: Labor Costs, with varied productivity assumptions, 50 hours worked over 2 10-week terms, and a median professional engineering charge of $125/hour.

User Manual For CAN Reader

Christopher Hoder
Grayson Zulauf
Ted Sumers

Thayer School of Engineering
Dartmouth College
Hanover, NH 03755
March 5, 2013

Contents

1 Introduction

The CAN Data reader will allow you to interact with the CAN bus through passive sniffing as well as active injection and experimentation. Below in Figure 1 is a screen shot of the entire window for the CAN reader:

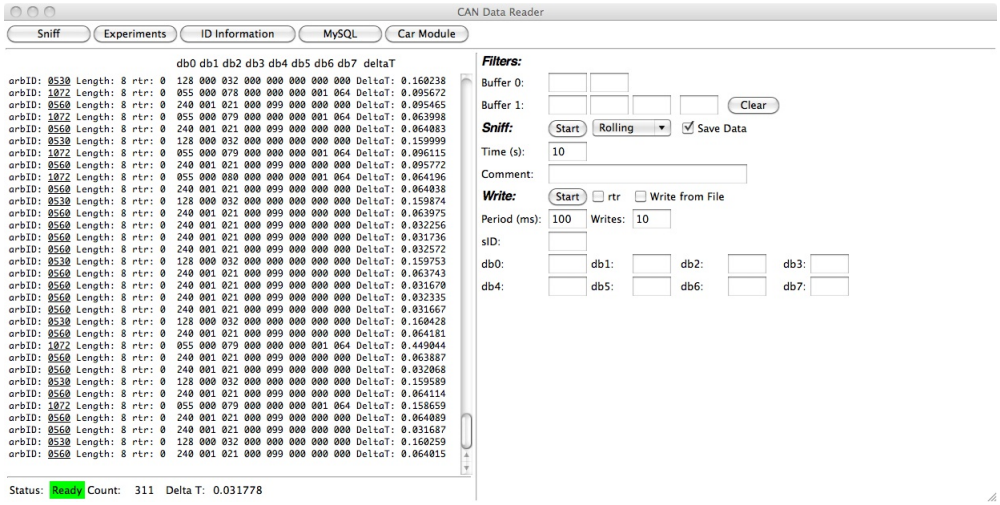


Figure 1: Screenshot of the the CAN Reader. This shows the screen the user will see upon starting the program. The sniffed data will be displayed as shown on the left side while the right side can be tabbed between five different screens.

The top menu buttons provide the user with five possible different tabs. These options will change the items in the right side frame. More detail about these options can be found in the tab's corresponding sections in this user guide. The left side of the window is a frame where the sniffed data will be streamed when the user is listening to CAN traffic. At the bottom of this frame, there is a status indicator that allows the user to know if the board is connected, ready to use, or in use.

Additionally there are some simple statistics about packets coming into the program such as the number of packets received (*count*) and time gap between packets received (*Delta T*). The *Delta T* for each packet will be specific to each arbitration ID and will tell you how long it has been since the last packet with that ID has been received. The initial value is -1 if the ID has never been seen before in a particular sniff session.

2 Installation

The CAN reader requires the installation of several python packages. In order to run the client software for communicating with the GoodThopter10 board you will need python-sqlite3 and python-serial packages. For Windows, you need a 32-bit version of Python because there is no 64-bit python serial for Windows.

The CAN reader also requires the MYSQL python package as well as MySQLdb. MYSQL can be found at the following link: http://dev.mysql.com/downloads/mysql/. MySQLdb can be found here: http://sourceforge.net/projects/mysql-python/.

Mac users may encounter an issue installing these packages. After installing, run the following commands in terminal:

```
shell>> python -c "import MySQLdb"
```

If you do not get an error message the package has been installed. If you get the following error message:

```
import MySQLdb
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/Library/Frameworks/EPD64.framework/Versions/6.3/lib/python2.6/site-packages/
MySQL_python-1.2.3-py2.6-macosx-10.5-x86_64.egg/MySQLdb/__init__.py",
line 19, in <module>
import _mysql
ImportError: dlopen(/Library/Frameworks/EPD64.framework/Versions/6.3/lib/
python2.6/site-packages/MySQL_python-1.2.3-py2.6-macosx-10.5-x86_64.egg/
_mysql.so, 2): Library not loaded: libmysqlclient.18.dylib
Referenced from: /Library/Frameworks/EPD64.framework/Versions/6.3/lib/
python2.6/site-packages/MySQL_python-1.2.3-py2.6-macosx-10.5-x86_64.egg/_mysql.so
Reason: image not found
```

You can fix this by resetting the $DYLD\_LIBRARY\_PATH$ setting. First you will need to locate your mysql_config file. This is most likely in the /mysql/bin folder from the package downloaded when installing mysql.

Once located you need to go to the downloaded MySQLdb folder and open setup_posix.py and find the line:

```
mysql_config.path= "INSERT path to mysql\_config file"
```

Then in your *.bash_profile* file add the following line:

```
export DYLD_LIBRARY_PATH="INSERT path to mysql_config file"
```

Now go back and reinstall MySQLdb.

The source code for the CAN reader can be downloaded from the Goodfet source forge page, http://goodfet.sourceforge.net/ and is located in the ./goodfet/contrib/hoder directory. The main display can be run from

```
python mainDisplay.py
```

You may need root privileges to access the USB ports for communication with the Goodthoper10.

3 Sniff/Write Tab

This tab in the CAN Reader will allow the user to perform the basic interactions with the CAN bus through the GoodThopter10 board. The user will be able to sniff and write to the bus as well as set filters on various arbitration IDs. Below in Figure 2 you can see a screenshot of the right frame when the sniff tab is on top. Each method is discussed below.

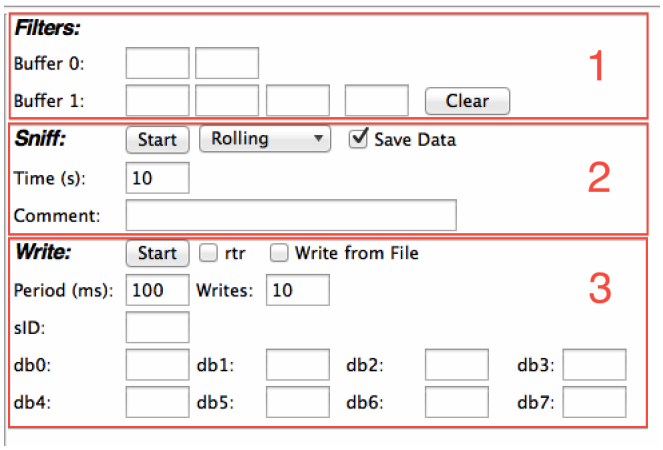


Figure 2: Screenshot of the Sniff/Write table on the CAN Reader. The different options have been boxed in red and numbered for reference

1. Filters These inputs provide the option of setting filters for sniffing the CAN bus. Note that at least 3 filters must be set to *only* get the given ids. This is because there are 2 buffers on the GoodThopter10 and filters need to be set to both in order to only get those packets. However, as long as one filter is set this ArbID will be filtered for in one buffer.

2. Sniff These inputs allow the user to sniff the CAN traffic. The drop down menu allows the user to choose between two different types of view of the incoming packets. *Rolling* will print a new line for each packet that is received. *Fixed* will maintain one line for each ID and update this line when a new packet arrives. The save data checkbox gives the option of saving data or not (the default is to save data). See File Management for information on how sniffed packets are saved. The sniff button will set any filters provided for *time* seconds and attach the given *comment* to all packets that are received. These are saved to the data location path specified in the settings file.

3. Write These inputs allow the user to inject data packets onto the CAN bus. The user can inject either a regular data packet or a Remote Transmission Request (RTR) by

selecting the checkbox. The user can specify the delay between packets in milliseconds with *period* as well as the number of writes of the data packet with *writes*. sID is the standard ID of the packet and $db0$, $db1$, ..., $db7$ are the data bytes. The inputs are assumed to be in decimal form.

Also, the user can inject a series of data packets by selecting the $Write\ from\ file$ option. This will open a file dialog where the user can select a comma separated value (.csv) file to inject. The format of the file should be a .csv file where each row corresponds to a separate packet. Any row with a '#' will be ignored. The columns of a row containing a data packet are assumed to be in the following format. All values are assumed to be in decimal format:

- Column 0: Delay time from previous row (seconds). If no delay, set to 0.
- Column 1: Standard ID
- Column 2: Data Length (0-8)
- Column 3: Data byte 0

  ...
- Column 7: Data byte 7

4 Experiments Tab

This tab in the CAN Reader will allow the user to perform more advanced interactions with the CAN bus. The experiments allow the user to gain insight into the higher level protocols and interactions that are taking place on the bus. Figure 3 has a screenshot of the experiments frame, followed by a discussion of each method.

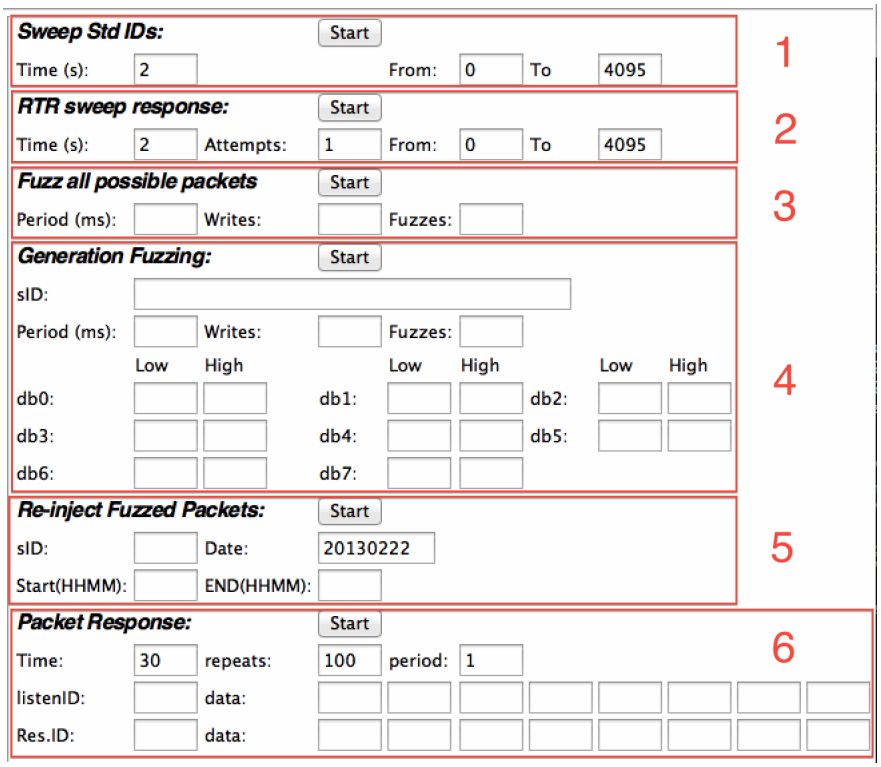


Figure 3: Screenshot of the Experiments tab on the CAN Reader. The different experiments have been boxed in red and numbered for reference.

1. Sweep Std IDs This option will allow the user to sniff a range of arbitration IDs and see what messages are on the bus. The method will sniff in groups of six IDs for *time* seconds. If at least one message is received, the program will cycle through the six IDs and listen for *time* seconds.

2. RTR Sweep Response This option allows the user to perform a Remote Transmission Request (RTR) on each byte in the range given. Each RTR will be sent *attempts* times and then the bus will be sniffed for *time* seconds for a response.

3. Fuzz all possible packets This option will create randomly generated, correctly formatted packets with a random standard ID in the range 0 to 4095 (all possible values). The generated packet will then be written to the bus *writes* number of times with a wait of *period* milliseconds between re-injections. This process will be repeated $Fuzzes$ number of times.

4. Generation Fuzzing This option will create randomly generated packets where each byte's value is randomly chosen in the range specified from *low* to *high*. The standard id will be chosen randomly from the standard ids input (sID). They must be input with comma separations. The generated packet will then be written to the bus *writes* number of times with a wait of *period* milliseconds between re-injections. This process will be repeated $Fuzzes$ number of times.

5. Re-Inject Fuzzed Packets This option will allow the user to re-inject messages that were fuzzed in either method 3 or 4. The only required input is the date. The date is used to locate the file from which contains the packets we wish to inject. If the user inputs a standard ID (sID) then only this standard ID will be re-injected (can only take one ID). If a start time is specified in the HHMM format, only packets that were injected after the given time will be injected. Likewise, only inject packets originally injected before the end time will be reinjected.

6. Packet Response This option will allow you to do a bit of interaction with the bus. You can set a packet to listen for and then respond with the set packet *repeats* number of times at an interval of *period* milliseconds. $ListenID$ is the standard ID one wishes to listen for. If the corresponding data fields are left blank then this will simply listen for the ID and respond when it is read in. If there is a specific packet specified then that packet will need to be read in. $Res.ID$ is the standard ID that will be written back to the bus along with its corresponding data packets.

5 ID Information

This section describes how the ID Information tab can be used to document a user's knowledge of the packets on the CAN bus. This feature allows the user to add and edit information and record their thoughts straight from the CAN Reader. The data is stored in a .json file that the user sets in the Settings Dialog. For every ID, there are three sub tabs included that provide information on the ID: general information, byte specific information, and known useful packets. Figure 4 below shows the ID information tab.

Figure 4: Screenshot of the ID Information tab in the CAN Reader. This tab allows the user to record and read known information about each arbitration ID..

General Info In the general information tab the user can record the frequency at which the given ID transmits, the CAN bus speed they are reading it on, any known correlations to the ID, commentTags for experiments that have been run, and note any general comments. The General Info tab is above in Figure 4.

Byte Info This tab allows the user to specify specific information about the bytes of a given ID. For every ID the user can record the following: if the values are continuous, if the data has been observed changing, any known correlations, and general comments. Figure 5 shows a screen shot of this tab.

Packets This tab will allow the user to see packets that they want to record for a given ID. These packets can be saved here. When the packets are loaded into the text box the description will be linked so that you can click on the description and it will autofill the write packet section of the sniff tab. These packets are assumed to be in decimal form and should be input in the following form:

```
packet description: db0 db1 db2 db3 db4 db5 db6 db7
```

A screen shot of the Packets sub-tab can be seen below in Figure 6.

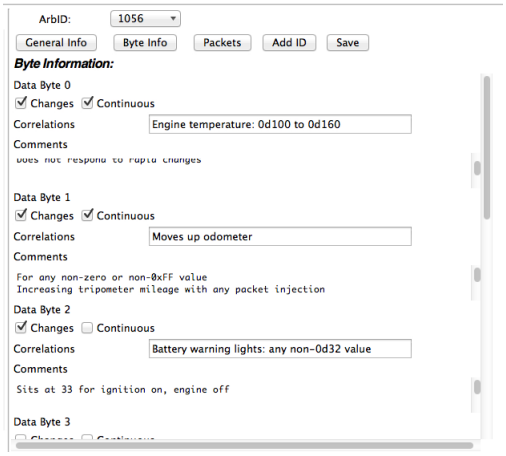


Figure 5: Screenshot of the ID Information tab sub tab that shows our specific knowledge of the bytes for a given ID.

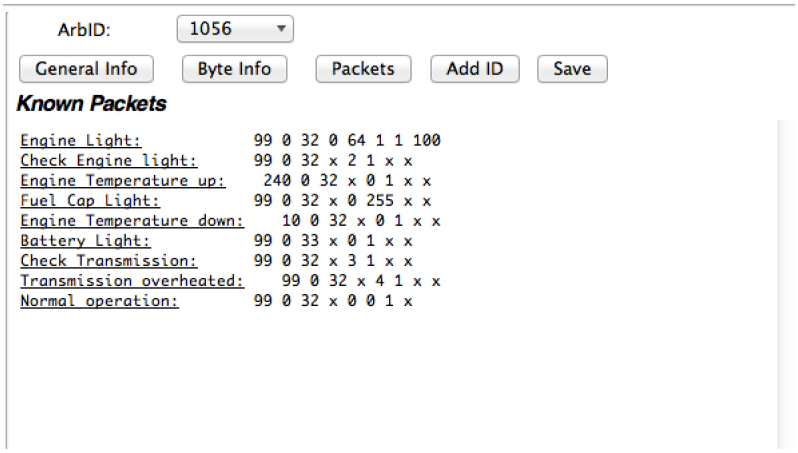


Figure 6: Screenshot of the Packets sub-tab that allows the user to store data packets for a given ID.

6 Settings Dialog

The settings dialog can be accessed by pressing Control-Z or by clicking on the option in the menu bar. This dialog allows the user to select persistent settings in the CAN Reader. An image of the Settings dialog is included below:

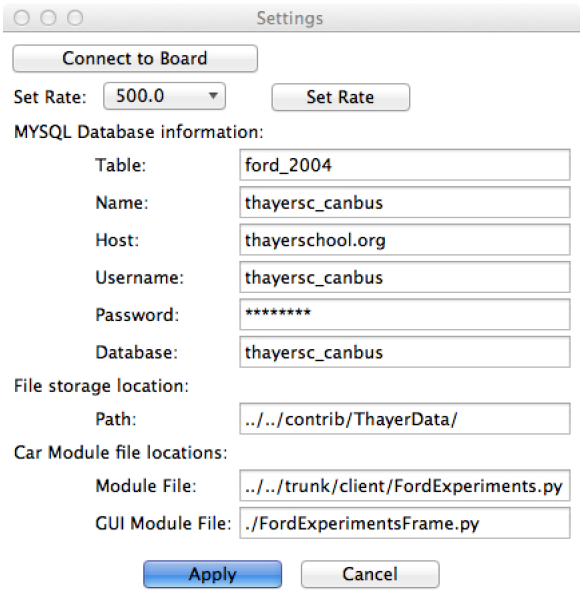


Figure 7: Settings Dialog for the CAN Reader

Here the user can connect the GoodThopter10, set the rate at which the board communicates with the CAN bus, set MySQL database information, designate the root path for all data collection and designate the paths for a car module to the CAN Reader. These settings are saved in the file ./Settings.ini and are loaded each time the program is opened. For a further discussion see section 10 for a discussion on creating custom modules.

7 Database and MySQL Communication Tab

The MySQL tab will allow the user to interact with the database directly from the CAN Reader. See the section on the MySQL database (8) for more information on the database. By pressing this tab button or using the shortcut Control-u the following appears on the right side frame:

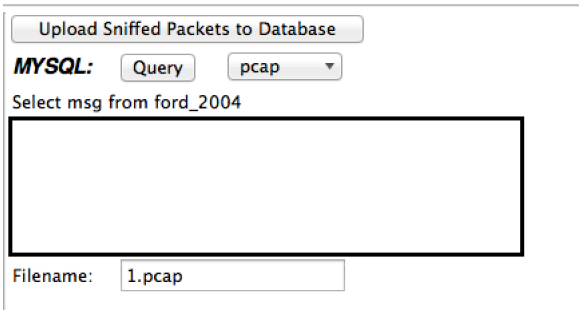


Figure 8: Screenshot of the Database and MySQL tab on the CAN Reader

The **Upload Sniffed Packets to Database** button will upload all files in the file storage path set in the settings. All uploaded files will be moved into a sub-directory whose name is the date of upload. The filename will also be modified to alert the user that the file has been uploaded. See the File Management (9) section for more information.

The second option for the user is to query the database itself. The data that is returned from the query can be saved into two different file formats, comma separated variable file or a .pcap file. The .pcap file will return the data packets in the correct format to be loaded into Wireshark for analysis.

Since the methods for writing to a .pcap file require a particular form of the data from the user should only enter the conditional part of the MySQL query. The "SELECT msg FROM table where" will automatically be added for the user. Should the user select to save the data to a .csv file they can query anything they want. In either case, the data will be saved to the given filename in the SQLData sub folder. The filename is assumed to have the .csv or .pcap ending.

8 MySQL Database

The CAN Reader is designed to work in coordination with a MySQL database. The MySQL server information can be set in the settings dialog and then all the sniffed packets can easily be uploaded to these databases. Table 1 below describes the fields for the MySQL table:

Table 1: Data Fields for MySQL database

Field	Type	Length	Unsigned	Allow Null	Purpose
id	int	11	yes	no	Unique identifier
time	double	20,5	yes	no	Timestamp for the packet
stdID	int	5	yes	no	Standard ID
exID	int	5	yes	yes	Extended ID (optional)
length	int	1	yes	no	Data length
error	bit	1	-	no	Error boolean for packet
remoteFrame	bit	1	-	no	Remote Transmission Request Boolean
db0	int	3	yes	yes	Data Byte 0
db1	int	3	yes	yes	Data Byte 1
db2	int	3	yes	yes	Data Byte 2
db3	int	3	yes	yes	Data Byte 3
db4	int	3	yes	yes	Data Byte 4
db5	int	3	yes	yes	Data Byte 5
db6	int	3	yes	yes	Data Byte 6
db7	int	3	yes	yes	Data Byte 7
msg	varchar	30	-	no	Raw Message
comment	varchar	500	-	yes	Comment Tag
filter	bit	1	-	no	1 if Filtered for
readTime	int	11	yes	no	Length of sniff when read

A new table of this format can be made by using the *createTable* method in the DataManage class contained in DataManage.py. However there is currently no option to do this in the CAN Reader.

9 File Management

The files management for the CAN reader can be done entirely automatically to avoid lost data and confusion when uploading files to the MySQL database. This will serve as a brief discussion about how the data is stored during various experiments.

All data that is used or created by the CAN reader is kept in the directory set by the user in the settings file (6). The sniff method will save all of the packets that were sniffed off the bus if the **Save Data** checkbox is checked. There will be a new .csv file created for every day. The naming convention is making the file the days date of the form YYYYMMDD (an example file would be 20130222.csv). If the file already exists in the folder, the packets will be appended to the end.

When the user uploads the data to the MySQL database via the **Upload Sniffed Packets to Database** button, the files will be renamed with the "_uploaded" tag at the end and moved into a sub folder that is named by the day's date in the same YYYYMMDD format. If there is already a file of the same name in this folder an additional tag will be added to the end of the filename to make it unique. This will be "_i" where i is the first number that leads to a unique filename.

When the user runs a fuzzing experiment (generation or general), the injected data will be saved to a file but in the sub directory folder "./InjectedData/" The filename will again be based on the days date in YYYYMMDD format. However, the following tag will be added to the end: _GenerationFuzzedPackets.csv. An example filename would be: 20130221_GenerationFuzzedPackets.csv.

Data that the user queries for from the MySQL database will be saved to the SQLData subfolder. The filename for these files will be specified by the user. If the file already exists then a tag will be added to the filename of the form "_i" where i is the first integer that leads to a unique filename.

Currently the settings are for the files to be saved in the /goodfet/contrib/ThayerData/ folder. The sub folders InjectedData and SQLData are already part of the svn. However, there is no code to add newly acquired data because the data can be uploaded to the MySQL database and accessed anywhere.

10 Creating Modules

The user can create their own custom modules for this GUI. This allows the user to create options that are specific to their car or needs. This allows the CAN Reader to remain extensible to multiple different cars but allow for specific hacks to be developed by the user without needing multiple windows or programs.

The car module can be created by creating two python files that provide the functionality needed. The *Module File* in the settings dialog refers to the experimentation python file. This file should include a class of the same name that extends the experiments class. This file can contain the car specific methods the user wants to implement and since it is a sub-class of experiments, the CAN Reader will not loose any functionality.

The second file that must be specified is the *GUI Module File*. This file should contain a class of the same name that will construct the tkinter frame for display in the program. The class takes two parameters as inputs: the tkinter frame or canvas that any widgets will be placed on and a pointer an instance of the *Module File* that contains the car specific methods.

The *__init__* method of the *GUI Module File* should place all desired widgets on the given frame. and set any bindings to methods. The provided frame will be an additional tab on the right side of the CAN Reader and when connected, the *Car Module* button on the top menu bar will become active and be used to make this frame visible. The CAN Reader is built using Tkinter in python and the *GUI Module File* is expected to do the same.

For examples the settings currently have a 2004 Ford Tarus specific module setup. The *Module File* is found in goodfet/trunk/client/FordExperiments.py and the *GUI Module File* can be found in goodfet/contrib/hoder/FordExperimentsFrame.py

The example module can be seen below in Figure 9:

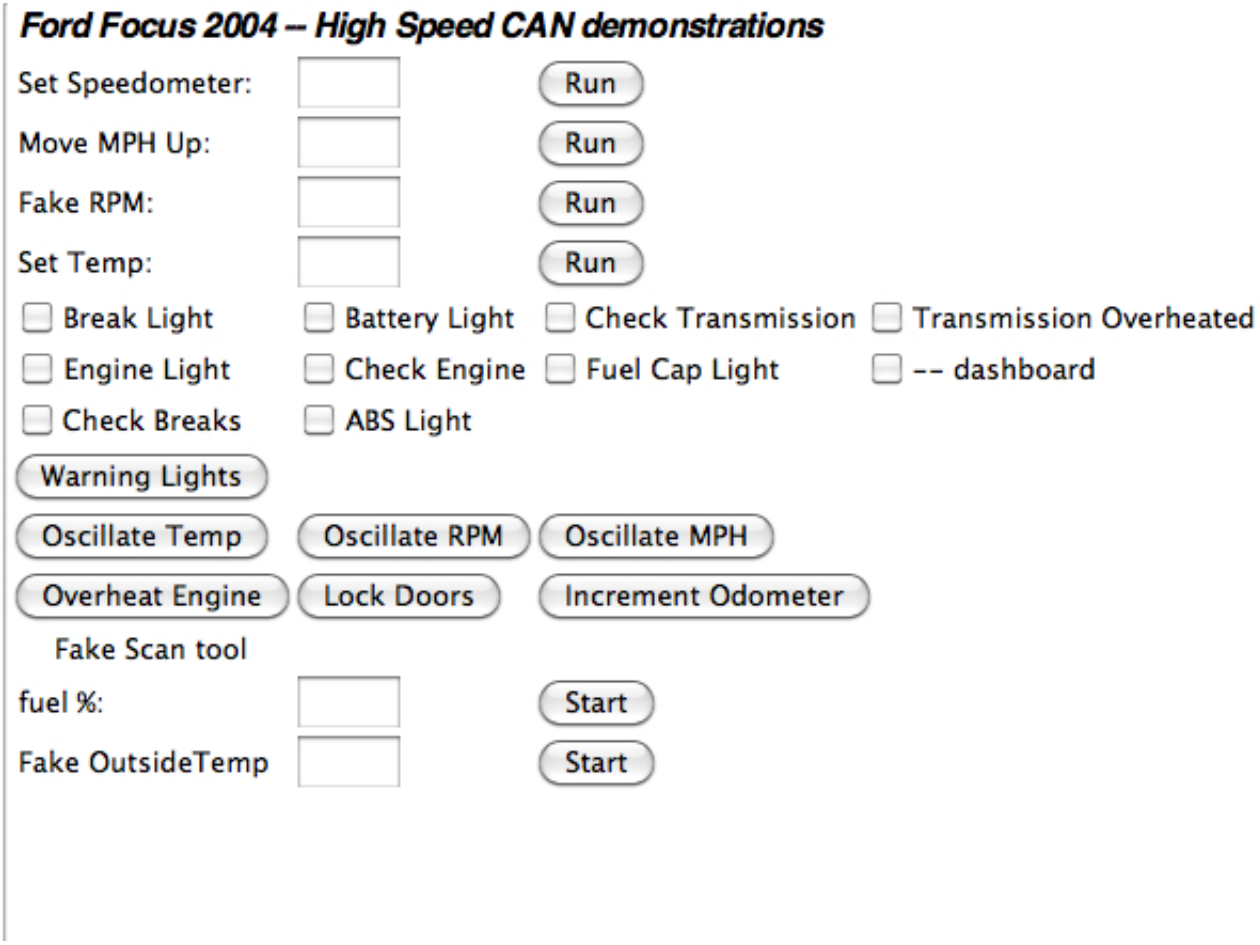


Figure 9: Screen shot of the CAN reader with the car specific module created for a 2004 Ford Taurus that allows for several different car specific hack demonstrations.

11 Wireshark Integration

The software package features integration with the open-source network protocol analyzer, Wireshark. Wireshark supports the analysis of a number of different protocols via packet dissection, filtering, and a simple graphical program. The current download of Wireshark, for UNIX, Windows, or OS X, can be found at http://www.wireshark.org/download.html.

Wireshark supports the analysis of the CAN protcol via two included dissectors, CANopen and socketCAN. Neither of these dissectors supports the analysis of CAN packets that are formatted exactly how our packets are received from the GoodThopter10 board, so we chose to alter the socketCAN dissector to support the project.

Replace the code in the existing socketcan dissector with the *packet-socketcan.c* file included in your package. The code for the socketCAN dissector is located at *.epan\dissectors\packet-socketcan.c*. Copy and paste from the included file to replace the code.

After completing the update to the code, we must ensure that Wireshark builds using the correct version of socketCAN. Use the following commands to perform this build, while in the Wireshark directory.

```
shell>> ./autogen.sh
shell>> ./configure --with-lua
shell>> make
```

To open Wireshark, use:

```
shell>> sudo WIRESHARK_RUN_FROM_BUILD_DIRECTORY=1 ./wireshark
```

Wireshark must always be opened using this command to ensure it opens the build package, not the one from the install location.

In the GUI, enabling the "write to pcap" checkbox will automatically generate a correctly formatted .pcap file from the SQL command generated. After creating this file, use the *build directory* command to open Wireshark, and use

```
File->Open->[filename].pcap
```

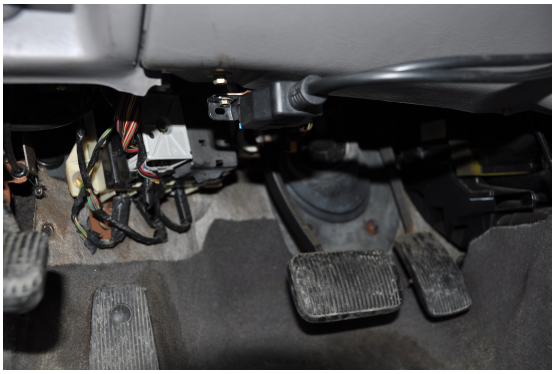
to import the .pcap file. Wireshark's tools may now be used for packet analysis.

12 Experimental Setup

Below are figures outlining the experimental setup used in order to run experiments on a vehicle using our CAN Reader software. OBD-II to db9 connector cables are necessary to connect the GoodThopter10 board to the port. Figure 10 shows the OBD-II connector port on a 2004 Ford Taurus which is located underneath the steering wheel.

(a)

(b)

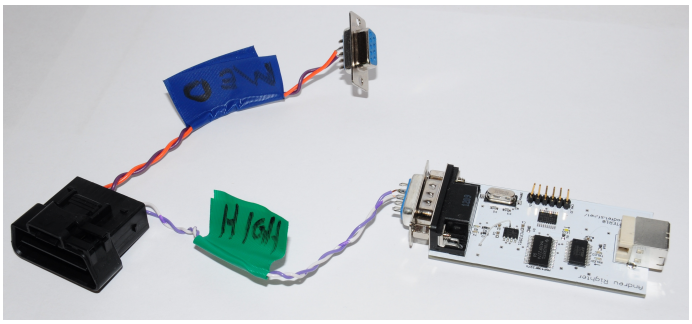


(c)

Figure 10: Figure (a) shows an experimental setup on a 2004 Ford Taurus where the GoodThopter10 boards are connected to the car's CAN bus through the OBD-II port. The boards are then connected to the laptop shown on the right side of the picture. Figure (b) is an image of the OBD-II port on the car, located below the steering wheel and above the peddles. Figure (c) shows this OBD-II port with our connector cables attached

To connect directly to both the high speed CAN and the medium speed present on the

Ford Taurus custom connectors were created and can be shown below in Figure 11. Here we demonstrate two types of connectors that have been made.



(a)

(b)

Figure 11: Pictures of the custom cables created to connect the GoodThopter10 board to the OBD-II port of a vehicle. Figure (a) shows a custom OBD-II connector that was made which allows for connection to the medium speed CAN and high speed CAN on the 2004 Ford Taurus through one connector. Figure (c) shows the connector made which makes use of a commercially available OBD-II to db9 cable.

These components can all be hooked up together to create the experimental setup shown in Figure 10a. An additional image of the cables connecting from the OBD-II cables to the GoodThopter10 board can be seen below in Figure 12.

When creating costume connectors for the OBD-II to GoodThopter10 board, the user should be aware that many different wirings exist for OBD-II to db9 cables. The pinouts

Figure 12: Photo of the connection of our GoodThopter10 boards to the vehicle next to the gear shifter. These boards are connected to the OBD-II port via the db9 connection port on the GoodThopter10 and our computers through the USB port on the board.

for both a standard OBD-II port and for the GoodThopter10 board can be seen below in Figure 13.

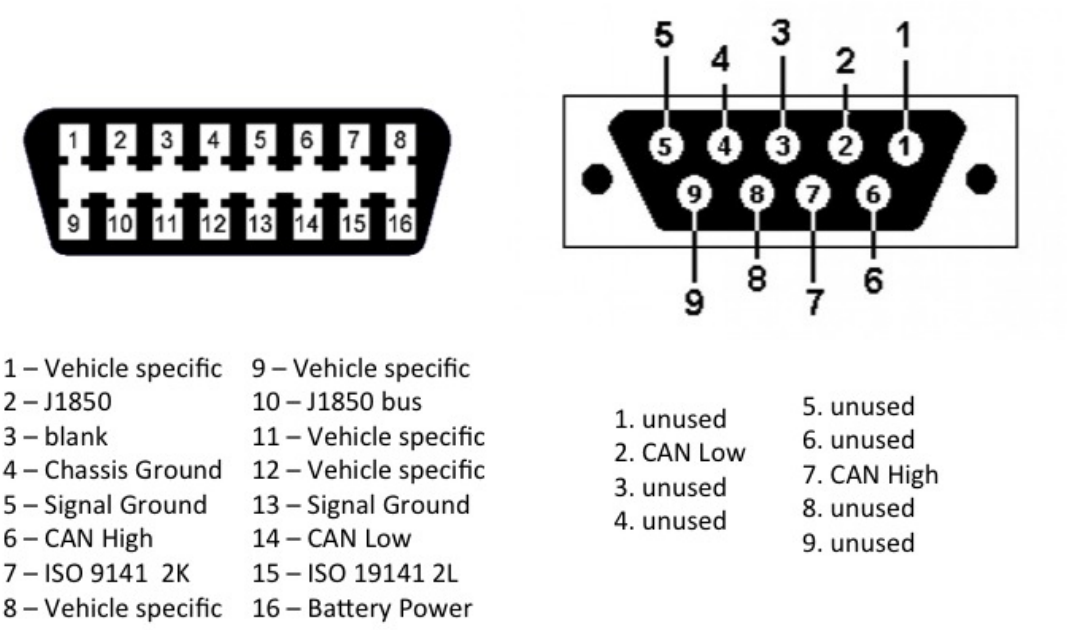


Figure 13: The image on the left shows the pinout for the OBD-II port on a commercial vehicle. These are the standard pins but several have been left for vehicle specific features. For example, the 2004 Ford Taurus used pins 3 and 11 for a medium speed CAN High and Low respectively. The image on the right shows the db9 connections for the GoodThopter10 board. Pins 7 and 2 are used for CAN High and Low respectively.

13 Shortcuts

Command - q	Quits the CAN Reader
Control - q	Quits the CAN Reader
Command - s	Saves the ID Information
Control - z	Opens the Settings Dialog
Control - s	Lifts the sniff tab to view
Control - e	Lifts the experiments tab to view
Control - u	Lifts the MySQL tab to view
Control - i	Lifts the ID Information tab to view

An Extensible CAN Bus Reverse-Engineering Methodology

Christopher Hoder
Grayson Zulauf
Ted Sumers

Thayer School of Engineering
Dartmouth College
Hanover, NH 03755

March 5, 2013

Contents

1 Introduction

Siege requested an extensible reverse-engineering methodology for cars' CAN buses as a deliverable for the project. To be successful, a hacker needs a large library of information about the target car, detailing the operation and protocols for each ECU, as well as their respective interactions. This information, however, is vehicle-specific, and lacks value when applied to exploiting the vulnerabilities of other vehicles. Instead, Siege requested the *methodology* behind obtaining the information, which can then be used to build knowledge and eventually hack any vehicle utilizing the CAN protocol. This methodology was developed in tandem with our software package and refined through experimentation.

At the highest level, the methodology follows a familiar process for system characterization. Any system can be characterized by its response to input signals, and a thorough understanding of our system (i.e. the electrical network present on our experimental car) would allow the a) prediction of the packets outputted in response to a physical input and b) prediction of the physical response ("output") in response to inputted packets. Our methodology first requires the characterization of the unperturbed, background state, before developing the transfer function to predict output responses based on various inputs, as shown in Figure 1. This method allows the user to quickly move from the black-box state of a car to an impactful hack.

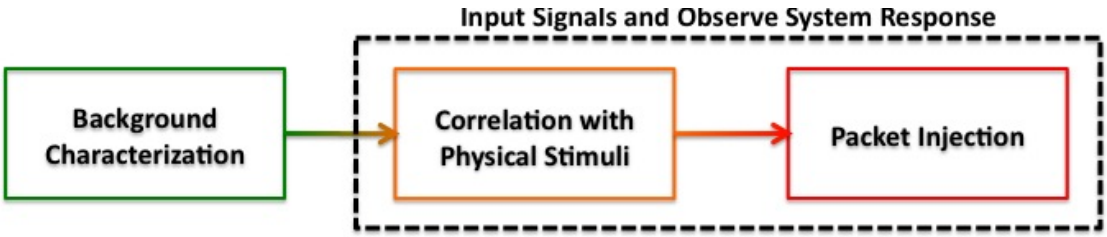


Figure 1: High Level Methodology Flowchart for hacking an automobile's CAN bus. The three main components to reverse-engineering the system are outlined and the appropriate experiments are addressed inturn

This packet includes a rigorous outline of each general experiment necessary to hack the CAN bus on a modern automobile, with the format of the experiments drawn from Montgomery's *Design and Analysis of Experiments, 7th Edition.* Each experiment includes detail of the necessary inputs and expected outputs, as well as a methodology of how to conduct the experiment for both hack expediency and completeness. Our team followed

the methodology, starting with a black-box car, to implement a successful hack on a 2004 Ford Taurus in under 4 hours.

2 Background Analysis Experiments

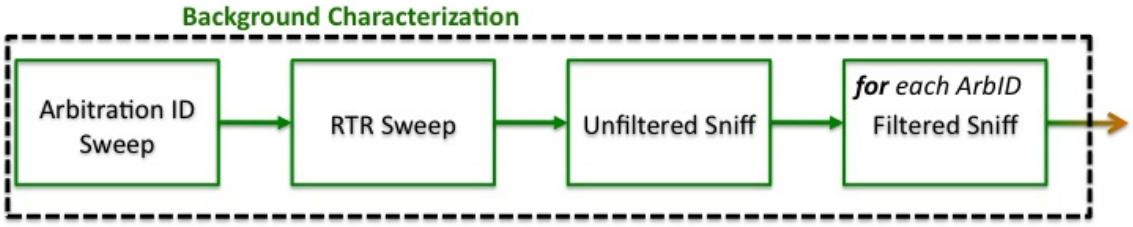


Figure 2: The four experiments required to characterize the background state of the car. The ArbID and RTR Sweeps ensure that the user captures all ECUs on the network, while the sniffs characterize the background packet transmissions and frequencies.

2.1 Arbitration ID Sweep

Objective: Characterize the ArbIDs that are transmitting on the network during steady-state operation.

Control Variables: Car is on in park, all adjustable components are turned off; doors are shut, seatbelts are clicked in; parking brake is up; brakes are released.

Response Variables

1. Observed data packets on CAN bus

Methodology: Sweep across all Arbitration IDs, using `Sweep Std IDs` function included in the GUI. This experiment will filter on each Arbitration ID across the range specified by the user, filtering on each for the amount of seconds inputted in Time.

Repeat for: None.

Inputs:

1. Seconds per ArbID
2. ArbID range

Outputs:

1. Catalogue of ArbIDs that transmit during steady-state operation

Analysis Questions: What ArbIDs are communicating when the car is in this state? How active is the bus with no external stimuli introduced?

2.2 Remote Transmission Request Sweep

Objective: Characterize the ArbIDs that are on the network but not transmitting during normal operation by sending Remote Transmission Requests to all possible ArbID values.

Control Variables: Car is on in park, all adjustable components are turned off; doors are shut, seatbelts are clicked in; parking brake is up; brakes are released.

Response Variables

1. Observed data packets on CAN bus

Methodology: Sweep across all Arbitration IDs and request a response from each, using `RTR sweep response` in the GUI. This experiment will send a remote transmission request (RTR) to each ArbID in the inputted range, and then listen for a response from that ArbID for the user-specified time.

Repeat for: None.

Inputs:

1. Seconds per ArbID
2. ArbID range

Outputs:

1. Catalogue of ArbIDs on the network but do not transmit during steady-state operation

Analysis Questions: What ArbIDs do not transmit during normal operation? How many ArbIDs are on the network in total?

2.3 Unfiltered Passive Listening

Objective: : Characterize the common ArbIDs and their "steady-state" data bytes. Characterize the holistic network operation in the steady-state.

Control Variables: Car is on in park, all adjustable components are turned off; doors are shut, seatbelts are clicked in; parking brake is up; brakes are released.

Response Variables

1. Observed data packets on CAN bus

Methodology: `Sniff` for 120 seconds with no physical stimuli introduced to the car and no set filters. Use this to characterize the unfiltered background steady state of the car in three separate modes: with the car on, with the ignition on but the engine off, and with the car off. Especially note IDs discovered in prior experiments that do not appear here.

Repeat for: None.

Inputs:

1. Seconds

Outputs:

1. Updated ArbID catalogue, in all three car modes.
2. Background data packets for each ArbID

Analysis Questions: What is the percent idle time? What ArbIDs are communicating in each state? For each ArbID, what data packets can we expect to see? How active is the bus with no intentional stimuli introduced?

2.4 Filtered Passive Listening

Objective: : Characterize the common ArbIDs and their "steady-state" data bytes.

Control Variables: Car is on in park, all adjustable components are turned off; doors are shut, seatbelts are clicked in; parking brake is up; brakes are released.

Response Variables

1. Observed data packets on CAN bus

Methodology: For each Arbitration ID discovered above, filter and `sniff` for 15 seconds with no physical stimuli introduced to the car, in each of the three modes described above. Use these packets to characterize the background behavior for each ID, graphically (plot each data byte versus input time) and with the range of values for each data byte.

Repeat for: Each ArbID, car engine off, battery on.

Inputs:

1. Seconds
2. ArbID Catalogue

Outputs:

1. Updated ArbID catalogue, in all three car modes.
2. Background data packets for each ArbID

Analysis Questions: What is the percent idle time? What ArbIDs are communicating in each state? For each ArbID, what data packets can we expect to see? How active is the bus with no intentional stimuli introduced? What are the background packets for each ArbID?

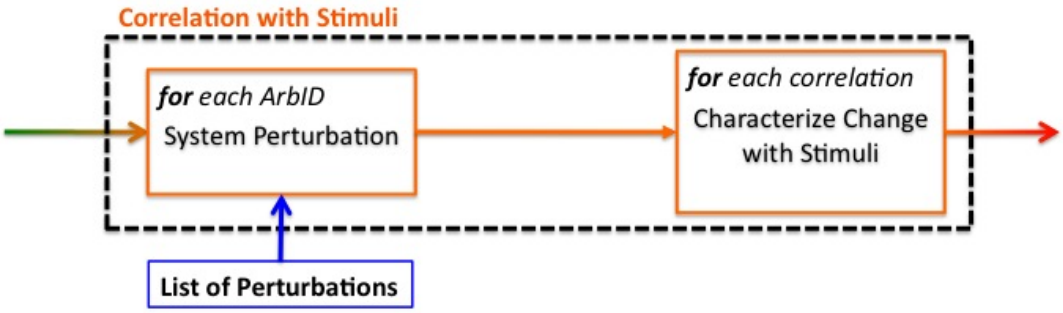


Figure 3: The two experiments used to predict packets from a physical stimulus. All perturbations will be run for each Arbitration ID on the network, and then each ArbID-Perturbation combination must be characterized. The list of perturbations is developed from determining a list of components or actions that are believed to be communicated on the electrical system

3 Correlation with Stimuli Experiments

3.1 System Perturbation

Objective: : Observe CAN bus packet response to injection of physical stimuli to the system (i.e. revving the engine). This will allow us to begin correlating ArbIDs and their data payloads to the change in state of physical components of the car.

Control Variables: Car is on in park, all adjustable components are turned off (except for the component under test); doors are shut, seatbelts are clicked in; parking brake is up; brakes are released.

Response Variables

1. Observed data packets on CAN bus

Methodology: Do a complete system perturbation run to see if each ID is correlated with physical stimuli. It is recommended to order a wiring diagram of the car, and document the modules connected to the CAN bus. Through this documentation, a systematic order should be defined for all of the components on the bus, so that for each Arbitration ID, the same physical components are altered, in a set order. If a change is observed in reaction

to any physical stimuli (via observation in the GUI, in the SQL Database, or graphically), repeat the fiddling until the exact physical stimuli have been identified.

Repeat for: All Arbitration IDs

Inputs:

1. ArbID Catalogue
2. List of system perturbations to be introduced

Outputs:

1. For each ID, a list of the physical stimuli that cause an observable change in the packets transmitted or the frequency at which they appear.

Analysis Questions: 1. Were there any new arbIDs? Do any specific arbIDs always follow each other? Is there a response delay? Is there a refresh rate of the packet where the same packet is repeated on the bus during component stimulation? Does only a particular data byte change? How many bytes change? What is their range? Are the bytes discrete or continuous?

3.2 Characterize Change with Stimuli

Objective: : From the prior experiment, further refine our understanding of those ArbIDs that do respond to stimuli, to the point that we may predict the relationship between a physical stimulus and the exact packets that appear on the bus.

Control Variables: Car is on in park, all adjustable components are turned off (except for the component under test); doors are shut, seatbelts are clicked in; parking brake is up; brakes are released.

Response Variables

1. Observed data packets on CAN bus

Methodology: For each observed correlation and each ArbID, `sniff` and filter for only the respective ID and only change the stimulus recorded above. Use these experiments to characterize the change from the background state for each ID, and ideally for each data byte in the respective ID. Note how different states/value of the physical stimulus change the values transmitted by the ID (e.g. consider when the wipers are on and off, but also look at the intermediate values).

Repeat for: All Arbitration IDs, all perturbations.

Inputs:

1. Background characterization for each ID
2. List of physical stimuli that caused a change in each ArbID

Outputs:

1. Deviation from background characterization for each ID and physical stimulus pair

Analysis Questions: 1. Were there any new arbIDs? Do any specific arbIDs always follow each other? Is there a response delay? Is there a refresh rate of the packet where the same packet is repeated on the bus during component stimulation? Does only a particular data byte change? How many bytes change? What is their range? Are the bytes discrete or continuous?

4 Packet Injection Experiments

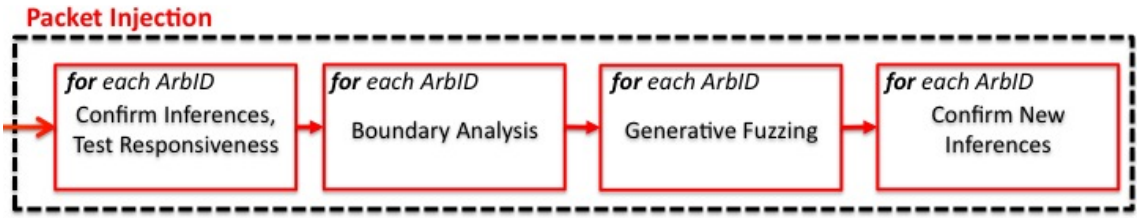


Figure 4: The four experiments used in the packet injection portion of the experimental methodology. These experiments outline a methodology for testing hypotheses about component correlations with network communication as well as fuzzing experiments to discover new ones.

4.1 Confirm Inferences, Test Responsiveness

Objective: Gain confirmation of our inferences about what data is being transmitted by a given ArbID by injecting the ArbID-packet combinations discovered in the last experiment into the CAN bus and monitoring the responses.

Control Variables: Car is on in park, all adjustable components are turned off; doors are shut, seatbelts are clicked in; parking brake is up; brakes are released.

Response Variables

1. Observed data packets on CAN bus
2. Physical response from component

Methodology: Confirm inferences from *Correlate Stimuli*, Section 3, by injecting packets and observing physical stimuli. In this stage, we should be able to predict the physical response to certain packets. For some ArbIDs, however, the predicted physical stimulus will not occur this does not necessarily imply our inference from *Correlate Stimuli* was incorrect, and instead likely implies that this ArbID is listen only or state-dependent.

Repeat for: All Arbitration IDs-Packet combinations discovered in *Correlate Stimuli*.

Inputs:

1. Physical stimulus and packet correlation for each ArbID

Outputs:

1. ArbIDs and stimuli that confirm inferences
2. ArbIDs and stimuli that change our inferences
3. ArbIDs and stimuli that do not respond to packet injection

Analysis Questions: Does physical response match expected response based on prior experimentation? If it is different, how so? Is the component listen-only? Was the same component affected? Does CAN traffic change after injection? Does this change match previously observed CAN traffic?

4.2 Boundary Analysis

Objective: Systems have a tendency to fail on boundaries; thus, we will test the boundaries of observed data bytes for a given arbID. There are 4 boundary values of importance: just before and just beyond the expected range of values, and the absolute minimum and maximum possible data values (0d00 and 0d255, respectively).

Control Variables: Car is on in park, all adjustable components are turned off; doors are shut, seatbelts are clicked in; parking is break up; brakes are released.

Response Variables

1. Observed data packets on CAN bus
2. Physical response from component.

Methodology: Perform boundary analysis on all ArbIDs, by injecting 0d00, 0d255, and the "one-beyond" values into the relevant data bytes using the `Write` feature in the GUI. Observe any physical stimuli in response to these injections. If physical stimuli are observed, go back to *Correlate Stimuli*, Section 3, to further refine your understanding of these particular packet and physical response combinations.

Repeat for: All Arbitration IDs on the bus, all data bytes on each ArbID.

Inputs:

1. ArbIDs catalogue

Outputs:

1. ArbID and boundary value combinations that have observable responses

Analysis Questions: Are there any abnormal behaviors? Component failures? Error lights? Any new packets on bus? Does traffic increase? How can we change the packet to induce a different response? How many bits do you need to change? Are the bit changes Boolean? Or linear? How sensitive is the component to a change in bit? How many bits need to be changed to generate a noticeable value shift?

4.3 Generative Fuzzing

Objective: Explore the protocol structure for signals not yet discovered by randomly creating correctly formatted data packets.

Control Variables: Car is on in park, all adjustable components are turned off; doors are shut, seatbelts are clicked in; parking brake up; brakes are released.

Response Variables

1. Observed data packets on CAN bus (percent traffic, actual packets).
2. Physical response from component.

Methodology: For each ID, use generative fuzzing (`Experiments->Generative Fuzzing`) to test for further physical stimuli. Place in a time that is equal to or faster than the ID frequency characterized in Experiment 4. Use the full range of 0d00 to 0d255 as a default for the values to fuzz over, but refine this range based on observed correlations or to test only certain data byte values. Note any new physical responses observed, the time at which they occurred, and save the associated fuzzing file with the injected packets.

Repeat for: All Arbitration IDs on the bus.

Inputs:

1. ArbIDs catalogue
2. Known correlations
3. Fuzzing ranges

Outputs:

1. ArbIDs with responses to fuzzing
2. Timestamps of physical responses
3. File with Injection Packets

Analysis Questions: Do we cause any unexpected behavior? Do we have an ArbID correlated to the component affected? Did it respond to the input itself? If not, have we seen the response received before?

4.4 Confirm New Inferences

Objective: Generate and confirm the new inferences predicted during *Boundary Analysis*, Section 4.2, and *Generative Fuzzing*, Section 4.3.

Control Variables: Car is on in park, all adjustable components are turned off; doors are shut, seatbelts are clicked in; parking brake up; brakes are released.

Response Variables

1. Observed data packets on CAN bus (percent traffic, actual packets).
2. Physical response from component.

Methodology: Generate and confirm new inferences from fuzzing reduce packets from successful fuzzing until exact data bytes and packet combinations are identified. Re-inject the saved fuzzing file (`Write->Inject from file`), reducing the number of packets included in the file until the packet(s) causing the physical response has/have been identified. Return to *Correlate Stimuli*, Section 3, to confirm these inferences, by predicting the response to appropriate physical stimuli.

Repeat for: All new stimuli observed in response to fuzzing and boundary analysis.

Inputs:

1. ArbIDs catalogue
2. Known correlations
3. Fuzzing ranges
4. Fuzzing packet files
5. New correlations

Outputs:

1. ArbID-packet combinations for new physical responses

Analysis Questions: Which packet or combination of packets is causing the new response? Was the fuzzing only affecting one ID? If not, what combination of IDs is necessary to generate the physical response? Can this be repeated by reducing the number of packets in the inject-from file? Does this occur every time fuzzed packets are injected, or only sometimes?

API Documentation

API Documentation

March 6, 2013

Contents

1 Module DataManage

1.1 Variables

Name	Description
__package__	**Value:** `None`

1.2 Class DataManage

This class will do the data Management for the CAN analysis. This includes loading data up to the MYSQL database, downloading data from the MYSQL database, converting data to pcap, loading in packets.

To Do:

- change the way to upload data to run bulk inserts and not item by item as it is now.
- Allow for the use of both hexidecimal and decimal numbers for all methods and provide better integration

1.2.1 Methods

__init__(*self*, *host*, *db*, *username*, *password*, *table*, *dataLocation*)

Constructor method.

Parameters

`table:`	SQL table to add data to *(type=string)*
`host:`	Host for MYSQL table *(type=string)*
`username:`	MYSQL username *(type=string)*
`password:`	MYSQL username password *(type=string)*
`dataLocation:`	path to the folder where data will be saved and loaded from *(type=String)*
`db:`	*(type=String)*

getSQLLocation(*self*)

This method returns the path to folder where all sql queries will be saved

Return Value

path to the folder where sql data is stored. this will be a sub folder of the data location path that was passed into the constructor, `__init__`.

(type=string)

getDataLocation(*self*)

This method returns the path to the main folder where all data will be stored and read from.

Return Value

path to the main folder where data will be saved and loaded from

(type=String)

getInjectedLocation(*self*)

This method returns the path to the sub folder where data that was injected will be read from.

Return Value

Path, relative to dataLocation, give in the constructor `__init__`, where the injection data will be stored

(type=String)

createTable(*self*, *table*)

This method will create a new table in the MYSQL database. There is no error checking so be careful when adding in a new table

Parameters

`table`: Name of the new table in the database. The database information was set in the constructor, `__init__`

(type=String)

changeTable(*self*, *table*)

Changes the dtable that we are going to be reading from and uploading to in the MYSQL database

Parameters

`table`: String that is the name of the table we want to communicate with in the MYSQL database

(type=String)

getTable(*self*)

Returns the name of the table that we are set to upload to.

Return Value

The table we are set to communicate with

(type=String)

addData(*self*, *cmd*)

This method will insert data into the MYSQL database based on the given command and the MYSQL information provided in the constructor, `__init__`. This method is designed for the insertion of data and not for retrieving data. Use the method `getData` to retrieve data from the MYSQL database.

Parameters

`cmd`: MYSQL command that we want executed. This is designed for the insertion of data.

(type=String)

getData(*self*, *cmd*)

This method is designed to grab data from the MYSQL database and return it to the user. It is not designed for insertions. See `getData` for a method to insert data to the database.

Parameters

`cmd`: MYSQL command requesting data

(type=String)

Return Value

SQL data that you requested. The format will be a list of all the rows of data. Each row will be a list of the columns you requested.

(type=List of Lists)

addDataPacket(*self*, *data*, *time*, *error*)

This method will take in a data packet (such as one read from our sniff file) and then upload it to the MYSQL database that is set for the class.

Parameters

`data`: This is a list that is one packet as it read off by GooDFETMCPCAN rxpacket method. It is 14 elements long and each byte is stored to an element as an ASCII character .

(type=List)

`time`: The time stamp of the packet

(type=float)

`error`: An additional boolean to set if there was an error detected with this packet during its parsing.

(type=Boolean)

getCmd(*self*, *packet*, *time*, *error*, *duration*, *filter*, *comment*=`None`)

This method will create a sql insertion command based on the given data and the MYSQL information stored in the class. This is designed for inserting 1 packet.

Parameters

`packet:` This is a dictionary that contains the information for an entire data packet sniffed off of the CAN bus it has already been parsed from the format that it is saved as in the sniff method from GoodFETMCPCANCommuniation, `GoodFETMCPCANCommunication.sniff`. See `parseMessageInt` to parse the message and for information on the packet Dictionary. The keys expected to be contained in this dictionary are as follows sID, length,rtr,error, ide, eID (optional), db0 (optional), ... , db7 (optional). See the parsing method for information on all components

(type=Dictionary)

`time:` Time stamp of the packet

(type=float)

`error:` 1 if there is an error in the packet, 0 otherwise. This is a field in the MYSQL database

(type=Boolean)

`duration:` The length of the observation time during which this packet was sniffed off of the CAN bus.

(type=Float)

`filter:` 1 if there was filtering applied during this experiment, 0 otherwise

(type=Boolean)

`comment:` Comment tag that can be assigned to the field in the MYSQL database

(type=string)

Return Value

SQL command for insertion of the packet to the MYSQL database table set in the class

(type=String)

writeDataCsv(*self*, *data*, *filename*)

This method will write the given data to the given filename as a csv file. This is designed to write packet data to files

Parameters

`data:` A list of the data we wish to write to the csv file. The format is that each element in the list is a considered a row. and then each element in the row is a column in the csv file. This can handle both string and numeric elements in the lists. This is the format that is returned by `opencsv`.

(type=List of Lists)

writePcapUpload(*self*, *filenameUp*, *filenameWriteTo*)

This method will create a pcap file of the data contained in a given csv file. The csv file is assumed to be of the format saved by the sniff method in the GoodFETMCPCANCommunication class, `GoodFETMCPCANCommunication.sniff`.

Parameters

`filenameUp`: path/filename for the csv file that will be read and used for making a pcap file. This assumes that the .csv is included in the input.

(type=String)

`filenameWriteTo`: path/filename for the pcap to be saved to. This assumes that there is the .pcap ending included in the input.

(type=String)

writeToPcap(*self*, *filenameWriteTo*, *data*)

This method will create a pcap formatted file from the data that was supplied.

Parameters

`filenameWriteTo`: path/filename for the pcap to be saved to. This assumes that there is the .pcap ending included in the input.

(type=String)

`data`: A list of the data we wish to write to the csv file. The format is that each element in the list is a considered a row. and then each element in the row is a column in the csv file. This can handle both string and numeric elements in the lists. This is the format that is returned by `opencsv`.

(type=List of Lists)

testSQLPCAP(*self*)

This is an internal test method

writetoPcapfromSQL(*self*, *filenameWriteto*, *results*)

This method will create a pcap formatted file from the data that was supplied.

Parameters

`filenameWriteto`: path/filename for the pcap to be saved to. This assumes that there is the .pcap ending included in the input.

(type=String)

`results`: A list of the data we wish to write to the csv file. The format is that each element in the list is a considered a row. and then each element in the row is a column in the csv file. This can handle both string and numeric elements in the lists. This is the format that is returned by a MYSQL query.

(type=List of Lists)

parseMessage(*self*, *data*)

This method will parse a row of the data packets that is of the form of the packet as returned by the GoodFETMCPCAN rxpacket method will simply convert each databyte to integers and then call `parseMessageInt`. See this method for more information.

Parameters

`data:` This is a list that is one packet as returned by GoodFETMCPCAN rxpacket method. It is 13 bytes long and ASCII value.

(type=List)

Return Value

This is a dictionary that contains the information for an entire data packet sniffed off of the CAN bus it has already been parsed from the format that it is saved as in the sniff method from GoodFETMCPCANCommuniation. See `parseMessageInt` to parse the message and for information on the packet Dictionary. The keys expected to be contained in this dictionary are as follows sID, length,rtr,error, ide, eID (optional), db0 (optional), ... , db7 (optional).

1. sID: standard ID of the packet
2. length: length of the data packet (number of data bytes). This will be 0 for an Remote Transmission Request frame
3. rtr: boolean that is 1 if the message is a Remote Transmission Request, 0 otherwise
4. ide: Boolean that is 1 if the message has an extended id. 0 otherwise
5. eID: Extended ID. Only included if one exists
6. db0: Databyte 0. Only included if one exists

   —

7. db7: Databyte 7. Only included if one exists

(type=Dictionary)

parseMessageInt(*self*, *data*)

Parameters

`data:` This is a list that is one packet as it read off by `opencsv` method. The Elements will have been convereted to integers.

(type=List)

Return Value

This is a dictionary that contains the information for an entire data packet sniffed off of the CAN bus it has already been parsed from the format that it is saved as in the sniff method from GoodFETMCPCANCommuniation. See `parseMessageInt` to parse the message and for information on the packet Dictionary. The keys expected to be contained in this dictionary are as follows sID, length,rtr,error, ide, eID (optional), db0 (optional), ... , db7 (optional).

1. sID: standard ID of the packet
2. length: length of the data packet (number of data bytes). This will be 0 for an Remote Transmission Request frame
3. rtr: boolean that is 1 if the message is a Remote Transmission Request, 0 otherwise
4. ide: Boolean that is 1 if the message has an extended id. 0 otherwise
5. eID: Extended ID. Only included if one exists
6. db0: Databyte 0. Only included if one exists

   —

7. db7: Databyte 7. Only included if one exists

(type=Dictionary)

packet2str(*self*, *packet*)

Converts the packet, in the form of the datapacket provided by GoodFETMCPCAN rxpacket and each element is a string ASCII character corresponding to the Hex number of the databyte. This will convert the list to a string of the hex bytes.

Parameters

`packet:` This is a list that is one packet as it read off by the GoodFETMCPCANComunication class. It is 13 bytes long and string character is in each element.

(type=List)

Return Value

String that is the data packet printed out where each byte has been converted to the hex characters.

(type=String)

uploadData(*self*, *filename*)

This method will upload all the data contained in the given file path to the MYSQL database.

Parameters

`filename:` This is the path to the file that the user wishes to upload. It is assumed to contain the .csv ending and be a file in the format of the data that is saved by GoodFETMCPCANCommunication sniff method.

(type=String)

opencsv(*self*, *fileObj*)

This method will load packet data from a csv file. The format of the data will be of the form of the data that was saved by the GoodFETMCPCANCommunication sniff method. See `GoodFETMCPCANCommunication.sniff` for more information on the meaning of the databytes. The elements in each row of the csv file will be parsed as if they are the following types:

1. float
2. string
3. integer 4 Hex value (i.e. 'ff' or 'af')

This will ignore any lines in the csv document that begin with # in the first column. This allows for comments to be added.

Parameters

`fileObj:` This is the file object (i.e. what is returned by the open command) of the csv document that is to be read.

(type=Object of the file)

Return Value

The csv file parsed by rows and columns. The format will be that each element in the return list will be one row of the csv file. All entries will be stored as a number except column 2 which will be stored as a string (see above).

(type=List of lists)

readWriteFileHex(*self*, *filename*)

This method will be used for reading a file for of packets to be written to the the CAN bus. The format of the csv layout is expected to be as follows with each element stored as a hex except for the time delay:

- col 0: delay time from previous row (0 if no delay)
- col 1: Standard ID
- col 2: Data Length (0-8)
- col 3: db 0 —
- col 7: db7 (as needed, based on data length)

Parameters

`filename:` path/filenae to the csv file to be uploaded. This is expected to include the .csv ending. The format of the rows is described above.

(type=String)

Return Value

This will simply turn the csv file to a list of lists where the elements of the outer list are rows in the csv file. Each row is a list of all the columns. The time delay will be converted to a float while all other columns are converted to an integer.

(type=List of Lists)

readWriteFileDEC(*self*, *filename*)

This method will be used for reading a file for of packets to be written to the the CAN bus. The format of the csv layout is expected to be as follows with each element stored as an integer value except for the time delay which is expected as a float:

- col 0: delay time from previous row (0 if no delay)
- col 1: Standard ID
- col 2: Data Length (0-8)
- col 3: db 0 —
- col 7: db7 (as needed, based on data length)

Additionally, this can now take in the packets that the sniff method writes out, `GoodFETMCPCANCommunication.sniff`. It will detect either the header that is added to these files or that the length is longer than those written by the fuzz methods.

Parameters

`filename:` path/filenae to the csv file to be uploaded. This is expected to include the .csv ending. The format of the rows is described above.

(type=String)

Return Value

This will simply turn the csv file to a list of lists where the elements of the outer list are rows in the csv file. Each row is a list of all the columns. The time delay will be converted to a float while all other columns are converted to an integer.

(type=List of Lists)

readInjectedFileDEC(*self*, *filename*, *startTime*=`None`, *endTime*=`None`, *id*=`None`)

This method will read packets from a file that was saved from packets that were injected onto the bus. An example of a method that does this would be the experiments method `experiments.generationFuzzer`. The file is assumed to be a csv file with the .csv included matching this format. Any lines that begin with a # in the first column will be ignored. The user can specify start and end times to only get a subset of the packet. If no startTime is provided then it will return packets starting from the begining up until the endTime. If no endTime is provided then it will return all packets in the file after the startTime. An id can be provided and it will return only the packets with the given id.

When parsing the data it will assume the data is of the following form where the first column is parsed as a float and all subsequent are parsed as a decimal ineteger:

1. time of injection
2. standard Id
3. 8 (data length) @todo: extend to varied length
4. db0

   —
5. db7

Parameters

`filename:` path/filenae to the csv file to be uploaded. This is expected to include the .csv ending. The format of the rows is described above.

(type=String)

`startTime:` This is a timestamp of the start time for the earliest time for packets you want. The method will not return any packets that were injected before this start time. This input is optional.

(type=timestamp)

`endTime:` This is the timestamp of the latest inject time for packets you want. The method will return no packets with an endTime after this timestamp. This input is optional.

(type=timestamp)

`id:` This is an optional parameter that allows you to positively filter for the id that you want of packets. Only packets with the given id will be returned.

(type=Integer)

Return Value

This imports the data on the csv file. Each element in the list will correspond to a row. each element in the row will correspond to the column. The types of the column will be a float for the first columna and integer for all subsequent columns.

(type=List of Lists)

uploadFiles(*self*)

This method will upload all data files from sniffing experiments that have been saved. The sniff would be called by `GoodFETMCPCANCommunication.sniff` and it will be all .csv files saved in the self.DATALOCATION which is input by the user when the class is initiated.

This method will find all .csv files in the folder and attemp to upload them all. It will call the `uploadData` method on all the filenames. Once a file has been uploaded (or attempted to) the file will be moved to a subfolder that is named by todays date in the following format: YYYYMMDD. If the folder does not exist, it will be created. The filename will then be changed by adding "_Uploaded" to the end of the csv. To ensure that no files are overwritten, the program will ensure that the file does not exist. if one does of the same filename it will append "_i" where i is the first integer that does not have a conflicting filename. This allows the user to upload multiple times during the same day without loosing the origional data files.

saveJson(*self*, *filename*, *data*)

This method will dump a json data structure to the filename given.

Parameters

`filename:` path to the .json file that the data will be saved to
(type=String)

`data:` Data to be saved to a json file
(type=json data)

loadJson(*self*, *filename*)

This method will load a json file into memory.

Parameters

`filename:` path to the .json file that is to be loaded into memory
(type=String)

1.2.2 Instance Variables

Name	Description
DATALOCATION	Location of main data folder
SQLDDATALOCATION	Location where MYSQL data will be stored
INJECTDATALOCATION	Location where injection data will be stored
MIN_TIME_DELAY	This is the minimum time between 2 packets that we will consider there to be a delay between injection of the packets.
MAX_TIME_DELAY	This is the maximum time between 2 packets that we will consider as a time delay between the two packets

2 Module FordExperiments

2.1 Variables

Name	Description
__package__	**Value:** `None`

2.2 Class FordExperiments

GoodFETMCPCANCommunication.GoodFETMCPCANCommunication
└ experiments.experiments
└ **FordExperiments.FordExperiments**

This class is a subclass of experiments and is a car specific module for demonstrating and testing hacks.

2.2.1 Methods

__init__(*self*, *dataLocation*=`'../../contrib/ThayerData/'`)

Constructor

Parameters

`data_location`: path to the folder where data will be stored

Overrides: GoodFETMCPCANCommunication.GoodFETMCPCANCommunication.__init__

mimic1056(*self*, *packetData*, *runTime*)

cycledb1_1056(*self*, *runTime*)

getBackground(*self*, *sId*)

This method gets the background packets for the given id. This is a simple "background" retriever in that it returns the packet that is of the given id that was sniffed off the bus.

cycle4packets1279(*self*)

oscillateMPH(*self*, *runTime*)

oscillateTemperature(*self*, *runTime*)

fakeVIN(*self*)

setScanToolTemp(*self*, *temp*)

setEngineTemp(*self*, *temp*)

overHeatEngine(*self*)

runOdometer(*self*)

setDashboardTemp(*self*, *temp*)

warningLightsOn(*self*, *checkEngine*, *checkTransmission*, *transmissionOverheated*, *engineLight*, *battery*, *fuelCap*, *checkBreakSystem*, *ABSLight*, *dashB*)

fakeScanToolFuelLevel(*self*, *level*)

fakeOutsideTemp(*self*, *level*)

fakeAbsTps(*self*, *level*)

mphToByteValue(*self*, *mph*)

ByteValuToMph(*self*, *value*)

setMPH(*self*, *mph*)

speedometerHack(*self*, *inputs*)

rpmToByteValue(*self*, *rpm*)

ValueTorpm(*self*, *value*)

setRPM(*self*, *rpm*)

rpmHack(*self*, *inputs*)

This method will increase the rpm by the given rpm amount.

Parameters

`inputs`: Single element of a list that corresponds to the amount the user wishses to

(type=List)

imbeethovenbitch(*self*)

engineDiagnostic(*self*, *data*)

Inherited from experiments.experiments(Section 5.2)

filterStdSweep(), generalFuzz(), generationFuzzer(), packetRespond(), rtrSweep(), sweepRandom()

Inherited from GoodFETMCPCANCommunication.GoodFETMCPCANCommunication(S 4.2)

addFilter(), filterForPacket(), freqtest(), isniff(), multiPacketSpit(), multiPacketTest(), printInfo(), reset(), setRate(), sniff(), sniffTest(), spit(), spitSetup(), spitSingle(), test(), writeData()

2.2.2 Instance Variables

Name	Description
Inherited from GoodFETMCPCANCommunication.GoodFETMCPCANCommunication (Section 4.2)	
INJECT_DATA_LOCATION	

3 Module GoodFETMCPCAN

3.1 Variables

Name	Description
__package__	**Value:** `None`

3.2 Class GoodFETMCPCAN

GoodFET.GoodFET
GoodFETSPI.GoodFETSPI
GoodFETMCPCAN.GoodFETMCPCAN

This class uses the regular SPI app to implement a CAN Bus adapter on the Goodthopter10 hardware.

3.2.1 Methods

MCPsetup(*self*)

Sets up the ports.

MCPreset(*self*)

Reset the MCP2515 chip.

MCPsetrate(*self*, *rate*=125)

Sets the data rate in kHz.

MCPcanstat(*self*)

Get the CAN Status.

MCPcanstatstr(*self*)

Read the present status as a string.

MCPcanstatint(*self*)

Read present status as an int.

MCPreqstat(*self*, *state*)

Set the CAN state.

MCPreqstatNormal(*self*)

Set the CAN state.

MCPreqstatSleep(*self*)

Set the CAN state.

MCPreqstatLoopback(*self*)

Set the CAN state.

MCPreqstatListenOnly(*self*)

Set the CAN state.

MCPreqstatConfiguration(*self*)

Set the CAN state.

MCPrxstatus(*self*)

Reads the RX Status by the SPI verb of the same name.

MCPreadstatus(*self*)

Reads the Read Status by the SPI verb of the same name.

readrxbuffer(*self*, *packbuf*=`0`)

Reads the RX buffer. Might have old data.

fastrxpacket(*self*)

rxpacket(*self*)

Reads the next incoming packet from either buffer. Returns None immediately if no packet is waiting.

MCPrts(*self*, *TXB0*=`False`, *TXB1*=`False`, *TXB2*=`False`)

Requests to send one of the transmit buffers.

writetxbuffer(*self*, *packet*, *packbuf*=0)

Writes the transmit buffer.

txpacket(*self*, *packet*)

Transmits a packet through one of the outbound buffers. As usual, the packet should begin with SIDH. For now, only TXB0 is supported.

simpleParse(*self*, *packet*)

This will print a simple parsing of the data with the standard Id and the extended id parsed.

packet2str(*self*, *packet*)

Converts a packet from the internal format to a string.

packet2parsed(*self*, *data*)

This method will parse the packet that was received via `rxpacket`.

Parameters

`data`: data packet read off of the MCP2515 from the CAN bus. format will be 14 bytes where each element is a character whose unicode integer value corresponds to the hex value of the byte (use the ord() method).

(type=List)

Return Value

Dictionary of the packet parsed into various components. The key values will be as follows

1. ide : 1 if the message is an extended frame. 0 otherwise
2. eID : extended ID. Not included if not an extended frame
3. rtr : 1 if the message is a remote transmission request (RTR)
4. sID : Standard ID.
5. length: packet length (between 0 and 8)
6. db0 : Data byte 0 (not included if RTR message)

   —
7. db7 : Data byte 7 (not included if RTR message)

(type=Dictionary)

packet2parsedstr(*self*, *data*)

This will return a string that is the parsed CAN message. The bytes will be parsed for the standard ID, extended ID (if one is present), rtr, length and databytes (if present). The message will be placed into a string as decimal integers not hex values. This method calls `packet2parsed` to do the packet parsing.

Parameters

`data`: Data packet as returned by the `rxpacket`

(type=List)

Return Value

String that shows the data message in decimal format, parsed.

(type=String)

peek8(*self*, *adr*)

Read a byte from the given address. Untested.

Overrides: GoodFET.GoodFET.peek8

poke8(*self*, *adr*, *val*)

Poke a value into RAM. Untested

MCPbitmodify(*self*, *adr*, *mask*, *data*)

Writes a byte with a mask. Doesn't work for many registers.

Inherited from GoodFETSPI.GoodFETSPI

SPIsetup(), SPItrans(), SPItrans8()

Inherited from GoodFET.GoodFET

MONpeek16(), MONpeek8(), MONpoke16(), __init__(), bslResetZ1(), btInit(), call(), dir(), dump(), dumpmem(), eeprompeek(), erase(), execute(), findbaud(), flash(), getConsole(), getbuffer(), getpc(), getsecret(), glitchApp(), glitchRate(), glitchTime(), glitchVerb(), glitchVoltages(), glitchstart(), glitchstarttime(), halt(), infostring(), loadsymbols(), lock(), mon_connected(), monitor_info(), monitor_list_apps(), monitor_ram_depth(), monitor_ram_pattern(), monitorclocking(), monitorecho(), monitorgetclock(), monitorsetclock(), monitortest(), name2adr(), out(), peek(), peek16(), peek32(), peekblock(), peekbysym(), picROMclock(), picROMfastclock(), poke16(), pokeblock(), pokebysym(), pokebyte(), pyserInit(), readbyte(), readcmd(), readpicROM(), resume(), serClose(), serInit(), setBaud(), setsecret(), setup(), silent(), start(), status(), telosBReset(), telosI2CStart(), telosI2CStop(), telosI2CWriteBit(), telosI2CWriteByte(), telosI2CWriteCmd(), telosSetSCL(), telosSetSDA(), test(),

testleds(), timeout(), writecmd(), writepicROM()

3.2.2 Class Variables

Name	Description
MCPMODES	**Value:** `['Normal', 'Sleep', 'Loopback', 'Listen-only', 'Configura...`
MCPrates	**Value:** `[10.4, 41.6, 83.3, 100, 125, 250, 500, 1000]`
Inherited from GoodFETSPI.GoodFETSPI APP	
Inherited from GoodFET.GoodFET GLITCHAPP, MONITORAPP, app, baudrates, besilent, connected, count, data, symbols, verb, verbose	

4 Module GoodFETMCPCANCommunication

4.1 Variables

Name	Description
__package__	**Value:** `None`

4.2 Class GoodFETMCPCANCommunication

Known Subclasses: experiments.experiments

4.2.1 Methods

__init__(*self*, *dataLocation*=`'../../contrib/ThayerData/'`)

printInfo(*self*)

This method will print information about the board to the terminal. It is good for diagnostics.

reset(*self*)

Reset the chip

sniff(*self*, *freq*, *duration*, *description*, *verbose*=`True`, *comment*=`None`, *filename*=`None`, *standardid*=`None`, *debug*=`False`, *faster*=`False`, *parsed*=`True`, *data*=`None`, *writeToFile*=`True`, *db0*=`None`, *db1*=`None`)

sniffTest(*self*, *freq*)

This method will preform a test to see if we can sniff corretly formed packets from the CAN bus.

Parameters

`freq`: frequency of the CAN bus

(type=number)

freqtest(*self*, *freq*)

This method will test the frequency provided to see if it is the correct frequency for this CAN bus.

Parameters

`freq`: The frequency to listen to the CAN bus.

(type=Number)

isniff(*self*, *freq*)

An intelligent sniffer, decodes message format

test(*self*)

This will perform a test on the GOODTHOPTER10. Diagnostic messages will be printed out to the terminal

addFilter(*self*, *standardid*, *verbose*=`True`)

This method will configure filters on the board. Filters are positive filters meaning that they will only store messages that match the ids provided in the list of standardid. Since there are 2 buffers and due to the configuration of how the filtering works (see MCP2515 documentation), at least 3 filters must be set to guarentee you do not get any unwanted messages. However even with only 1 filter set you should get all messages from that ID but the other buffer will store any additional messages.

Parameters

`standardid`: List of standard ids that need to be set. There can be at most 6 filters set.

(type=list of integers)

`verbose`: If true it will print out messages and diagnostics to terminal.

(type=Boolean)

Return Value

This method does not return anything

(type=None)

To Do: rename setFilters

filterForPacket(*self*, *standardid*, *DB0*, *DB1*, *verbose*=`True`)

This method will configure filters on the board to listen for a specific packet originating from standardid with data bytes 0 and 1. It will configure all six filters, so you will not receive any other packets.

Parameters

`standardid:` standardID to listen for

(type=integer)

`standardid:` DB0 contents to filter for

(type=integer)

`standardid:` DB1 contents to filter for

(type=integer)

`verbose:` If true it will print out messages and diagnostics to terminal.

(type=Boolean)

`DB1:` *(type=integer)*

`DB0:` *(type=integer)*

Return Value

This method does not return anything

(type=None)

multiPacketTest(*self*)

multiPacketSpit(*self*, *packet0*=`None`, *packet1*=`None`, *packet2*=`None`, *packet0rts*=`False`, *packet1rts*=`False`, *packet2rts*=`False`)

This method writes packets to the chip's TX buffers and/or sends the contents of the buffers onto the bus.

Parameters

`packet0`: A list of 13 integers of the format [SIDhigh SIDlow 0 0 DLC DB0-7] to be loaded into TXBF0

(type=list of integer)

`packet1`: A list of 13 integers of the format [SIDhigh SIDlow 0 0 DLC DB0-7] to be loaded into TXBF1

(type=list of integer)

`packet2`: A list of 13 integers of the format [SIDhigh SIDlow 0 0 DLC DB0-7] to be loaded into TXBF2

(type=list of integer)

`packet0rts`: If true the message in TX buffer 0 will be sent

(type=Boolean)

`packet0rts`: If true the message in TX buffer 1 will be sent

(type=Boolean)

`packet0rts`: If true the message in TX buffer 2 will be sent

(type=Boolean)

`packet2rts`: *(type=Boolean)*

spitSetup(*self*, *freq*)

This method sets up the chip for transmitting messages, but does not transmit anything itself.

spitSingle(*self*, *freq*, *standardid*, *repeat*, *writes*, *period*=`None`, *debug*=`False`, *packet*=`None`)

This method will spit a single message onto the bus. If there is no packet information provided then the message will be sent as a remote transmission request (RTR). The packet length is assumed to be 8 bytes The message can be repeated given number of times with a gap of period (milliseconds) between each message. This will continue for the the number of times specified in the writes input. This method will setup the bus and call the spit method, `spit`. This method includes a bus reset and initialization.

Parameters

`freq`: The frequency of the bus

(type=number)

`standardid`: This is a single length list with one integer elment that corresponds to the standard id you wish to write to

(type=list of integer)

`repeat`: If true the message will be repeatedly injected. if not the message will only be injected 1 time

(type=Boolean)

`writes`: Number of writes of the packet

(type=Integer)

`period`: Time delay between injections of the packet in Milliseconds

(type=Integer)

`debug`: When true debug status messages will be printed to the terminal

(type=Boolean)

`packet`: Contains the data bytes for the packet which is assumed to be of length 8. Each byte is stored as an integer and can range from 0 to 255 (8 bits). If packet == None then an RTR will be sent on the given standard id.

(type=List)

spit(*self*, *freq*, *standardid*, *repeat*, *writes*, *period*=`None`, *debug*=`False`, *packet*=`None`)

This method will spit a single message onto the bus. If there is no packet information provided then the message will be sent as a remote transmission request (RTR). The packet length is assumed to be 8 bytes The message can be repeated a given number of times with a gap of period (milliseconds) between each message. This will continue for the the number of times specified in the writes input. This method does not include bus setup, it must be done before the method call.

Parameters

`freq`: The frequency of the bus

(type=number)

`standardid`: This is a single length list with one integer element that corresponds to the standard id you wish to write to

(type=list of integer)

`repeat`: If true the message will be repeatedly injected. if not the message will only be injected 1 time

(type=Boolean)

`writes`: Number of writes of the packet

(type=Integer)

`period`: Time delay between injections of the packet in Milliseconds

(type=Integer)

`debug`: When true debug status messages will be printed to the terminal

(type=Boolean)

`packet`: Contains the data bytes for the packet which is assumed to be of length 8. Each byte is stored as an integer and can range from 0 to 255 (8 bits). If packet == None then an RTR will be sent on the given standard id.

(type=List)

setRate(*self*, *freq*)

This method will reset the frequency that the MCP2515 expects the CAN bus to be on.

Parameters

`freq`: Frequency of the CAN bus

(type=Number)

writeData(*self*, *packets*, *freq*)

This method will write a list of packets to the bus at the given frequency. This method assumes a packet length of 8 for all packets as well as a standard id.

Parameters

`packets`: The list of packets to be injected into the bus. Each element of packets is a list that is a packet to be injected onto the bus. These packets are assumed to be in the following format:

```
row[0] = time delay relative to the last packet.
         if 0 or empty there will be no delay
row[1] = Standard ID (integer)
row[2] = Data Length (0-8)
          (if it is zero we assume an Remote Transmit Request)
row[3] = Data Byte 0
row[4] = Data Byte 1
...
row[10] = Data Byte 7
```

(type=List of Lists)

`freq`: Frequency of the CAN bus

(type=number)

4.2.2 Instance Variables

Name	Description
INJECT_DATA_LOCATION	stores the sub folder path where injected data will be stored

5 Module experiments

5.1 Variables

Name	Description
__package__	**Value:** `None`

5.2 Class experiments

GoodFETMCPCANCommunication.GoodFETMCPCANCommunication ┐
experiments.experiments

Known Subclasses: FordExperiments.FordExperiments

This class provides methods for reverse-engineering the protocols on the CAN bus network via the GOODTHOPTER10 board, *http://goodfet.sourceforge.net/hardware/goodthopter10/*

5.2.1 Methods

__init__(*self*, *data_location*=`'../../contrib/ThayerData/'`)

Constructor

Parameters

`data_location`: path to the folder where data will be stored
(type=string)

Overrides: GoodFETMCPCANCommunication.GoodFETMCPCANCommunication.__init__

filterStdSweep(*self*, *freq*, *low*, *high*, *time*=5)

This method will sweep through the range of standard ids given from low to high. This will actively filter for 6 ids at a time and sniff for the given amount of time in seconds. If at least one message is read in then it will go individually through the 6 ids and sniff only for that id for the given amount of time. This does not save any sniffed packets.

Parameters

`freq`: The frequency at which the bus is communicating

(type=number)

`low`: The low end of the id sweep

(type=integer)

`high`: The high end of the id sweep

(type=integer)

`time`: Sniff time for each trial. Default is 5 seconds

(type=number)

Return Value

A list of all IDs found during the sweep.

(type=list of numbers)

sweepRandom(*self*, *freq*, *number*=5, *time*=5)

This method will choose random values to listen out of all the possible standard ids up to the given number. It will sniff for the given amount of time on each set of ids on the given frequency. Sniffs in groups of 6 but when at least one message is read in it will go through all six individually before continuing. This does not save any sniffed packets.

Parameters

`freq:` The frequency at which the bus is communicating

(type=number)

`number:` High end of the possible ids. This will define a range from 0 to number that the ids will be chosen from

(type=integer)

`time:` Sniff time for each trial. Default is 5 seconds

(type=number)

Return Value

A list of all IDs found during the sweep and a list of all the IDs that were listened for throughout the test

(type=list of numbers, list of numbers)

rtrSweep(*self*, *freq*, *lowID*, *highID*, *attempts*=1, *duration*=1, *verbose*=True)

This method will sweep through the range of ids given by lowID to highID and send a remote transmissions request (RTR) to each id and then listen for a response. The RTR will be repeated in the given number of attempts and will sniff for the given duration continuing to the next id.

Any messages that are sniffed will be saved to a csv file. The filename will be stored in the DATA_LOCATION folder with a filename that is the date (YYYYMMDD)_rtr.csv. If the file already exists it will append to the end of the file The format will follow that of `GoodFETMCPCANCommunication.sniff` in that the columns will be as follows:

```
1. timestamp:     as floating point number
2. error boolean: 1 if there was an error detected of packet formatting (not exha
3. comment tag:   comment about experiments as String
4. duration:      Length of overall sniff
5. filtering:     1 if there was filtering. 0 otherwise
6. db0:           Integer

   ---
7. db7:           Integer
```

Parameters

`freq`: The frequency at which the bus is communicating

(type=number)

`lowID`: The low end of the id sweep

(type=integer)

`highID`: The high end of the id sweep

(type=integer)

`attempts`: The number of times a RTR will be repeated for a given standard id

(type=integer)

`duration`: The length of time that it will listen to the bus after sending an RTR

(type=integer)

`verbose`: When true, messages will be printed out to the terminal

(type=boolean)

Return Value

Does not return anything

(type=None)

generationFuzzer(*self*, *freq*, *standardIDs*, *dbLimits*, *period*, *writesPerFuzz*, *Fuzzes*)

This method will perform generation based fuzzing on the bus. The method will inject properly formatted, randomly generated messages at a given period for a *writesPerFuzz* number of times. The packets that are injected into the bus will all be saved in the following path DATALOCATION/InjectedData/(today's date (YYYYMMDD))_GenerationFuzzedPackets.csv. An example filename would be 20130222_GenerationFuzzedPackets.csv Where DATALOCATION is provided when the class is initiated. The data will be saved as integers. Each row will be formatted in the following form:

```
row = [time of injection, standardID, 8, db0, db1, db2, db3, db4, db5, c
```

Parameters

`freq:` The frequency at which the bus is communicating

(type=number)

`standardIDs:` List of standard IDs the user wishes to fuzz on. An ID will randomly be chosen with every new random packet generated. If only 1 ID is input in the list then it will only fuzz on that one ID.

(type=list of integers)

`dbLimits:` This is a dictionary that holds the limits of each bytes values. Each value in the dictionary will be a list containing the lowest possible value for the byte and the highest possible value. The form is shown below:

```
dbLimits['db0'] = [low, high]
dbLimits['db1'] = [low, high]
...
dbLimits['db7'] = [low, high]
```

(type=dictionary)

`period:` The time gap between packet inejctions given in milliseconds

(type=number)

`writesPerFuzz:` This will be the number of times that each randomly generated packet will be injected onto the bus before a new packet is generated

(type=integer)

`Fuzzes:` The number of packets to be generated and injected onto bus

(type=integer)

Return Value

This method does not return anything

(type=None)

generalFuzz(*self*, *freq*, *Fuzzes*, *period*, *writesPerFuzz*)

The method will inject properly formatted, randomly generated messages at a given period for a *writesPerFuzz* number of times. A new random standard id will be chosen with each newly generated packet. IDs will be chosen from the full range of potential ids ranging from 0 to 4095. The packets that are injected into the bus will all be saved in the following path DATALOCATION/InjectedData/(today's date (YYYYMMDD))_GenerationFuzzedPackets.csv. An example filename would be 20130222_GenerationFuzzedPackets.csv Where DATALOCATION is provided when the class is initiated. The data will be saved as integers. Each row will be formatted in the following form:

```
row = [time of injection, standardID, 8, db0, db1, db2, db3, db4, db5, c
```

Parameters

`freq:` The frequency at which the bus is communicating

(type=number)

`period:` The time gap between packet inejctions given in milliseconds

(type=number)

`writesPerFuzz:` This will be the number of times that each randomly generated packet will be injected onto the bus before a new packet is generated

(type=integer)

`Fuzzes:` The number of packets to be generated and injected onto bus

(type=integer)

Return Value

This method does not return anything

(type=None)

packetRespond(*self*, *freq*, *time*, *repeats*, *period*, *responseID*, *respondPacket*, *listenID*, *listenPacket*=`None`)

This method will allow the user to listen for a specific packet and then respond with a given message. If no listening packet is included then the method will only listen for the id and respond with the specified packet when it receives a message from that id. This process will continue for the given amount of time (in seconds). and with each message received that matches the listenPacket and ID the transmit message will be sent the *repeats* number of times at the specified *period*. This message assumes a packet length of 8 for both messages, although the listenPacket can be None

Parameters

`freq`: Frequency of the CAN bus

(type=number)

`time`: Length of time to perform the packet listen/response in seconds.

(type=number)

`repeats`: The number of times the response packet will be injected onto the bus after the listening criteria has been met.

(type=Integer)

`period`: The time interval between messages being injected onto the CAN bus. This will be specified in milliseconds

(type=number)

`responseID`: The standard ID of the message that we want to inject

(type=Integer)

`respondPacket`: The data we wish to inject into the bus. In the format where respondPacket[0] = databyte 0 ... respondPacket[7] = databyte 7 This assumes a packet length of 8.

(type=List of integers)

`listenID`: The standard ID of the messages that we are listening for. When we read the correct message from this ID off of the bus, the method will begin re-injecting the responsePacket on the responseID

(type=Integer)

`listenPacket`: The data we wish to listen for before we inject packets. This will be a list of the databytes, stored as integers such that listenPacket[0] = data byte 0, ..., listenPacket[7] = databyte 7. This assumes a packet length of 8. This input can be None and this will lead to the program only listening for the standardID and injecting the response as soon as

Inherited from GoodFETMCPCANCommunication.GoodFETMCPCANCommunication(S 4.2)

addFilter(), filterForPacket(), freqtest(), isniff(), multiPacketSpit(), multiPacketTest(), printInfo(), reset(), setRate(), sniff(), sniffTest(), spit(), spitSetup(), spitSingle(), test(), writeData()

5.2.2 Instance Variables

Name	Description
Inherited from GoodFETMCPCANCommunication.GoodFETMCPCANCommunication (Section 4.2)	
INJECT_DATA_LOCATION	

6 Module mainDisplay

6.1 Variables

Name	Description
__package__	**Value:** `None`

6.2 Class DisplayApp

This is the main display for the graphical user interface (GUI). This GUI is designed to aid the user in their work listening to CAN traffic via the GOODTHOPTER10 board, *http://goodfet.sourceforge.net/hardware/goodthopter10/*. There are no inputs to this class but all default data is loaded from the settings file.

6.2.1 Methods

__init__(*self*)

writeiniFile(*self*, *filename*, *section*, *option*, *value*)

Writes the given settings to the given settings filename. If the section does not exist in the settings file then it will be created. The file is assumed to be a .ini file. This method is a modified version of the one found on the following website:
http://bytes.com/topic/python/answers/627791-writing-file-using-configparser

Parameters

`filename:` path to the settings file
(type=string)

`section:` section heading in the settings file
(type=string)

`option:` string

`option:` The option in the given section in the settings file that will be set

`value:` The value of the option we are saving.

ConfigSectionMap(*self*, *Config*, *section*)

This method has been implemented based on the following exmaple, *http://wiki.python.org/moin/ConfigParserExamples*.

Parameters

`Config:` ConfigParser instance that has already read the given settings filename.

`section:` Section that you want to get all of the elements of from the settings file that has been read by the Config parser and passed in as Config.

(type=string)

Return Value

Dictionary where they keys are the options in the given section and the values are the corresponding settings value.

(type=Dictionary)

buildMenus(*self*)

This method will build the menu bars

selectall(*self*, *event*)

This method is called when the user wishes to select the entire text box.

buildExperimentCanvas(*self*)

This method will build out the experiment frame which will display to the user the various experiments that can be run. This will be a list of experiments with input values and is kept inside the `self.RightSideCanvas` and is one of the tabs that will be created but buried from view until the user selects the experiments button

buildInfoFrame(*self*)

This builds the tab that displays our information about the arbitration ids and any known information about the packets at the moment. This will itself have 3 sub tabs: General Information, Bytes, Packets. These are rebuilt for each arbitration id chosen.

buttonTest(*self*)

deleteArbID(*self*)

This method will delete an ID from the JSON file that stores packet information. The user will first be prompted to ask if they really want to. If they say yes then the another dialog box will open for the user to input the id they want to delete. This will delete the ID, save the new JSON document, and update the ID drop down menu.

addArbID(*self*)

This method will add a new arbitration id to the json file. It will first prompt the user as to what id it wants to add. Then the new database entry will be created, added to the structure, saved and updated on the GUI.

buildByteInfoFrame(*self*, *i*)

This method will build the frame that will display our information about the specific bytes of an arbitration ID.

Parameters

`i`: This is the row to add our Frame to. This is for the grid formation

(type=Integer)

buildPacketInfoFrame(*self*, *i*)

This method will build the frame that will display our information about the specific packets of an arbitration ID. These are packets that we know what they are telling us or broadcasting to the bus. Each packet description will be click-able. This will fill the write options under the sniff tab and then change focus to this tab. This will allow for easy injection of these packets.

Parameters

`i`: This is the row to add our Frame to. This is for the grid formation

(type=Integer)

buildGeneralInfoFrame(*self*, *i*)

This method will build the frame that will display our information about our general knowledge of an arbitration ID.

Parameters

`i`: This is the row to add our Frame to. This is for the grid formation

(type=Integer)

liftGeneralInfo(*self*)

This method lifts the General information tab to the top and makes it viewable to the screen.

liftBytesInfo(*self*)

This method lifts the Bytes information tab to the top and makes it viewable to the screen.

liftPackets(*self*)

This method lifts the Packets information tab to the top and makes it viewable to the screen.

updateQueryInfo(*self*, *name*, *index*, *mode*)

This method will update the query tab so that it is correct and so that the user does not try to save a .pcap file to .csv or vice versa. It will also update the beginning of the MYSQL command for the .pcap. This is because of how the write to .pcap method is written. See `DataManage.writetoPcapfromSQL`.

saveJsonInfo(*self*, *event*=`None`)

This file will save all of the user's information to the json file specified in settings. The data will be saved to json data structure in memory and then dumped to the file via the DataManager method, `DataManage.saveJson`.

updateInfo(*self*, *name*=`None`, *index*=`None`, *mode*=`None`)

This method is called when the user changes the option menu for arbitration IDs under our packetInformation tab. This will update these packet information tabs with our knowledge of the new ID. If the set ID does not exist (was set when the user clicked on the id from sniffed data) then a warning dialog will appear.

injectPacket(*self*, *data*)

This method will take the data that is given and fill the write method section in the sniff tab. It will also switch the view to this tab so that the user can easily inject this packet. This method does not inject the packet onto the CAN bus. It simple makes the entry ready for the user to do so.

Parameters

`data`: This is a list that contains the data packet that we want to inject. data[0] contains the arbitration id as a string. data[1] contains the data bytes as a string. The format of the data within the string is assumed to be in decimal form with spaces between the bytes.

(type=List)

buildCarFrame(*self*)

This method will build the car module tab. It will build the frame but will only build the module if one has been set in the settings file. This allows users to create their own module that is car specific but can still be run from the GUI. See `connectBus` for information on the module connection.

buildCarModule(*self*)

This will build the car module. This is specific to a car and the class method to do this can be set by the user. The user's class should take in the frame where it will write all of it's components as well as a reference to the experiment file that must also be specified in the settings file. This experiment file class is a sub class of experiments which is a sub class of GoodFETMCPCANCommunication.

buildSQLCanvas(*self*)

This method will build the SQL tab frame where some of the database information is stored. It will contain the upload to db method as well as a way to download data from the database and convert to .pcap

buildCanvas(*self*)

This builds the Frame that controls the sniffing and writing with the CAN bus. It is contained within the `self.RightSideCanvas` and is in the same grid layout as the SQL, EXPERIMENTS tabs.

buildDataCanvas(*self*)

This methods will build the left frame that contains a display for information pulled off of the bus.

addtextToScreen(*self*, *text*)

buildControls(*self*)

This method builds out the top frame bar which allows the user to switch tabs between the experiments, sniff/write, MYSQL and Arbitration id tabs

setBindings(*self*)

This method will set bindings on the window. This includes mouse clicks and key presses

handleKeys(*self*, *event*)

This will handle an event for when a key is pressed

handleQuit(*self*, *event*=`None`)

This method is called when the user quits the program. It terminates the display and exits

setCarModule(*self*, *experimentFileLocation*, *experimentGUILocation*)

This method will save the new car module file locations that can be set in the settings dialog box. They will be saved to the settings ini file but will not update the current window. The program must be restarted for changes to take effect. A warning will appear to indicate this.

Parameters

`experimentFileLocation:` path to the experiment file that will be part of the car module. This should be a file that contains a class of the same name. The class is assumed to be a sub class of the experiments class.

(type=String)

`experimentGUILocation:` path to the experiment gui module addition. This will be a file that contains a class of the same name. This class will print the module's addition to the GUI for the user to run the experiments included in the experimentFileLocation class.

(type=String)

setDataManage(*self*, *table*, *name*, *host*, *username*, *password*, *database*)

This method will update the stored information for accessing the MYSQL database. The settings will be saved to the settings file.

Parameters

`table:` SQL table to add data to
(type=string)

`name:` Name for SQL account
(type=string)

`host:` Host for MYSQL table
(type=string)

`username:` MYSQL username
(type=string)

`password:` MYSQL username password
(type=string)

`database:` *(type=string)*

handleSettings(*self*, *event*=`None`)

This method will open the settings dialog box for the user to change various components of the GUI.

handleModQ(*self*, *event*)

This method will quit the GUI

setRunning(*self*)

This method sets the running boolean when a method is communicating with the bus

unsetRunning(*self*)

This method unsets the running boolean when a method is done communicating with the bus

getRate(*self*)

This method returns the rate that the GOODTHOPTER10 is set to

testConnect(*self*)

this is to test module files

connectBus(*self*)

This method will try to reconnect with the GOODTHOPTER10. It will first check to make sure that no method is currently communicating with the bus.

checkComm(*self*)

This method check to see if the program is able to begin communication with the GOODTHOPTER10 board. This method should be called before anything begins to try and communicate. It will check first to see if the board is connected and will then check to see if the self.running boolean is set or not.

Return Value

False if the board is either not connected or if there is currently a script communicating with the board. True otherwise

(type=Boolean)

getExperimentFileLocations(*self*)

getDataLocation(*self*)

Returns the path to the data location

Return Value

Data location path

(type=string)

setDataLocation(*self*, *location*)

Sets the data location path in the program as well as saved to the settings file.

Parameters

`location:` path to new location to save data to

(type=string)

setRate(*self*, *freq*)

This method will set the rate that the board communicates with the CAN Bus on.

Parameters

`freq:` Frequency of CAN communication

(type=number)

clearFilters(*self*)

This method will clear the filters that the user has input into the dialog spots. It does not reset the chip and clear them on the board.

sniff(*self*)

This method will sniff the CAN bus. It will take in the input arguments from the GUI and pass them onto the sniff method in the `GoodFETMCPCANCommunication.sniff` file. The method will take in any filters that have been set on the GUI, as well as the sniff length and comment off the display. This method will call `sniffControl` which will be run as a thread.

sniffControl(*self*, *freq*, *duration*, *description*, *verbose*=`False`, *comment*=`None`, *filename*=`None`, *standardid*=`None`, *debug*=`False`, *faster*=`False`, *parsed*=`True`, *data*=`None`, *writeToFile*=`True`)

This method will actively do the sniffing on the bus. It will call `GoodFETMCPCANCommunication.sniff`. This method will be called by the `sniff` method when started by the user in the GUI. It will set up the display for the incoming sniffed data as well as reset the counters. The input parameters to this method are the same as to the sniff method in the GoodFETMCPCANCommunication class.

Parameters

`freq:` Frequency of the CAN communication

(type=number)

`description:` This is the description that will be put in the csv file. This gui will set this to be equal to the comments string

(type=string)

`verbose:` This will trigger the sniff method to print out to the terminal. This is false by default since information is printed to the GUI.

(type=Boolean)

`comment:` This is the comment tag for the observation. This will be saved with every sniffed packet and included in the data uploaded to the SQL database

(type=string)

`filename:` filename with path to save the csv file to of the sniffed data. By default the sniff method in the GoodFETMCPCANCommunication file will automatically deal with the file management and this can be left as None.

(type=String)

`standardid:` This will be a list of the standard ids that the method will filter for. This can be a list of up to 6 ids.

(type=List of integers)

`debug:` *(type=Boolean)*

updateStatus(*self*, *name*, *index*, *mode*)

This method will update the status indicator for the user in the bottom of of the data Frame. This will tell the user whether the bus is actively being communicated with, if the board is disconnected, or if it is ready to go. There is no checking for if somebody disconnects the board after it has been connected.

updateCanvas(*self*)

This method will update the data to be displayed on the screen (in the text box) to the user. It will get the data form the queue that the `GoodFETMCPCANCommunication.sniff` method was passed. These packets are then printed to the screen in either a fixed or rolling viewpoint. Rolling meaning that each packet is given it's own line while fixed means that each ID has it's own line and that line is updated with each new packet. These can be changed during a sniff run.

arbIDInfo(*self*, *id*)

This method is called when the user clicks on an arbID in the data textboxes. It will open up the data information frame (on the RightSideFrame) and display our knowledge of this arbitration ID to the user

loadJson(*self*)

This method will load our packet informtion from the Json file specified in the settings.

write(*self*)

This method handles injecting packets onto the bus. It will load the user inputs with very basic error checking. It will then call writeControl in a thread. This method will call `GoodFETMCPCANCommunication.spit` if not writing from a file and will call `GoodFETMCPCANCommunication.writeData` if we are reading in data from a file first.

writeControl(*self*, *freq*, *sID*, *repeat*, *writes*, *period*, *debug*=`False`, *packet*=`None`)

This method will spit a single message onto the bus. If there is no packet information provided then the message will be sent as a remote transmission request (RTR). The packet length is assumed to be 8 bytes The message can be repeated a given number of times with a gap of period (milliseconds) between each message. This will continue for the the number of times specified in the writes input. This method does not include bus setup, it must be done before the method call. This method will do this by calling `GoodFETMCPCANCommunication.spit` to inject the message onto the bus.

Parameters

`freq`: The frequency of the bus

(type=number)

`sID`: This is a single length list with one integer elment that corresponds to the standard id you wish to write to

(type=list of integer)

`repeat`: If true the message will be repeatedly injected. if not the message will only be injected 1 time

(type=Boolean)

`writes`: Number of writes of the packet

(type=Integer)

`period`: Time delay between injections of the packet in Milliseconds

(type=Integer)

`debug`: When true debug status messages will be printed to the terminal

(type=Boolean)

`packet`: Contains the data bytes for the packet which is assumed to be of length 8. Each byte is stored as an integer and can range from 0 to 255 (8 bits). If packet == None then an RTR will be sent on the given standard id.

(type=List)

uploaddb(*self*)

This method will upload all files that have been sniffed to the designated folder `self.DATA_LOCATION` to the MYSQL data base that was specified in the settins. See `DataManage.uploadFiles` for more information.

infoFrameLift(*self*, *event*=`None`)

This method is used in the tabbing of the right side frames. This will lift the arbitration id info frame to the top for the user to see. It will also move all the other frames form visability by moving the blank canvas up to the second to top frame.

sqlFrameLift(*self*, *event*=`None`)

This method is used in the tabbing of the right side frames. This will lift the MYSQL frame to the top for the user to see. It will also move all the other frames form visability by moving the blank canvas up to the second to top frame.

sniffFrameLift(*self*, *event*=`None`)

This method is used in the tabbing of the right side frames. This will lift the SNIFF/WRITE frame to the top for the user to see. It will also move all the other frames form visability by moving the blank canvas up to the second to top frame.

experimentFrameLift(*self*, *event*=`None`)

This method is used in the tabbing of the right side frames. This will lift the experiment frame to the top for the user to see. It will also move all the other frames form visability by moving the blank canvas up to the second to top frame.

ourCarFrameLift(*self*, *event*=`None`)

idInfo(*self*)

This method will open an info box for the user to gain information on a known arbID

sqlQuery(*self*)

This method will take in the user's inputed MYSQL query and query the database for it. When the user is saving to a .pcap the user must get the full message and therefore can only specify the second half of the MYSQL query and the first part:

```
SELECT msg FROM table where
```

will be added automatically. This does not happen with the csv option where the user can query whatever they want. BE CAREFUL there is no error checking or prevention from querying something bad that could delete information. Data will be saved to the MYSQL location specified by the DataManage class, `self.dm`. This is self.DATA_LOCATION/SQLData/.

isConnected(*self*)

This method checks to see if the GOODTHOPTER10 is connected

handleCmd1(*self*)

handleCmd2(*self*)

handleCmd3(*self*)

generalFuzz(*self*)

This method grabs user input to perform a general fuzz. Loose error checking is performed to ensure that the data is of the correct format but it is not exhaustive. See `generalFuzzControl` for more information on generalFuzz.

generalFuzzControl(*self*, *freq*, *Fuzzes*, *period*, *writesPerFuzz*)

The method will inject properly formatted, randomly generated messages at a given period for a *writesPerFuzz* number of times. A new random standard id will be chosen with each newly generated packet. IDs will be chosen from the full range of potential ids ranging from 0 to 4095. The packets that are injected into the bus will all be saved in the following path DATALOCATION/InjectedData/(today's date (YYYYMMDD))_GenerationFuzzedPackets.csv. An example filename would be 20130222_GenerationFuzzedPackets.csv Where DATALOCATION is provided when the class is initiated. The data will be saved as integers. This method will call `experiments.generalFuzz` to perform this Each row will be formatted in the following form:

```
row = [time of injection, standardID, 8, db0, db1, db2, db3, db4, db5, c
```

Parameters

`freq:` The frequency at which the bus is communicating

(type=number)

`period:` The time gap between packet inejctions given in milliseconds

(type=number)

`writesPerFuzz:` This will be the number of times that each randomly generated packet will be injected onto the bus before a new packet is generated

(type=integer)

`Fuzzes:` The number of packets to be generated and injected onto bus

(type=integer)

Return Value

This method does not return anything

(type=None)

packetResponse(*self*)

This method compiles the user input for running the Packet Response method. This will perform loose error checking to ensure that the messages are the correct type, it will then pass all the information on as a thread to `packetResponseControl`. See this method for more information.

packetResponseControl(*self*, *freq*, *time*, *repeats*, *period*, *responseID*, *respondPacket*, *listenID*, *listenPacket*=`None`)

This method will allow the user to listen for a specific packet and then respond with a given message. If no listening packet is included then the method will only listen for the id and respond with the specified packet when it receives a message from that id. This process will continue for the given amount of time (in seconds). and with each message received that matches the listenPacket and ID the transmit message will be sent the *repeats* number of times at the specified *period*. This message assumes a packet length of 8 for both messages, although the listenPacket can be None. This method will call `experiments.packetRespond` to perform this experiment.

Parameters

`freq:` Frequency of the CAN bus

(type=number)

`time:` Length of time to perform the packet listen/response in seconds.

(type=number)

`repeats:` The number of times the response packet will be injected onto the bus after the listening criteria has been met.

(type=Integer)

`period:` The time interval between messages being injected onto the CAN bus. This will be specified in milliseconds

(type=number)

`responseID:` The standard ID of the message that we want to inject

(type=Integer)

`respondPacket:` The data we wish to inject into the bus. In the format where respondPacket[0] = databyte 0 ... respondPacket[7] = databyte 7 This assumes a packet length of 8.

(type=List of integers)

`listenID:` The standard ID of the messages that we are listening for. When we read the correct message from this ID off of the bus, the method will begin re-injecting the responsePacket on the responseID

(type=Integer)

`listenPacket:` The data we wish to listen for before we inject packets. This will be a list of the databytes, stored as integers such that listenPacket[0] = data byte 0, ..., listenPacket[7] = databyte 7. This assumes a packet length of 8. This input can be None and this will lead to the program only listening for the

reInjectFuzzed(*self*)

This method compiles user input to be able to re-inject packets that were injected as part of the fuzzing methods. There is loose error checking to ensure that the data is of the correct type but is not very exhaustive. This method will then call `reInjectFuzzed` as a thread to perform this action. See this method for more information on what it does.

The date is input in the form YYYYMMDD. This is used to find the name of the file that contains the fuzzed data that we are looking for.

The startTime and endTime are input in the form HHMM. These are then converted to timestamp form.

reInjectFuzzedControl(*self*, *filename*, *startTime*=`None`, *endTime*=`None`, *id*=`None`)

This method will re-inject data that was already injected to the bus as part of a Fuzzing method. This will load the data searching for the specified arbID that it was given. The reInjectFuzzed will take the user's input of the date to find the filename that will contain the packets. The other inputs are optional. If startTime is provided this will only re-inject messages that were written after the given time until the endTime (if specified). If the endTime is not specified then it will re-inject all packets from the startTime (or start of file) to the end of the file. Likewise the user could also specify the id to search for. if the id is specified then this method will only re-inject packets that had this id.

Parameters

`filename:` path to the inejction file that we are going to read our data from

(type=String)

`startTime:` Optional parameter that specifies an earliest injection time for packets to be re-injected

(type=timestamp)

`endTime:` Optional parameter that specifies the lastest injection time for packets to be re-injected

(type=timestamp)

`id:` Optional parameter that specifies a specific arbitration id that we want to re-inject

(type=Integer)

GenerationFuzz(*self*)

This method takes in the user inputs to perform generation fuzzing. There is loose error checking to ensure that data is of the right type but little else. This method then calls `GenerationFuzzControl` as a thread. See this method for more information on how it works.

GenerationFuzzControl(*self*, *freq*, *sID*, *dbInfo*, *period*, *writesPerFuzz*, *Fuzzes*)

This method will perform generation based fuzzing on the bus. The method will inject properly formatted, randomly generated messages at a given period for a *writesPerFuzz* number of times. This method will call `experiments.generationFuzzer` to perform this.

The packets that are injected into the bus will all be saved in the following path DATALOCATION/InjectedData/(today's date (YYYYMMDD))_GenerationFuzzedPackets.csv. An example filename would be:

`20130222_GenerationFuzzedPackets.csv`

Where DATALOCATION is provided when the class is initiated. The data will be saved as integers. Each row will be formatted in the following form:

`row = [time of injection, standardID, 8, db0, db1, db2, db3, db4, db5, c`

Parameters

`freq:` The frequency at which the bus is communicating

(type=number)

`sID:` List of standard IDs the user wishes to fuzz on. An ID will randomly be chosen with every new random packet generated. If only 1 ID is input in the list then it will only fuzz on that one ID.

(type=list of integers)

`dbInfo:` This is a dictionary that holds the limits of each bytes values. Each value in the dictionary will be a list containing the lowest possible value for the byte and the highest possible value. The form is shown below:

```
dbInfo['db0'] = [low, high]
dbInfo['db1'] = [low, high]
...
dbInfo['db7'] = [low, high]
```

(type=dictionary)

`period:` The time gap between packet inejctions given in milliseconds

(type=number)

`writesPerFuzz:` This will be the number of times that each randomly generated packet will be injected onto the bus before a new packet is generated

(type=integer)

`Fuzzes:` The number of packets to be generated and injected onto bus

(type=integer)

RTRsweepID(*self*)

This method takes in user inputs for a RTR sweep experiment. There is loose error checking to ensure that the input data is of the right type but not very secure checking. This will perform the RTR experiment by calling `RTRsweepIDControl`. See this for more information on the method.

RTRsweepIDControl(*self*, *freq*, *lowI*, *highI*, *attemptsI*, *sT*, *verbose*)

This method will sweep through the range of ids given by lowID to highID and send a remote transmissions request (RTR) to each id and then listen for a response. The RTR will be repeated in the given number of attempts and will sniff for the given duration continuing to the next id.

This method will perform this by calling `experiments.rtrSweep`. The user can specify the range of ids to sweep in a continuous integer region. The options are the number of attempts to request a response and the sniff time post request.

Any messages that are sniffed will be saved to a csv file. The filename will be stored in the DATA_LOCATION folder with a filename that is the date (YYYYMMDD)_rtr.csv. If the file already exists it will append to the end of the file The format will follow that of `GoodFETMCPCANCommunication.sniff` in that the columns will be as follows:

1. timestamp: as floating point number
2. error boolean: 1 if there was an error detected of packet formatting (not exhaustive check). 0 otherwise
3. comment tag: comment about experiments as String
4. duration: Length of overall sniff
5. filtering: 1 if there was filtering. 0 otherwise
6. db0: Integer

   —
7. db7: Integer

Parameters

`freq`: The frequency at which the bus is communicating
(type=number)

`lowI`: The low end of the id sweep
(type=integer)

`highI`: The high end of the id sweep
(type=integer)

`attemptsI`: The number of times a RTR will be repeated for a given standard id
(type=integer)

`sT`: The length of time that it will listen to the bus after sending an RTR
(type=integer)

`verbose`: When true, messages will be printed out to the terminal
(type=boolean)

Return Value

Does not return anything
(type=None)

sweepID(*self*)

This method will gather user inputs for sweeping through standard ids in the range of integers specified by the user inputs. The data will be gathered and loosely error checked to ensure that the data is of the correct type but the checking is not very exhaustive. This will then call `sweeIDControl` in a thread. See this method for more inforamtion on the method.

sweeIDControl(*self*, *freq*, *lowI*, *highI*, *sT*)

This method will sweep through the range of standard ids given from low to high. This will actively filter for 6 ids at a time and sniff for the given amount of time in seconds. If at least one message is read in then it will go individually through the 6 ids and sniff only for that id for the given amount of time. All the data gathered will be saved. This does not save any sniffed packets but the messages are printed out to the terminal. This method will call `experiments.filterStdSweep`

Parameters

`freq`: The frequency at which the bus is communicating

(type=number)

`lowI`: The low end of the id sweep

(type=integer)

`highI`: The high end of the id sweep

(type=integer)

`sT`: Sniff time for each trial. Default is 5 seconds

(type=number)

Return Value

A list of all IDs found during the sweep.

(type=list of numbers)

main(*self*)

This method is the loop that runs the display.

6.2.2 Instance Variables

Name	Description
BOLDFONT	Bold font that all headers will use
SETTINGS_FILE	This stores the location of the file where settings are saved

continued on next page

Name	Description
DATA_LOCATION	Stores the location where sniffing information and injection information will be stored
SQL_NAME	Holds SQL name
SQL_HOST	holds SQL host name
SQL_USERNAME	holds SQL username
SQL_PASSWORD	holds SQL password
SQL_DATABASE	holds SQL database name
SQL_TABLE	holds SQL table name
dm	Data Manager class. This can do all the data manipulation/storage/retrieval
initDx	Total window width
initDy	Total window height
freq	Bus frequency
packetInformationFile	This file stores the user's known information about the packets. This is a json file
experimentFile	This is the experiment file that is car specific. this is for adding a car specific module
experimentGUIFile	This is the GUI file that is needed to add the car specific module to the CAN Reader
CarExtention	This is false when there is no car extension, true otherwise
root	Stores the tk object for the window
RightSideCanvas	Canvas for the entire right side (all tabs are inside of this canvas, in the grid format)
blankCanvas	This is a blank canvas to cover hidden layers
running	This is a boolean value which when false tells you that there is a thread communicating with the bus at the moment
experimentFrame	Experiment frame
generalFuzzData	This is a dictionary that will store all the information needed to run the general fuzz method. keys: period, writesPerFuzz, Fuzzes. All the values are Tkinter.StringVar() and contained the information input by the user
fuzzData	This is a dictionary that will store all the information needed to run the generation fuzz method. keys: sIDs (string of all sIDs to fuzz on), period, writes, writesPerFuzz, Fuzzes, dB0,db1,..,db7 (stores the limits as a list [low, high]. All the values are Tkinter.StringVar() and contained the information input by the user

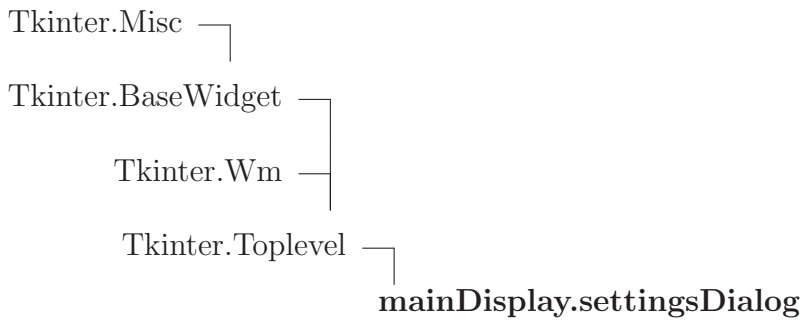
continued on next page

Name	Description
reInjectData	This is a dictionary that will store all the information needed to run the re-injecting of fuzzed data experiment. keys: sID, date, startTime, endTime. All the values are Tkinter.StringVar() and contained the information input by the user
packetResponseData	This is a dictionary that will store all the information needed to run packet response experiment. keys: time, repeats, period, listenID, listen_db0,...,listen_db7,responseID, Response_db0,...,Response_db7. All the values are Tkinter.StringVar() and contained the information input by the user
infoFrame	Frame the holds the display widgets for our packet information
options	These are the known arbitration ids
blankCanvasInfoFrame	This is a blank canvas to hide frames when they are not on top
byteInfoFrame	Byte info frame
packetInfoFrame	Packet Info tab
packetInfoText	This is the text widget where all the packets will be displayed to
packetHyperlink	This contains the links so that we can inject packets in this list by clicking on the the provided description
generalInfoFrame	general information tab
generalInfoVars	This will store all of the variables for resetting the General Info's page
carFrame	Car module frame
filterIDs	This contains a list of the Tkinter.StringVar's that hold the filters that the user inputs
SniffChoice	Type of sniff to display. Rolling will print a new line for each packet. Fixed will keep each packet on it's own individual line
saveInfo	Boolean to let the user decide if they want to save the sniff data or not
writeData	This is a dictionary containing all of the data for writing onto the bus
dataFrame	This is the frame that has all the data display on it
statusString	This is a status update to alert the user if the GOODTHOPTER 10 is connected, in use or ready.

continued on next page

Name	Description
statusLabel	The label that contains the status
msgCount	This counts the number of messages received in a sniff run
msgPrev	This is the time received for the previous message so that we can compute the time between messaages
msgDelta	This Displays the time difference between the last 2 packets sniffed off the bus
topFrame	This frame is just to provide a visual so the user knows what bytes are which in the stream of data
dataText	This is the actual textbox that all the packets are written to
comm	Stores the communication with the CAN class methods

6.3 Class settingsDialog



This class creates a dialog that allows the user to set the settings for the GUI. These settings are saved in /Settings.ini which allows for persistent information even after the user closes the window. You can set the bus frequency, SQL information, data locations.

6.3.1 Methods

__init__(*self*, *parent*, *dClass*, *data*, *title*=`None`)

Constructor method for the window. This is a child window of the mainDisplay

Parameters

`parent:` Tkinter.Tk() for the main window

`dClass:` main window

`data:` unused

`title:` title for the child window

Overrides: Tkinter.BaseWidget.__init__

body(*self*, *master*)

This method builds the main part of the window with the settings options and input locations

setRate(*self*)

This will set the rate the GOODTHOPTER 10 listens to the CAN bus

buttonbox(*self*)

buttons in the display. Apply/Cancel buttons

ok(*self*, *event*=`None`)

This method will gather all the SQL information and save it to the settings file as well as update the information within the GUI itself for the current instance.

cancel(*self*, *event*=`None`)

Closes the window if the user wants to cancel settings changes

validate(*self*)

apply(*self*)

Inherited from Tkinter.BaseWidget

destroy()

Inherited from Tkinter.Misc

__contains__(), __getitem__(), __setitem__(), __str__(), after(), after_cancel(), after_idle(), bbox(), bell(), bind(), bind_all(), bind_class(), bindtags(), cget(), clipboard_append(), clipboard_clear(), clipboard_get(), colormodel(), columnconfigure(), config(), configure(), deletecommand(), event_add(), event_delete(), event_generate(), event_info(), focus(), focus_displayof(), focus_force(), focus_get(), focus_lastfor(), focus_set(), getboolean(), getvar(), grab_current(), grab_release(), grab_set(), grab_set_global(), grab_status(), grid_bbox(), grid_columnconfigure(), grid_location(), grid_propagate(), grid_rowconfigure(), grid_size(), grid_slaves(), image_names(), image_types(), keys(), lift(), lower(), mainloop(), nametowidget(), option_add(), option_clear(), option_get(), option_readfile(), pack_propagate(), pack_slaves(), place_slaves(), propagate(), quit(), register(), rowconfigure(), selection_clear(), selection_get(), selection_handle(), selection_own(), selection_own_get(), send(), setvar(), size(), slaves(), tk_bisque(), tk_focusFollowsMouse(), tk_focusNext(), tk_focusPrev(), tk_menuBar(), tk_setPalette(), tk_strictMotif(), tkraise(), unbind(), unbind_all(), unbind_class(), update(), update_idletasks(), wait_variable(), wait_visibility(), wait_window(), waitvar(), winfo_atom(), winfo_atomname(), winfo_cells(), winfo_children(), winfo_class(), winfo_colormapfull(), winfo_containing(), winfo_depth(), winfo_exists(), winfo_fpixels(), winfo_geometry(), winfo_height(), winfo_id(), winfo_interps(), winfo_ismapped(), winfo_manager(), winfo_name(), winfo_parent(), winfo_pathname(), winfo_pixels(), winfo_pointerx(), winfo_pointerxy(), winfo_pointery(), winfo_reqheight(), winfo_reqwidth(), winfo_rgb(), winfo_rootx(), winfo_rooty(), winfo_screen(), winfo_screencells(), winfo_screendepth(), winfo_screenheight(), winfo_screenmmheight(), winfo_screenmmwidth(), winfo_screenvisual(), winfo_screenwidth(), winfo_server(), winfo_toplevel(), winfo_viewable(), winfo_visual(), winfo_visualid(), winfo_visualsavailable(), winfo_vrootheight(), winfo_vrootwidth(), winfo_vrootx(), winfo_vrooty(), winfo_width(), winfo_x(), winfo_y()

Inherited from Tkinter.Wm

aspect(), attributes(), client(), colormapwindows(), command(), deiconify(), focusmodel(), frame(), geometry(), grid(), group(), iconbitmap(), iconify(), iconmask(), iconname(), iconposition(), iconwindow(), maxsize(), minsize(), overrideredirect(), positionfrom(), protocol(), resizable(), sizefrom(), state(), title(), transient(), withdraw(), wm_aspect(), wm_attributes(), wm_client(), wm_colormapwindows(), wm_command(), wm_deiconify(), wm_focusmodel(), wm_frame(), wm_geometry(), wm_grid(), wm_group(), wm_iconbitmap(), wm_iconify(), wm_iconmask(), wm_iconname(), wm_iconposition(), wm_iconwindow(), wm_maxsize(), wm_minsize(), wm_overrideredirect(), wm_positionfrom(), wm_protocol(), wm_resizable(), wm_sizefrom(), wm_state(), wm_title(), wm_transient(), wm_withdraw()

6.3.2 Class Variables

Name	Description
Inherited from Tkinter.Misc	
noarg	

7 Module tkHyperlinkManager

7.1 Variables

Name	Description
__package__	**Value:** `None`

7.2 Class HyperlinkManager

Wrapper class for setting hyperlink bindings within the text box. This code was modified from the opensource website *http://effbot.org/zone/tkinter-text-hyperlink.htm* and carries the following copyright:

Copyright 1995-2010 by Fredrik Lundh

See the website for additional information on the functionality of this class.

7.2.1 Methods

__init__(*self*, *text*)

reset(*self*)

remove all hyperlinks

add(*self*, *action*, *id*)

Add a new hyper link

Parameters

`action:` method that will be called for this hyperlink

`id:` the arbitration id that we are associating this action.

Index