SYSTEMS ENGINEERING

COLORADO STATE UNIVERSITY

Student CyberTruck Experience Manual

Jeremy Daily

David Nnaji

Ben Ettlinger

Subhojeet Mukhergee

June 26, 2020

A special thank you goes out to Urban Jonson at the National Motor Freight Traffic Association, Inc. (NMFTA) for being a champion of the program. His passion for cybersecurity and education was insiprational in creating the content and program called the Student CyberTruck Experience. The Executive Director the NMFTA, Paul Levine, deserves special thanks as he provided resources and guidance on bringing the Student CyberTruck Experience to life. Our initial sponsors at the University of Tulsa include the NMFTA, Geotab, and PeopleNet. The continued support from Geotab through Ryan Brander and Glenn Atkinson are gratefully appreciated.

The students who significantly contributed to this manual, both in writing and participating in the Student CyberTruck Experience include: Hayden Allen, Duy Van, John Maag, Ryan Corley.

Contents

1 Introduction to the Student CyberTruck Experience

1.1 Trucking as a Critical Infrastructure

Transporting material by truck is a critical aspect of modern life. The Transportation Systems Sector has been declared a critical infrastructure sector by the Department of Homeland Security (DHS) [1]. Therefore, protecting trucks, trailers, vans, buses and the logistic engines supporting these vehicles is of the utmost importance for our way of life.

In the mid 1990s, the technology used on heavy vehicle transitioned from mechanical controls to digital controls. For many years, these digital controls and embedded systems were isolated from the rest of the world unless accessed by a technician. This is known as an air gap. However, the trucking industry realized process improvements through remote monitoring of the vehicles and their status. Technologies like over-the-air updates have become part of the truck and its supporting systems. This broke down the air gap defense and trucks may be connected to the Internet by way of a cellular modem. With the truck network connected to an outside network, there is a possibility for a vulnerability to be exploited and an attacker to remotely execute code on the truck. This is problematic, because the truck systems are trusting of the internal network and will react without regard the authenticity of a message. This means a well crafted message on the internal network can wreak havoc on the truck and potentially shut it down... or worse. Keep in mind, a truck traveling at 70 miles per hour carrying 80,000 lb has an large amount of energy. Here is a quick example of the kinetic energy calculation for moving truck:

$$KE = \frac{1}{2}mv^2$$

$$KE = \frac{1}{2}\frac{W}{g}(1.466S)^2$$

$$KE = \frac{80,000}{2(32.2)}[1.466(70)]^2$$

$$KE = 13,081,819 \text{ ft-lb}$$

For comparison, a .30-06 rifle can propel a 0.308 inch diameter bullet at 3000 ft/second, which is around 3000 ft-lbs of energy. In other words, a truck has as much energy as 4360 rifles being fired at once!

The ability for a truck to cause serious damage in a crash is significant. Crashes are perhaps the most

spectacular outcome of a cyber attack, but other subversive results can be achieved, like theft, logistics disruption, a traffic congestion. One particularly scary scenario is if a cyber attack affects many trucks at the same time. The tow and recovery services can easily respond to a couple broken-down trucks, but if thousands of truck came to a halt at the same time, the highway system would be broken. This means medical supplies, food, and fuel are no longer distributed. Its only a few days before serious civil unrest would start if trucking gets shut down. Brainstorming terrible scenarios is somewhat trivial, but our goal is to develop solutions. A compelling industry report from the NMFTA provides much of the rational to address the problem of heavy vehicle cybersecurity [2]. The reason for this book and the accompanying Student CyberTruck Experience is to teach the engineers and scientists the necessary elements needed to improve the cybersecurity posture of the transportation industry.

How the Student CyberTruck Experience Started

In 2016, Urban Jonson of the National Motor Freight Traffic Association, Inc. (NMFTA) reached out to Dr. Jeremy Daily while he was teaching mechanical engineering at the University of Tulsa (TU) to discuss some research findings related to heavy vehicle cybersecurity. In this conversation, Dr. Daily confirmed many of the hypotheses in the NMFTA whitepaper on the state of heavy vehicle cybersecurity. This conversation led to an invitation for Dr. Daily to attend the first NMFTA Heavy Vehicle Cyber Security (HVCS) meeting in Virginia. One of the results of the meeting was a recognition for the need to build the human talent needed to address the challenges associated with cybersecurity of heavy vehicles. This initiative is how the Student CyberTruck Experience came into being. A detailed account of the lessons learned in developing heavy vehicle cybersecurity talent was presented at the ESCAR USA 2017 conference [3].

1.2 Program Mission

The mission of the Student CyberTruck Experience is to *develop the talent necessary to improve the cybersecurity posture of heavy vehicles*.

This mission is intended to be high level and focused on the big picture. However, it is too general and hard to measure the actual achievement of the mission. There are a few indicators that the program is succeeding in its mission:

1. Graduates of the program have been hired into the industry in a cybersecurity capacity.
2. Graduates have gone onto graduate school and created scholarly works contributing to the body of science around heavy vehicle cybersecurity.
3. Student participants have attended and participated in the CyberTruck Challenge, thus building the overall awareness around heavy vehicle cybersecurity.
4. External sponsors continue to sponsor the research project, which enables undergraduates to focus summer research efforts towards heavy vehicle cybersecurity.

5. The ideas and projects from the Student CyberTruck Experience have been used as ideas for other funded research projects at the federal level.
6. Students consistently receive positive feedback at the HVCS bi-annual meetings.

While this is great feedback as to the success of the program, we need a more measured approach to help students and researchers progress through the program and maximize their potential. There is a famous saying:

> You can only improve what you measure.

This calls for some measurable program learning objectives.

1.3 Program Learning Objectives

As suggested by Sara Rathburn from Colorado State University's The Institute for Teaching and Learning, learning objectives are written statements organizing and defining the specific knowledge, skill-sets, and/or abilities students should acquire [4]. These learning objectives help students assess their skills and provide specifics for the learning journey to become a productive cybersecurity practitioner. An effective learning objective has three parts:

1. A description of what is expected.
2. The conditions necessary to demonstrate proficiency.
3. The criteria for evaluating performance.

For example, a Program Learning Example could be

- Create a program to reassemble messages that arrive out of order from a J1939 Transport Protocol - Data Transfer to properly receive the data in the same order in which it was sent.

Looking at this objective, there is a clear output of what output is expected. Specifically, the output is the computer program. However, the language writing the code is not specified, so any language would be acceptable. The conditions for the exercise is an out of order message stream from J1939 network. Success will be achieved when the original byte stream matches the reassembled message. The primary action verb in this objective is the word create. Different action verbs suggest different levels of learning, which has been popularized as Bloom's Taxonomy.

1.3.1 The Taxonomy of Educational Objectives (Bloom's Taxonomy)

When developing new course for university, we have to generate a set of course objectives. A well reasoned approach is to create objectives that start with the phrase "By the end of this course, students should be able to..." and the rest of the sentence is the learning objective. To create the learning objective, an action verb is used followed by the object of knowledge they are expected to acquire or construct. There are multiple

dimensions of knowledge as well as different levels of thinking about things. A succinct info-graphic was produced at Iowa State University under a Creative Commons License and displayed in Figure 1.3.1 on the following page. The examples and breakdown of the learning objective space shown in the figure will be used to span the basis for talent generation for the Student CyberTruck Experience.

The learning objectives should focus on accomplishing the program mission. Since the topic of cybersecurity is rapidly changing in the modern era, the learning objectives may need to change to accommodate advances in technology, knowledge, and application. For example, the advent of quantum computing promises to create a paradigm shift in modern cryptography as many "hard" problems that are used as the basis for modern cryptography can be calculated quickly, thus nullifying the underlying algorithms. As of this writing, there have been no practical exploits based on quantum computing.

Often Bloom's Taxonomy is constructed as a pyramid or cake as shown in Figure 1.3.2 on page 12. This is a simpler graphic to digest and use as a model for your own level of knowledge. Often the base layers are necessary to accomplish a mastery of a higher layer. For example, to apply a cryptographic algorithm on a heavy truck network, you would need to understand the limitations of J1939 and have knowledge of the truck system. The larger foundations are needed to achieve the higher order thinking. This means learning objectives that are focused on the top of the pyramid, like evaluate and create, require a mastery of the layers below. Thus students should strive to achieve the highest level of contemplation in an effort to have the broadest experience.

However, when a new student is introduced to a field, like heavy vehicle cybersecurity, they cannot immediately go to the top layer. Moreover, the process of building the knowledge foundation is critical to success. As such, the learning objectives in the program will vary across the cognitive process dimension. This enables a student to enter the program at their level and still have a objectives to challenge them. For example, a computer science student may have training and experience using data structures in C and an engineering student may have practical experience fixing air brake systems. These students would each bring their own level of expertise to the program and enter the quest for knowledge at different levels. The CS student may be able to quickly declare a structure for a CAN data frame in a C header file, but the ME student may need to learn about C data structures and data types, then understand how to create a struct. In other words, the level of effort in achieving mastery of the learning objectives may be different for each student.

1.3.2 Program Learning Objectives

To achieve the program mission, students who go through the Student CyberTruck Experience (CyTeX) should be able to...

1. List the major Original Equipment Manufacturers (OEM) and Tier 1 Suppliers for on-highway heavy vehicles.
2. Recognize a unified diagnostic service (UDS) message in SAE J1939 communication.
3. Recall the different layers of the Open Systems Interconnection (OSI) networking model.

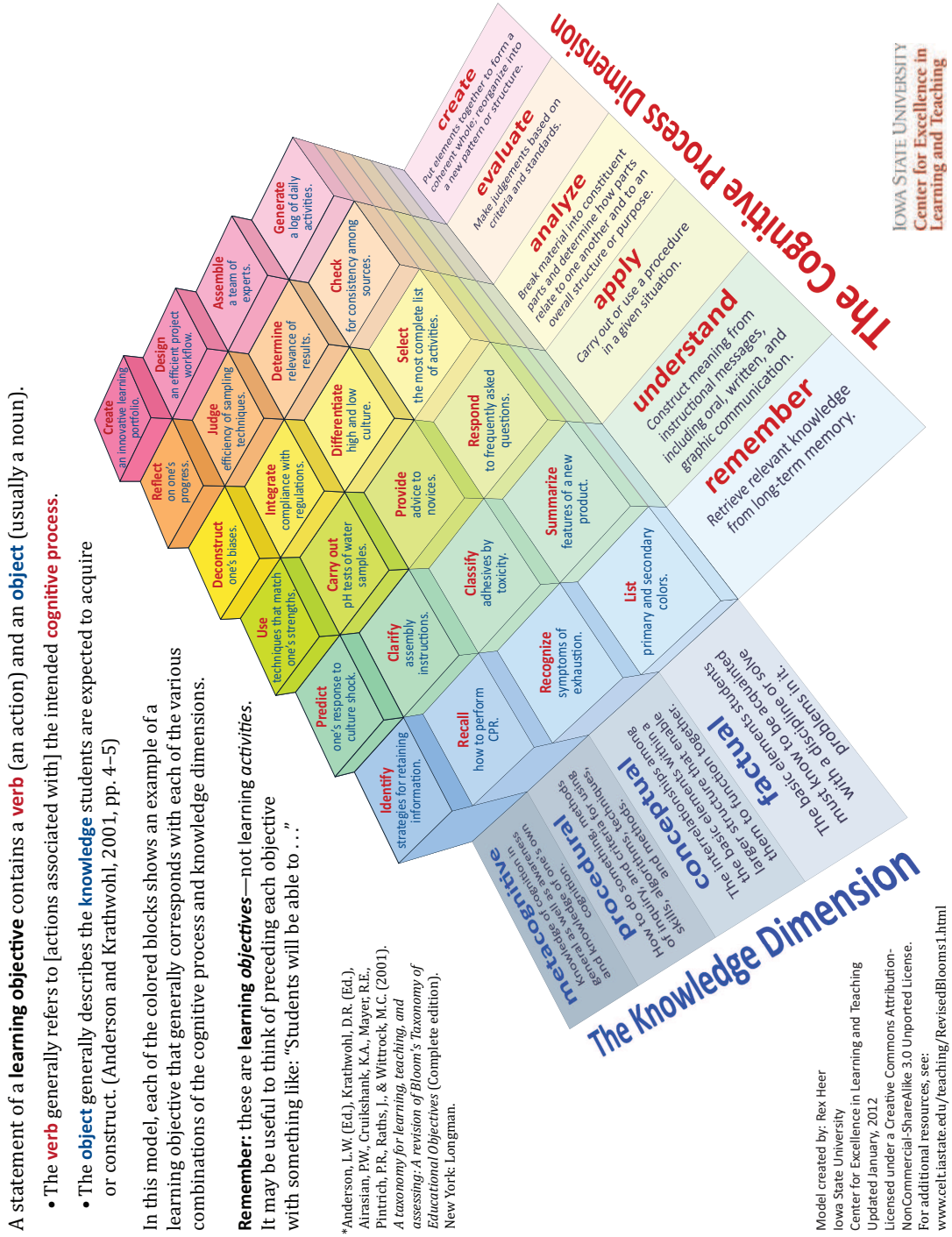

Figure 1.3.1: Revised Bloom's Taxonomy used in creating learning objectives [5].

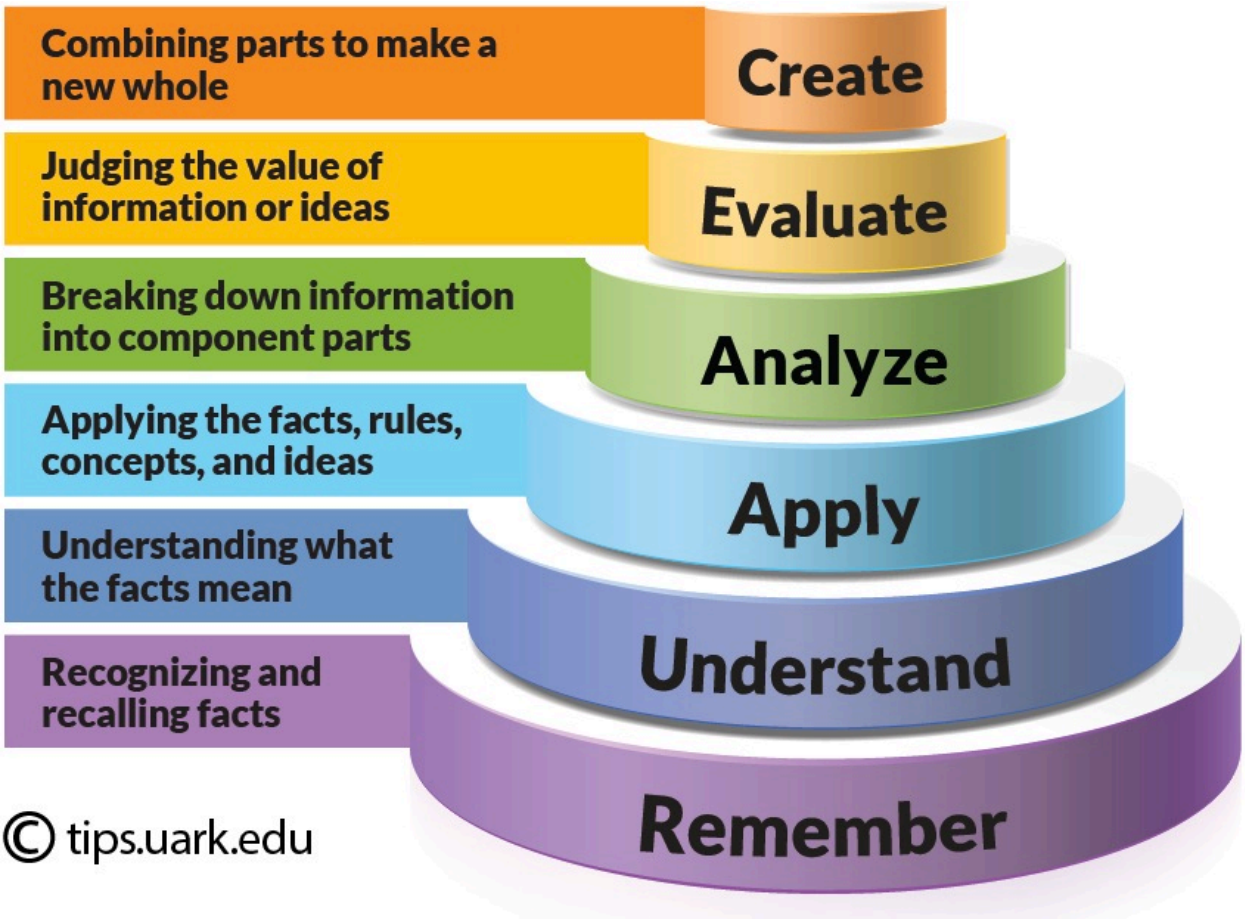

Figure 1.3.2: Bloom's Taxonomy from Jessica Shabatura.

4. Identify the different electronic control units (ECUs) used on heavy vehicles.
5. Summarize the strengths and weaknesses of the controller area network.
6. Classify trucks and trailers by weight class and use.
7. Clarify the way CAN bus data is used for the freight carriers, OEMs, tier 1 suppliers and telematics providers.
8. Predict potential cybersecurity attacks through an electronic logging device (ELD).
9. Respond to J1939 request messages for VIN, Component Identification, and other J1939 on request messages.
10. Provide a secure API for a CAN to Ethernet electronic device to network log data on a computer.
11. Carry out cryptographic message authentication for intra-vehicle communications.
12. Use certificates and public key infrastructure (PKI) to securely update a vehicle connected device.
13. Select the fastest cryptographic approach to secure diagnostic communication between a gateway and a diagnostic application.
14. Differentiate between best practices for IT security and heavy vehicle network security.
15. Integrate X.509 Certificates and PKI into embedded hardware security modules connected to vehicles.
16. Deconstruct heavy vehicle event data recorder (HVEDR) data sets from raw binary to time history graphs.
17. Check and verify digitally signed data from a CAN Logger
18. Determine the contents of a CAN frame based on an oscilloscope trace of the signals on the CAN bus wires.
19. Judge the real-time performance of HMAC and CMAC approaches to verifying the integrity and authenticity of CAN messages.
20. Reflect on the challenges for implementing secure solutions to legacy vehicles already on the road.
21. Generate a solution to stream maximum bit rate CAN data over an IP network.
22. Assemble a system to detect and defend against intrusions on the CAN bus.
23. Design a printed circuit board to bridge multiple networks while securely storing cryptographic key material.
24. Create an original research poster advancing the cybersecurity posture of heavy vehicles.

Notice these objective make use of every verb in the graphic shown in Figure 1.3.1 on page 11. Additional objectives will be enumerated within each chapter of the manual as the specifics of the content and exercises are matured.

1.4 Exercises

While the learning objectives are fantastic for what we want to achieve, the specific activities to accomplish those goals need to be defined. Through the manual, we will present exercises that focus on helping students to achieve a learning objective. These exercises are created at different levels of the taxonomy, which enables a learner to start at a level appropriate to their skills. Successful completion of all the exercises should lead to the accomplishment of all learning objectives. The idea of successful completion means we have to have a measurement for success.

Exercise 1.1. [Evaluate] Reflect on on the program learning objectives in Section 1.3.2 and determine what is missing. Write a new learning objective using a verb that is not already in the list. Justify the reason for the new objective and submit it to your cohort for discussion.

For each exercise, the

1.5 Assessments

In school, formal assessments typically come in the form of quizzes, exams, and essays. These tools are part of a traditional pedagogy where students learn to achieve the grade and the rewards associated with the good grades. In this model, the teacher is primarily responsible for the assessment tools. However, the teacher feedback and assessment needs to transition to self or peer assessment when schooling ends and graduates enter the work force. Since the Student CyberTruck Experience mission is workforce centric, these self assessment skills are necessary to develop.

The exercises used throughout the program should have measurable outputs. Many times the outputs are binary (e.g. the approach worked or it didn't), but some outputs are qualitative or quantitative. For quantitative outputs, there are typically thresholds of performance to measure success. For example, if an exercise is to build a CAN to Ethernet translator device, the quantitative measure of success could be the message rate through puts of the device. The exercise should include the testing functions required to perform the assessments; and the outputs of the tests should be compared to a threshold or standard of performance. For qualitative results, the arguments and analysis need to be presented clearly to peers, mentors, and instructors to ensure the concept is effectively communicated. Feedback is the primary form of assessing qualitative outputs.

1.5.1 Formative Assessments

Formative assessments are frequent small tests of ideas and concepts with low-stakes. They are meant to have little to no pressure and provide quick feedback on the progress towards achieving mastery of the learning objectives. Some types of formative assessments include:

- Conversations to determine levels of knowledge, like trivia questions

- What-if scenarios
- Bouncing ideas off peers
- Confirming concepts with mentors
- Sketching diagrams and flow charts

Formative assessments are frequently used in writing code and are synonymous with debug statements. Simply put, a debug statement is a small test to ensure the code is working according to your idea and design.

Students should develop their own set of self-critiquing skills and frequently perform their own formative assessments. The more of these frequent assessments that take place, the more likely for success. Furthermore, the documentation and communication of these formative assessment results are important to "managing your boss," which is an important workplace behavior where a subordinate will frequently update the manager on progress and challenges. Articulating what you've done, how you tested it, and explaining the results are welcome communications for any good manager.

An example of the formative assessment for Exercise 1.1 is the discussion with your peers and instructor. There are no grades associated with the exercise and the questions and discussion around your proposed objective will give you feedback on if it is clear and measurable.

1.5.2 Summative Assessments

In school, summative assessments are less frequent, high stakes exercises like examinations and term papers. Since the focus of the Student CyberTruck Experience is project based, the summative assessment is the successful completion of the project. This begs the question of what success means. In an engineering design, a customer will voice a need or desire. One of the first jobs of the design engineer is to restate the problem into a set of measurable requirements. Since a good requirement is measurable, the project essentially assesses itself. The project is completed once all the requirements are demonstrated to be satisfied. This means it is possible to perform a summative assessment completely by yourself. However, an external review of the project is highly recommended.

For the CyTeX program, the realization of the summative assessment is the research poster presented at the NMFTA's fall HVCS meeting. These posters are viewed by all the attendees, which range from motor carriers, to government researchers, military scientists, and cybersecurity professionals. Students are expected to explain their poster contents and address questions from the attendees.

An example of the summative assessment for Exercise 1.1 is the acceptance of the proposed learning objective into the program. Perhaps it will be included in the next version of this manual!

1.6 Research

Every year we continue to research topics that can be relevant and helpful to improving the cybersecurity posture of the trucking industry. These projects are curated from the needs of the trucking companies, current technologies, topics of interest, and achievability. The list of projects is curated in the fall of each year and proposed to the sponsors. Once agreed upon, the projects are presented to the students. Student select the projects based on interest, but often they don't have enough background to know where to go. In those cases, the exercises herein can lay the foundation of knowledge and experience to take steps towards accomplishing the research.

Research can lead new discoveries and novel methods, but the likely output of the research for this program is a well thought out engineering design project. The engineering design process is as follows:

Ask Understand the needs of the customer, which means you have to know who the customer and their use cases. Restate and define the problem with a series of measurable engineering specifications, criteria, and constraints.

Imagine Brainstorm and explore different possible solutions to the problem.

Plan Select the most promising solution and draw/sketch out a solution. Determine the resources needed

Create Implement a prototype the solution. Acquire the parts and materials needed and build the solution

Test Evaluate the prototype to determine if it works and satisfies the design constraints

Iterate the process and improve the implementation and refine the requirements.

Share Communicate your design and solution to the appropriate audience.

There are many info-graphics on the Internet to visualize this process. The final poster presentation is the final step in sharing your results. The poster should capture the engineering design process and the things you've learned along the way, even if you learned that something doesn't work.

1.7 Systems Engineering Approach

The engineering design process is often visualized in an iterative loop. Another visualization, popular among systems engineers, is the systems V ("vee") diagram. The V-diagram is used by in SAE J3061: Cybersecurity Guidebook for Cyber-Physical Vehicle Systems to describe the process taken to design and develop more secure cyber-physical systems [6]. The graphic for this process is shown in Figure 1.7.1.

The V diagram is setup so the design phases are on the left and the testing phases are on the right. The top of the diagram contains the systems level approach, while the bottom V is separated into hardware and software designs. Many processes are iterative and follow the engineering design process within their own blocks. Often the hardware and software teams need to work closely together. For example, a systems level requirement may be to securely store cryptographic keys. Often this is done with a dedicated hardware

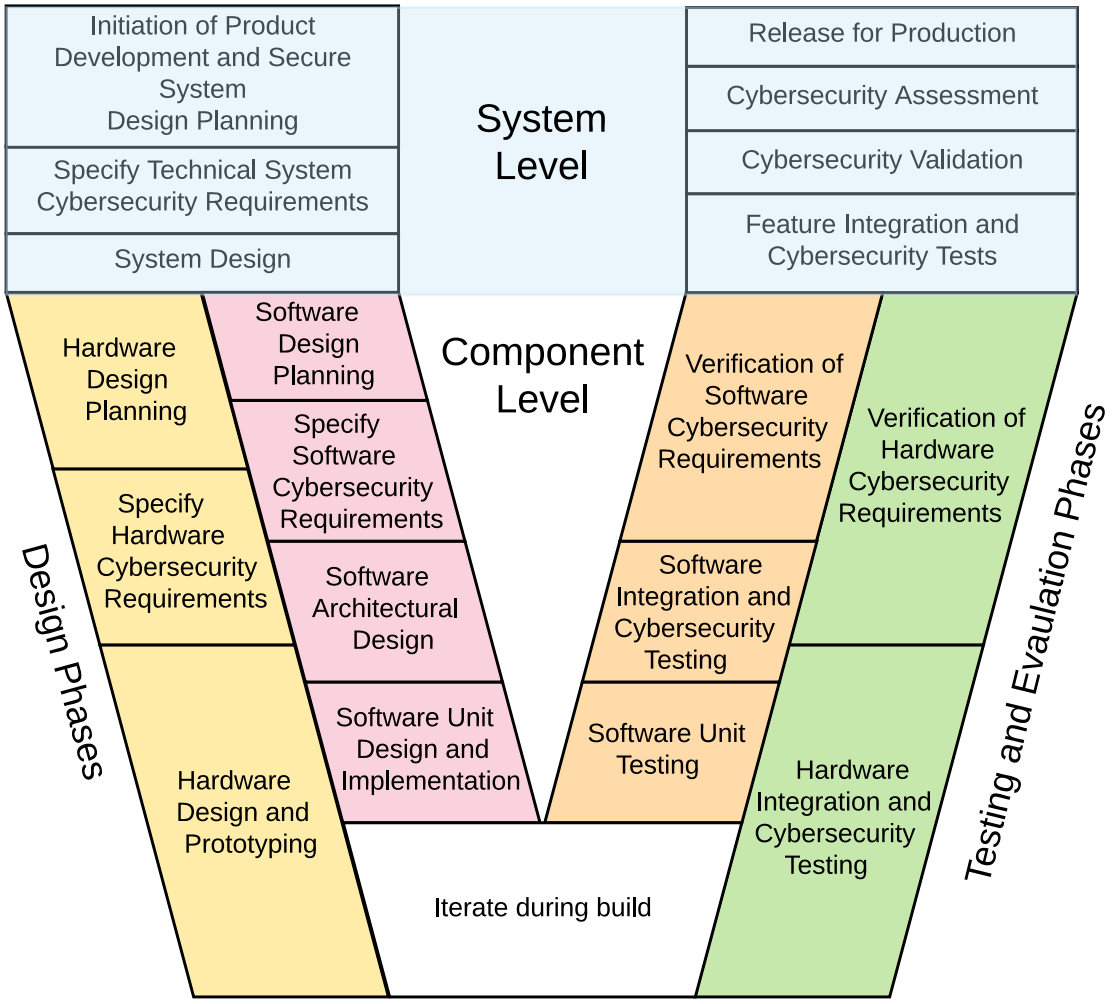

Figure 1.7.1: Systems engineering process diagram for designing more secure cyber-physical systems.

security module (HSM). The hardware team needs to build the circuit to support the HSM and the software team needs to use the key material stored on the HSM. The software team may have to use evaluation hardware prototypes to implement the communication protocols and product algorithms.

During your work in the Student CyberTruck Experience, take some time to reflect where you are working in the diagram of Figure 1.7.1. This reflection should help you understand the different levels of effort and teamwork needed to bring secure solutions into fruition.

1.8 Anticipated Schedule

The training for which this maual is designed is focused on a 16 week course, which is typical of a college semester. Many students will come into the training with sufficient background in some topics, but others should be fruitful for exploration. However, the schedule and contents are intended to develop the necessary talent needed to succeed in heavy vehicle cybersecurity research.

Week 1 Receive and build hardware kits with CAN, LEDs, and the potentiometer. Setup software and the tool chain needed to perform the exersises.

Week 2 Perform some basic programming with Arduino by writing some short test scripts to demonstrate hardware functionality.

Week 3 Gather CAN data from an running SAE J1939 network on a heavy vehicle. Parse single frame data according to SAE J1939. Plot the Engine Speed data.

Week 4 Understand and parse SAE J1939 Transport Protocol Data.

Week 5 SAE J1939 Address Claims, launch cyberattacks on a vehicle.

Week 6 Gather diagnostic data with a vehicle diagnostic adapter using RP1210.

Week 7 Parse data from a Unified Diagnostic Session (UDS) using ISO-14229.

Week 8 Explore seed-key exchanges for diagnostic sessions.

Week 9 Legacy networks J1708 and J1587.

Week 10 Introduction to Ethernet and networking concepts.

Week 11 IP based networks, TCP, and UDP.

Week 12 Build a CAN to Ethernet Bridge

Week 13 Cryptography and Cybersecurity

Week 14 Symmetric Algorithms (AES) and Message Authentication Codes

Week 15 Asymmetric Algorithms, Certificates, Digital Signatures and Diffie-Hellman Key Exchanges

Week 16 Implementing Security with Hardware Security Modules

2 Introduction to Trucking

2.1 Objectives

What is the trucking industry?

What is a truck?

Who makes trucks?

Cybersecurity for Trucking

2.2 Trucking Resources

The National Motor Freight Traffic Association, Inc. (NMFTA) has produced a great whitepaper [2]

3 Overview of Heavy Vehicle Systems

Weight Classes

3.1 Smart Sensor Simulators

SSS2 Descrpiption

3.1.1 Updating the firmware

Coco fills in this part

4 Prototyping Electronics

The purpose of this chapter is to give students skills necessary to work with modern electronics and build their own hardware tools.

4.1 Intro to Microcontrollers

A **microcontroller (MCU)** is a tiny computer that can run one program at a time, over and over again. More often than not, they are connected to sensors and actuators allowing them to listen and interact with the physical world.

Sensors convert physical phenomena such as wheel speed, fuel level, and oil temperature into electrical signals. **Actuators** convert electrical energy back into physical energy such as linear motion, light, heat, and motor torque.

MCUs listen to sensors and talk to actuators. They decide what to do based on the instructions you have written and stored to its memory. MCUs and the devices connected to them form the basis of the **Electronic Control Unit (ECU)**.

4.2 Arduino and Teensy

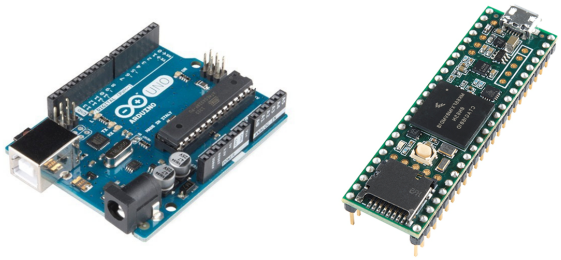

Figure 4.2.1: The Arduino Uno (left) and Teensy 3.6 (right)

Arduino offers some of the most recognizable open-source microcontroller boards today. This includes the Arduino Uno. When it comes to prototyping, these boards have many advantages including: widespread availability, low cost, thorough documentation, and a global community of users and forums. The Arduino

Uno uses the ATMEGA328P, has 32 KB of flash memory and runs at 16 MHz which are acceptable hardware specifications for simple projects.

Programs are written and compiled in the Arduino IDE program, a special text-editor, and uploaded via USB. When the Arduino is connected to an external power source it will run the instructions you uploaded.

The Teensy is another USB-based microcontroller development system. Although it is slightly more expensive than the Arduino, the 3.2 and 3.6 versions are preferred within CyTeX for their additional capabilities and features. Most notably, they support more communication protocols such as CAN, I2C, SPI, and Ethernet. The Teensy 3.6 footprint is roughly a third the size of the UNO, the MK66 Processor, has 1024 kB of flash memory, and runs at 180 MHz.

All programming is done via the USB port and the Teensy Loader Application. It is run automatically when using Verify or Upload within the Arduino software. Teensy Loader is available for Windows, Mac, and Linux.

Full documentation for the Teensyand Arduino UNOcan be found on their respective websites.

4.3 Intro to Microcomputers

A single-board computer (SBC) is a complete computer built on a single circuit board, with microprocessor(s), memory, input/output (I/O) and other features required of a functional computer. You *could* call it a "micro-computer." SBCs have the ability to run multiple programs, run an operating system, and can often support a simplified version of a typical desktop GUI.

The Raspberry Pi by the Raspberry Pi Foundation in the UK is by far the most common prototyping SCB. For automotive applications, the Beaglebone Black by Texas Instruments and the NXP S32K have been used within CyTeX. Both of these devices are capable of more advanced functionality.

4.4 Recommended Hardware Kit

Each student should have access to the parts in the basic kit. These items are minimum to successfully accomplish the programming and learning exercises.

4.4.1 Basic Kit

The Basic Kit comes in a plastic conductive container and all the parts needed to accomplish many of the code and prototyping projects.

Qty	Label	Description	Supplier	Supplier Part Number
1	A	Teensy 4.0 Development Board	PJRC	TEENSY40_PINS[1]
2	B	MPC2562 CAN Transceiver	Digi-Key	MCP2562FD-E/P[2]
1	C	Solderless Breadboard	Digi-Key	BKGS-830-ND[3]
5	D	120 Ohm Axial Resistors	Digi-Key	S120CACT-ND[4]
1	E	Wiz850IO Ethernet Expansion Board	Digi-Key	1278-1043-ND[5]
1	F	30 Pack of Male-Male Breadboard Wires	Sparkfun	PRT-14284[6]
1	G	20 Pack of Male-Female Breadboard Wires	Sparkfun	PRT-12794[7]
1	H	Ethernet Cat6 Cable, 3 ft.	Sparkfun	CAB-08915[8]
1	J	USB micro Cable, 6 inch	Sparkfun	CAB-13244[9]
1	K	SOIC8 to DIP Converter Board	Sparkfun	BOB-13655[10]
1	L	ATECC608A Crypto Authentication Module	Digi-Key	ATECC608A-SSHDA[11]
1	M	ATECC608 Crypto Co-Processor Breakout	Sparkfun	SPX-15838[12]
1	N	Trimpot 10K Ohm with Knob	Sparkfun	COM-09806[13]
1	P	Break Away Headers - Straight	Sparkfun	PRT-00116[14]
1	Q	Ambient Temperature Sensor	Sparkfun	SEN-14049[15]
4	R	4.7k ohm axial resistors	Digi-Key	CF14JT4K70[16]
3	S	LEDs	Digi-Key	LTL-4223[17]
1	T	SerLCD 16x2 Character Display	Sparkfun	LCD-14072[18]
1	U	Conductive 18 Compartment Organizer	Flambeau	C618[19]

Table 4.4.1: Basic Hardware Kit (Either use M or a the combination of K and L.)

Newer laptop computers may not have a physical Ethernet port, so a USB to Ethernet adapter may be necessary.

Exercise 4.1. [Remember] Download the Datasheet for the Teensy 4.0 processor. Answer the following questions:

1. What is the name of the processor?
2. What technology is the processor core built on?

[1] https://www.pjrc.com/store/teensy40_pins.html
[2] https://www.digikey.com/product-detail/en/microchip-technology/MCP2562FD-E-P/MCP2562FD-E-P-ND/4842807
[3] https://www.digikey.com/products/en?keywords=BKGS-830-ND
[4] https://www.digikey.com/product-detail/en/stackpole-electronics-inc/RNMF14FTC120R/S120CACT-ND/2617441
[5] https://www.digikey.com/products/en?keywords=Wiz%20850
[6] https://www.sparkfun.com/products/14284
[7] https://www.sparkfun.com/products/12795
[8] https://www.sparkfun.com/products/8915
[9] https://www.sparkfun.com/products/13244
[10] https://www.sparkfun.com/products/13655
[11] https://www.digikey.com/products/en?keywords=ATECC608A-SSHDA-TCT-ND
[12] https://www.sparkfun.com/products/15838
[13] https://www.sparkfun.com/products/9806
[14] https://www.sparkfun.com/products/116
[15] https://www.sparkfun.com/products/14049
[16] https://www.digikey.com/products/en?keywords=CF14Jt4k70CT-ND
[17] https://www.digikey.com/product-detail/en/lite-on-inc/LTL-4223/160-1127-ND/200395
[18] https://www.sparkfun.com/products/14072
[19] https://www.flambeaucases.com/18-compartment-box-1095.aspx

Figure 4.4.1: Photograph of Example Parts Kit using M in place of K and L.

3. How many bits does the processor use when executing instructions?
4. How fast does the processor run?
5. How many CAN channels does the micoprocessor have?
6. What are the register number bases for the CAN Channels?

Exercise 4.2. [Understand] After reviewing the datasheet for the MCP2562, address the following questions:

1. Why do we need to connect pin 5 on the MCP2562 to 3.3V?

Exercise 4.3. [Apply] Build the circuit in Figure 4.4.2 on the next page using a solderless breadboard. For the inital build, ignore the PWM op amp and the power conditioning. Testing the circuit will use USB for power.

4.4.2 Basic Tools

Multimeter Any modern digital multimeter is acceptable to get started. There are many on Amazon that are suitable.

Soldering Iron A Weller WES51 or a Hakko FX888 with fine tips are good.

Solder Lead based solder with a flux core is much easier to work with as opposed ot lead free solder. Be sure to use proper protective equiment (gloves and safety glasses) and work in a well ventilated

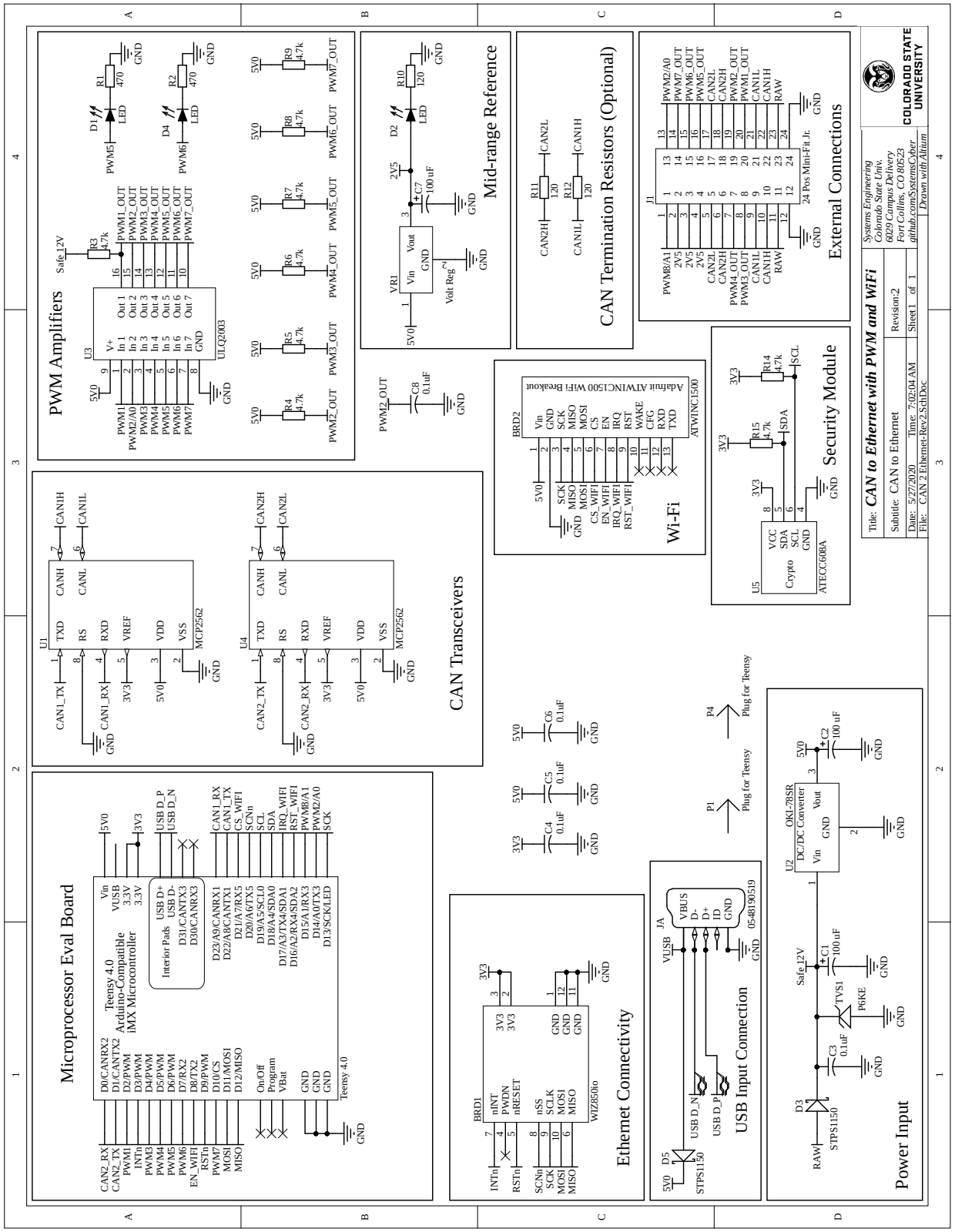

Figure 4.4.2: CAN to LAN Evaluation Schematic

Qty	Description	Supplier	Supplier Part Number	URL
1	Teensy 4.1	PJRC	TEENSY41	[20]
1	Beagle Bone Black			
1	Quadrature Encoder with breakout board. Bournes PEC09			

Table 4.4.2: Advanced Hardware Kit

Qty	Description	Supplier	Supplier Part Number	URL
1	Smart Sensor Simulator 2	DG Technologies	SSS2	[21]
1	Vehicle Diagnostics Adapter	DG Technologies	DPA5 Pro	
1	ECU	??		

Table 4.4.3: Vehicle ECU Testing Kit

area. Sn63Pb37 0.020" (0.50mm) wire is about right for many electronic assemblies. Flux cored wire helps.

Solder flux A flux pen is helpful for surface mount parts to get good solder flow and adhesion.

Solder braid Solder wick or braid helps remove excess solder. This is really helpful when hand soldering surface mount parts and the legs of the component become bridged with excess solder.

Tweezers A necessary tool to help place surface mount parts.

Wire Cutters Flush cutting diagonal cutters, like the Hakko CHP-170, are nice.

Wire Strippers When working with electronics, smaller gauge strippers are helpful. The Hakko CSP-30-1 can strip 20 to 30 gauge wire.

Carrying Tote The Flambeau Utility Tote is helpful for carrying around the small parts in the base of the tote and an open top to hold bulkier project items.

30 gauge Wire Thin wire is necessary to connect to the small legs of integrated circuits and making circuit board repairs. It's also used to connect the internal USB of a Teensy to an external jack.

4.4.3 Advanced Hardware Kits

4.5 Electronic Hardware Exercises

The first thing to do once the hardware is put together is to test the circuit. These tests are designed to ensure proper connectivity, sufficient power, and completeness. Building the testing skills is critically important in understanding and troubleshooting new designs. These tests help create an idea of what to expect and lay the foundation for more advanced harware reverse engineering skills. When examining and testing a piece of hardware for a truck, there are some basic assumptions. Inevitably there will be a time when something

[20] https://www.pjrc.com/store/teensy41.html
[21] https://www.pjrc.com/store/teensy41.html

doesn't work as expected. A sound strategy is to always examine and test your assumptions. These basics are necesary to perform advanced troubleshooting or functional testing. The following is a list of common assumptions and how to test them:

1. Power is applied to the circuit. This power is often measured on designate testpoints or pins known in the schematic. Use a multi-meter to measure the 5 volt and 3.3 volt circuits. Voltage measurements are always relative, so the ground (black) probe is connected to ground and the red probe is connected to the voltage to be measured.
2. Key is on. The ignition signal is often required on heavy vehicle ECUs to enable the regulator. The key switch signal is often a +12 v signal that can be measured with a multimeter. Also, CAN traffic is present when the ignition signal is present.
3. Each chip has power and ground connected. This can be tested with the multimeter.
4. The CAN network is terminated properly. There should be two 120 ohm resistors on each end of the network. However, the CAN is fairly tolerant of deviations from this when working on a desktop system. However, there must be a resistor between CAN Hand CAN L. This can be measured with an ohm meter when power to the circuit is off.
5. The CAN Transceivers are powered and connected. This can be tested with a volt meter as the CAN H and CAN L are both around 2.5V to ground when there is no CAN traffic. When CAN traffic is present, the CAN H line will go above 2.5 V average and CAN L will drop below 2.5 V DC.
6. Check the continuity of each hookup wire before using it. They can be fragile and break easily. Even though it looks like the wires are connected, if there is a bad hookup wire, the assumption of continuity is violated.

4.5.1 Getting Started in the Ardunio IDE with Teensy

These exercises should serve to get you "up to speed" on Arduino hardware. Little to no computer programming experience is needed for this section. Most of the code is based on examples or lightly modifying the examples. You should be able download and install Arduino and Teensyduino (in that order) before starting this section.

Exercise 4.4. [Remember] Blink Without Delay

This sketch should be able to turn on and off an LED without using any delay functions. This enables you to keep a schedule for events based on time. Since this may be one of your first sketches, we will show some screenshots to ensure you have the correct example and configuration. The screenshot in Figure 4.5.1 on the following page shows how to select the example. The screenshot in Figure 4.5.2 on the next page shows how to select the board that matches your hardware. Once the sketch is open, the board is selected, and the USB is plugged in, select Sketch, then Upload to compile and upload the program. The LED built into the Teensy 4.0 should blink on 1 second itervals.

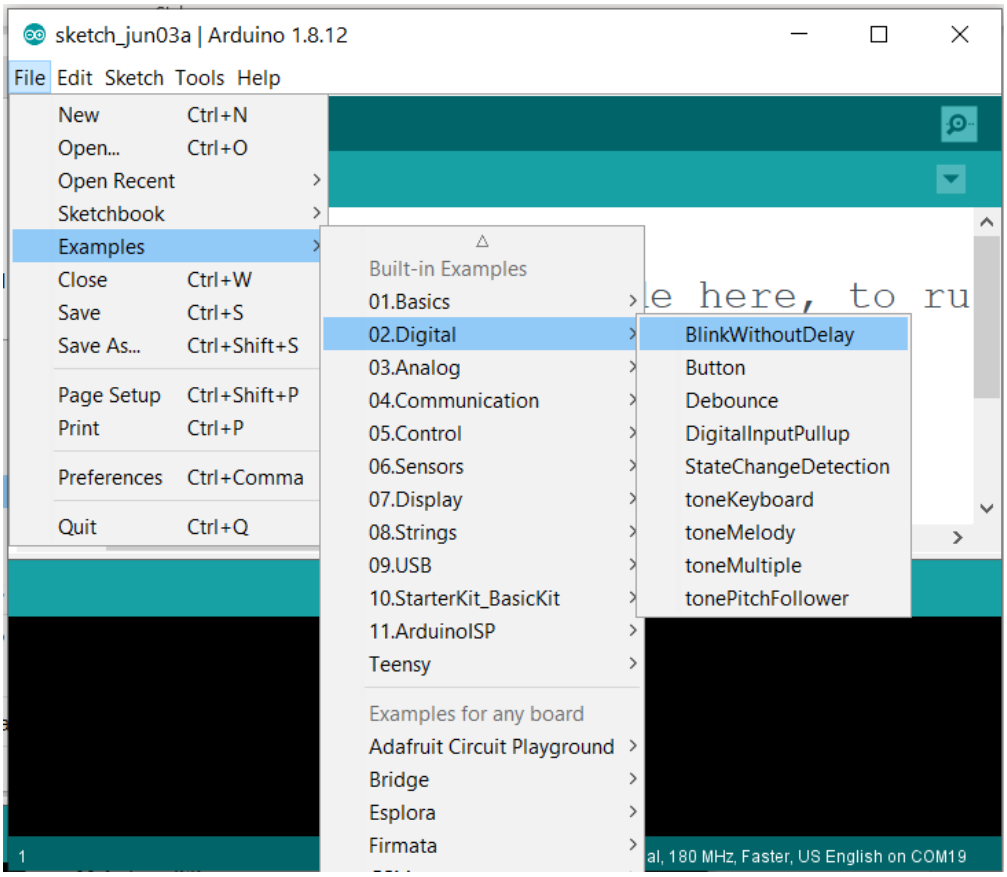

Figure 4.5.1: Opening the Blink Without Delay example in Arduino

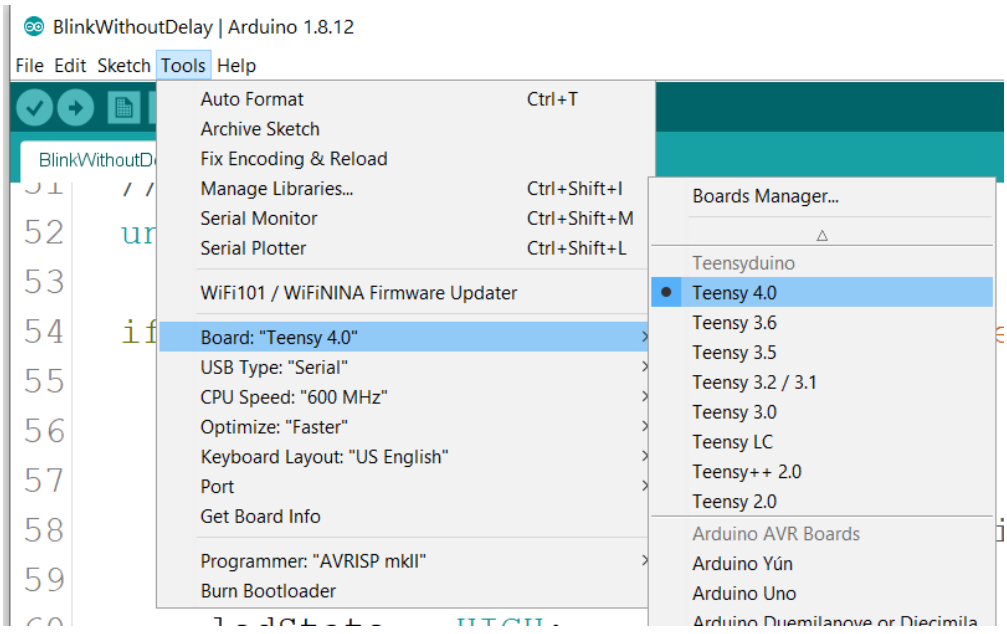

Figure 4.5.2: The Arduino environment must know the type of board it is connected to. In this case, select the Teensy 4.0.

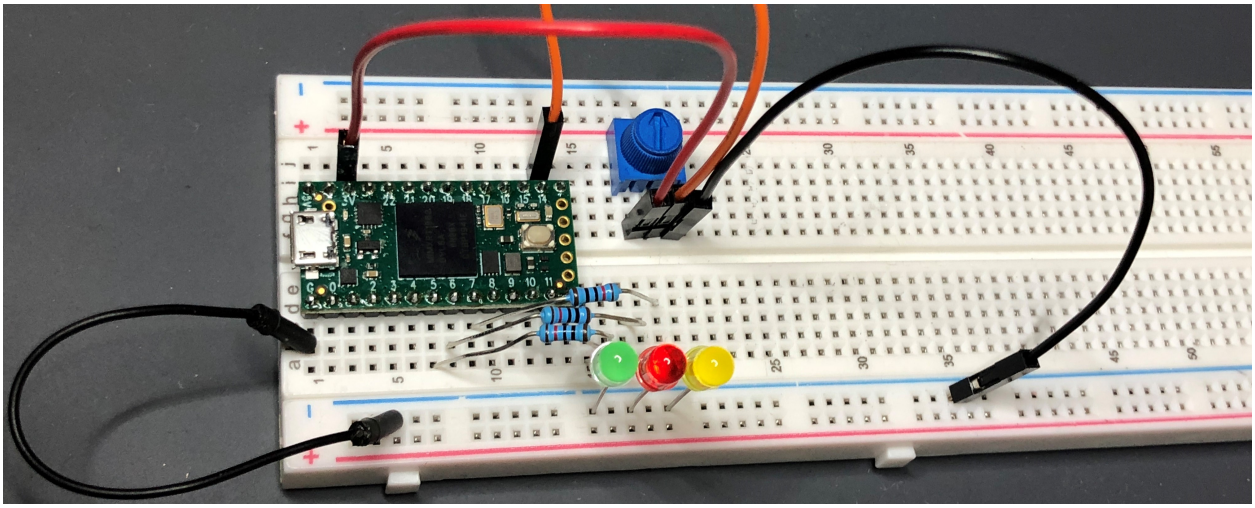

Figure 4.5.3: Example breadboard connections with a Teensy 4.0, LEDs with 120-ohm resistors, and a potentiometer.

Exercise 4.5. [Understand] Read the documentation on the elapsedMillis function and convert the Blink Without Delay to use elapsedMillis.

Exercise 4.6. Add the external discrete LEDs in series with a 120-ohm resistor and ensure you can blink them without using a delay function. Connect your LEDs to pins D5, D6, and D7 as shown in Figure 4.5.3. In this exercise, you will have declare pinModes and set new pins to outputs. Use the PWM tutorial in Arduino shown in Figure 4.5.4 on the next page as a basis to get started.

1. [Understand] Explain why a resistor is needed in series with the LED. What would happen if the
2. [Understand] Draw a sketch of the waveform being generated from the redPin in you circuit. Confirm your sketch with an oscilloscope trace (if available).
3. [Analyze] Determine the default frequency of the pulse width modulated (PWM) signals.
4. [Apply] Change the analogWriteFrequency to 2, 20, 200, 2000, and 20,000 cycles per second (Hz). Observe the visual effect on the LEDs.

Exercise 4.7. Connect the center of the trim potentiometer to Pin A0 (D14), one side to 3.3V, and the other side to ground as shown in Figure 4.5.3. Note, the pins on the Teensy 4.0 are not 5V tolerant, so you cannot use the +5V supply as suggested by the Arduino example.

1. [Understand] Run the AnalogInOutSerial example sketch in the Analog Built-in Example. Explain how and why the map() function is used.
2. [Analyze] What would happen if the map function was not used?

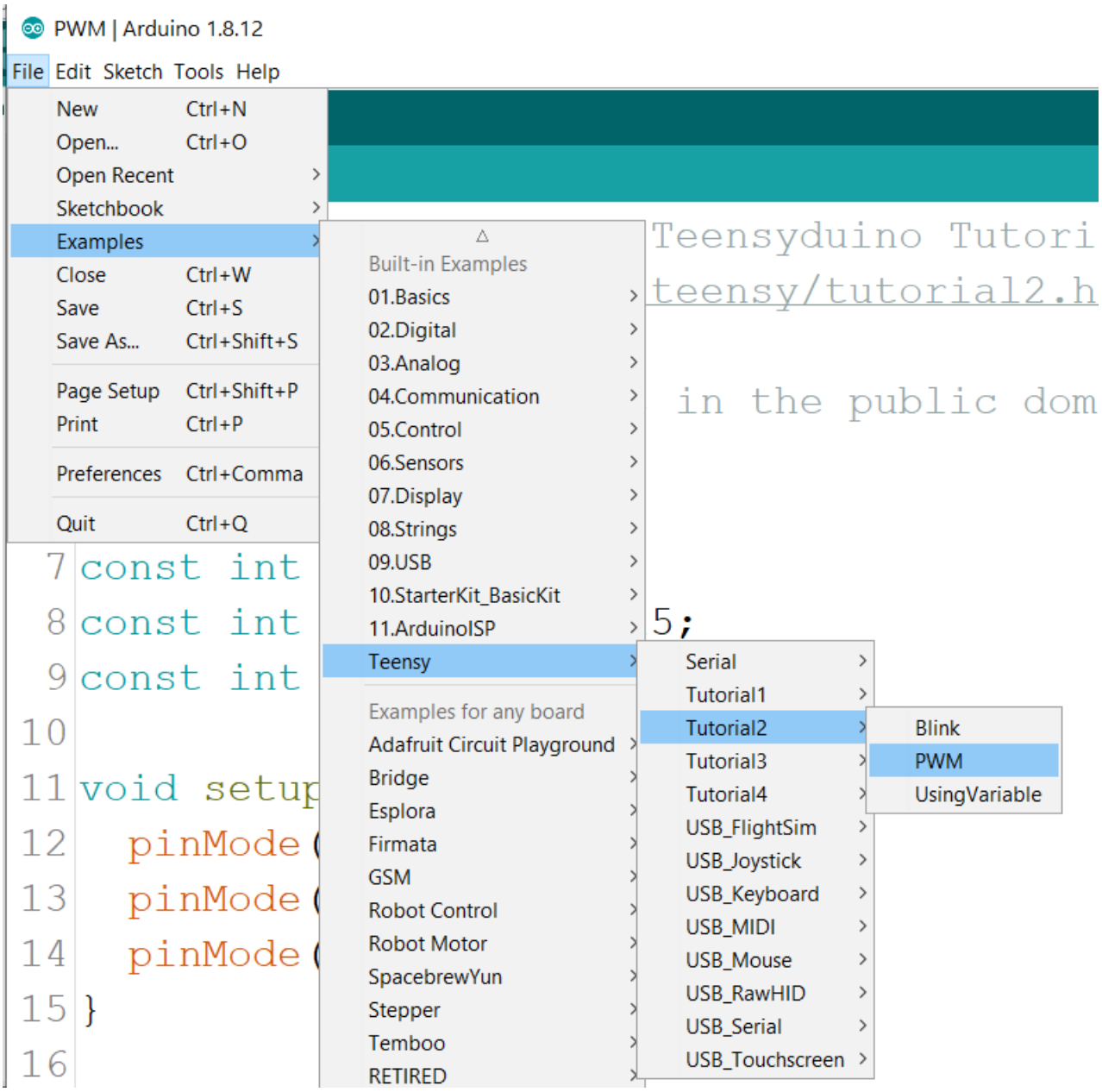

Figure 4.5.4: Finding the Teensy PWM tutorial.

3. [Apply] Convert the example sketch to update the serial output 50 times per second using the elapsedMillis function.
4. [Apply] Control brightness of the LEDs with the potentiometer.
5. [Create] Convert your potentiometer-based circuit into a PWM-based truck accelerator pedal.

4.5.2 Intro to CAN with Teensy and Arduino

Exercise 4.8. Connect two MCP2562 CAN Transcievers to the Teensy board. Connect the two transceivers together so they can communicate. Be sure to add terminating resistors.

1. [Apply] Using the FlexCAN_T4 library, write a program to send CAN messages from one node to another and display the result. Confirm it works by changing your code to see the output in the serial monitor change.
2. [Apply] Encapsulate the output value for the LEDs in a CAN message. For example, if you have 3 LEDs, we can define a 3 byte CAN message that broadcasts the LED status every 100 milliseconds. You can choose and CAN ID you want for the output value.

Exercise 4.9. Send the potentiometer value from one CAN channel to another.

1. [Apply] On the receiving channel, act on that potentiometer value by changing the brightness of the LEDs.
2. [Analyze] How fast can you update the LED brighness based on the potentiometer using CAN?
3. [Evaluate] How fast do you need to update the brightness of the LED?

Exercise 4.10. Connect your breadboard Teensy CAN to a truck or truck testbed and read J1939 messages.

1. [Remember] What is the J1939 bit rate for your setup?
2. [Remember] How many bits are used for the J1939 Arbitration ID?
3. [Analyze] How many different CAN IDs are used?
4. [Apply] Convert the output of your serial monitor to match the Linux SocketCAN and the can-utils candump format. Hint: use the Serial.printf function for formatting. The candump format has the channel designator followed by the can ID follwoed by the hash symbol and then the bytes of data. All CAN bytes are in hex format.

5 Truck Systems for Computer Scientists

6 Computer Programming Fundamentals (Computer Programming for Engineers)

Computer programming may seem like a daunting field for beginner programmers. It does not need to be. At its core, computer programming is providing a set of instructions to a computer. Each line of code contains one or more instructions. Sometimes instructions interfere with each other and cause errors. Imagine telling someone who has never seen a sandwich before how to make a peanut butter and jelly sandwich. The steps may be:

1. Get Bread, Peanut Butter, Jelly, Knife, and Plate.
2. Place plate.
3. Place two pieces of bread side by side on plate. Call them Slice A and Slice B.
4. Put peanut butter on knife.
5. Use knife to spread peanut butter on Slice A.
6. Clean knife.
7. Put jelly on knife.
8. Use knife to spread jelly on Slice B.
9. Combine Slice A with Slice B by placing Slice B on Slice A.
10. Eat Sandwich.
11. Clean Knife
12. Clean Plate

Provided no external changes, this set of instructions will always lead to a proper peanut butter and jelly sandwich. However, some people may argue about if you should use grape jelly or strawberry jelly, chunky or smooth peanut butter. Some people are adamant that you must put Jelly on Slice A, never peanut butter. Some argue that if one starts with a loaf of bread instead of slices then it produces a better sandwich. These options represent the different ways that people program. There are many different sets of instructions that produce peanut butter and jelly sandwiches. Programming is very smiliar. There are many different ways to code a program that will lead to the same result.

However, there are ways which are more efficient than others, and there are ways to make code easier to understand. When writing code, it is important to write with a consistent programming style that allows others to understand your code. Code must be maintained over time, and may transition between programmers.

This chapter should serve to introduce the basics of computer programming as a platform for the Cyber Truck Experience program.

6.1 Arduino Programming Language

6.2 Python 3 Programming Language

An important tool in any hackers toolkit is the ability to interact with collected data. In this section, we will be using a programming language called Python to help with this.

6.2.1 Background

Python is a general purpose, versatile, and popular programming language. It is great as a first language because it is concise and easy to read. It is also a good language to have in any programmer's stack as it can be used for everything from web development to software development and data science applications.

This is different than what we have done to this point in Arduino. We will generally use the Arduino devices to directly interface with systems. These devices will then send data back to a computer or other device running a python script.

6.2.2 Codecademy

We will begin by having you learn the basics of python. We will be using an external system to aid in this. Navigate over to hereand create an account using your email and fill out any information they require. Please ensure that you do not pay for this service at this time. We will only need the services that the free version offers.

Please complete this course. Take notes throughout the course over functions that may be useful (importing and exporting files, searching through lists, data structures, etc.) Periodically, there will be additional trial material to supplement your learning.

6.2.3 Editors and IDE

6.2.4 Threading

6.2.5 PySerial

6.2.6 matplotlib

6.2.7 Making GUIs with PyQt5

6.3 C++ Programming Language

6.3.1 Codecademy

6.4 Computer Programming Fundamentals Exercises

7 Serial Communication with SAE J1708 and J1587

Exercise 7.1. CAN Frame Decoding

Capture a CAN Frame using an oscilloscpope on a J1939 network and decode it according to SAE J1939.

Exercise 7.2. Read Live Engine RPM Challenge

Using a BeagleBone Black and a Truck Cape, connect to an engine contoller that is broadcasting non-zero engine RPM. Gather this data using candump. Interpret the raw CAN frames and extract information for Engine RPM, or J1939 SPN 190. Plot 20 seconds of changing RPM with matplotlib. Print the properly labeled plot to PDF and show it to your instructor. Objectives

Learn how to interface with Linux SocketCAN and can-utils Be able to look up a signal definition in the J1939 Digital Annex (spreadsheet) Use grep to search for specific strings from a candump Have a reliable CAN datalogger for use in future projects Plot data using matplotlib in Python.

Suggested Materials

This exercise can be run with any Linux device with CAN hardware. An example of a commercial product with these features is the DG Technologies' Beacon device. An example of a hand built project is the BeagleBone Black with a TU TruckCape. Resources

J1939DA Internet Access (you may want to share your PC's connection sharing)

Exercise 7.3. Man in the Middle

Build a man-in-the middle board and box that takes CAN signals into one can channel and sends them out on another. Start a diagnostics session with a PC and RP1210 device to perform maintenance. Create a forwarding system that inspects and forwards network traffic in both directions. Attempt to hijack a a diagnostic session and affect a parameter change started with the PC diagnostics software.

Using DDEC Reports, try to prevent resetting the CPC clock during a data extraction on a CPC.

8 CAN Communication with SAE J1939

8.1 Objectives

- Learn how to interface with Linux SocketCAN and can-utils
- Be able to look up a signal definition in the J1939 Digital Annex (spreadsheet)
- Use grep to search for specific strings from a candump
- Have a reliable CAN datalogger for use in future projects
- Plot data using matplotlib in Python.

8.1.1 Online Learning Resources

https://www.kvaser.com/about-can/higher-layer-protocols/j1939-introduction/

8.1.2 Suggested Hardware Materials

This exercise can be run with any device with CAN hardware. An example of a commercial product is the DG Technologies' Beacon device. An example of a hand-built project is the BeagleBone Black with a TruckCape.

8.1.3 Software Resources

You will need access to a database that containse the meaning for the CAN messages. The SAE J1939 digital annex is a spreadsheet version of the J1939-71 document, which contained application layer meaning for the parsed data. The SAE J1939-21 document contains information needed to decode and parse the J1939 Transport Protocol (TP) messages.

The exercises will use Python with access to information needed to decode a J1939 message. This information could be specific knowledge on the just a few messages, or the entire database. The source of the decoding information is usually the digital annex of SAE J1939.

8.2 SAE J1939 CAN Frame Format

J1939 message use the CAN 2.0 specification that enables 29-bit arbitration identifiers. These identifiers are used to explain the meaning of the data in the CAN frame. In SAE J1939-21, the J1939 message ID structure is defined and summarized in Table 8.2.1 on the next page. Messages in J1939 are categorized into four different types: 1) Broadcast, 2) Request, 3) Command, 4) Acknowledgement. By far, the most common message is a broadcast message. The network behavior is setup to work in two different ways: 1) point-to-point messages and 2) broadcast messages. Even though the copper wires and voltage signaling are seen by all the ECUs on the network, a point-to-point message in J1939 is intended for messages to be used between just 2 nodes, or a 1-to-1 messaging system. Broadcast messages are 1-to-many in nature with no specific destination in mind. An example of a broadcast message is the Cruise Control/Vehicle Speed message that reports the wheel-based vehicle speed. Usually the engine controller sends this message out to the entire network. An instrument cluster on the network would consume this message an display the speed on the speedometer. A telematics device could report the speed back to the main office, which means the same message was used by multiple modules. An example for a point-to-point message, on the other hand, would be a torque control command message from the brake controller to the engine controller instructing the engine to reduce torqe output when the brakes are trying to stop the vehicle. Even though every node sees the message, only the engine controller can act on it. Using functionality as a criteria for broadcast vs

The J1939 message formats are defined as protocol data units (PDUs). There are two formats that correspond to the point-to-point messaging or broadcast message. PDU1 message formats are used for point-to-point messaging and must contain a source address and a destination address. These two data elements are shown as the Destination Address (DA) and Source Address (SA) in the PDU1 format of Table 8.2.1 on the following page. The PDU2 messages are broadcast to the network without regard to the recipient. This means their destination address is always the global address, which is defined as 0xFF or 255 in decimal. Since this is always the case for a broadcast message, there is no need to use the destination address field. This enables the PDU2 format to include the PDU Specific (PS) field and use 18 bits to define the parameter group number (PGN). The PGN for PDU1 still uses 18 bits, but only the upper 10 bits are used to denote values as the lower 8 bits are always set to zero.

Exercise 8.1. [Evaluate] Show that the number of PGNs available in J1939 based on the two different PDUs is 17,344. (Hint: if the Data Page and Extended Data Page bits are omitted, then there are only 4336 PGNs.)

Table 8.2.1 on the next page includes an example of a PDU 1 message and an example of a PDU 2 message. The table contains the bit positions across the top row that are indexed from zero. The presentation of the bits in the table from left to right is the order in which they are recieved on the network. In other words, the bit in postion 28 appears on the nework first. This means the first three bits recieved by an ECU are the priority bits. These are designated by the network designers to create priorities for arbitration should any two nodes try to transmit at the same time. The lowest numerical value will have the highest priority. Since there are 3 bits available, there are a total of $2^3 = 8$ priority levels, usually noted as 0-7. To determine the priority using a computer program, we need to extract the portion of the message ID corresponding to the

Bit	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Byte	Priority			EDP	DP	PDU Format								PDU Specific								Source Address							
PDU1	Priority			PGN (10 Most Signficant Bits)										Destination Address								Source Address							
Binary	1	1	1	0	0	1	1	1	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0	1	1
Hex	1			C		E				C				0				3				0				B			
PDU2	Priority			Parameter Group Number (18 bits)																		Source Address (8 bits)							
Binary	1	1	0	0	0	1	1	1	1	1	1	1	0	1	1	1	1	0	0	0	1	0	0	1	1	0	0	0	1
Hex	1			8		F				E				F				1				3				1			

Table 8.2.1: CAN Message Identifier for J1939. The Protocol Data Unit (PDU) is the defintion of the CAN frame.

Bit	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Byte	Priority			EDP	DP	PDU Format								PDU Specific								Source Address							
Message Binary	1	1	0	0	0	1	1	1	1	1	1	1	0	1	1	1	1	0	0	0	1	0	0	1	1	0	0	0	1
Message Hex	1			8		F				E				F				1				3				1			
Mask Binary	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Mask Hex	1			C		0				0				0				0				0				0			
AND Result	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AND Result	1			8		0				0				0				0				0				0			

Table 8.2.2: An example of using the AND boolean operation to mask the priortity bits of a J1939 message.

priority and then perform a right shift until the priority fiels is represented in bit positions 0, 1, and 2. This procedure is called *masking* and *bit-shifting*.

8.2.1 Masking

The process of masking is the process of applying a boolean AND operation to select only the bits of interest. A logic AND operation says that when both logic values are true then the result is true. If any arguments are false, then the result of an AND is false. The application is a programmer can block any bits using a 0 and accept any bits using a 1. A logic AND will keep the values of the passing bits. If we apply this logic to the examples in Table 8.2.1, we could isolate the bits responsible for the different fields in the J1939 message ID. The example in Table 8.2.2 shows the process of performing the AND operation on an example message. The mask is 0x1C000000 and the message is 0x18FEF131. The result of the AND operation is 0x18000000.

8.2.2 Bit Shifting

Once the bit fields are masked, the value needs to be determined. However, the position of the masked bits are retained. If we converted the masked result shown in Table 8.2.2 directly to a value, would get a priority of 402,653,184, which is clearly not an allowable value for the priority. For the computer to calculate the value, the binary needs to be shifted to the right 26 positions. This means the binary of 0b0001 1000 0000 0000 0000 0000 0000 0000 becomes 0b0110, which is the integer of 6. This makes for a valid priority. After

working J1939 messages, you will come to recognize that any message that starts with a 0x18 will have a priority of 6.

Implementing a mask followed by a bit shift in a Python computer program makes use of so-called bit-wise operators (AND, OR, XOR) and bit shifts as shown in the following code snippet.

```
#!/usr/bin/env python3
id = 0x18FEF131
PRIORITY_MASK = 0x1C000000
priority = (id & PRIORITY_MASK) >> 26
print(priority)
```

Implementing the same process in a complete Arduino sketch would be the following:

```
// Arduino (Teensy) example sketch to mask and bit-shift
#define PRIORITY_MASK 0x1C000000

void setup(){
  // This is empty for this example
}

void loop(){
  uint32_t id = 0x18FEF131; // Normally this would be from CAN
  // Mask and bit shift
  uint8_t priority = (id & PRIORITY_MASK) >> 26;
  Serial.println(priority);
  delay(1000); // Wait 1 second
}
```

In the Arduino sketch, the parentheses matter because the bit shift operator takes precedence over the AND operator. This means

```
uint8_t priority = id & (PRIORITY_MASK >> 26);
```

and

```
uint8_t priority = id & PRIORITY_MASK >> 26;
```

would both evaluate to the same thing. In this case the output is 1 because the mask is bit shifted before the AND operation takes place. Always be sure to include the parentheses around the masking operation first.

8.2.3 Parsing a J1939 Identifier

Given the example in Table 8.2.2 on the previous page, we needto determine the following:

- Message Priority
- Parameter Group Number
- Destination Address
- Source Address

To find these, we need to pull out the necessary elements using bitshifting and masking while determining if the message is a PDU1 or PDU2.

8.2.3.1 Using Arduino or C++

A complete Arduino Sketch to perform this operation is listed in Listing 8.1 on the following page. This listing starts as many programs do with defined constants. The #define tag simply replaces everywhere the name on the left is found with the value on the right. This way, there are no uses of the so-called magic numbers in your code. Every constant is defined and has a name. This makes code much more readable. Notice the #defines do not need a semicolon, whereas the other lines in C/C++ do need them.

The serial monitor output for this sketch is repeating every second.

```
1 Priority, PGN, DA, SA
2 6, 65265, 255, 49
3 7, 60416, 3, 11
```

This example computed the results from parsing the CAN identifier according to J1939. We went from one 29-bit word to four data parameters.

8.2.3.2 Parsing a J1939 message in Python

Often a data file with CAN is available to parse that came from a J1939 enabled network. Let's build a function that can parse the CAN ID based on J1939. The output will be a Python dictionary, which is useful to be used in many other contexts. The Python listing is shown in Listing 8.2 on page 43.

The program starts with a comment statement known in the unix world as a "she-bang." This symbol enables a shell to open the correct interpreter to run the file. In this case, we want to use the python3 interpreter. This line has no effect when running on Windows. The next lines setup defined constants. Python is an interpreted language, so it doesn't have compiler primitves like #define. Instead, each contant is assigned to a variable name at the top level (i.e. a global constant) and it can be used in all the functions. The first function is called main(). This function is called by the last line. Notice the second to last line tests to see if the name assigned to the module is called __main__. This only happens when the script is run as the top level script (i.e. it is called directly). We can use this feature for testing testing the script, but don't have to worry about main() executing if we import the module to use the intended function, which is parseJ1939id().

The printed results from the Python script in Listing 8.2 on page 43 is as follows:

Listing 8.1: Arduino C++ code to demonstrate parsing two types of J1939 Protocol Data Units.

```
// Arduino (Teensy) Example Sketch to Parse J1939 IDs
#define PRIORITY_MASK          0x1C000000
#define PDU_FORMAT_MASK        0x00FF0000
#define PDU_SPECIFIC_MASK      0x0000FF00
#define SOURCE_ADDRESS_MASK    0x000000FF
#define DATA_PAGE_MASK         0x01000000
#define EXT_DATA_PAGE_MASK     0x02000000
#define PDU1_PGN_MASK          0x03FF0000
#define PDU2_PGN_MASK          0x03FFFF00
#define PRIORITY_BIT_SHIFT     26
#define PDU_FORMAT_BIT_SHIFT   16
#define PDU2_THRESHOLD         240
#define PGN_BIT_SHIFT          8
#define DEST_BIT_SHIFT         8
#define GLOBAL_SOURCE_ADDR     0xFF

//Create a function to parse J1939 CAN IDs
void parse_j1939_id(uint32_t id,
                    uint32_t &pgn,
                    uint8_t &da,
                    uint8_t &sa,
                    uint8_t &priority){
    sa = id & SOURCE_ADDRESS_MASK;
    priority = (id & PRIORITY_MASK) >> PRIORITY_BIT_SHIFT;
    uint8_t pf = (id & PDU_FORMAT_MASK) >> PDU_FORMAT_BIT_SHIFT;
    if (pf < PDU2_THRESHOLD) { // PDU 1 format uses values lower than 240
        da = (id & PDU_SPECIFIC_MASK) >> DEST_BIT_SHIFT;
        pgn = (id & PDU2_PGN_MASK) >> PGN_BIT_SHIFT;
    }
    else { // PDU 2 format
        da = GLOBAL_SOURCE_ADDR;
        pgn = (id & PDU1_PGN_MASK) >> PGN_BIT_SHIFT;
    }
}

void setup(){
}

void loop(){
    uint32_t pgn; // Parameter Group Number
    uint8_t da;  // Destination Address
    uint8_t sa;  // Source Address
    uint8_t priority;

    //Normally this would come from CAN
    uint32_t user_input_id = 0x18FEF131;
    parse_j1939_id(user_input_id, pgn, da, sa, priority);
    Serial.printf("%d,%d,%d,%d\n",priority,pgn,da,sa);

    user_input_id = 0x1CEC030B;
    parse_j1939_id(user_input_id, pgn, da, sa, priority);
    Serial.printf("%d,%d,%d,%d\n",priority,pgn,da,sa);
    delay(1000);
}
```

Listing 8.2: Python program to demonstrate parsing a 29-bit CAN ID based on J1939.

```
#!/usr/bin/env python3
PRIORITY_MASK        = 0x1C000000
PDU_FORMAT_MASK      = 0x00FF0000
PDU_SPECIFIC_MASK    = 0x0000FF00
SOURCE_ADDRESS_MASK  = 0x000000FF
DATA_PAGE_MASK       = 0x01000000
EXT_DATA_PAGE_MASK   = 0x02000000
PDU1_PGN_MASK        = 0x03FF0000
PDU2_PGN_MASK        = 0x03FFFF00

PRIORITY_OFFSET     = 26
PDU_FORMAT_OFFSET   = 16
DA_OFFSET           = 8
PGN_OFFSET          = 8
PDU2_THRESHOLD      = 240
GLOBAL_SOURCE_ADDR  = 0xFF

def main():
    print(parseJ1939id(0x18FEF131))
    print(parseJ1939id(0x1CEC030B))

def parseJ1939id(id):
    sa = id & SOURCE_ADDRESS_MASK
    priority = (id & PRIORITY_MASK) >> PRIORITY_OFFSET
    pf = (id & PDU_FORMAT_MASK) >> PDU_FORMAT_OFFSET
    if (pf < PDU2_THRESHOLD):  # See SAE J1939-21
        # PDU 1 format uses values lower than 240
        da = (id & PDU_SPECIFIC_MASK) >> DA_OFFSET
        pgn = (id & PDU1_PGN_MASK) >> PGN_OFFSET
    else:          # PDU 2 format
        da = GLOBAL_SOURCE_ADDR
        pgn = (id & PDU2_PGN_MASK) >> PGN_OFFSET
    return {'sourceAddress': sa,
            'priority': priority,
            'destinationAddress': da,
            'PGN': pgn}

if __name__ == '__main__':
    main()
```

```
1 {'source_address': 49, 'priority': 6, 'destination_address': 255, 'PGN': 65024}
2 {'source_address': 11, 'priority': 7, 'destination_address': 3, 'PGN': 60419}
```

The next step is to figure out what how to derive meaning for those four numbers. Specifically, we will label the addresses and PGNs with English names.

8.2.4 Source and Destination Addresses

The source addresses for J1939 are standardized by SAE. The Javascript Object Notation (JSON) in Listing 8.3 on the next page and Listing 8.4 on page 46 creates a dictionary lookup map the numerical values of the source or destination address to a description in English. The keys of the dictionary are the decimal values of the source address. These are the same for the destination address as well.

8.2.5 Using a Parameter Group Number (PGN)

Once the PGN is known based on the CAN ID, we can look up what the message data contains based on Suspect Parameter Numbers (SPNs). The SAE J1939-71 document contains many maps from PGN to SPN. Let's see how this works with an example. A log of CAN data was collected from a truck that went through a startup sequence, then the driver depressed the accelerator pedal and sped up the engine. Finally the engine was turned off. The data was collected using an embedded Linux device called the BeagleBone Black with a Truck Cape. The data is in a raw text form with the following format:

```
(unix timestamp) channel CAN_ID [N] B0 B1 ... BN
```

In this entry the unix timestamp is the number of seconds from the unix epoch, which is midnight on January 1, 1970 UTC. The channel is based on the device settings in Linux. The CAN ID is in hexadecimal is 8 characters for 29-bit ids. The data length code (DLC) is denoted as N with N bytes of data following. There can be at most 8 bytes of data per CAN frame. An example from the text file is as follows:

```
(011.802193) can1 0CF00331 [8] FF FF FF FF FF FF FF FF
(011.822872) can1 18EEFF03 [8] 64 00 40 00 00 03 03 10
(011.840437) can1 18EAFF00 [3] EB FE 00
(011.842907) can1 18FE4A03 [8] 03 5F 4F FF FF F3 FF FF
(011.849202) can1 0CF00331 [8] FF FF FF FF FF FF FF FF
(011.852511) can1 18EEFF00 [8] F4 B8 4E 01 00 00 00 00
(011.853073) can1 18EEFF29 [8] F4 B8 4E 01 00 0C 00 00
(011.861682) can1 18F0010B [8] CF FF F0 FF FF DC FF FF
(011.871828) can1 18FEBF0B [8] 00 00 7D 7D 7D 7D FF FF
(011.877482) can1 18FEDF00 [8] 7D A0 28 7D 7D FF FF FC
(011.879632) can1 1CECFF31 [8] 20 17 00 04 FF EB FE 00
(011.888282) can1 0CF00400 [8] 0E 7D 7D 00 00 FE 00 7D
```

Listing 8.3: J1939 Source and Destination Addresses

```
{
    "0": "Engine #1",
    "1": "Engine #2",
    "2": "Turbocharger",
    "3": "Transmission #1",
    "4": "Transmission #2",
    "5": "Shift Console - Primary",
    "6": "Shift Console - Secondary",
    "7": "Power TakeOff - (Main or Rear)",
    "8": "Axle - Steering",
    "9": "Axle - Drive #1",
    "10": "Axle - Drive #2",
    "11": "Brakes - System Controller",
    "12": "Brakes - Steer Axle",
    "13": "Brakes - Drive axle #1",
    "14": "Brakes - Drive Axle #2",
    "15": "Retarder - Engine",
    "16": "Retarder - Driveline",
    "17": "Cruise Control",
    "18": "Fuel System",
    "19": "Steering Controller",
    "20": "Suspension - Steer Axle",
    "21": "Suspension - Drive Axle #1",
    "22": "Suspension - Drive Axle #2",
    "23": "Instrument Cluster #1",
    "24": "Trip Recorder",
    "25": "Passenger-Operator Climate Control #1",
    "26": "Alternator/Electrical Charging System",
    "27": "Aerodynamic Control",
    "28": "Vehicle Navigation",
    "29": "Vehicle Security",
    "30": "Electrical System",
    "31": "Starter System",
    "32": "Tractor-Trailer Bridge #1",
    "33": "Body Controller",
    "34": "Auxiliary Valve Control or Engine Air System Valve Control",
    "35": "Hitch Control",
    "36": "Power TakeOff (Front or Secondary)",
    "37": "Off Vehicle Gateway",
    "38": "Virtual Terminal (in cab)",
    "39": "Management Computer #1",
    "40": "Cab Display #1",
    "41": "Retarder, Exhaust, Engine #1",
    "42": "Headway Controller",
    "43": "On-Board Diagnostic Unit",
    "44": "Retarder, Exhaust, Engine #2",
    "45": "Endurance Braking System",
    "46": "Hydraulic Pump Controller",
    "47": "Suspension - System Controller #1",
    "48": "Pneumatic - System Controller",
```

Listing 8.4: J1939 Source and Destination Addresses (continued)

```
    "49": "Cab Controller - Primary",
    "50": "Cab Controller - Secondary",
    "51": "Tire Pressure Controller",
    "52": "Ignition Control Module #1",
    "53": "Ignition Control Module #2",
    "54": "Seat Control #1",
    "55": "Lighting - Operator Controls",
    "56": "Rear Axle Steering Controller #1",
    "57": "Water Pump Controller",
    "58": "Passenger-Operator Climate Control #2",
    "59": "Transmission Display - Primary",
    "60": "Transmission Display - Secondary",
    "61": "Exhaust Emission Controller",
    "62": "Vehicle Dynamic Stability Controller",
    "63": "Oil Sensor",
    "64": "Suspension - System Controller #2",
    "65": "Information System Controller #1",
    "66": "Ramp Control",
    "67": "Clutch/Converter Unit",
    "68": "Auxiliary Heater #1",
    "69": "Auxiliary Heater #2",
    "70": "Engine Valve Controller",
    "71": "Chassis Controller #1",
    "72": "Chassis Controller #2",
    "73": "Propulsion Battery Charger",
    "74": "Communications Unit, Cellular",
    "75": "Communications Unit, Satellite",
    "76": "Communications Unit, Radio",
    "77": "Steering Column Unit",
    "78": "Fan Drive Controller",
    "79": "Seat Control #2",
    "80": "Parking brake controller",
    "81": "Aftertreatment #1 system gas intake",
    "82": "Aftertreatment #1 system gas outlet",
    "83": "Safety Restraint System",
    "84": "Cab Display #2",
    "85": "Diesel Particulate Filter Controller",
    "86": "Aftertreatment #2 system gas intake",
    "87": "Aftertreatment #2 system gas outlet",
    "88": "Safety Restraint System #S",
    "89": "Atmospheric Sensor",
    "248": "File Server / Printer",
    "249": "Off Board Diagnostic-Service Tool #1",
    "250": "Off Board Diagnostic-Service Tool #2",
    "251": "On-Board Data Logger",
    "252": "Reserved for Experimental Use",
    "253": "Reserved for OEM",
    "254": "Null Address",
    "255": "GLOBAL (All-Any Node)"
}
```

PGN 61444 ***Electronic Engine Controller 1*** ***EEC1***

Engine related parameters

Transmission Repetition Rate:	engine speed dependent		
Data Length:	8		
Extended Data Page:	0		
Data Page:	0		
PDU Format:	240		
PDU Specific:	4	PGN Supporting Information:	
Default Priority:	3		
Parameter Group Number:	61444 (0x00F004)		

Start Position	Length	Parameter Name	SPN
1.1	4 bits	Engine Torque Mode	899
1.5	4 bits	Actual Engine - Percent Torque High Resolution	4154
2	1 byte	Driver's Demand Engine - Percent Torque	512
3	1 byte	Actual Engine - Percent Torque	513
4-5	2 bytes	Engine Speed	190
6	1 byte	Source Address of Controlling Device for Engine Control	1483
7.1	4 bits	Engine Starter Mode	1675
8	1 byte	Engine Demand – Percent Torque	2432

Figure 8.2.1: Data entry in SAE J1939 for the Electronic Engine Controller 1 message.

PGN 61444 (0xF004) is for the Electronic Engine Control 1 that contains the engine speed, which is shown as the bottom entry in the short section of the log above. This is a frequent message and is ubiquitus. It is responsible for displaying the engine speed on the instrument cluster. However, how do we translate the CAN message into some engineering value that is readable on an instrument or able to be plotted. Information from the SAE J1939-71 document as shown in Figure 8.2.1 and Figure 8.2.2 on the following page can help us decode the message. The complete J1939 database can be obtained from SAE.

In J1939, all positions are indexed from 1, whereas in C/C++ and Python, the starting index is zero. This is important when setting up constants for bit and byte positions in the different measurements. Our interest is in the engine speed, which is shown as SPN 190 in positions 4 and 5. The numbers in J1939 that are multiple bytes in length are represented in the big endian format, or reverse byte order. This means the least significant byte in the engine speed is in position 4 and the most significant byte is in postion 5. This is also known as the Intel format. This is opposite they way humans normally read numbers, which is from left to right. In Python, we can make use of the struct library. There is a format string for structs that enable multibyte numbers to be interpreted in the correct endianness. The listing of the format structures is available in the Python documentation.

Armed with the standard decoding strategy, we can now write a program to examine the log file for engine speeds. Listing 8.5 on page 49 is a complete example for creating the resulting plot of the engine speed shown in Figure 8.2.3 on page 51. When we look at Listing 8.5, you'll notice a carry over of the code in Listing 8.2 to parse CAN IDs into J1939 fields. We also had to add a function to parse the candump file format we used for our log file. This function relies on the structure of each line to produce all the parts of the CAN message. The data is returned as a dictionary and has the ability to handle data length entries less than 8. The main function is called when this program is run as itself. If the python file is imported into another program, then the main function would not be called, but the functions would be available in

SPN 190 ***Engine Speed***

Actual engine speed which is calculated over a minimum crankshaft angle of 720 degrees divided by the number of cylinders.

Data Length:	2 bytes	
Resolution:	0.125 rpm/bit, 0 offset	
Data Range:	0 to 8,031.875 rpm	Operational Range: same as data range
Type:	Measured	
Supporting Information:		
PGN reference:	61444	

Figure 8.2.2: Data entry in SAE J1939-71 for Engine Speed (SPN 190).

memory.

The main function starts with setting up empty lists to keep track of the data from the file. To run this script successfully, you will need access to the log file called KWTruck.txt.

8.3 Examples and Exercises

Exercise 8.2. [Apply] Read Live Engine RPM Challenge

Using the hardware of your choice, connect to an engine contoller that is broadcasting non-zero engine RPM. Gather this data in a text format where each line has the following elements:

1. Timestamp in the number of seconds from the Unix epoch
2. CAN Channel
3. CAN Arbitration Identifier in hexadecimal format
4. CAN message data length code (DLC)
5. CAN data in hex format.

Interpret the raw CAN frames and extract information for Engine RPM, or J1939 SPN 190. Plot 20 seconds of changing RPM with matplotlib. Print the properly labeled plot to PDF and show it to your instructor.

Exercise 8.3. [Evaluate] Deconstruct the voltage trace of a CAN frame recorded with an ocilloscope. Decode the signal into J1939 SPNs with values and units.

Exercise 8.4. Man in the Middle

Build a man-in-the middle board and box that takes CAN signals into one can channel and sends them out on another. Start a diagnostics session with a PC and RP1210 device to perform maintenance. Create a forwarding system that inspects and forwards network traffic in both directions. Attempt to hijack a a diagnostic session and affect a parameter change started with the PC diagnostics software.

Using DDEC Reports, try to prevent resetting the CPC clock during a data extraction on a CPC.

Listing 8.5: Python code to plot engine speed vs time for a CAN log.

```
#!/usr/bin/env python3
import struct
import matplotlib.pyplot as plt

PRIORITY_MASK       = 0x1C000000
PDU_FORMAT_MASK     = 0x00FF0000
PDU_SPECIFIC_MASK   = 0x0000FF00
SOURCE_ADDRESS_MASK = 0x000000FF
PDU1_PGN_MASK       = 0x03FF0000
PDU2_PGN_MASK       = 0x03FFFF00

PRIORITY_OFFSET    = 26
PDU_FORMAT_OFFSET  = 16
DA_OFFSET          = 8
PGN_OFFSET         = 8
PDU2_THRESHOLD     = 240

GLOBAL_SOURCE_ADDR = 0xFF
ENGINE_SA          = 0

CANDUMP_TIMESTAMP_ADDR = 0
CANDUMP_CHANNEL_ADDR   = 1
CANDUMP_ID_ADDR        = 2
CANDUMP_DLC_ADDR       = 3
DATA_START_ADDR        = 4

PGN_EEC1 = 61444
SPN190_SCALE = 0.125 #RPM/bit
SPN190_OFFSET = 0

def main():
    spn190_times = []
    spn190_values = []
    filename = 'KWTruck.txt'
    with open(filename,'r') as f:
        for line in f:
            # We knew this data file was from Linux using candump.
            j1939_frame = parse_candump_line(line)
            # Add the additional results from parsing the id
            j1939_frame.update(parseJ1939id(j1939_frame['id']))
            # PGN for electronic engine control 1 messsage
            if (j1939_frame['pgn'] == PGN_EEC1 and
                j1939_frame["source_address"] == ENGINE_SA):
                # Unpack the engine speed data in big endian format and
                # convert to engineering values
                rpm = (struct.unpack('<H',j1939_frame['data'][3:5])[0]
                         * SPN190_SCALE + SPN190_OFFSET)
                spn190_values.append(rpm)
                # Include the timestamp for time series data
                spn190_times.append(j1939_frame['timestamp'])
    #Plot the engine speed
    plt.plot(spn190_times,spn190_values,'-',label="Engine RPM")
    plt.xlabel("Time (sec.)")     plt.ylabel("Engine Speed (RPM)")
    plt.title("Engine Speed for {}".format(filename))
    plt.grid()
    plt.legend()
    plt.show()
    plt.savefig('EngineSpeedGraphFrom{}.pdf'.format(filename))
```

Listing 8.6: Python code to plot engine speed vs time for a CAN log (cont).

```
1
2  def parseJ1939id(id):
3      sa = id & SOURCE_ADDRESS_MASK
4      priority = (id & PRIORITY_MASK) >> PRIORITY_OFFSET
5      pf = (id & PDU_FORMAT_MASK) >> PDU_FORMAT_OFFSET
6      if (pf < PDU2_THRESHOLD):  # See SAE J1939-21
7          # PDU 1 format uses values lower than 240
8          da = (id & PDU_SPECIFIC_MASK) >> DA_OFFSET
9          pgn = (id & PDU1_PGN_MASK) >> PGN_OFFSET
10     else:
11         # PDU 2 format
12         da = GLOBAL_SOURCE_ADDR
13         pgn = (id & PDU2_PGN_MASK) >> PGN_OFFSET
14     return {'source_address': sa,
15             'priority': priority,
16             'destination_address': da,
17             'pgn': pgn}
18
19 def parse_candump_line(line):
20     # Strip the newline characters and white space off the ends
21     # then split the string into a list based on whitespace.
22     data = line.strip().split()
23     # can_dump formats use parentheses to wrap the floating
24     # point timestamp. We just want the numbers so use [1:-1] to slice
25     time_stamp = float(data[CANDUMP_TIMESTAMP_ADDR][1:-1])
26     # physical CAN channel
27     channel = data[CANDUMP_CHANNEL_ADDR]
28     # determine the can arbitration identifier as an integer
29     can_id = int(data[CANDUMP_ID_ADDR],16)
30     # Data length code is a single byte wrapped in []
31     dlc = int(data[CANDUMP_DLC_ADDR][1])
32     # Build the data field as a byte array
33     data_bytes = b''
34     for byte_string in data[DATA_START_ADDR:]:
35         # Convert text into a single byte
36         data_bytes += struct.pack('B',int(byte_string,16))
37     #assert dlc == len(data_bytes)
38     can_frame = {'id':can_id,
39                  'dlc':dlc,
40                  'data':data_bytes,
41                  'timestamp':time_stamp,
42                  'channel':channel}
43     return can_frame
44
45 if __name__ == '__main__':
46     main()
```

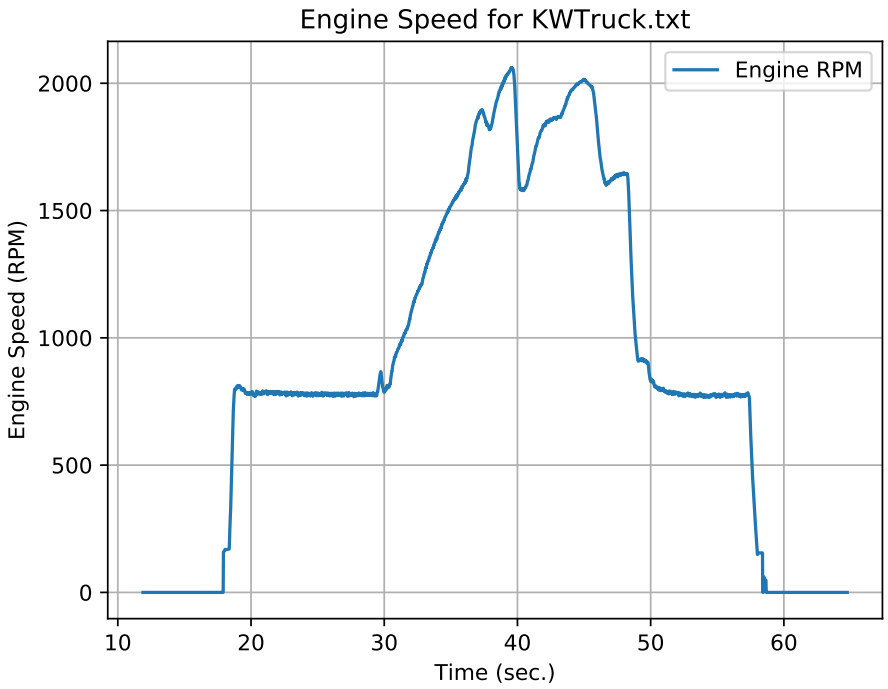

Figure 8.2.3: Engine speed graph produced by the Python source code in Listing 8.5 on page 49

9 Princples of Cybersecurity

The Triads of Cybersecurity

Confidentiality, Integrity, and Availability

Or,

Authentication, Authorization, and Auditing

9.1 Introduction to Cryptography

9.2 Message Authentication

9.3 Encryption on CAN

10 Heavy Vehicle Digital Forensics

11 Challenge Problems

Problem 11.1. Transfer a JPEG image of the CSU logo over CAN and display it.

Nomenclature

air gap A physically disconnected network with no access to the outside world.

API Application Programming Interface

CMAC Cryptographic Message Authentication

CyTeX Student CyberTruck Experience

ECU Electronic Control Unit

ELD Electronic Logging Device

HMAC Hash based message authentication

HSM Hardware Security Module

HVCS Heavy Vehicle Cyber Security

HVEDR Heavy Vehicle Event Data Recorder

IT Information Technology

JSON Javascript Objet Notation

NMFTA National Motor Freight Traffic Association, Inc.

OEM Orignial Equipment Manufacturer

OSI Open Systems Interconnection

PKI Public Key Infrastructure

PWM pulse width modulated

TU The University of Tulsa

UDS Unified Diagnostic Services

VIN Vehicle Identification Number

Bibliography

[1] Department of Homeland Security, *Transportation Systems Sector*, Original release date: July 06, 2009; Last revised: May 09, 2019; Accessed: May 24, 2020. "https://www.cisa.gov/transportation-systems-sector. 7

[2] National Motor Freight Traffic Association, Inc., *A Survey of Heavy Vehicle Cyber Security*, September 2015. http://www.nmfta.org/documents/hvcs/nmfta%20heavy%20duty%20vehicle%20cyber%20security%20whitepaper%20v1.0.3.6.pdf. 8, 19

[3] J. Daily, U. Jonson, and R. Gamble, "Talent generation for vehicle cybersecurity," in *Embedded Security for Cars (ESCAR) USA*, 2017. http://www.nmfta.org/documents/hvcs/vehiclecybersecurityeducationalinitiatives.pdf. 8

[4] S. Rathburn, *Developing Course Learning Objectives*, Accessed: May 24, 2020. https://tilt.colostate.edu/TipsAndGuides/Tip/92. 9

[5] Center for Excellence in Learning and Teaching, Iowa State University, *Revised Bloom's Taxonomy*, Accessed: May 24, 2020. https://www.celt.iastate.edu/teaching/effective-teaching-practices/revised-blooms-taxonomy/. 11

[6] S. International, *Cybersecurity Guidebook for Cyber-Physical Vehicle Systems*, Jan 2016. https://www.sae.org/standards/content/j3061_201601/. 16