16:03 Saving My '97 Chevy by Hacking It

by Brandon L. Wilson

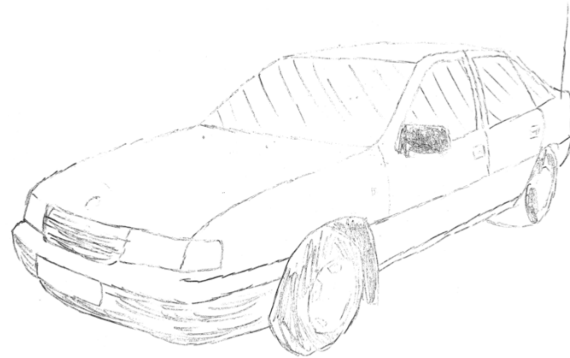
Hello everyone!

Today I tell a story of both joy and woe, a story about a guy stumbling around and trying to fix something he most certainly does not understand. I tell this story with two goals in mind: first to entertain you with the insane effort that went into fixing my car, then also to motivate you to go to insane lengths to accomplish something, because in my experience, the crazier it is and the crazier people tell you that you are to attempt it, the better off you'll be when you go ahead and try it.

Let me start by saying, though: do not hack your car, at least not the car that you actually drive. I cannot stress that enough. Do keep in mind that you are messing with the code that decides whether the car is going to respond to the steering wheel, brakes, and gas pedal. Flip the wrong bit in the firmware and you might find that *YOU* have flipped, in your car, and are now in a ditch. Don't drive a car running modified code unless you are certain you know what you're doing. Having said that, let's start from the beginning.

Once upon a time, I came into the possession of a manual transmission 1997 Chevrolet Cavalier. This car became a part of my life for the better part of 315,000 miles.[2] One fine day, I got in to take off somewhere, turned the key, heard the engine fire up—and then immediately cut off.

Let me say up front that when it comes to cars, I know basically nothing. I know how to start a car, I know how to drive a car, I know how to put gas in a car, I know how to put oil in a car, but in no way am I an expert on repairing cars. Before I could even begin to understand why the car wouldn't start, I had to do a lot of reading to understand the basics on how this car runs, because every car is different.



In the steering column, behind the steering wheel and the horn, you have two components physically locked into each other: the ignition lock cylinder and the ignition switch. First, the key is inserted into the ignition lock cylinder. When the key is turned, it physically rotates inside the ignition lock cylinder, and since the ignition switch is locked into it, turning the key also activates the ignition switch. The activation of that switch supplies power from the battery to everywhere it needs to go for the car to actually start.

But that's not the end of the story: there's still the anti-theft system to deal with. On this car, it's something called the PassLock security system. If the engine is running, but the computer can't detect the car was started legitimately with the original key, then it disables the fuel injectors, which causes the car to die.

Since the ignition switch physically turning and supplying battery power to the right places is what makes the car start, stealing a car would normally be as simple as detaching the ignition switch, sticking a screwdriver in there, and physically turning it the same way the key turns it, and it'll fire right up.[3]

So the PassLock system needs to prevent that from working somehow. The way it does this starts with the ignition lock cylinder. Inside is a resistor of a certain resistance, known by the instrument panel cluster, which is different from car to car. When physically turning the cylinder, that certain resis-

[2]Believe it or not, those miles were all on the original clutch. You can see why I might want to save it.

[3]*This is helpfully described by Deviant Ollam on page 17. –PML*

tance is applied to a wire connected to the instrument panel cluster. As the key turns, a signal is sent to the instrument panel cluster. The cluster knows whether that resistance is correct, and if and only if the resistance is correct, it sends a password to the PCM (Powertrain Control Module), otherwise known as the main computer. If the engine has started, but the PCM hasn't received that "password" from the instrument panel cluster, it makes the decision to disable the fuel injectors, and then illuminate the "CHECK ENGINE" and "SECURITY" lights on the instrument panel cluster, with a diagnostic trouble code (DTC) that indicates the security system disabled the car.

So an awful lot of stuff has to be working correctly in order for the PCM to have what it needs to not disable the fuel injectors. The ignition lock cylinder, the instrument panel cluster, and the wiring that connects those to each other and to the PCM all has to be correct, or the car can't start.

Since the engine in my car does turn over (but then dies), and the "SECURITY" warning light on the instrument panel cluster lights up, that means something in the whole chain of the PassLock system is not functioning as it should.

Naturally, I start replacing parts to see what happens. First, the ignition lock cylinder might be bad – so I looked up various guides online about how to "bypass" the PassLock system. People do that by installing their own resistor on the wires that lead to the instrument panel cluster, then triggering a thirty-minute "relearn" procedure so that the instrument panel cluster will accept the new resistor value.[4] Doing that didn't seem to help at all. Just in case I messed that up somehow, I decided to buy a brand new ignition lock cylinder and give that a try. Didn't help.

Then I thought maybe the ignition switch is bad, so I put a new one of those in as well. Didn't help. Then I thought maybe the clutch safety switch had gone bad (the last stop for battery power on its way from the ignition switch to the rest of the car) – checking the connections with a multi-meter indicated it was functioning properly.

I even thought that maybe the computer had somehow gone bad. Maybe the pins on it had corroded or something – who knows, anything could be causing it not to get the password it needs from the instrument panel cluster. There is a major problem with replacing this component however, and that is that the VIN, Vehicle Identification Number, unique to this particular car, is stored in the PCM. Not only that, but this password that flies around between the PCM and instrument panel cluster is generated from the VIN number. The PCM and panel are therefore "married" to each other; if you replace one of them, the other needs to have the matching VIN number in it or it'll cause the same problem that I seem to be experiencing.

Fortunately, one can buy replacement PCMs on eBay, and the seller will actually pre-flash it with the VIN number that the buyer specifies. I bought from eBay and slapped it in the car, but it still didn't work.

At this point, I have replaced the ignition lock cylinder, the ignition switch, even the computer itself, and still nothing. That only leaves the instrument panel cluster, which is prohibitively expensive, or the wiring between all these components. There are dozens upon dozens of wires connecting all this stuff together, and usually when there's a loose connection somewhere, people give up and junk the whole car. These bad connections are almost impossible to track down, and even worse, I have no idea how to go about doing it.

So I returned all the replacement parts, except for the PCM from eBay, and tried to think about what to do next. I have a spare PCM that only works with my car's VIN number. I know that the PCM disables the fuel injectors whenever it detects an unauthorized engine start, meaning it didn't get the correct password from the instrument panel cluster. And I also know that the PCM contains firmware that implements this detection, and I know that dealerships upgrade this firmware all the time. If that's the case, what's to stop me from modifying the firmware and removing that check?

Tune In and Drop Out

I began reading about a community of car tuners, people who modify firmware to get the most out of their cars. Not only do they tweak engine performance, but they actually disable the security system of the firmware, so that they can transplant any engine from one car to the body of another car. That's exactly what I want to do; I want to disable that feature entirely so that the computer doesn't care what's going on outside it. If they can do it, so can I.

[4] This is how old remote engine start kits work.

How do other people disable this check? According to the internet, people "tune" their cars by loading up the firmware image in an application called, oddly enough, TunerPro. Then they load up what's called an XDF file, or a definition file, which defines the memory addresses for configuration flags for all sorts of things – including, of course, the enabling and disabling of the anti-theft functionality. Then all they have to do is tell TunerPro "hey, turn this feature off", and it knows which bits or bytes to change from the XDF file, including any necessary checksums or signatures. Then it saves the firmware image back out, and tuners just write that firmware image back to the car.

It sounds easy enough – assuming the car provides an easy mechanism for updating the firmware. Most tuners and car dealerships will update the firmware through the OBD2 diagnostic port under the steering column, which is on all cars manufactured after 1996 (yay for me). Unfortunately, each car manufacturer uses different protocols and different tools to actually connect to and use the diagnostic port. For example, General Motors, which is what I need to deal with, has a specific device called a Tech2 scan tool, which is like a fancy code reader, which can be plugged into the OBD2 port. It's capable of more than just reading diagnostic trouble codes, though; it can upload and download the firmware in the PCM. There's just one problem: it's ridiculously expensive. This thing runs anywhere from a few hundred for the Chinese clone to several thousands of dollars!

I spent some time looking into what protocol it uses, so that I could do what it does myself – but no such luck. It seems to use some sort of proprietary obfuscated algorithm so the PCM has to be "unlocked" before it can be read from or written to. GM really doesn't want me doing myself what this tool does. Even worse, after doing a little googling, it seems there is no XDF file for my particular car, so I have to find these memory addresses myself.

The first step is to get at the firmware. If I can't simply plug into the OBD2 port and read or write the firmware, I'm going to have to get physical. I find the PCM, unplug it from the car, unscrew the top cover, and start starting at what's underneath.

Luckily, there appears to be a 512KB flash chip on board. I know from googling about TunerPro and others' experience with firmware from the late nineties that this is exactly the right size to hold the PCM firmware image. Fortunately, I have managed to physically extract chips like this before, so I de-soldered the chip, inserted it into an old Willem EEPROM programmer, and managed to dump the entire 512KB of memory. What now?

Thankfully, Google has come to the rescue and presented me with a series of forum posts that tell me how to interpret this firmware dump. These old

posts were pretty much the only help I could find on the subject, so I had to decipher some guy's notes and do the best I could.

Apparently the processor in this PCM and others of its era is a Motorola 68332. I just so happen to have a history with the Motorola 68K series CPUs. Ever since high school I have messed with BASIC and assembly programming for Texas Instruments graphing calculators, some of which have a Motorola 68K CPU, and I enjoy collecting and tinkering with old game consoles, which is good because the Sega Genesis just so happens to have a Motorola 68K CPU.

It sure would be nice to confirm in some way if this file really was dumped correctly and this really is Motorola 68K firmware being executed by this PCM. There ought to be a vector table at the beginning of memory, containing handler addresses that the CPU executes in response to certain events. For example, when the CPU first gets power, it has to start executing from the value at address `0x000004`, which holds what is called the Reset Vector. Looking at that address, I see `00 00 40 04`. I fire up IDA Pro, go to address `0x4004`, and hit C to start analyzing code at that address – but I get total garbage.

That's strange – since that didn't pan out, I start looking for human-readable strings. I find only one, which appears to be a 17-character VIN number, except that it's not a VIN number.

```
String:      1G1J11C72V24767321
Actual VIN:  1G1JC1272V7476231
```

I stared at this until I realized that if I swap every two characters, or bytes, in the actual VIN number, I get the string from the disassembly. It seems the image is a little jumbled up – googling for meaning behind this reveals that the image is byte-swapped. This is how the bytes are actually stored on the chip, but this isn't what I want – I want the bytes back in the original order, the way they're being executed. After swapping every pair of bytes and then looking at address `0x000004`, I don't see `00 00 40 04` – I see `00 00 04 40`. If I go to `0x440` in IDA Pro and start analyzing, I see an explosion of readable code. In fact, I see a beautiful graph of how cleanly this file disassembled.

I'm ecstatic that I have a clean and proper firmware image loaded into IDA Pro, but what now? It would take years for me to properly and truly understand all this code.

I have to remind myself that my goal is to disable the check on whether we've received the password or not from the instrument panel cluster – but I have absolutely no idea where in the firmware that check is. There doesn't seem to exist an XDF file for my 1997 Chevrolet Cavalier. But – maybe one does exist for a very similar car. If I can know the memory address I want to change in somebody else's firmware image, and it's similar enough to mine, maybe that'll give me clues to finding the memory address in my own image.

After doing lots…and lots…of googling, the closest firmware image I could find which had a matching XDF file was for the 2001 Pontiac Trans Am. I load up this firmware image in TunerPro along with the corresponding XDF file, and a particular setting jumps out at me called "Option byte for vehicle theft deterrent" – with a memory address of `0x1E5CC`. I fire up IDA Pro against the 2001 Pontiac Trans Am image and go to that memory address, which puts me in the middle of a bunch of bytes that are referenced all over the place in the code. This is some sort of "configuration" area, which controls all the features of the car's computer. If I change this byte in TunerPro and save the firmware image, it updates two things: one, this option byte at `0x1E5CC`, and also a checksum word (two bytes) that protects the configuration area from corruption or tampering. So to turn off the anti-theft system, I have to flip a bit, update the checksums, write those changes back to the car computer, and voila, I'm done. Now all that's left is to find the same code that uses that bit in my 1997 Chevrolet Cavalier firmware image. Sounds simple enough.

```
IsVATSPresent_IThinkD0NZIfPresent:
7a754:    cmpi.b  #2, (VATS_type).l
7a75c:    sne     d0
7a75e:    neg.b   d0
7a756:    and.b   (byte_FFFF8BE5).w, d0
7a764:    rts
```

The byte at `0x1E5CC` is referenced all over the place – but there's only one place in particular with a small subroutine that looks at the specific bit we care about. If I can find this same subroutine in my own firmware image, I'm in business.

I look for these exact instructions in my own firmware image, but they isn't there. I look for any comparison to bit 2 of a particular byte, but there are none. I look for "`sne d0`" followed by "`neg.b`

d0" – but no dice. I look for the same instructions acting on any register at all – but no matches. I try dozens and dozens of other code matching patterns – but no matches.

I thought it would be really simple to look for the same or a similar code pattern in my firmware image and I'd have no trouble finding it, but apparently not. These TunerPro XDF definition files get created by somebody, right? How do they find all these memory addresses of interest, so they can build these XDF files?

According to the forum posts I found,[5] they first look for a particular piece of functionality: the handling of OBD2 code reader requests. The PCM is what's responsible for receiving the commands from a code reader, generating a response, and then sending it back over the OBD2 port to the code reader tool. Somewhere in this half-megabyte mess is all the code that handles these requests.

These OBD2 tools are capable of retrieving more than just diagnostic trouble codes. Not only can they upload and download firmware images for the PCM, but they can also retrieve all sorts of real-time engine information, telling you exactly what the computer's doing and how well it's doing it. It can also return the anti-theft system status. So if I can understand the OBD2 communication code, I can find my way to the option flag in the 2001 Pontiac Trans Am firmware. And if I can navigate my way to the option flag in that firmware, then I can just apply that same logic to my own firmware.

How can I find the code that handles these requests? According to the "PCM hacking 101" forum guide, I should start by looking for the code that actually interacts with the OBD2 port.

So how does a Motorola 68K CPU interact with the OBD2 port, or any hardware for that matter? It uses something called memory-mapped I/O. In other words, the hardware is wired in such a way, that when reading from or writing to a particular memory address, it isn't accessing bytes in the firmware on the flash chip or in RAM; it's manipulating actual hardware.

In any given device, there is usually a range of address space dedicated just to interacting with hardware. I know it has to be outside the range of where the firmware exists, and I know it has to be outside the range of where the RAM exists.

I know how big the firmware is, and since it disassembled so cleanly, I know it starts out at address 0, so that means the firmware goes from 0 all the way up to 0x07FFFF.

I also know from poking around in the disassembly that the RAM starts at 0xFF0000, but I don't know how big it is or where it ends. As a quick and dirty way of getting close to an answer, I use IDA Pro to export a .asm file, then have sed rip out the memory addresses accessed by certain instructions, then sort that list of memory addresses.

This way, I discover that typical RAM accesses only go up to a certain point, and then things start getting weird. I start seeing loops on reading values contained at certain memory addresses, and no other references to writes at those memory addresses. It wouldn't make sense to keep reading the same area over and over, expecting something to change, unless that address represents a piece of hardware that can change. When I see code like that, the only explanation is that I'm dealing with memory-mapped I/O. So while I don't have a complete memory map just yet, I know where the hardware accesses are likely to be.

Consulting the forum guide again, I learn that one of the chips on the PCM circuit board is responsible for handling all the OBD2 port communication. I don't mean it handles the high-level request; I mean it deals with all the work of interpreting the raw signals from the OBD2 pins and translating that into a series of bytes going back and forth between the firmware and the device plugged into the OBD2 port. All it does is tell the firmware "Hey, something sent 5 bytes to us. Please tell me what bytes you want me to send back," and the firmware deals with all the logic of figuring out what those bytes will be.

This chip has a name – the MC68HC58 data link controller – and lucky for me, the datasheet is readily available.[6] It's fairly comprehensive documentation on anything and everything I ever wanted to know about how to interact with this controller. It even describes the memory-mapped IO registers which the firmware uses to communicate with it. It tells me everything but the actual number, the actual memory address the firmware is using to interact with it, which is going to be unique for the device in which it's installed. That's going to be up to me to figure out.

After printing out the documentation for this chip and some sleepless nights reading it, I figured

[5] https://www.thirdgen.org/forums/diy-prom/507563-pcm-hacking-101-step.html
[6] unzip pocorgtfo16.pdf mc68hc58.pdf

out some bytes that the firmware must be writing to certain registers (to initialize the chip), otherwise it can't work, so I started hunting down where these memory accesses were in the firmware. And sure enough, I found them, starting at address `0xFFF600`.

So now that I've found the code that receives a command from an OBD2 code reader, it should be really easy to read the disassembly and get from there to code that accesses our option flag, right?

I wish! The firmware actually buffers these requests in RAM, and then de-queues them from that buffer later on, when it's able to get to it. And then, after it has acted on the request and calculated a response, it buffers that for whenever the firmware is able to get around to sending them back to the plugged-in OBD2 device. This makes sense; the computer has to focus on keeping the engine running smoothly, and not getting tied up with requests on how well the engine is performing.

Unfortunately, while that makes sense, it also makes it a nightmare to disassemble. The forum guide does its best to explain it, but unfortunately its information doesn't apply 100% to my firmware, and it's just too difficult to extrapolate what I need in order to find it. This is where things start getting really nutty.

Emulation

If I can't directly read the disassembly of the code and understand it, then my only option is to execute and debug it.

There are apparently people out there that actually do this by pulling the PCM out of the car and putting it on a workbench, attaching a bunch of equipment to it to debug the code in real-time to see what it's doing. But I have absolutely no clue how to do that. I don't have the pinouts for the PCM, so even if I did know what I was doing, I wouldn't know how to interface with this specific computer. I don't know anything about the hardware, I don't know anything about the software – all I know about is the CPU it's running, and the basics of a memory map for it. That is at least one thing I have going for me – it's extremely similar to a very well-known CPU (the Motorola 68K), and guaranteed to have dozens of emulators out there for it, for games if nothing else.

Is it really possible I have enough knowledge about the device to create or modify an emulator to execute it? All I need the firmware to do is boot just well enough that I can send OBD2 requests to it and see what code gets executed when I do. It doesn't actually have to keep an engine running, I just need to see how it gets from point A, which is the data link controller code, to point B, which is the memory access of the option flag.

If I'm going to seriously consider this, I have to think about what language I'm going to do this in. I think, live, breathe, and dream C♯ for my day job, so that is firmly ingrained into my brain. If I'm really going to do this, I'm going to have to hack the crap out of an existing emulator, I need to be able to gut hardware access code, add it right back, and then gut it again with great efficiency. So I want to find a Motorola 68K emulator in C♯.

You know you've gone off the deep end when you start googling for a Motorola 68K emulator in a managed language, but believe it or not, one does

[7] https://www.codeproject.com/Articles/998595/CPS-NET-a-Csharp-based-CPS-MAME-emulator

exist. There is an old Capcom arcade system called the CPS1, or Capcom Play System 1. It was used as a hardware platform for Street Fighter II and other classic games. Somebody went to the trouble of creating an emulator for this thing, with a full-featured debugger, totally capable of playing the games with smooth video and sound, right on Code Project.[7]

I began to heavily modify this emulator, completely gutting all the video-related code and display hardware, and all the timers and other stuff unique to the CPS1. I spent a not-insignificant amount of time refactoring this application so it was just a Motorola 68K CPU core, and with the ability to extend it with details about the PCM hardware.[8]

Once I had this Motorola 68K emulator in C♯, it was time to get it to boot the 2001 Pontiac Trans Am image. I fire it up, and find that it immediately encounters an illegal instruction. I can't say I'm very surprised – I proceed to take a look at what's at that memory address in IDA Pro.

When going to the memory address of the illegal instruction, I saw something I didn't expect to see. . . a TBLU instruction. What in the world? I know I've never seen it before, certainly not in any Sega Genesis ROM disassembly I've ever dealt with. But, IDA Pro knew how to display it to me, so that tells me it's not actually an illegal instruction. So, I look in the Motorola 68332 user manual,[9] and look up the TBLU instruction.

Without getting too into the weeds on instruction decoding, I'll just say that this instruction basically performs a table lookup and calculates a value based on precisely how far into the table you go, utilizing both whole and fractional components. Why in the world would a CPU need an instruction that does this? Actually it's very useful in exactly this application, because it lets the PCM store complex tables of engine performance information, and it can quickly derive a precise value when communicating with various pieces of hardware.

It's all very fascinating I'm sure, but I just want the emulator to not crash upon encountering this instruction, so I put a halfway-decent implementation of that instruction into the C♯ emulator and move on. Digging into Motorola 68K instruction decoding enabled me to fix all sorts of bugs in the CPS1 emulator that weren't a problem for the games it was emulating, but it was quite a problem for me.

```
   6e328:  mov.b   (byte_73dec).l,  ($FFFFFd48).w
 2 6e330:  mov.b   (byte_73ded).l,  ($FFFFFd49).w
   6e338:  mov.b   (byte_73dee).l,  ($FFFFFd4a).w
 4 6e340:  mov.b   (byte_73dee).l,  ($FFFFFd4b).w
   6e348:  mov.b   (byte_73dee).l,  ($FFFFFd4c).w
 6 6e350:  mov.b   (byte_73dee).l,  ($FFFFFd4d).w
   6e358:  mov.b   (byte_73def).l,  ($FFFFFd4e).w
 8 6e360:  mov.b   (byte_73de4).l,  ($FFFFFc1a).w
   6e368:  mov.b   (byte_73de8).l,  ($FFFFFc1c).w
10 6e370:  andi.b  #$F0,   ($FFFFFC1C).w
   6e376:  ori.b   #$E,    ($FFFFFC1C).w
12 6e37c:  bclr    #7,     ($FFFFFC1F).w
   6e382:  bset    #7,     ($FFFFFC1A).w
14   loop88:
   6e388:  btst    #7,     ($FFFFFC1F).w
16 6e38e:  beq.s   loop88
   6e390:  unlk    a6
18 6e392:  rts
```

Once I got past the instructions that the emulator didn't yet have support for, I'm now onto the next problem. The emulator's running. . . but now it's stuck in an infinite loop. The firmware appears to keep testing bit 7 of memory address `0xFFFC1F` over and over, and won't continue on until that bit is set. Normally this code would make no sense, since there doesn't appear to be anything else in the firmware that would make that value change, but since `0xFFFC1F` is within the range that I think is memory-mapped I/O, this probably represents some hardware register.

What this code does, I have no idea. Why we're waiting on bit 7 here, I have no idea. But, now that I have an emulator, I don't have to care one bit.[10]

[8]`git clone https://github.com/brandonlw/pcmemulator`
[9]`unzip pocorgtfo16.pdf mc68332um.pdf`
[10]*We the editors politely apologize for this pun, which is entirely the fault of the author. –PML*
[11]To be more accurate, I do this a few dozen more times and then happily move on.

I fix this by patching the emulator to always say the bits are set when this memory address is accessed, and we happily move on.[11] Isn't emulation grand?

```
else if(address == 0xFFF70F)
    return 0x02|0x01;
else if(address == 0xFFFC1F)
    return -1; //0xFF
else if(address == 0xFFF60E)
    //...
```

Now I've finally gotten to the point that the firmware has entered its main loop, which means it's functioning as well as I can expect, and I'm ready to begin adding code that emulates the behavior of the data link controller chip. Since I now know what memory addresses represent the hardware registers of the data link controller, I simply add code that pretends there is no OBD2 request to receive, until I start clicking buttons to simulate one.

I enter the bytes that make up an OBD2 request, and tell the emulator to simulate the data link controller sending those bytes to the firmware for processing. Nothing happens. Imagine that, yet another problem to solve!

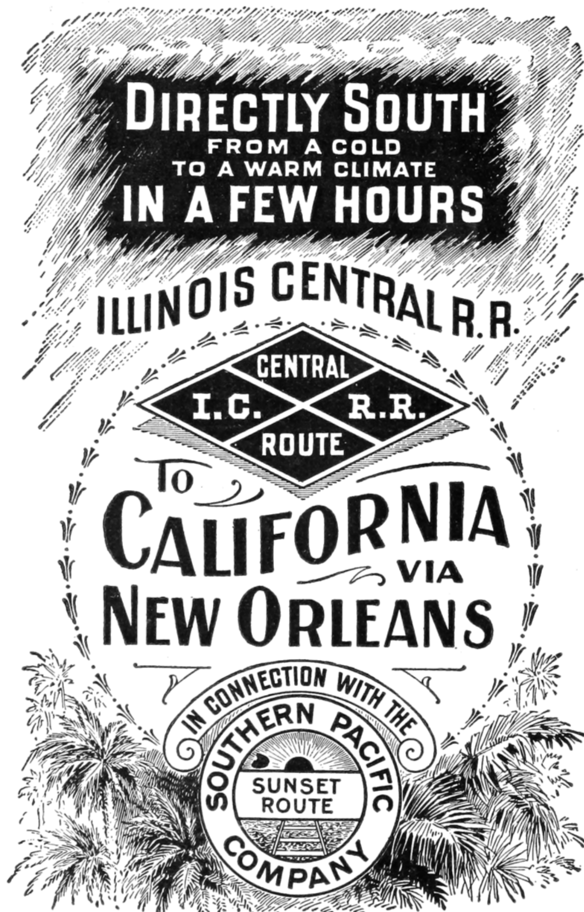
I scratched my head on this one for a long time, but I finally remembered something from the forum guide: the routines that handle OBD2 requests are executed by "main scheduling routines." If the processing of messages is on a schedule, then that implies some sort of hardware timer. You can't schedule something without an accurate timer. That means the firmware must be keeping track of the number of accurate ticks that pass. So if I check the vector table, where the handlers for all interrupts are defined, I ought to find the handler that triggers scheduling events.

```
move.b  #1,(InterruptVector108Flag).w
move.l  (InterruptVector108FlagCounter).w, d3
addq.l  #1, d3
move.l  d3, (InterruptVector108FlagCoutner).w
cmpi.l  #$7FFFFFFF, d3
bne.s   lov_2a18c
jsr     (Stop2700).l
  loc_2a18c:
jsr     DoLotsOfHardwareRegisterReadsWrites
tst.b   (byte_FFFFAE6E).w
bne.s   locret_2A19E
jsr     sub_71FC2
  locret_2A19E:
rts
```

This routine, whenever a specific user interrupt fires, will set a flag to 1, and then increment a counter by 1. As it turns out, this counter is checked within the main loop – this is actually the number of ticks since the firmware has booted. The OBD2 request handling routines only fire when a certain number of ticks have occurred. So all I have to do is simulate the triggering of this interrupt periodically, say every few milliseconds. I don't know or care what the real amount of time is, just as long as it keeps happening. And when I do this, I find that the firmware suddenly starts sending the responses to the simulated data link controller! Finally I can simulate OBD2 requests and their responses.

Now all I need to do is throw together some code to brute-force through all the possible requests, and set a "breakpoint" on the code that accesses the option flag.

Many hours later, I have it! With an actual request to look at, I can do some googling and see that it utilizes "`mode $22`," which is where GM stuffs non-standard OBD2 requests, stuff that can potentially change over time and across models. Request `$1102` seems to return the option flag, among other things.



THE ONLY TRUE WINTER ROUTE

PULLMAN BUFFET SLEEPING CAR

connecting with Southern Pacific Company's famous "Sunset Limited," from Chicago every Tuesday and Saturday night. Through reservations to the coast.

THROUGH PULLMAN TOURIST CAR

from Chicago to San Francisco every Wednesday night.

Particulars of agents of connecting lines, or by addressing A. H. HANSON, General Passenger Agent, Illinois Central R. R., Chicago.

Christmas Superdeals!

ATARI 520STFM Super Pack £359.00
Including VAT and NEXT DAY DELIVERY!

Atari 520STFM Super Pack includes:

- ★ Built-in TV modulator allowing you to use the 520STFM with your domestic TV set.
- ★ Built-in 1 megabyte disc drive for fast loading and saving of programs.
- ★ £450 worth of free games software including MARBLE MADNESS, TEST DRIVE, ARKANOID 2, BUGGY BOY, WIZBALL and 16 more.
- ★ ORGANISER Business Software worth £50.
- ★ FREE JOYSTICK!
- ★ And to enable you to have your ST running within minutes, a free fitted power plug!

ALSO AVAILABLE WITH JUST ONE FREE GAME £279

WE ACCEPT EXPRESS VOUCHERS

commodore AMIGA A500 £389.00
Including VAT and NEXT DAY DELIVERY!

Amiga Pack includes:

- ★ Built-in 1 megabyte disc drive for fast loading and saving of programs.
- ★ FREE TV modulator worth £24.99 enabling you to use the AMIGA with your domestic TV set.
- ★ FREE Game Software worth £230 including BUGGY BOY, MERCENARY, WIZBALL and seven more games.
- ★ FREE PHOTON PAINT graphics package worth £69.95.
- ★ And to enable you to unpack and use your AMIGA straight away, a free fitted power plug!

ALSO AVAILABLE WITHOUT FREE GAMES £369.00

CREDIT CARD ORDERLINE: 0908 663708 9am–8pm
To order: telephone the credit card orderline above with your ACCESS or VISA number
OR make Cheque or P.O. payable to Digicom Computer Services Ltd and send your order to:

Full range of Atari and Commodore hardware and software available at discount prices.

WE ACCEPT EXPRESS VOUCHERS

DIGICOM
170 Bradwell Common Boulevard, MILTON KEYNES MK13 8BG

VISA

WE ACCEPT EXPRESS VOUCHERS

Now that I've found the OBD2 request in the 2001 Pontiac Trans Am, I can emulate my own firmware image and send the same request to it. Once I see where the code takes me, I can modify the byte appropriately, recalculate the firmware checksum, reflash the chip in my programmer, resolder it back into the PCM, reassemble it and reattach it to the car, hop in, and turn the key and hope for the best.

I'm sorry to say that this doesn't work.

Why? Who can say for sure? There are several possibilities. The most plausible explanation is that I just screwed up the soldering. A flash chip's pins can only take so much abuse, especially when I'm the one holding the iron.

Or, since I discovered that this anti-theft status is returned via a non-standard OBD2 request, it's possible that the request might just do something different between the two firmware images. It doesn't bode well that the two images were so different that I couldn't find any code patterns across both of them. My Cavalier came out in 1997 when OBD2 was brand new, so it's entirely possible that the firmware is older than when GM thought to even return this anti-theft status over OBD2.

What do I do now? I finally decide to give up and buy a new car. But if I could do it over again, I would spend more time figuring out exactly how to flash a firmware image through the OBD2 port. With that, I would've been free to experiment and try over and over again until I was sure I got it right. When I have to repeatedly desolder and resolder the flash chip several times for each attempt, the potential for catastrophe is very high.

If you take anything away from this story, I hope it's this: if you're faced with a problem, and you come up with a really crazy idea, don't be afraid to try it. You might be surprised, it just might work, and you just might get something out of it. The car may still be sitting in a garage collecting dust, but I did manage to get a functioning car computer emulator out of it. My faithful companion did not die in vain. And who knows, maybe someday he will live again.