The Car Hacker's Handbook: A Guide for the Penetration Tester - Craig Smith (2016)

Chapter 6. ECU HACKING

by Dave Blundell

A vehicle typically has as many as a dozen or more electronic controllers, many of which are networked to communicate with each other. These computerized devices go by many different names, including *electronic control unit* or *engine control unit (ECU)*, *transmission control unit (TCU)*, or *transmission control module (TCM)*.

While these terms may have specific meanings in a formal setting, similar terms are often used interchangeably in practice. What may be a TCU to one manufacturer is a TCM to another, yet both electronic controllers perform the same or extremely similar functions.

Most automotive control modules have measures in place to prevent you from altering their code and operation; these range from very strong to laughably weak. You won't know what you're dealing with until you investigate a particular system. In this chapter, we'll take a closer look at particular security mechanisms, but first we'll examine strategies for gaining access to these systems. Then in Chapter 8 we'll look at some more specific ECU hacks, like glitch attacks and debugging. The attack vectors for ECUs fall into three different classes:

Front door attacks Commandeering the access mechanism of the original equipment manufacturer (OEM)

Backdoor attacks Applying more traditional hardware-hacking approaches

Exploits Discovering unintentional access mechanisms

We'll look at an overview of these attack classes, and then analyze the data you find. It's worth remembering that while the goal for ECU and other control module hacking is often the same—to gain access in order to reprogram and change behavior—it’s unlikely there’ll be a “master key” for all controllers. However, OEMs are generally not very creative and seldom change their ways, so insight into one controller likely applies to similar models from the same manufacturer. Also, few of today’s auto manufacturers develop their own automotive computers from scratch, instead licensing prefabricated solutions from third parties like Denso, Bosch, Continental, and others. Because of this design methodology, it’s relatively common to see vehicles from different auto manufacturers using very similar computer systems sourced from the same vendors.

Front Door Attacks

The OBD-II standard mandates that you be able to reprogram vehicles through the OBD-II connector, and reverse engineering the original method for programming is a guaranteed attack vector. We’ll examine J2534 and KWP2000 as examples of common protocols for programming.

J2534: The Standardized Vehicle Communication API

The SAE J2534-1 standard, or simply *J2534*, was developed to promote interoperability among digital tool vendors through the use of the J2534 API, which outlines the recommended way for Microsoft Windows to communicate with a vehicle. (You can purchase the J2534 API from the

SAE at *http://standards.sae.org/j2534/1_200412/*.) Prior to the adoption of the J2534 standard, each software vendor created its own proprietary hardware and drivers for communicating with a vehicle in order to perform computerized repairs. Because these proprietary tools weren't always available to smaller shops, the EPA mandated the adoption of the J2534 standard in 2004 to allow independent shops access to the same specialized computer tools used by dealerships. J2534 introduced a series of DLLs that map standard API calls to instructions necessary to communicate with a vehicle, thereby allowing multiple manufacturers to release software designed to work with J2534-compatible hardware.

Using J2534 Tools

J2534 tools provide a convenient way to observe OEM tools interacting with vehicle computers. Manufacturers often leverage J2534 to update computer firmware and sometimes to provide powerful diagnostic software. By observing and capturing information exchanged with a vehicle using J2534, you can see how OEMs perform certain tasks, which may provide you with information that you need to unlock the "front door."

When using J2534 tools to attack vehicle systems, the basic idea is to observe, record, analyze, and extend functionality. Of course, the first step is to obtain and configure a J2534 application and its corresponding interface hardware in order to perform a task you want to observe. Once you have your setup, the next step is to observe and record communications with the target while using the J2534 tools to perform an action on the target, like updating a configuration parameter.

There are two primary ways to observe J2534 transactions: by watching J2534 API calls on a PC using J2534 shim DLLs or by watching actual bus traffic using a separate sniffer tool to capture data.

J2534 tools are key to eavesdropping on the protocols built into the factory embedded vehicle systems, and they're one of the primary ways to attack the front door. Successful analysis of this communication will give you the knowledge you need to access vehicle systems the way the OEMs do. It'll also allow you to write applications with full access to read and reprogram systems, which will in turn enable you to communicate directly with a vehicle without having to use the J2534 interface or the OEM's J2534 software.

J2534 Shim DLLs

The J2534 shim is a software J2534 interface that connects to a physical J2534 interface and then passes along and logs all commands that it receives. This dummy interface is a kind of man-in-the-middle attack that allows you to record all API calls between the J2534 application and the target. You can then examine the log of commands to determine the actual data exchanged between the J2534 interface and the device.

To find an open source J2534 shim, search *code.google.com* for *J2534-logger*. You should also be able to find precompiled binaries.

J2534 with a Sniffer

You can also use J2534 to generate interesting traffic that you can then observe and record with a third party sniffer. There's no magic here: this is just an excellent example of how to generate juicy packets that might otherwise be difficult to capture. (See Chapter 5 for more information on monitoring network traffic.)

KWP2000 and Other Earlier Protocols

Before J2534, there were many flash-programmable ECUs and other control units, such as the Keyword Protocol 2000 (KWP2000 or ISO14230). From an OSI networking perspective, it's primarily an application protocol. It can be used on top of CAN or ISO9141 as the physical layer. You'll find a *huge* number of KWP2000 flasher tools that interface with a PC using a serial/ USB-serial interface and that support diagnostics and flashing using this protocol just by searching online. (For more on the Keyword Protocol 2000, see Chapter 2.)

Capitalizing on Front Door Approaches: Seed-Key Algorithms

Now that we've discussed how legitimate tools use the front door, it's time to capitalize on this attack vector by learning how to operate the figurative "lock on the gate." To do this, we must understand the algorithm that the embedded controller uses to authenticate valid users; this is almost always a seed-key algorithm. Seed-key algorithms usually generate a pseudorandom *seed* and expect a particular response, or *key*, for each seed before allowing access. A typical valid exchange could look something like this:

```
ECU seed: 01 C3 45 22 84
Tool key: 02 3C 54 22 48
```

or this:

```
ECU seed: 04 57
Tool key: 05 58
```

Unfortunately, there's no standard seed-key algorithm. You might have a 16-bit seed and 16-bit key, a 32-bit seed and 16-bit key, or a 32-bit seed and 32-bit key. The algorithm that generates a key from a given seed also varies from platform to platform. Most algorithms are a combination of simple arithmetic operations and one or more values used as part of the computation. There are several techniques for figuring out these algorithms in order to give you access to the ECU:

• Obtain the firmware for the device in question through other means. Disassemble it and analyze the embedded code to find the code responsible for generating seed-key pairs.

• Obtain a legitimate software tool—for example, J2534 reflash software—that's capable of generating legitimate seed-key pairs, and analyze the PC application code with a disassembler to determine the algorithm used.

• Observe a legitimate tool exchanging keys, and analyze the pairs for patterns.

• Create a device to spoof a legitimate tool into providing responses repeatedly. The main advantage of this method over purely passive observation is that it allows you to pick seeds for which you can reproduce the keys.

You can find more information about reverse engineering the seed-key algorithms used by General Motors at *http://pcmhacking.net/forums/viewtopic.php?f=4&t=1566&start=10*, and those used by VAG MED9.1 at *http://nefariousmotorsports.com/forum/index.php?*

topic=4983.0.

Backdoor Attacks

Sometimes front door attacks are too tricky; you may not have the right tools or the lock might be too hard to figure out. Don't despair—remember that automotive control modules are embedded systems, so you can use all the usual hardware-hacking approaches. In fact, using more direct-to-hardware backdoor approaches often makes more sense than trying to reverse engineer the front door lock placed by the factory, especially when trying to reprogram engine modules. If you can obtain a dump of the module, you can often disassemble and analyze it to figure out how the keys to the front door work. The first step in a hardware backdoor attack is analyzing the circuit board.

When reversing a circuit board of any system, you should start with the largest chips first. These larger processor and memory chips are likely to be the most complex. It's a good idea to make a list of part numbers to feed to Google, *datasheet.com*, or something similar, to obtain a copy of the data sheet. You'll sometimes encounter custom application-specific integrated circuits (ASICs) and one-off chips, especially with older ECUs, which will prove more difficult than off-the-shelf parts. In many cases, you'll have to infer the function of these parts based on how they're connected to identifiable parts.

It’s critical to look out for memory chips—SRAM, EEPROM, FlashROM, one-time-programmable ROM, serial EEPROM, serial flash, NVSRAM, and so on. The type of memory used varies immensely from one platform to another; every single variety listed here has been found in the wild. Newer designs are less likely to have parallel memories and more likely to have serial chips. Newer microcontrollers are less likely to have any external memories at all, as their internal flash capacities have dramatically increased. Any nonvolatile memory chip present can be removed from the circuit board, read, and then replaced. Chapter 8 goes into much more detail on reverse engineering the circuit board.

Exploits

Although arguably just another example of a backdoor approach, exploits deserve special attention. Rather than taking apart a computer, exploits involve feeding a system carefully crafted inputs to make it do things outside normal operation. Typically, exploits build on a bug or problem. This bug might cause a system to crash, reboot, or perform some undesirable behavior from the perspective of the vehicle user. Some of these bugs present the opportunity for buffer overflow attacks, which open the door for commandeering the vulnerable device merely by feeding it unexpected inputs. A cleverly crafted set of inputs triggers the bug, which then makes the device execute arbitrary code provided by the attacker instead of triggering the usual fault condition.

Not all bugs can be turned into exploits, however—some bugs only cause problems or shut down core systems. And while bugs are usually discovered by accident, most exploits require careful craft. It is unlikely that you'd be able to turn a known bug into an exploit without also having prior knowledge of the system, usually gained from firmware analysis. At a bare minimum, you'd need basic knowledge of the architecture in order to write the necessary code. Most of the time, this knowledge needs to be gathered through research prior to writing an exploit.

It's hard to find bugs that make suitable attack vectors and it's often just as difficult to write exploits for them, so exploits that build on bugs are fairly uncommon. While it is foolish to discount the relevance of exploits, the other methods presented here and in Chapter 8 are much more practical paths to understanding and reprogramming automotive systems in most cases.

Reversing Automotive Firmware

Hacking into an automotive control module far enough to retrieve its current firmware and configuration is really just the beginning of the adventure. At this point, you probably have anywhere from 4KB to 4MB of raw machine-ready code, with a mixture of various parameters and actual code that forms the program the processor will run. Let's say you have a binary blob in the firmware from one of the hacks in this chapter or the chapters later in this book. Next you need to disassemble the binary.

First, you must know which chip this binary is for. There are several free decompilers for different chips out on the Internet. Otherwise you can drop some cash and buy IDA Pro, which supports a large variety of chips. These tools will convert the hex values in the binary into assembler instructions. The next stage is to figure out what exactly you are looking at.

When you're starting to analyze raw data, a high-level understanding of the function of the devices you're reverse engineering will be key to knowing what to look for. You can follow a number of *breadcrumbs*, or clues, for starters; these breadcrumbs are almost guaranteed to lead you to interesting and useful material. Next, we'll look at a few specific examples of how to use common automotive controller functions to gain insight into their operation, which will hopefully allow us to change their behavior.

Self-Diagnostic System

Every engine controller has some type of self-diagnostic system that typically monitors most critical engine functions, and analyzing this is an excellent route to understanding firmware. A good first step in investigative disassembly is to identify the location of these procedures. This will provide you with insight into the memory locations involved in all of the sensors and functions that are checked for errors. Any modern vehicle should support OBD-II packets, which standardize the diagnostic data reported. Even controllers created prior to OBD-II standards have a way to report faults. Some have a system where an analog input is shorted to ground and either an internal LED or the "check engine" light flashes out the code. For example, knowing that code 10 refers to a failed intake air temperature sensor means you can find the piece of code that sets error code 10 to help you identify the internal variables associated with the air temperature sensor.

For more detailed information on using diagnostics, see Chapter 4.

Library Procedures

Being able to change the behavior of a control unit is often one of the primary goals of reverse engineering ECU firmware, and identifying data used by a controller is an important step in the process. Most ECUs have a set of library functions used for routine tasks throughout the code. Library functions used for table lookups are worth identifying early on in the reverse engineering process, as these can lead straight to the parameters you're interested in. Each time a table is used, a function is called to fetch a result. Calls to this type of function are among the most frequent, making them easy to spot.

Usually each type of data stored within the ECU—one-dimensional array of bytes; two-dimensional array of words; three-dimensional array of unsigned, signed, and float shorts; and so on—has a unique reference function. When called, each table lookup routine needs to be passed, at a minimum, the table index (or start address) and the axis variables. Often, table lookup routines can be reused to pass information about the structure of the table, such as how many rows and columns are present.

Calibration data is usually stored in program memory, along with the routines accessing them. Microcontrollers typically have special instructions to access program memory, which provide a unique signature to search for and make table lookup routines particularly easy to spot. A secondary characteristic of these lookup routines is that they tend to have lots of interpolation math. In addition, table lookup routines are often grouped closely together in program memory, making it even easier to find others after you've found one. After identifying reference routines, searching for all calls to them can provide a key to identifying the vast majority of data used by the controller to make decisions.

The arguments passed to these functions typically include the start address of a table, its structure or shape, and which variables index elements of the table. Armed with this information, you're much closer to being able to change the behavior of the controller.

Finding Known Tables

One way to identify tables is to leverage the specific physical and electrical characteristics of vehicle sensors, which will display identifiable characteristics within ECU firmware. For example, an ECU with a MAF sensor will have a table that translates raw readings of voltage or frequency from the MAF into airflow into the engine, providing an internal representation.

Fortunately for us, the signal output from an MAF is determined by physics—that is, King's Law—so the curve will always have a characteristic shape, though it'll be slightly different for each sensor. This will result in the tables having a characteristic set of values that can be observed in the ROM. Armed with the knowledge that there will be universal data to identify, let's take a closer look at how calibration data is displayed in different programs.

Figures 6-1 and 6-2 show similarly shaped Ford and Nissan sensor curves; the similarity they illustrate extends to multiple manufacturers.

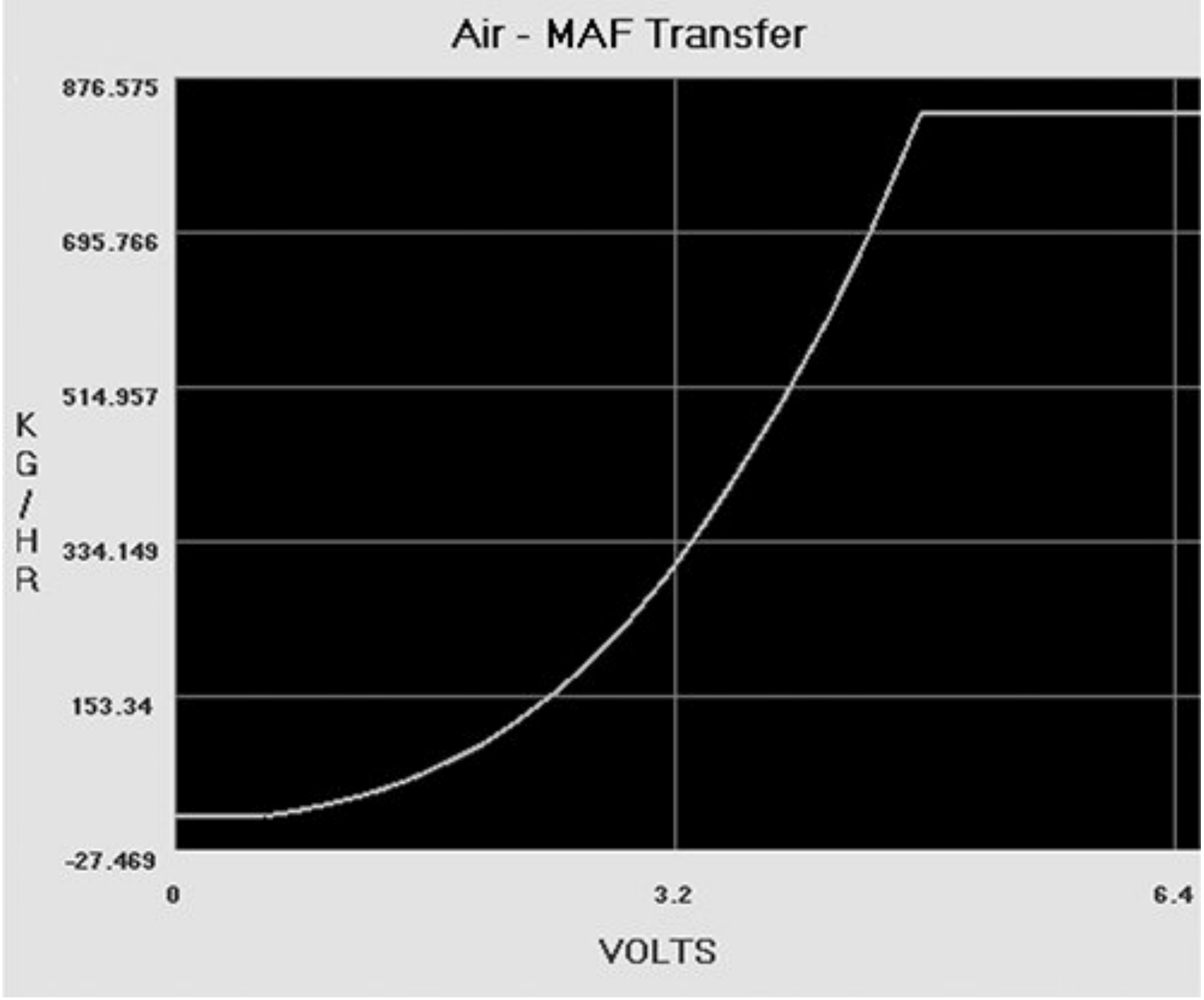


Figure 6-1: Ford MAF transfer graph

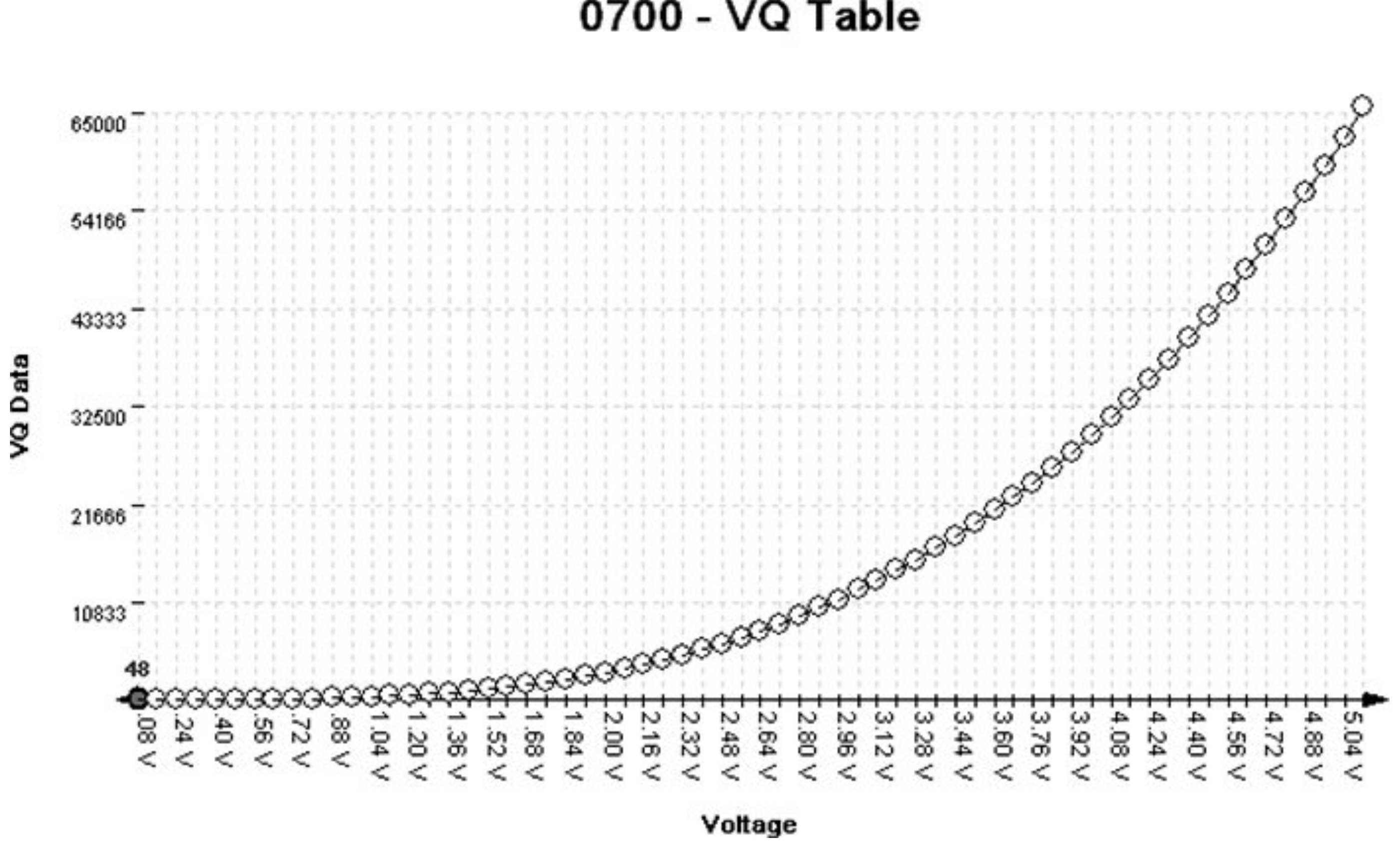
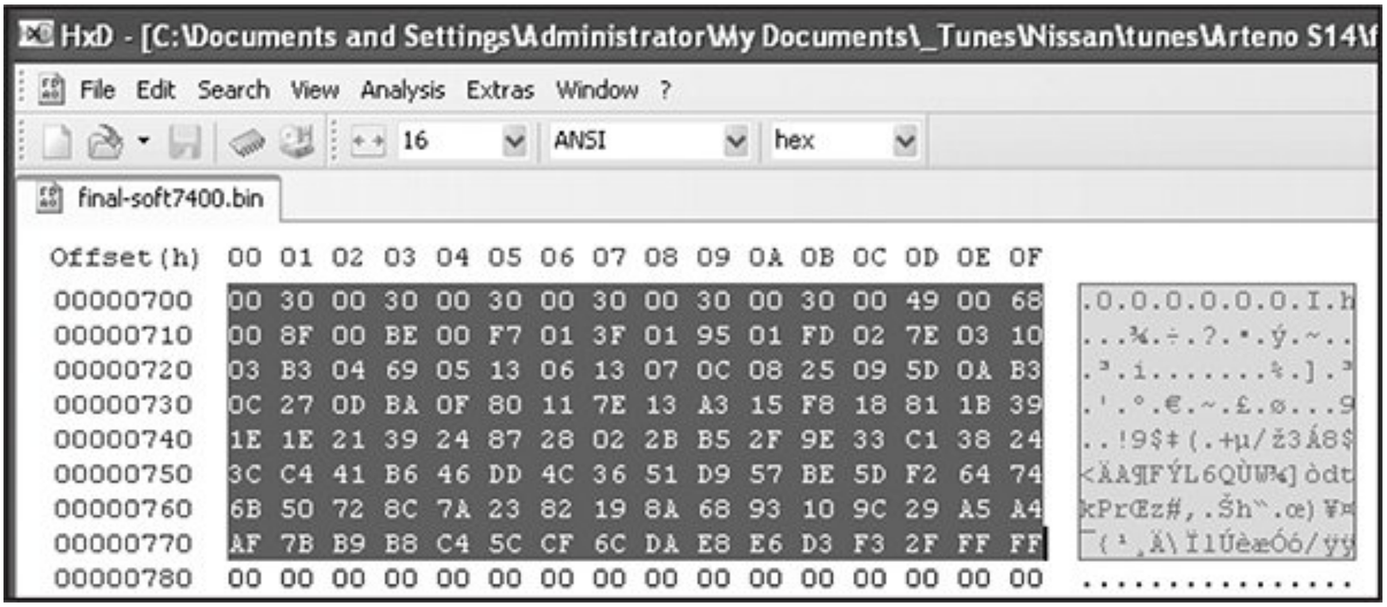


Figure 6-2: Nissan MAF VQ graph

Figures 6-2 through 6-6 show five different views of the same data. Figure 6-3 shows how the VQ curve pictured in Figure 6-2 would look in a hex editor.

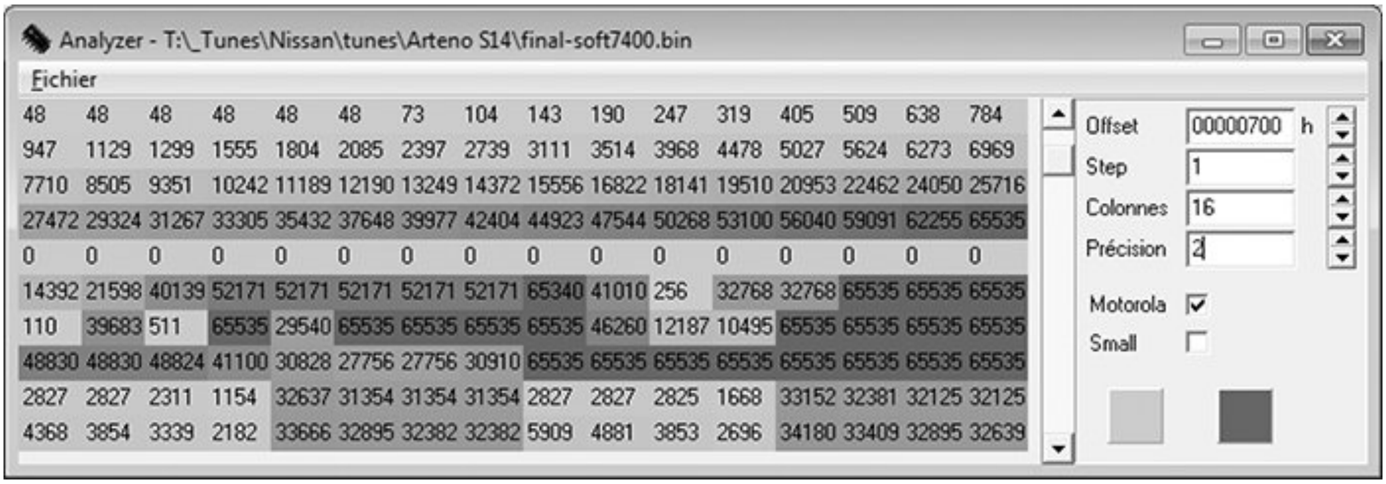

HxD - [C:\Documents and Settings\Administrator\My Documents_Tunes\Nissan\tunes\Arteno S14\f
File
Edit
Search
View
Analysis
Extras
Window
?
16
ANSI
hex
final-soft7400.bin

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000700  00 30 00 30 00 30 00 30 00 30 00 30 00 49 00 68  .0.0.0.0.0.0.I.h
00000710  00 8F 00 BE 00 F7 01 3F 01 95 01 FD 02 7E 03 10  ...¾.÷.?.•.ý.~..
00000720  03 B3 04 69 05 13 06 13 07 0C 08 25 09 5D 0A B3  .³.i.......%.].³
00000730  0C 27 0D BA 0F 80 11 7E 13 A3 15 F8 18 81 1B 39  .'.º.€.~.£.ø...9
00000740  1E 1E 21 39 24 87 28 02 2B B5 2F 9E 33 C1 38 24  ..!9$‡(.+µ/ž3Á8$
00000750  3C C4 41 B6 46 DD 4C 36 51 D9 57 BE 5D F2 64 74  <ÄA¶FÝL6QÙW¾]òdt
00000760  6B 50 72 8C 7A 23 82 19 8A 68 93 10 9C 29 A5 A4  kPrŒz#‚.Šh".œ)¥¤
00000770  AF 7B B9 B8 C4 5C CF 6C DA E8 E6 D3 F3 2F FF FF  ¯{¹¸Ä\ÏlÚèæÓó/ÿÿ
00000780  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Figure 6-3: VQ table in HxD hex editor: 128 bytes or 64- to

16-bit words

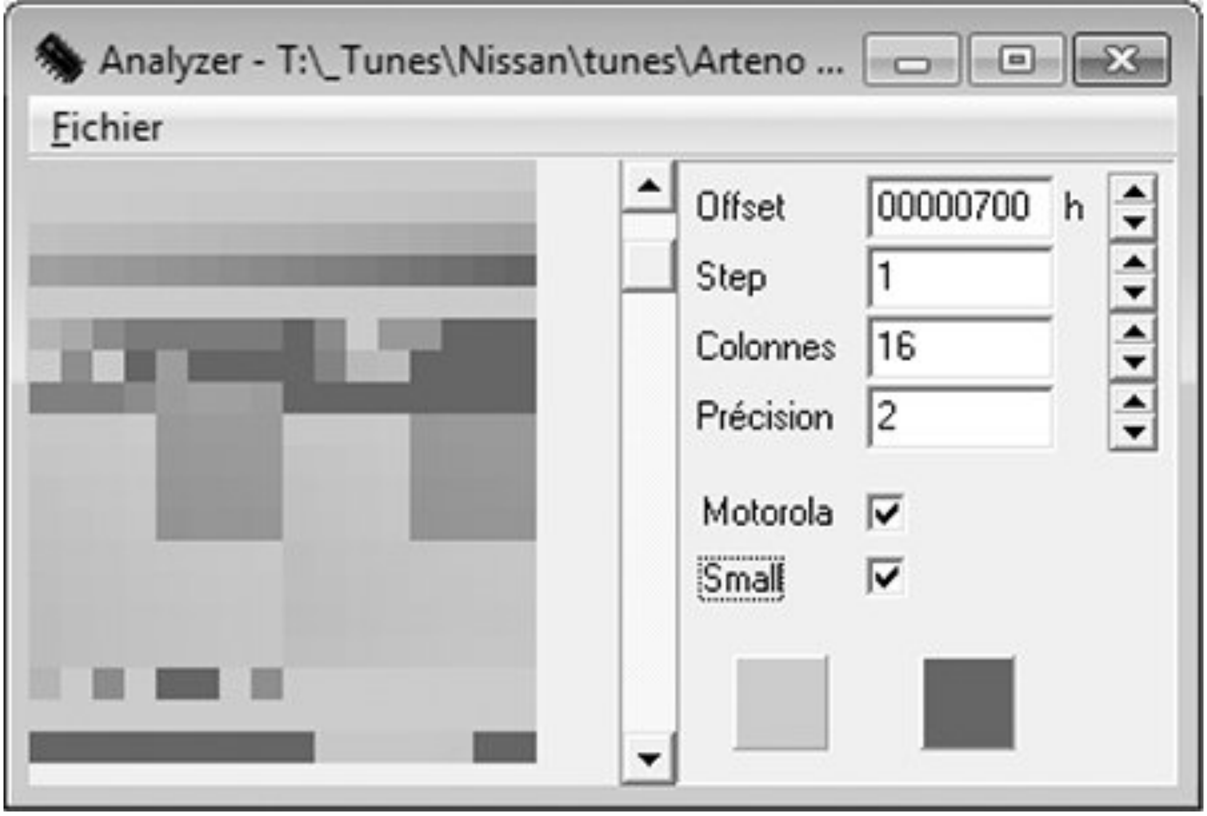
Figures 6-4 and 6-5 show the VQ table in analyze.exe available from *https://github.com/blundar/analyze.exe/*. A simple visualization tool, analyze.exe colors cells based on their numeric value. You can select the precision of the data —for example, 1 = 8-bit byte, 2 = 16-bit word, and 4 = 32-bit long—and how many rows and columns you want present. This simple visual arrangement often makes it easier to identify what is code and what is data than it is when you're using a hex editor, as in Figure 6-3.


Analyzer - T:_Tunes\Nissan\tunes\Arteno S14\final-soft7400.bin

48	48	48	48	48	48	73	104	143	190	247	319	405	509	638	784
947	1129	1299	1555	1804	2085	2397	2739	3111	3514	3968	4478	5027	5624	6273	6969
7710	8505	9351	10242	11189	12190	13249	14372	15556	16822	18141	19510	20953	22462	24050	25716
27472	29324	31267	33305	35432	37648	39977	42404	44923	47544	50268	53100	56040	59091	62255	65535
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14392	21598	40139	52171	52171	52171	52171	52171	65340	41010	256	32768	32768	65535	65535	65535
110	39683	511	65535	29540	65535	65535	65535	65535	46260	12187	10495	65535	65535	65535	65535
48830	48830	48824	41100	30828	27756	27756	30910	65535	65535	65535	65535	65535	65535	65535	65535
2827	2827	2311	1154	32637	31354	31354	31354	2827	2827	2825	1668	33152	32381	32125	32125
4368	3854	3339	2182	33666	32895	32382	32382	5909	4881	3853	2696	34180	33409	32895	32639

Offset	00000700 h
Step	1
Colonnes	16
Précision	2
Motorola	✓
Small	

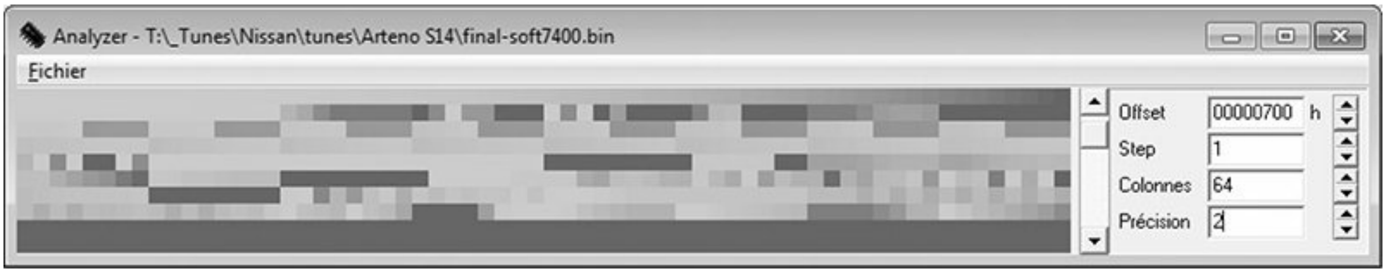
Figure 6-4: VQ table in analyze.exe: values from 48 to 65535 in first four rows of 16×16-bit values


Analyzer - T:_Tunes\Nissan\tunes\Arteno ...

Offset 00000700 h
Step 1
Colonnes 16
Précision 2
Motorola ✓
Small ✓

Figure 6-5: First four rows of 16x16-bit values

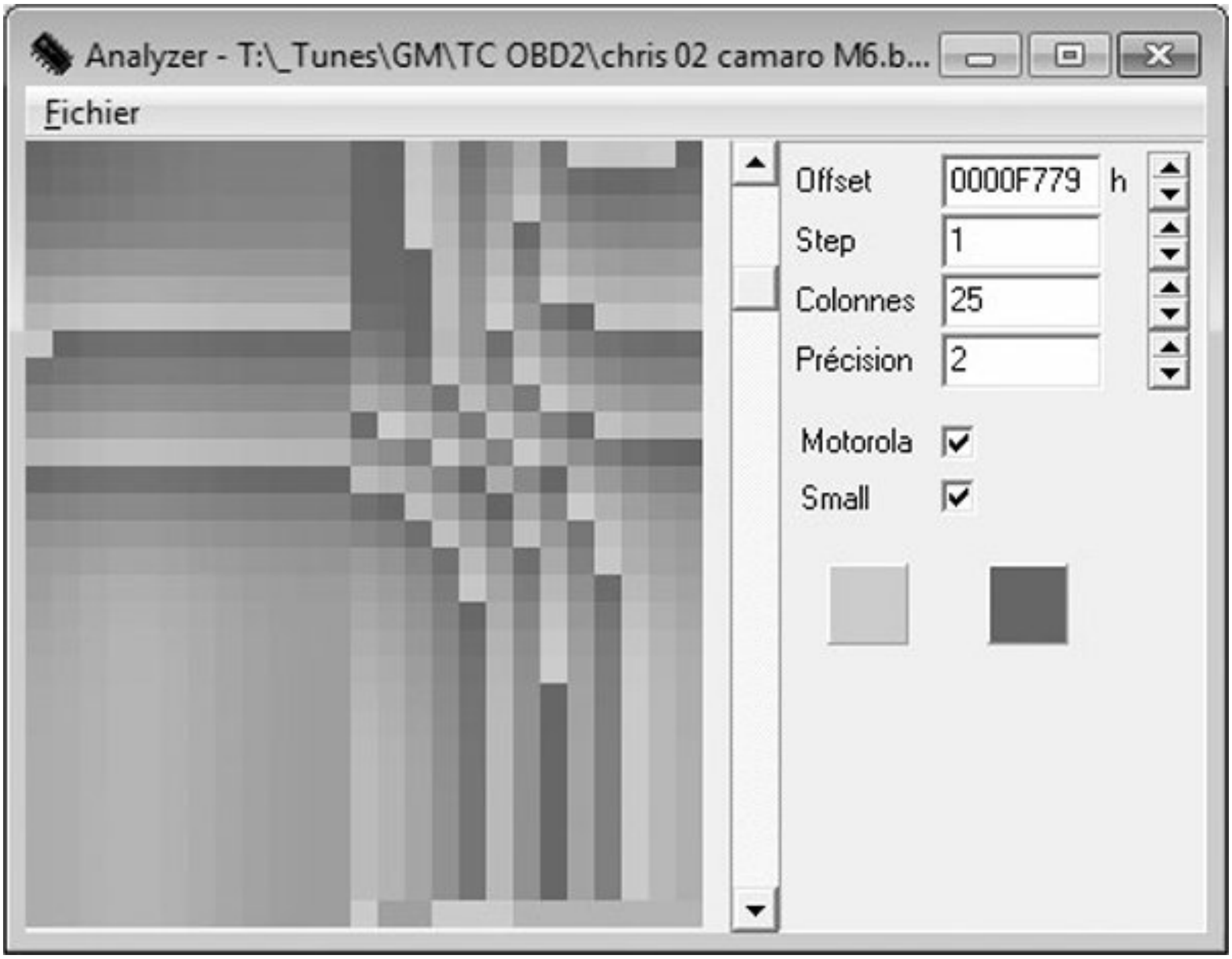
Look again at the first four rows of 16×16-bit values in Figures 6-4 and 6-5 shaded in analyze.exe. Notice how the smooth nonlinear curve in Figures 6-1 and 6-2 mimics the smooth nonlinear progression of values. Figure 6-6 shows the same values in a 64-column layout, so you can see the full gradient of the first four rows from Figure 6-5. No matter what type of vehicle you're looking at, the overall data structures will be similar.


Analyzer - T:_Tunes\Nissan\tunes\Arteno S14\final-soft7400.bin

Offset 00000700 h
Step 1
Colonnes 64
Précision 2

Figure 6-6: 64- to 16-bit words per row

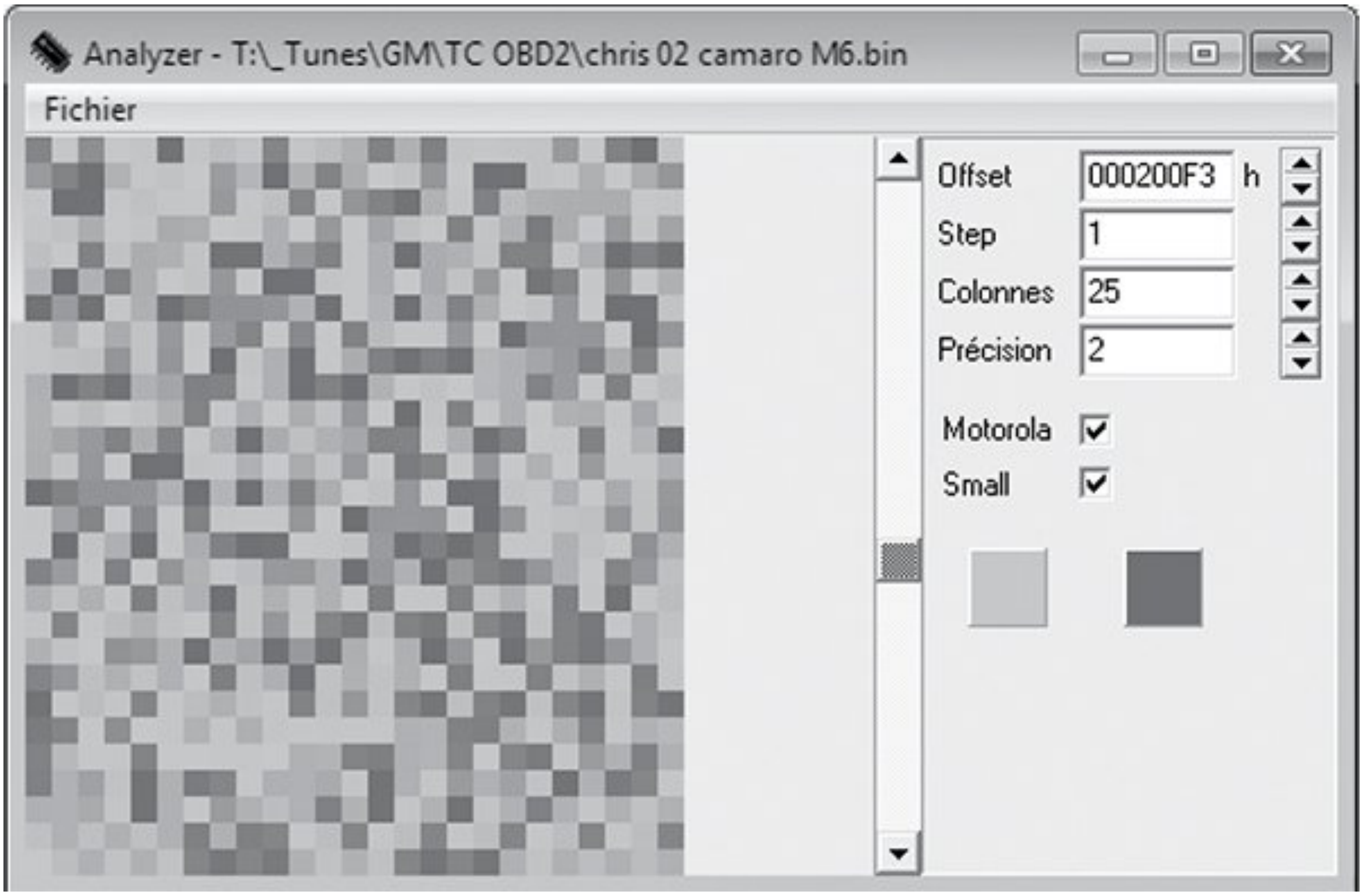
Data visualization tools like hex editors or analyze.exe can also be useful when you don't know the exact shape or pattern you are looking for. No matter what type of vehicle you're looking at, data structures will have orders and patterns that are not typically seen in executable code. Figure 6-7 shows an example of the clear visual pattern of data in analyze.exe—gradually changing values and repetition should stand out.



Offset 0000F779 h
Step 1
Colonnes 25
Précision 2
Motorola ✓
Small ✓

Figure 6-7: Patterns and gradual changes in table data appear in a 2002 Chevrolet Camaro ROM visualized with analyze.exe

On the other hand, when you look at code like that in Figure 6-8, there is a more random, chaotic appearance. (In Figures 6-7 and 6-8, precision is set to 2 because the microcontroller unit used is a 16-bit processor and it's reasonable to assume that a good chunk of the data items will be 16-bit as well.)



Offset 000200F3 h
Step 1
Colonnes 25
Précision 2
Motorola ✓
Small ✓

Figure 6-8: This random code doesn't have the neat, orderly patterns that are present in most tables.

More to Learn from the MCU

Hopefully, these examples help connect knowledge of the table data you expect to find with their representation within a binary blob. Learning the capabilities of the microcontroller unit (MCU) used in a target system can shed light on the types of data to expect when looking over the binary data.

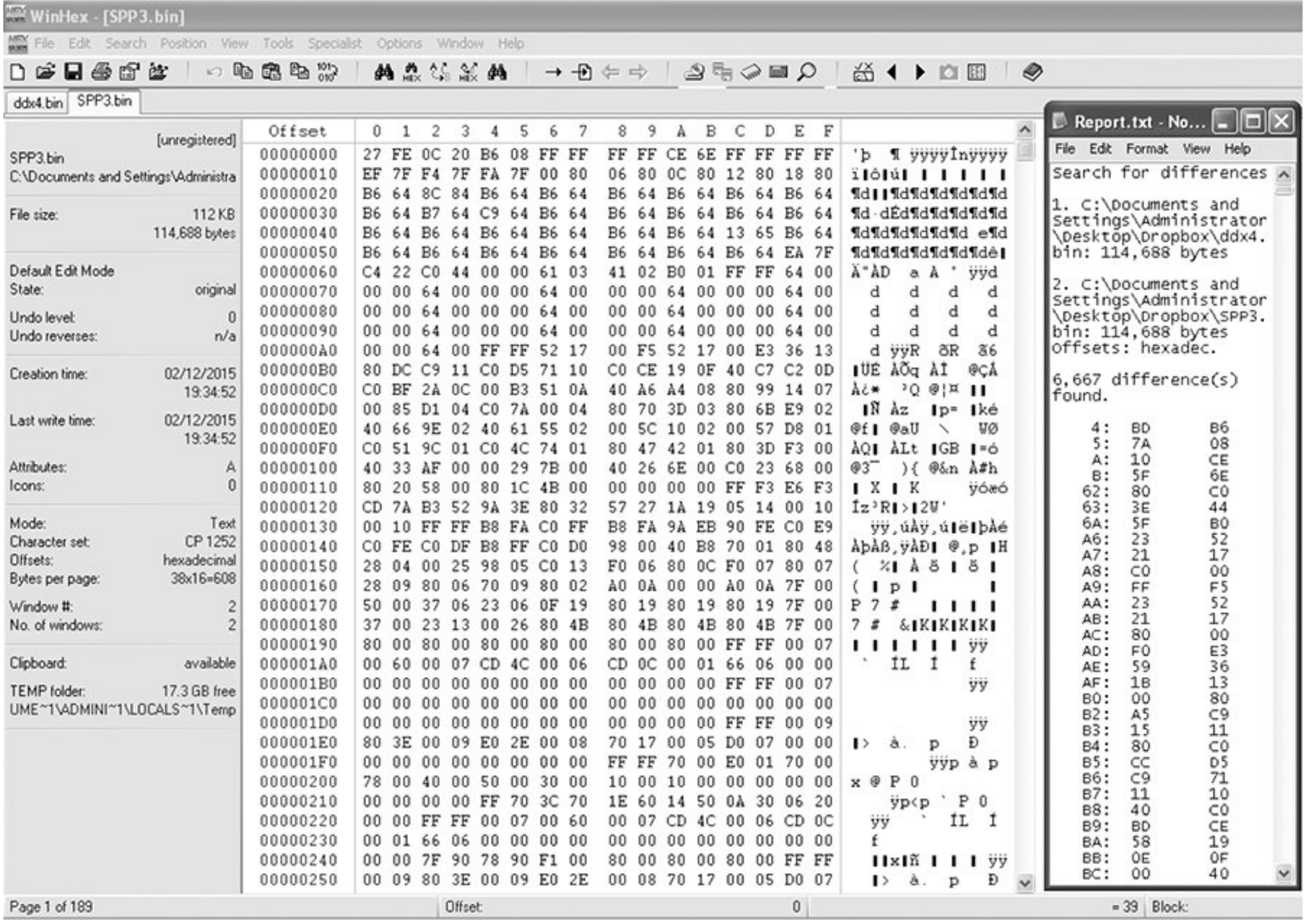
Generally, data representation formats are dictated by the hardware present. Knowing the size of registers on the MCU running the show can be a big help for identifying parameters. Most parameters tend to be the same size as or smaller than the registers of a given MCU. An 8-bit MCU, like a 68HC11, is likely to have lots of 8-bit data. It's unusual to see mostly 4-byte, or 32-bit, unsigned long integers on an 8-bit MCU. While 16-bit data becomes more common on MCUs like the 68332, 32-bit data becomes the norm with MPC5xx Power Architecture MCUs and so on. It's unusual to find floating-point data on an MCU that lacks a floating-point processor.

Comparing Bytes to Identify Parameters

It's often possible to get multiple bins that'll run on the same physical ECU. The more the better! Doing a simple compare in a hex editor will show which bytes differ between the files. It's common—but not guaranteed—for code to remain unchanged while parameters change. If less than 5 percent of the files differ, it's generally safe to assume that the differences are parameters. If you know what's been changed functionally between the two bins and you know which bytes have changed, you have further clues to help correlate changes in the ROM with changes in parameters.

Figures 6-9 and 6-10 compare a 1996 V8 Mustang and a 1997 V6 Thunderbird, showing 6,667 differences out of 114,688 bytes. This is an extreme example of having the same code with different parameters, but there's still only about a 5.8 percent difference compared to overall file size.

Most processors use an interrupt vector table defined by the processor being used. Referencing the processor's data sheet will define the structure of interrupt routines, allowing you to quickly identify the interrupt handlers. Tracing interrupt pins on the processor to circuitry within the ECU to pins you can reference in a vehicle wiring diagram can help you identify code blocks used to service such hardware functions as fuel and spark control, crank and cam signal processing, and idle functions.



WinHex - [SPP3.bin]

ddx4.bin | SPP3.bin

```
[unregistered]
SPP3.bin
C:\Documents and Settings\Administra

File size:                     112 KB
                        114,688 bytes

Default Edit Mode
State:                       original

Undo level:                         0
Undo reverses:                    n/a

Creation time:             02/12/2015
                             19:34:52

Last write time:           02/12/2015
                             19:34:52

Attributes:                         A
Icons:                              0

Mode:                            Text
Character set:                CP 1252
Offsets:                  hexadecimal
Bytes per page:           38x16=608

Window #:                           2
No. of windows:                     2

Clipboard:                  available

TEMP folder:             17.3 GB free
UME~1\ADMINI~1\LOCALS~1\Temp
```

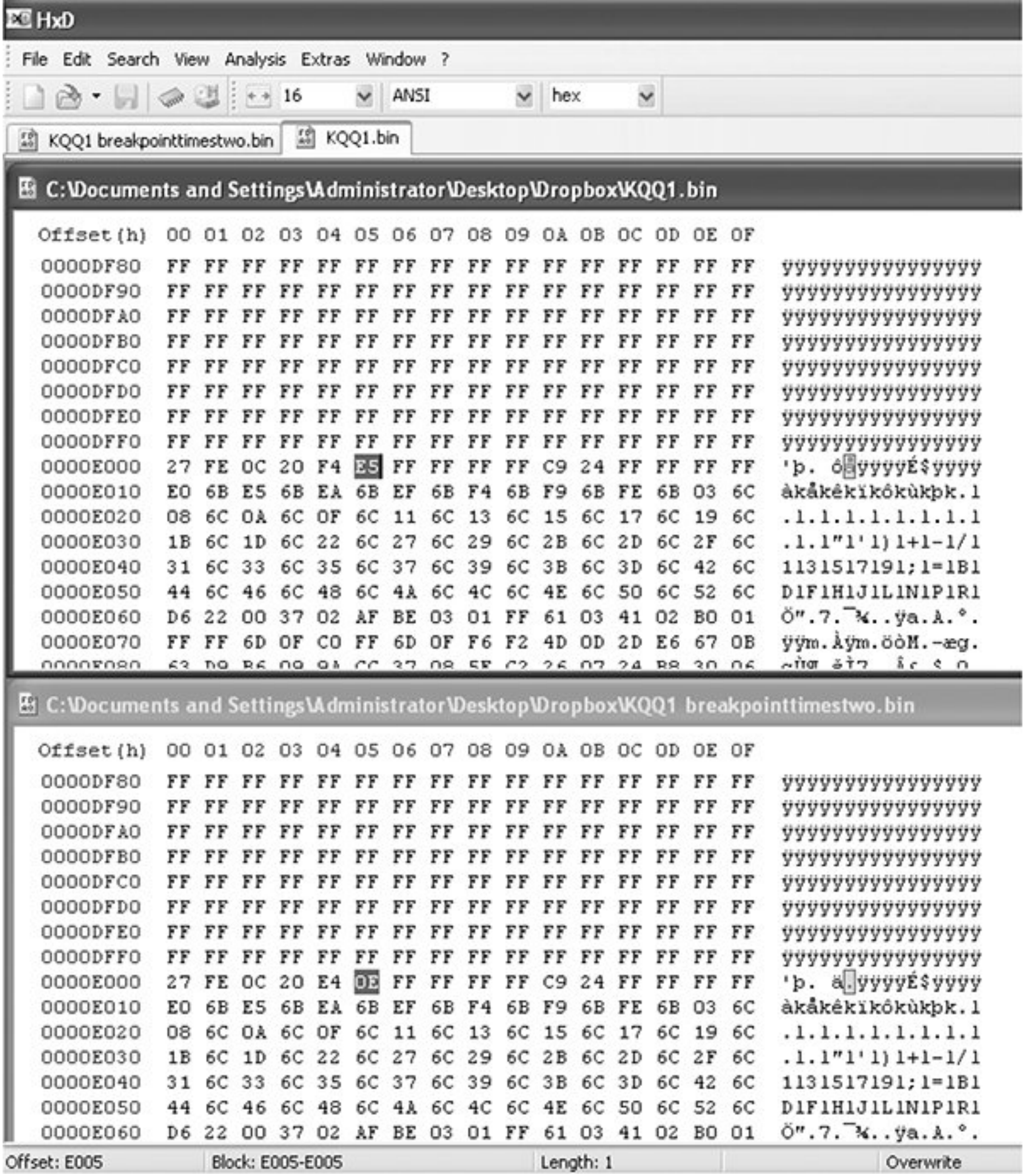
```
  Offset    0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00000000   27 FE 0C 20 B6 08 FF FF  FF FF CE 6E FF FF FF FF   'þ  ¶ ÿÿÿÿÎnÿÿÿÿ
00000010   EF 7F F4 7F FA 7F 00 80  06 80 0C 80 12 80 18 80   ï▮ô▮ú▮ ▮ ▮ ▮ ▮ ▮
00000020   B6 64 8C 84 B6 64 B6 64  B6 64 B6 64 B6 64 B6 64   ¶d▮▮¶d¶d¶d¶d¶d¶d
00000030   B6 64 B7 64 C9 64 B6 64  B6 64 B6 64 B6 64 B6 64   ¶d·dÉd¶d¶d¶d¶d¶d
00000040   B6 64 B6 64 B6 64 B6 64  B6 64 B6 64 13 65 B6 64   ¶d¶d¶d¶d¶d¶d e¶d
00000050   B6 64 B6 64 B6 64 B6 64  B6 64 B6 64 B6 64 EA 7F   ¶d¶d¶d¶d¶d¶d¶dê▮
00000060   C4 22 C0 44 00 00 61 03  41 02 B0 01 FF FF 64 00   Ä"ÀD  a A ° ÿÿd
00000070   00 00 64 00 00 00 64 00  00 00 64 00 00 00 64 00     d   d   d   d
00000080   00 00 64 00 00 00 64 00  00 00 64 00 00 00 64 00     d   d   d   d
00000090   00 00 64 00 00 00 64 00  00 00 64 00 00 00 64 00     d   d   d   d
000000A0   00 00 64 00 FF FF 52 17  00 F5 52 17 00 E3 36 13     d ÿÿR  õR  ã6
000000B0   80 DC C9 11 C0 D5 71 10  C0 CE 19 0F 40 C7 C2 0D   ▮ÜÉ ÀÕq ÀÎ  @ÇÂ
000000C0   C0 BF 2A 0C 00 B3 51 0A  40 A6 A4 08 80 99 14 07   À¿*  ³Q @¦¤ ▮▮
000000D0   00 85 D1 04 C0 7A 00 04  80 70 3D 03 80 6B E9 02    ▮Ñ Àz  ▮p= ▮ké
000000E0   40 66 9E 02 40 61 55 02  00 5C 10 02 00 57 D8 01   @f▮ @aU  \   WØ
000000F0   C0 51 9C 01 C0 4C 74 01  80 47 42 01 80 3D F3 00   ÀQ▮ ÀLt ▮GB ▮=ó
00000100   40 33 AF 00 00 29 7B 00  40 26 6E 00 C0 23 68 00   @3¯  ){ @&n À#h
00000110   80 20 58 00 80 1C 4B 00  00 00 00 00 FF F3 E6 F3   ▮ X ▮ K     ÿóæó
00000120   CD 7A B3 52 9A 3E 80 32  57 27 1A 19 05 14 00 10   Íz³R▮>▮2W'
00000130   00 10 FF FF B8 FA C0 FF  B8 FA 9A EB 90 FE C0 E9     ÿÿ¸úÀÿ¸ú▮ë▮þÀé
00000140   C0 FE C0 DF B8 FF C0 D0  98 00 40 B8 70 01 80 48   ÀþÀß¸ÿÀÐ▮ @¸p ▮H
00000150   28 04 00 25 98 05 C0 13  F0 06 80 0C F0 07 80 07   (  %▮ À ð ▮ ð ▮
00000160   28 09 80 06 70 09 80 02  A0 0A 00 00 A0 0A 7F 00   ( ▮ p ▮       ▮
00000170   50 00 37 06 23 06 0F 19  80 19 80 19 80 19 7F 00   P 7 #    ▮ ▮ ▮ ▮
00000180   37 00 23 13 00 26 80 4B  80 4B 80 4B 80 4B 7F 00   7 #  &▮K▮K▮K▮K▮
00000190   80 00 80 00 80 00 80 00  80 00 80 00 FF FF 00 07   ▮ ▮ ▮ ▮ ▮ ▮ ÿÿ
000001A0   00 60 00 07 CD 4C 00 06  CD 0C 00 01 66 06 00 00    `   ÍL  Í   f
000001B0   00 00 00 00 00 00 00 00  00 00 00 00 FF FF 00 07               ÿÿ
000001C0   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
000001D0   00 00 00 00 00 00 00 00  00 00 00 00 FF FF 00 09               ÿÿ
000001E0   80 3E 00 09 E0 2E 00 08  70 17 00 05 D0 07 00 00   ▮>  à.  p   Ð
000001F0   00 00 00 00 00 00 00 00  FF FF 70 00 E0 01 70 00           ÿÿp à p
00000200   78 00 40 00 50 00 30 00  10 00 10 00 00 00 00 00   x @ P 0
00000210   00 00 00 00 FF 70 3C 70  1E 60 14 50 0A 30 06 20       ÿp<p ` P 0
00000220   00 00 FF FF 00 07 00 60  00 07 CD 4C 00 06 CD 0C     ÿÿ   `   ÍL  Í
00000230   00 01 66 06 00 00 00 00  00 00 00 00 00 00 00 00     f
00000240   00 00 7F 90 78 90 F1 00  80 00 80 00 80 00 FF FF     ▮▮x▮ñ ▮ ▮ ▮ ÿÿ
00000250   00 09 80 3E 00 09 E0 2E  00 08 70 17 00 05 D0 07     ▮>  à.  p   Ð
```

Report.txt - No...

```
Search for differences

1. C:\Documents and
Settings\Administrator
\Desktop\Dropbox\ddx4.
bin: 114,688 bytes

2. C:\Documents and
Settings\Administrator
\Desktop\Dropbox\SPP3.
bin: 114,688 bytes
Offsets: hexadec.

6,667 difference(s)
found.

     4:  BD      B6
     5:  7A      08
     A:  10      CE
     B:  5F      6E
    62:  80      C0
    63:  3E      44
    6A:  5F      B0
    A6:  23      52
    A7:  21      17
    A8:  C0      00
    A9:  FF      F5
    AA:  23      52
    AB:  21      17
    AC:  80      00
    AD:  F0      E3
    AE:  59      36
    AF:  1B      13
    B0:  00      80
    B2:  A5      C9
    B3:  15      11
    B4:  80      C0
    B5:  CC      D5
    B6:  C9      71
    B7:  11      10
    B8:  40      C0
    B9:  BD      CE
    BA:  58      19
    BB:  0E      0F
    BC:  00      40
```

Page 1 of 189 | Offset: 0 | = 39 | Block:

Figure 6-9: Comparison of a 1996 V8 Mustang (DXE2.bin) *and a 1997 V6 Thunderbird* (SPP3.bin)


HxD
File Edit Search View Analysis Extras Window ?
16
ANSI
hex
KQQ1 breakpointtimestwo.bin
KQQ1.bin

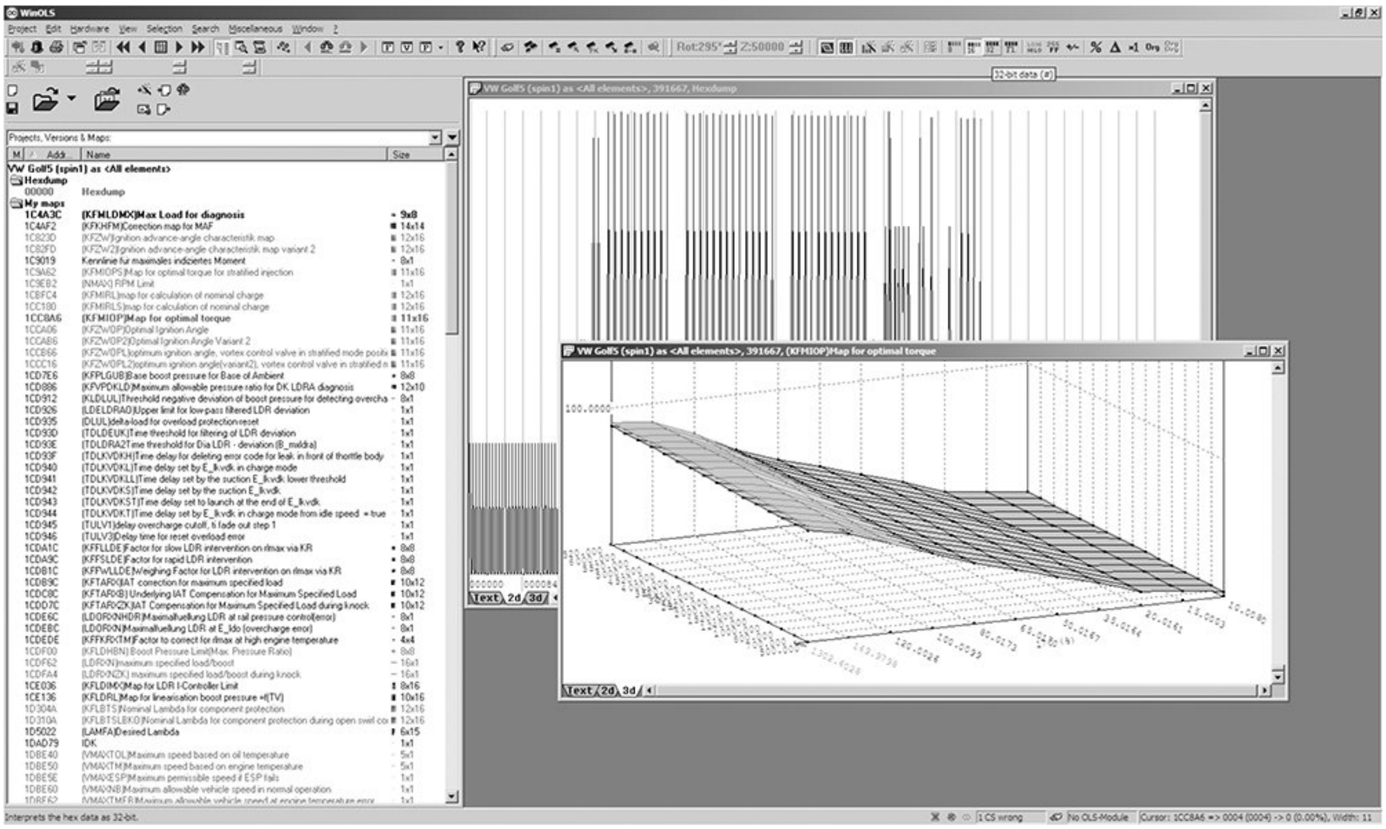
```
C:\Documents and Settings\Administrator\Desktop\Dropbox\KQQ1.bin

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

0000DF80  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000DF90  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000DFA0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000DFB0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000DFC0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000DFD0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000DFE0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000DFF0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000E000  27 FE 0C 20 F4 E5 FF FF FF FF C9 24 FF FF FF FF  'þ. ôåÿÿÿÿÉ$ÿÿÿÿ
0000E010  E0 6B E5 6B EA 6B EF 6B F4 6B F9 6B FE 6B 03 6C  àkåkêkïkôkùkþk.l
0000E020  08 6C 0A 6C 0F 6C 11 6C 13 6C 15 6C 17 6C 19 6C  .l.l.l.l.l.l.l.l
0000E030  1B 6C 1D 6C 22 6C 27 6C 29 6C 2B 6C 2D 6C 2F 6C  .l.l"l'l)l+l-l/l
0000E040  31 6C 33 6C 35 6C 37 6C 39 6C 3B 6C 3D 6C 42 6C  1l3l5l7l9l;l=lBl
0000E050  44 6C 46 6C 48 6C 4A 6C 4C 6C 4E 6C 50 6C 52 6C  DlFlHlJlLlNlPlRl
0000E060  D6 22 00 37 02 AF BE 03 01 FF 61 03 41 02 B0 01  Ö".7.¯¾..ÿa.A.°.
0000E070  FF FF 6D 0F C0 FF 6D 0F F6 F2 4D 0D 2D E6 67 0B  ÿÿm.Àÿm.öòM.-æg.

C:\Documents and Settings\Administrator\Desktop\Dropbox\KQQ1 breakpointtimestwo.bin

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

0000DF80  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000DF90  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000DFA0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000DFB0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000DFC0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000DFD0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000DFE0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000DFF0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000E000  27 FE 0C 20 E4 0E FF FF FF FF C9 24 FF FF FF FF  'þ. ä.ÿÿÿÿÉ$ÿÿÿÿ
0000E010  E0 6B E5 6B EA 6B EF 6B F4 6B F9 6B FE 6B 03 6C  àkåkêkïkôkùkþk.l
0000E020  08 6C 0A 6C 0F 6C 11 6C 13 6C 15 6C 17 6C 19 6C  .l.l.l.l.l.l.l.l
0000E030  1B 6C 1D 6C 22 6C 27 6C 29 6C 2B 6C 2D 6C 2F 6C  .l.l"l'l)l+l-l/l
0000E040  31 6C 33 6C 35 6C 37 6C 39 6C 3B 6C 3D 6C 42 6C  1l3l5l7l9l;l=lBl
0000E050  44 6C 46 6C 48 6C 4A 6C 4C 6C 4E 6C 50 6C 52 6C  DlFlHlJlLlNlPlRl
0000E060  D6 22 00 37 02 AF BE 03 01 FF 61 03 41 02 B0 01  Ö".7.¯¾..ÿa.A.°.

Offset: E005    Block: E005-E005    Length: 1    Overwrite
```

Figure 6-10: File compare function of the HxD hex editor

Identifying ROM Data with WinOLS

WinOLS is a popular commercial program for modifying bins. It combines a series of tools for calculating and updating checksums within a ROM with a set of tools for identifying tables. Figures 6-11 and 6-12 illustrate WinOLS in use.

If the ROM type is known, it has many templates that automatically identify configuration parameters. Most of the known built-in ROM types are geared toward Bosch Motronic ECUs. Templates and configurations can be saved, shared, and sold to enable users to make modifications to specific files with greater ease. WinOLS is arguably the most common software used for identifying interesting data within a ROM that doesn't involve code analysis. It's designed to facilitate rapid tuning changes to a controller.

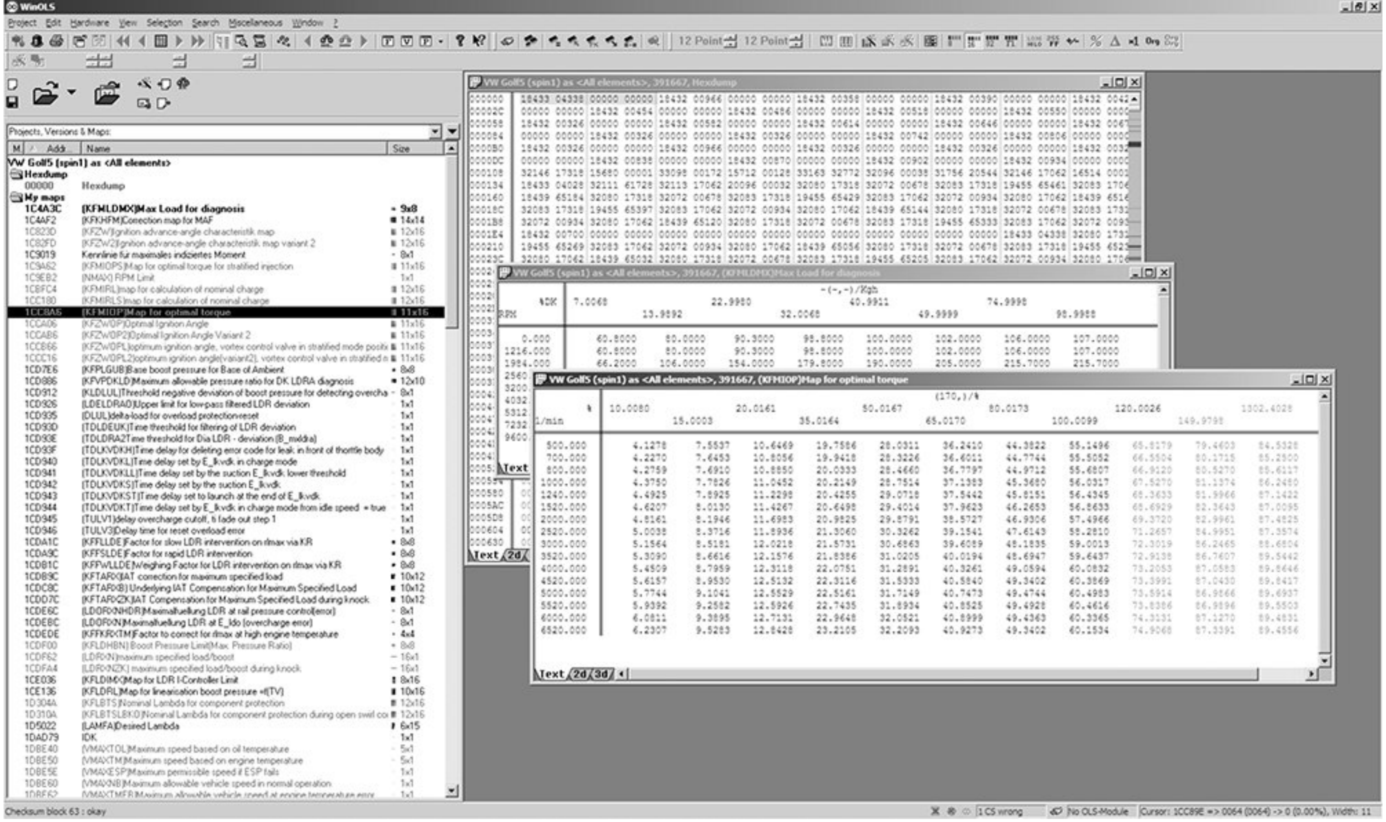

WinOLS
Project Edit Hardware View Selection Search Miscellaneous Window ?
Rot:295° Z:50000
32-bit data (#)
VW Golf5 (spin1) as <All elements>, 391667, Hexdump
VW Golf5 (spin1) as <All elements>, 391667, (KFMIOP)Map for optimal torque
100.0000
Text 2d 3d
Projects, Versions & Maps:
1 CS wrong
No OLS-Module
Cursor: 1CC8A6 => 0004 (0004) -> 0 (0.00%), Width: 11

VW Golf5 (spin1) as <All elements>

M	Addr.	Name	Size
	Hexdump		
	00000	Hexdump	
	My maps		
	1C4A3C	(KFMLDMX)Max Load for diagnosis	9x8
	1C4AF2	(KFKHFM)Correction map for MAF	14x14
	1C823D	(KFZW)Ignition advance-angle characteristik map	12x16
	1C82FD	(KFZW2)Ignition advance-angle characteristik map variant 2	12x16
	1C9019	Kennlinie für maximales indiziertes Moment	8x1
	1C9A62	(KFMIOPS)Map for optimal torque for stratified injection	11x16
	1C9EB2	(NMAX) RPM Limit	1x1
	1CBFC4	(KFMIRL)map for calculation of nominal charge	12x16
	1CC180	(KFMIRLS)map for calculation of nominal charge	12x16
	1CC8A6	(KFMIOP)Map for optimal torque	11x16
	1CCA06	(KFZWOP)Optimal Ignition Angle	11x16
	1CCAB6	(KFZWOP2)Optimal Ignition Angle Variant 2	11x16
	1CCB66	(KFZWOPL)optimum ignition angle, vortex control valve in stratified mode positi	11x16
	1CCC16	(KFZWOPL2)optimum ignition angle(variant2), vortex control valve in stratified m	11x16
	1CD7E6	(KFPLGUB)Base boost pressure for Base of Ambient	8x8
	1CD886	(KFVPDKLD)Maximum allowable pressure ratio for DK LDRA diagnosis	12x10
	1CD912	(KLDLUL)Threshold negative deviation of boost pressure for detecting overcha	8x1
	1CD926	(LDELDRAO)Upper limit for low-pass filtered LDR deviation	1x1
	1CD935	(DLUL)delta-load for overload protection-reset	1x1
	1CD93D	(TDLDEUK)Time threshold for filtering of LDR deviation	1x1
	1CD93E	(TDLDRA2Time threshold for Dia LDR - deviation (B_mldra)	1x1
	1CD93F	(TDLKVDKH)Time delay for deleting error code for leak in front of thottle body	1x1
	1CD940	(TDLKVDKL)Time delay set by E_lkvdk in charge mode	1x1
	1CD941	(TDLKVDKLL)Time delay set by the suction E_lkvdk lower threshold	1x1
	1CD942	(TDLKVDKS)Time delay set by the suction E_lkvdk	1x1
	1CD943	(TDLKVDKST)Time delay set to launch at the end of E_lkvdk	1x1
	1CD944	(TDLKVDKT)Time delay set by E_lkvdk in charge mode from idle speed = true	1x1
	1CD945	(TULV1)delay overcharge cutoff, ti fade out step 1	1x1
	1CD946	(TULV3)Delay time for reset overload error	1x1
	1CDA1C	(KFFLLDE)Factor for slow LDR intervention on rlmax via KR	8x8
	1CDA9C	(KFFSLDE)Factor for rapid LDR intervention	8x8
	1CDB1C	(KFFWLLDE)Weighing Factor for LDR intervention on rlmax via KR	8x8
	1CDB9C	(KFTARXX)IAT correction for maximum specified load	10x12
	1CDC8C	(KFTARXB) Underlying IAT Compensation for Maximum Specified Load	10x12
	1CDD7C	(KFTARXZK)IAT Compensation for Maximum Specified Load during knock	10x12
	1CDE6C	(LDORXNHDR)Maximalfuellung LDR at rail pressure control(error)	8x1
	1CDE8C	(LDORXN)Maximalfuellung LDR at E_ldo (overcharge error)	8x1
	1CDEDE	(KFFKRXTM)Factor to correct for rlmax at high engine temperature	4x4
	1CDF00	(KFLDHBN) Boost Pressure Limit(Max. Pressure Ratio)	8x8
	1CDF62	(LDRXN)maximum specified load/boost	16x1
	1CDFA4	(LDRXNZK) maximum specified load/boost during knock	16x1
	1CE036	(KFLDIMX)Map for LDR I-Controller Limit	8x16
	1CE136	(KFLDRL)Map for linearisation boost pressure =f(TV)	10x16
	1D304A	(KFLBTS)Nominal Lambda for component protection	12x16
	1D310A	(KFLBTSLBKO)Nominal Lambda for component protection during open swirl co	12x16
	1D5022	(LAMFA)Desired Lambda	6x15
	1DAD79	IDK	1x1
	1DBE40	(VMAXTOL)Maximum speed based on oil temperature	5x1
	1DBE50	(VMAXTM)Maximum speed based on engine temperature	5x1
	1DBE5E	(VMAXESP)Maximum permissible speed if ESP fails	1x1
	1DBE60	(VMAXNB)Maximum allowable vehicle speed in normal operation	1x1
	1DBE62	(VMAXTMFR)Maximum allowable vehicle speed at engine temperature error	1x1

Interprets the hex data as 32-bit.

Figure 6-11: WinOLS supports 2D and 3D table views, as shown in these alternate views.



```
Projects, Versions & Maps:
M  Addr     Name                                                                  Size
VW Golf5 (spin1) as <All elements>
Hexdump
   00000    Hexdump
My maps
   1C4A3C   (KFMLDMX)Max Load for diagnosis                                       9x8
   1C4AF2   (KFKHFM)Correction map for MAF                                        14x14
   1C823D   (KFZW)Ignition advance-angle characteristik map                       12x16
   1C82FD   (KFZW2)Ignition advance-angle characteristik map variant 2            12x16
   1C9019   Kennlinie für maximales indiziertes Moment                            8x1
   1C9A62   (KFMIOPS)Map for optimal torque for stratified injection              11x16
   1C9EB2   (NMAX) RPM Limit                                                      1x1
   1CBFC4   (KFMIRL)map for calculation of nominal charge                         12x16
   1CC180   (KFMIRLS)map for calculation of nominal charge                        12x16
   1CC8A5   (KFMIOP)Map for optimal torque                                        11x16
   1CCA06   (KFZWOP)Optimal Ignition Angle                                        11x16
   1CCAB6   (KFZWOP2)Optimal Ignition Angle Variant 2                             11x16
   1CCB66   (KFZWOPL)optimum ignition angle, vortex control valve in stratified mode positi 11x16
   1CCC16   (KFZWOPL2)optimum ignition angle(variant2), vortex control valve in stratified n 11x16
   1CD7E6   (KFPLGUB)Base boost pressure for Base of Ambient                      8x8
   1CD886   (KFVPDKLD)Maximum allowable pressure ratio for DK LDRA diagnosis      12x10
   1CD912   (KLDLUL)Threshold negative deviation of boost pressure for detecting overcha 8x1
   1CD926   (LDELDRA0)Upper limit for low-pass filtered LDR deviation             1x1
   1CD935   (DLUL)delta-load for overload protection-reset                        1x1
   1CD93D   (TDLDEUK)Time threshold for filtering of LDR deviation                1x1
   1CD93E   (TDLDRA2Time threshold for Dia LDR - deviation (B_mldra)              1x1
   1CD93F   (TDLKVDKH)Time delay for deleting error code for leak in front of thottle body 1x1
   1CD940   (TDLKVDKL)Time delay set by E_lkvdk in charge mode                    1x1
   1CD941   (TDLKVDKLL)Time delay set by the suction E_lkvdk lower threshold      1x1
   1CD942   (TDLKVDKS)Time delay set by the suction E_lkvdk                       1x1
   1CD943   (TDLKVDKST)Time delay set to launch at the end of E_lkvdk             1x1
   1CD944   (TDLKVDKT)Time delay set by E_lkvdk in charge mode from idle speed = true 1x1
   1CD945   (TULV1)delay overcharge cutoff, ti fade out step 1                    1x1
   1CD946   (TULV3)Delay time for reset overload error                            1x1
   1CDA1C   (KFFLLDE)Factor for slow LDR intervention on rlmax via KR             8x8
   1CDA9C   (KFFSLDE)Factor for rapid LDR intervention                            8x8
   1CDB1C   (KFFWLLDE)Weighing Factor for LDR intervention on rlmax via KR        8x8
   1CDB9C   (KFTARX)IAT correction for maximum specified load                     10x12
   1CDC8C   (KFTARXB) Underlying IAT Compensation for Maximum Specified Load      10x12
   1CDD7C   (KFTARXZK)IAT Compensation for Maximum Specified Load during knock    10x12
   1CDE6C   (LDORXNHDR)Maximalfuellung LDR at rail pressure control(error)        8x1
   1CDEBC   (LDORXN)Maximalfuellung LDR at E_ldo (overcharge error)               8x1
   1CDEDE   (KFFKRXTM)Factor to correct for rlmax at high engine temperature      4x4
   1CDF00   (KFLDHBN) Boost Pressure Limit(Max. Pressure Ratio)                   8x8
   1CDF62   (LDRXN)maximum specified load/boost                                   16x1
   1CDFA4   (LDRXNZK) maximum specified load/boost during knock                   16x1
   1CE036   (KFLDIMX)Map for LDR I-Controller Limit                               8x16
   1CE136   (KFLDRL)Map for linearization boost pressure =f(TV)                   10x16
   1D304A   (KFLBTS)Nominal Lambda for component protection                       12x16
   1D310A   (KFLBTSLBKO)Nominal Lambda for component protection during open swirl co 12x16
   1D5022   (LAMFA)Desired Lambda                                                 6x15
   1DAD79   IDK                                                                   1x1
   1DBE40   (VMAXTOL)Maximum speed based on oil temperature                       5x1
   1DBE50   (VMAXTM)Maximum speed based on engine temperature                     5x1
   1DBE5E   (VMAXESP)Maximum permissible speed if ESP fails                       1x1
   1DBE60   (VMAXNB)Maximum allowable vehicle speed in normal operation           1x1
   1DBE62   (VMAXTMER)Maximum allowable vehicle speed at engine temperature error 1x1
Checksum block 63 : okay
```

```
VW Golf5 (spin1) as <All elements>, 391667, (KFMLDMX)Max Load for diagnosis
                                     -(-,-)/Kgh
          %DK   7.0068          22.9980          40.9911          74.9998
RPM                    13.9892          32.0068          49.9999          98.9988
       0.000     60.8000   80.0000   90.3000   98.8000  100.0000  102.0000  106.0000  107.0000
    1216.000     60.8000   80.0000   90.3000   98.8000  100.0000  102.0000  106.0000  107.0000
    1984.000     66.2000  106.0000  154.0000  179.8000  190.0000  205.0000  215.7000  215.7000
```

```
VW Golf5 (spin1) as <All elements>, 391667, (KFMIOP)Map for optimal torque
                                                   (170,)/%
          %   10.0080          20.0161          50.0167          80.0173          120.0026          1302.4028
1/min                 15.0003          35.0164          65.0170          100.0099          149.9798
   500.000    4.1278   7.5537  10.6469  19.7586  28.0311  36.2410  44.3822  55.1496  65.8179  79.4603  84.5328
   700.000    4.2270   7.6453  10.8056  19.9418  28.3226  36.6011  44.7744  55.5052  66.5504  80.1715  85.2500
   800.000    4.2759   7.6910  10.8850  20.0333  28.4660  36.7797  44.9712  55.6807  66.9120  80.5270  85.6117
  1000.000    4.3750   7.7826  11.0452  20.2149  28.7514  37.1383  45.3680  56.0317  67.5270  81.1374  86.2480
  1240.000    4.4925   7.8925  11.2298  20.4255  29.0718  37.5442  45.8151  56.4345  68.3633  81.9966  87.1422
  1520.000    4.6207   8.0130  11.4267  20.6498  29.4014  37.9623  46.2653  56.8633  68.6929  82.3643  87.0095
  2000.000    4.8161   8.1946  11.6983  20.9825  29.8791  38.5727  46.9306  57.4966  69.3720  82.9961  87.4825
  2520.000    5.0038   8.3716  11.8936  21.3060  30.3262  39.1541  47.6143  58.2810  71.2657  84.9951  87.3574
  3000.000    5.1564   8.5181  12.0218  21.5731  30.6863  39.6089  48.1835  59.0013  72.3019  86.2465  88.6804
  3520.000    5.3090   8.6616  12.1576  21.8386  31.0205  40.0194  48.6947  59.6437  72.9138  86.7607  89.5442
  4000.000    5.4509   8.7959  12.3118  22.0751  31.2891  40.3261  49.0594  60.0832  73.2053  87.0583  89.8646
  4520.000    5.6157   8.9530  12.5132  22.3116  31.5333  40.5840  49.3402  60.3869  73.3991  87.0430  89.8417
  5000.000    5.7744   9.1041  12.5529  22.5161  31.7149  40.7473  49.4744  60.4983  73.5914  86.9866  89.6937
  5520.000    5.9392   9.2582  12.5926  22.7435  31.8934  40.8525  49.4928  60.4616  73.8386  86.9896  89.5503
  6000.000    6.0811   9.3895  12.7131  22.9648  32.0521  40.8999  49.4363  60.3365  74.3131  87.1270  89.4831
  6520.000    6.2307   9.5283  12.8428  23.2105  32.2093  40.9273  49.3402  60.1534  74.9068  87.3391  89.4556
Text 2d 3d
```

Figure 6-12: WinOLS being used on a 2006 Volkswagen 2.0Tsi ECU

Code Analysis

Code analysis can be a long, complicated task. If you're starting from scratch, with no experience, it will likely take hundreds of hours to analyze a complex piece of code. Modern control units often have upward of a megabyte or two of code, which is a huge amount of code when you're looking at it in assembly. An ECU from 1995 with 32 kilobytes (not megabytes) of code will have upward of 10,000 assembly instructions to sort out. Bottom line: do not underestimate how much work this approach will take. I'll briefly introduce a few tools, but I don't have the space to address the topic in sufficient depth for someone unfamiliar with the process. (After all, entire books have been written solely on code analysis.) Here, I'll just talk through specific tools and methods particularly applicable to automotive embedded systems.

When analyzing a new target, first identify the architecture you're working with. Knowing what processor executed the blob of binary will help you choose an appropriate software tool to further assist. If you can't identify a processor based on the markings on the chip itself, search online for data sheets to identify it.

To analyze code, you might need to find a disassembler. A quick Google search reveals that there are lots of them out there. Some target a single architecture—for example, Dis51—and some are custom-written for automotive reverse engineering—for example, Dis66k. Others, like CATS dasm, IDA Pro, Hopper, dasmx, and objdump from the GNU Binary Utilities (binutils), target multiple processors. IDA Pro supports more embedded targets than just about any other program, but it's also one of the most expensive disassemblers. GNU binutils also supports a pretty wide range of architectures, but the version included on most systems will be built only for the "native" architecture. Rebuilding binutils with all architectures enabled will open a few doors. Your budget and supported processors will determine which disassemblers are an option.

Bust out the disassembly tools and start trying to make sense of the mess, but as I warned earlier, this might take hundreds of hours. A divide-and-conquer mentality works best—focus on the smaller tasks rather than the project as a whole. If you obtained the binary by backdoor methods, you probably already took the ECU apart to identify the processor. If you cracked the J2534 programming routines, you might not have a clue what processor is running the show. In this case, you're going to need to keep running it through a disassembler over and over using different settings until you get something that makes sense.

You're looking for assembly code that disassembles cleanly, meaning that it looks like it makes logical sense. If you disassemble a binary for the wrong architecture or using the wrong settings, you'll still see assembly instructions, but the assembler actions won't make sense. Disassembly is a bit of an art, and it may take a little practice at seeing a "clean" assembler to get the hang of identifying when a dissassembler is providing the correct response, especially when nonexecutable tables and data are scattered among the code.

Here are some hints for making sense of disassembled code:

• OEMs love to patent stuff. If you can find the patents relevant to your system, you may end up with a guided tour of the code being disassembled. This is probably the most consistently available high-level procedural guide to help you understand the logic in an automotive computer. Patents usually lead production by at least one to two years, if not more.

• Look at any available software for manipulating the ECU at hand for insight into the structure and purpose of code segments. You can often infer a model of behavior from tables available to be modified in aftermarket software.

• Otherwise, start with a wiring diagram for the vehicle, and trace connections back through ECU circuitry to particular pins on the MCU. This should tell you which piece of MCU hardware handles which function. Cross reference the interrupt tables, or look for calls to service particular pieces of hardware in order to identify which piece(s) of code service that hardware function.

A plain, or old-style, disassembler will output very verbose text. Each individual instruction is parsed. Some disassemblers will attempt to mark areas referenced as data and void disassembling them. Other disassemblers need to be specifically told which areas are code and which areas are data.

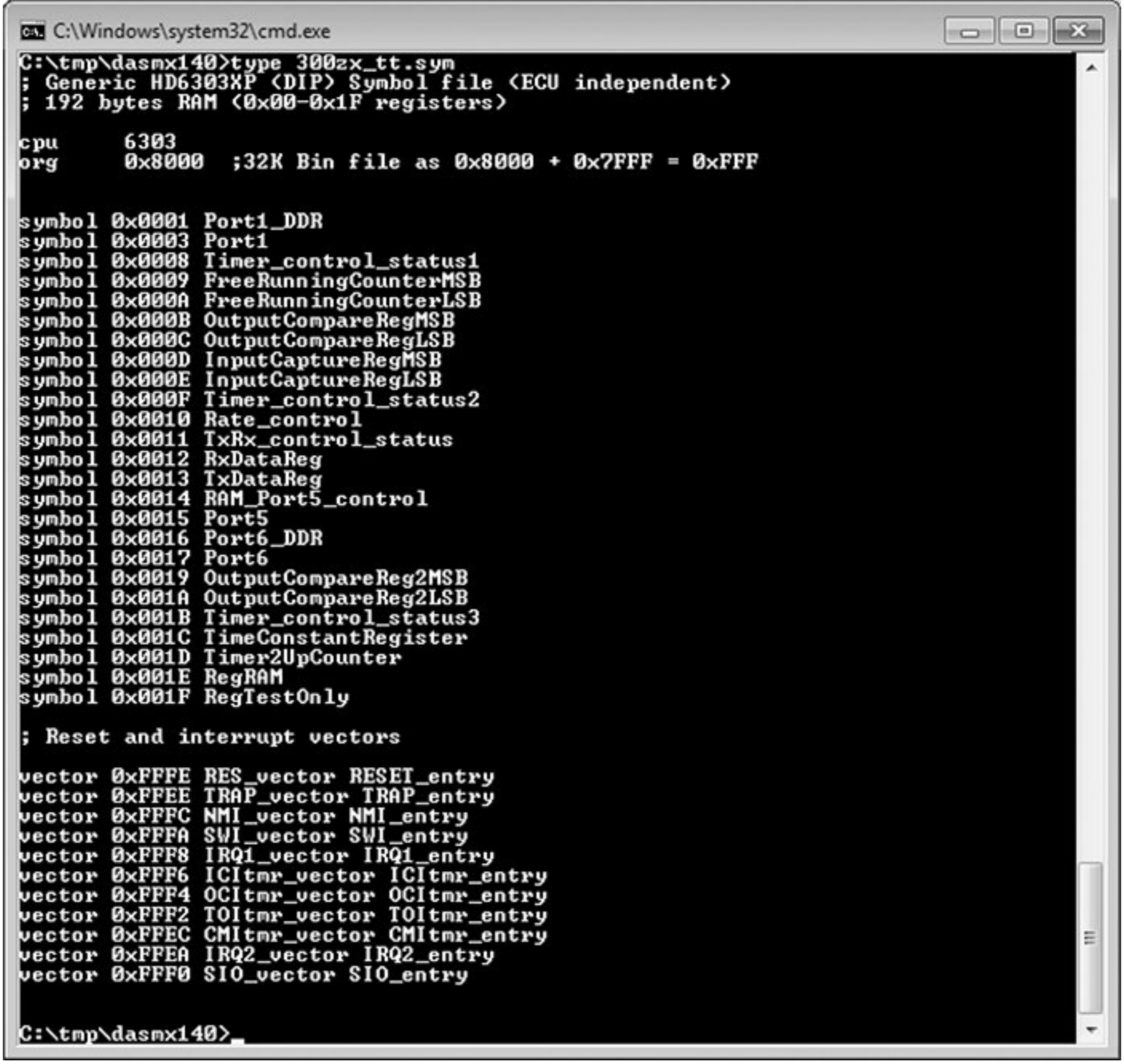
A Plain Disassembler at Work

To see disassembly in action, we'll look at a plain

disassembly of a 1990 Nissan 300ZX Twin Turbo ROM. This ECU has a 28-pin external 27C256 EPROM, so it's relatively easy to obtain its contents. This particular platform uses a HD6303 MCU, a derivative of the Motorola 6800 8-bit MCU that appears to be supported by the free disassembler DASMx
(see *http://www.16paws.com/ECU/DASMxx/DASMx.htm*). DASMx comes with minimal instructions: to disassemble *foo.bin*, create a file, *foo.sym*, that describes which platform is in use, and then create an entry point in memory to place the image, symbols you know about, and so on. Time for a crash course in the architecture!

A critical point about the memory structure is that the MCU can address 65535 bytes (64KB). This information tells you what to expect when looking at the addresses in your binary blob. Further reading suggests that the interrupt vector table lies at the *end* of addressable memory, with the reset vector —where every processor starts after a reset—at 0xFFFE/0xFFFF. Assuming that the 32KB (0x7FFF hex) binary blob we have from reading the EPROM contains the interrupt vector table, we can figure out that the binary image needs to start at memory address 0x8000 for it to end at 0xFFFF (0xFFFF – 0x7FFF = 0x8000). It also helps to search online to see whether others are trying to do something similar. For example, the post at *http://forum.nistune.com/viewtopic.php?f=2&t=417* is for a smaller 16KB binary based on settings for a 0xC000 entry point. The more legwork and research you do prior to actually invoking a disassembler, the more likely you are to get reasonable results.

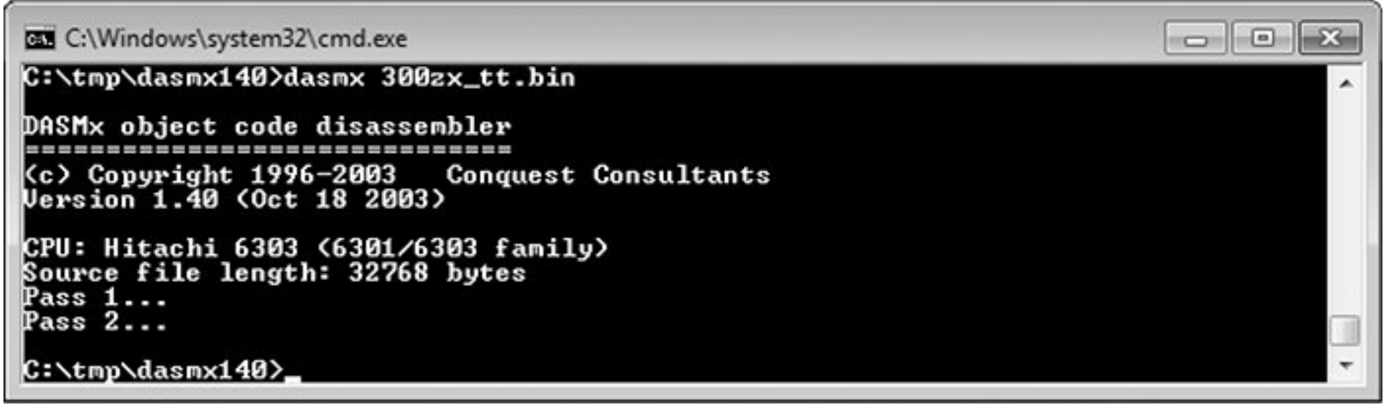
Figure 6-13 shows the symbol table for the 300ZX binary. Next to each symbol is the memory address used by the firmware. These memory addresses can hold values such as incoming data from different physical pins on the chip or internal information, like timing.



```
C:\tmp\dasmx140>type 300zx_tt.sym
; Generic HD6303XP (DIP) Symbol file (ECU independent)
; 192 bytes RAM (0x00-0x1F registers)

cpu     6303
org     0x8000  ;32K Bin file as 0x8000 + 0x7FFF = 0xFFF


symbol 0x0001 Port1_DDR
symbol 0x0003 Port1
symbol 0x0008 Timer_control_status1
symbol 0x0009 FreeRunningCounterMSB
symbol 0x000A FreeRunningCounterLSB
symbol 0x000B OutputCompareRegMSB
symbol 0x000C OutputCompareRegLSB
symbol 0x000D InputCaptureRegMSB
symbol 0x000E InputCaptureRegLSB
symbol 0x000F Timer_control_status2
symbol 0x0010 Rate_control
symbol 0x0011 TxRx_control_status
symbol 0x0012 RxDataReg
symbol 0x0013 TxDataReg
symbol 0x0014 RAM_Port5_control
symbol 0x0015 Port5
symbol 0x0016 Port6_DDR
symbol 0x0017 Port6
symbol 0x0019 OutputCompareReg2MSB
symbol 0x001A OutputCompareReg2LSB
symbol 0x001B Timer_control_status3
symbol 0x001C TimeConstantRegister
symbol 0x001D Timer2UpCounter
symbol 0x001E RegRAM
symbol 0x001F RegTestOnly

; Reset and interrupt vectors

vector 0xFFFE RES_vector RESET_entry
vector 0xFFEE TRAP_vector TRAP_entry
vector 0xFFFC NMI_vector NMI_entry
vector 0xFFFA SWI_vector SWI_entry
vector 0xFFF8 IRQ1_vector IRQ1_entry
vector 0xFFF6 ICItmr_vector ICItmr_entry
vector 0xFFF4 OCItmr_vector OCItmr_entry
vector 0xFFF2 TOItmr_vector TOItmr_entry
vector 0xFFEC CMItmr_vector CMItmr_entry
vector 0xFFEA IRQ2_vector IRQ2_entry
vector 0xFFF0 SIO_vector SIO_entry

C:\tmp\dasmx140>_
```

Figure 6-13: Symbol file for 32KB 300ZX binary disassembly with DASMx

We’ll use DASMx to disassemble the binary. As shown in Figure 6-14, DASMx reports a Hitachi 6303 MCU with a source file length, or size, of 32KB, which is 32768 bytes.

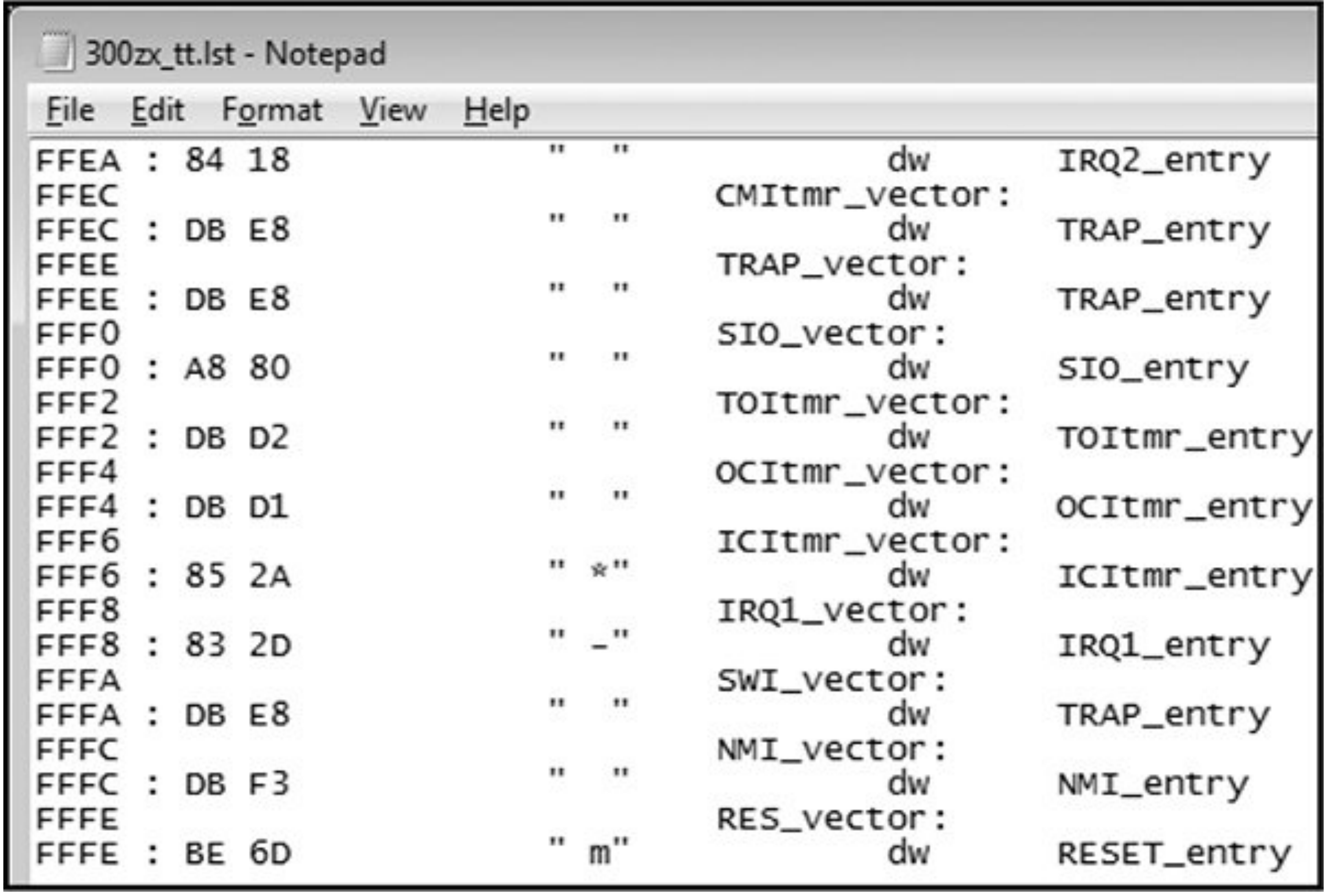


```
C:\tmp\dasmx140>dasmx 300zx_tt.bin

DASMx object code disassembler
================================
(c) Copyright 1996-2003   Conquest Consultants
Version 1.40 (Oct 18 2003)

CPU: Hitachi 6303 (6301/6303 family)
Source file length: 32768 bytes
Pass 1...
Pass 2...

C:\tmp\dasmx140>_
```

Figure 6-14: Running DASMx to disassemble 32KB 300ZX binary

Now cross your fingers and hope for a meaningful result!

The result is the vector table shown in Figure 6-15, which looks sane enough: all addresses are above the 0x8000 entry point specified. Notice that the reset vector (0xFFFE, RES-vector) has a pointer to the RESET_entry at 0xBE6D.

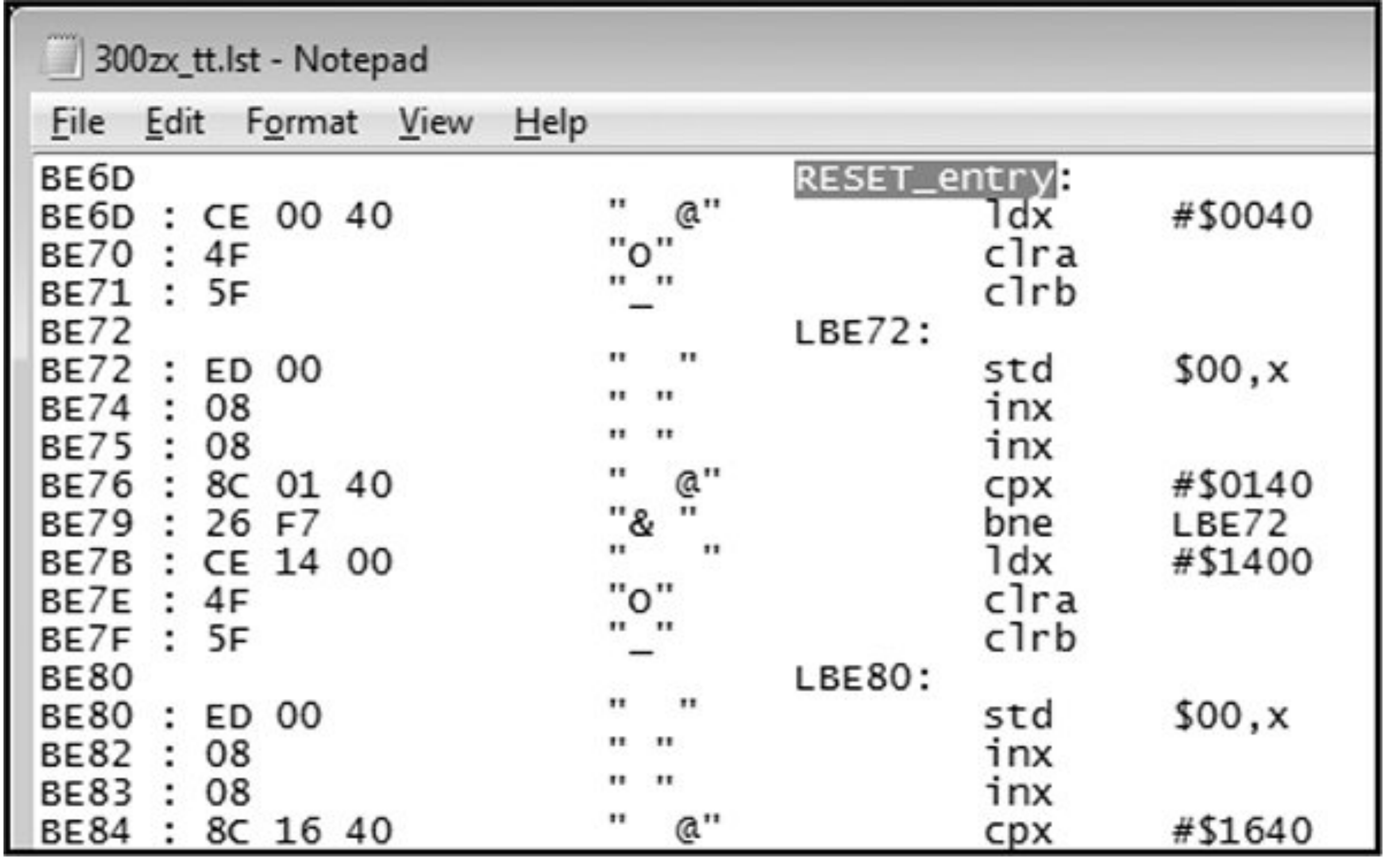

300zx_tt.lst - Notepad

File Edit Format View Help

```
FFEA : 84 18          "  "          dw      IRQ2_entry
FFEC                            CMItmr_vector:
FFEC : DB E8          "  "          dw      TRAP_entry
FFEE                            TRAP_vector:
FFEE : DB E8          "  "          dw      TRAP_entry
FFF0                            SIO_vector:
FFF0 : A8 80          "  "          dw      SIO_entry
FFF2                            TOItmr_vector:
FFF2 : DB D2          "  "          dw      TOItmr_entry
FFF4                            OCItmr_vector:
FFF4 : DB D1          "  "          dw      OCItmr_entry
FFF6                            ICItmr_vector:
FFF6 : 85 2A          " *"          dw      ICItmr_entry
FFF8                            IRQ1_vector:
FFF8 : 83 2D          " -"          dw      IRQ1_entry
FFFA                            SWI_vector:
FFFA : DB E8          "  "          dw      TRAP_entry
FFFC                            NMI_vector:
FFFC : DB F3          "  "          dw      NMI_entry
FFFE                            RES_vector:
FFFE : BE 6D          " m"          dw      RESET_entry
```

Figure 6-15: Disassembled vector table

We can disassemble the code at 0xBE6D for the reset vector, which is also the entry point for code. In Figure 6-16, we see a routine, RESET_entry, that looks like it wipes a chunk of RAM. This is a plausible part of the initial reset sequence because often when booting, firmware will initialize the data region to all 0s.



```
BE6D                                RESET_entry:
BE6D : CE 00 40         "  @"                  ldx     #$0040
BE70 : 4F               "O"                    clra
BE71 : 5F               "_"                    clrb
BE72                                LBE72:
BE72 : ED 00            "  "                   std     $00,x
BE74 : 08               " "                    inx
BE75 : 08               " "                    inx
BE76 : 8C 01 40         "  @"                  cpx     #$0140
BE79 : 26 F7            "& "                   bne     LBE72
BE7B : CE 14 00         "   "                  ldx     #$1400
BE7E : 4F               "O"                    clra
BE7F : 5F               "_"                    clrb
BE80                                LBE80:
BE80 : ED 00            "  "                   std     $00,x
BE82 : 08               " "                    inx
BE83 : 08               " "                    inx
BE84 : 8C 16 40         "  @"                  cpx     #$1640
```

Figure 6-16: Reset vector disassembly

We've taken this example as far as obtaining a disassembled binary image and looking for basic sanity. Now, for the hard part: following the code, breaking it into routines, and trying to figure out how it works.

Interactive Disassemblers

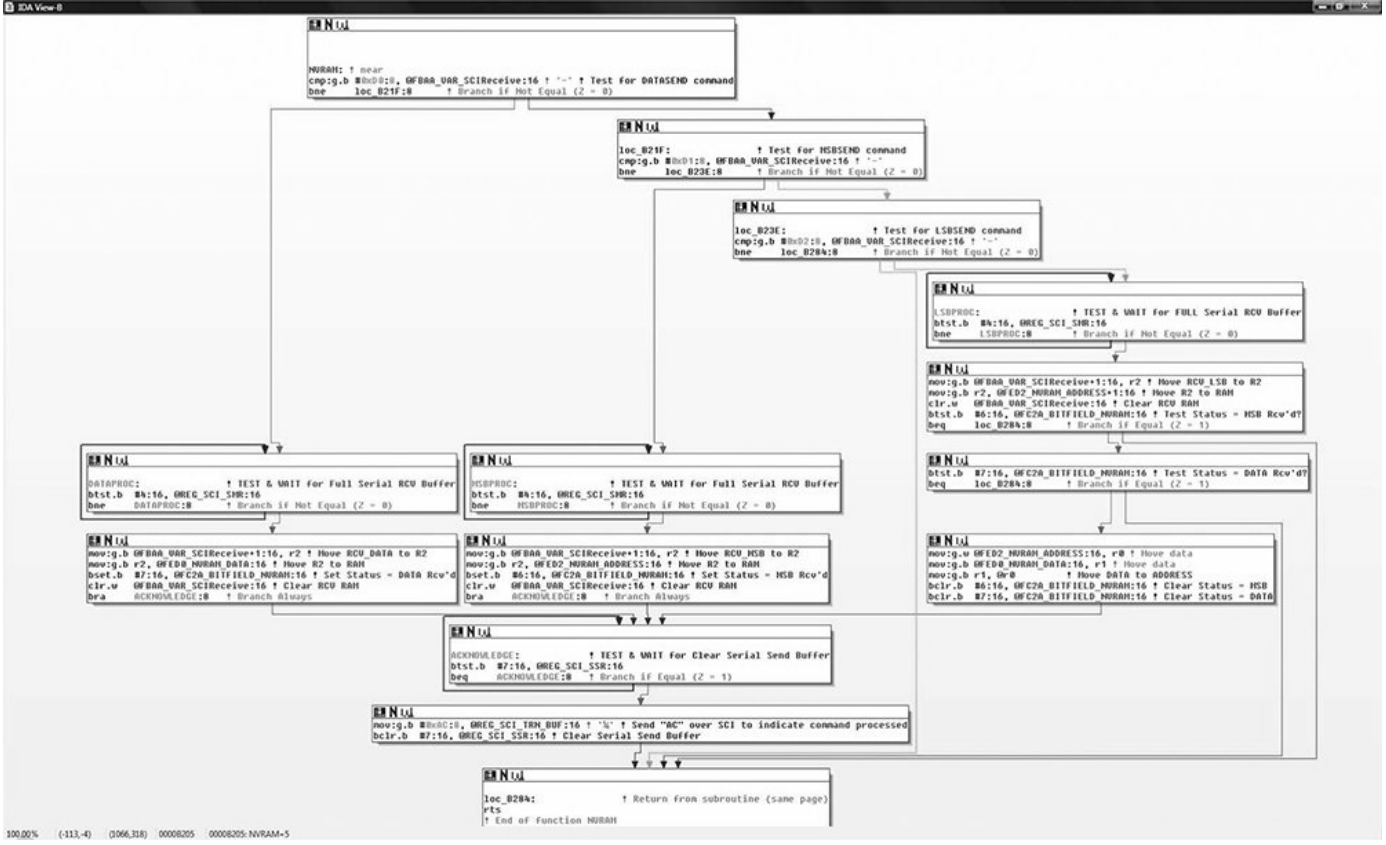
As of this writing, IDA Pro is the most popular interactive disassembler available. It performs the same tasks as the simple disassembler just discussed, and more. Specifically, IDA Pro names registers and variables; once IDA Pro identifies and names a variable, or memory address—for instance, $FC50–RPM—it gives all references to that variable within the code a descriptive name rather than a lessrecognizable plain hex address. IDA Pro also graphs code to visualize program flow.

One of the advantages of IDA Pro is that it's programmable to allow additional opcodes for customizing automotive processors and plugins for further processing disassembled code (for example, decompiling assembly into higher language code); it also lets you use structs, unions, classes, and other user-defined data types.

Lastly, IDA Pro supports more embedded platforms out of the box than just about any other disassembler currently available.

You don't necessarily need these functions to successfully analyze code, but they make things substantially easier. Figures 6-17 and 6-18 are screenshots from real code analysis with IDA Pro. Thanks to Matt Wallace for graciously posting these examples in a public forum.

The user in Figure 6-18 obtained Acura NSX ECU firmware through a combination of hardware-hacking approaches, took the code apart, analyzed it using IDA Pro, and rewrote it. Next, the user determined the necessary functions to log data from the ECU and alter its operation. The result allowed the user to use forced induction—that is, turbochargers and superchargers—with a factory computer; this would have been impossible without ECU modification.


IDA View-B

```
NVRAM: ! near
cmp:g.b #0xD0:8, @FBAA_VAR_SCIReceive:16 ! '-' ! Test for DATASEND command
bne     loc_821F:8       ! Branch if Not Equal (Z = 0)
```

```
loc_821F:                ! Test for MSBSEND command
cmp:g.b #0xD1:8, @FBAA_VAR_SCIReceive:16 ! '-'
bne     loc_823E:8       ! Branch if Not Equal (Z = 0)
```

```
loc_823E:                ! Test for LSBSEND command
cmp:g.b #0xD2:8, @FBAA_VAR_SCIReceive:16 ! '-'
bne     loc_8284:8       ! Branch if Not Equal (Z = 0)
```

```
LSBPROC:                 ! TEST & WAIT for FULL Serial RCV Buffer
btst.b  #4:16, @REG_SCI_SMR:16
bne     LSBPROC:8        ! Branch if Not Equal (Z = 0)
```

```
mov:g.b @FBAA_VAR_SCIReceive+1:16, r2 ! Move RCV_LSB to R2
mov:g.b r2, @FED2_NVRAM_ADDRESS+1:16 ! Move R2 to RAM
clr.w   @FBAA_VAR_SCIReceive:16 ! Clear RCV RAM
btst.b  #6:16, @FC2A_BITFIELD_NVRAM:16 ! Test Status = MSB Rcv'd?
beq     loc_8284:8       ! Branch if Equal (Z = 1)
```

```
btst.b  #7:16, @FC2A_BITFIELD_NVRAM:16 ! Test Status = DATA Rcv'd?
beq     loc_8284:8       ! Branch if Equal (Z = 1)
```

```
DATAPROC:                ! TEST & WAIT for Full Serial RCV Buffer
btst.b  #4:16, @REG_SCI_SMR:16
bne     DATAPROC:8       ! Branch if Not Equal (Z = 0)
```

```
MSBPROC:                 ! TEST & WAIT for Full Serial RCV Buffer
btst.b  #4:16, @REG_SCI_SMR:16
bne     MSBPROC:8        ! Branch if Not Equal (Z = 0)
```

```
mov:g.b @FBAA_VAR_SCIReceive+1:16, r2 ! Move RCV_DATA to R2
mov:g.b r2, @FED0_NVRAM_DATA:16 ! Move R2 to RAM
bset.b  #7:16, @FC2A_BITFIELD_NVRAM:16 ! Set Status = DATA Rcv'd
clr.w   @FBAA_VAR_SCIReceive:16 ! Clear RCV RAM
bra     ACKNOWLEDGE:8    ! Branch Always
```

```
mov:g.b @FBAA_VAR_SCIReceive+1:16, r2 ! Move RCV_MSB to R2
mov:g.b r2, @FED2_NVRAM_ADDRESS:16 ! Move R2 to RAM
bset.b  #6:16, @FC2A_BITFIELD_NVRAM:16 ! Set Status = MSB Rcv'd
clr.w   @FBAA_VAR_SCIReceive:16 ! Clear RCV RAM
bra     ACKNOWLEDGE:8    ! Branch Always
```

```
mov:g.w @FED2_NVRAM_ADDRESS:16, r0 ! Move data
mov:g.b @FED0_NVRAM_DATA:16, r1 ! Move data
mov:g.b r1, @r0          ! Move DATA to ADDRESS
bclr.b  #6:16, @FC2A_BITFIELD_NVRAM:16 ! Clear Status = MSB
bclr.b  #7:16, @FC2A_BITFIELD_NVRAM:16 ! Clear Status = DATA
```

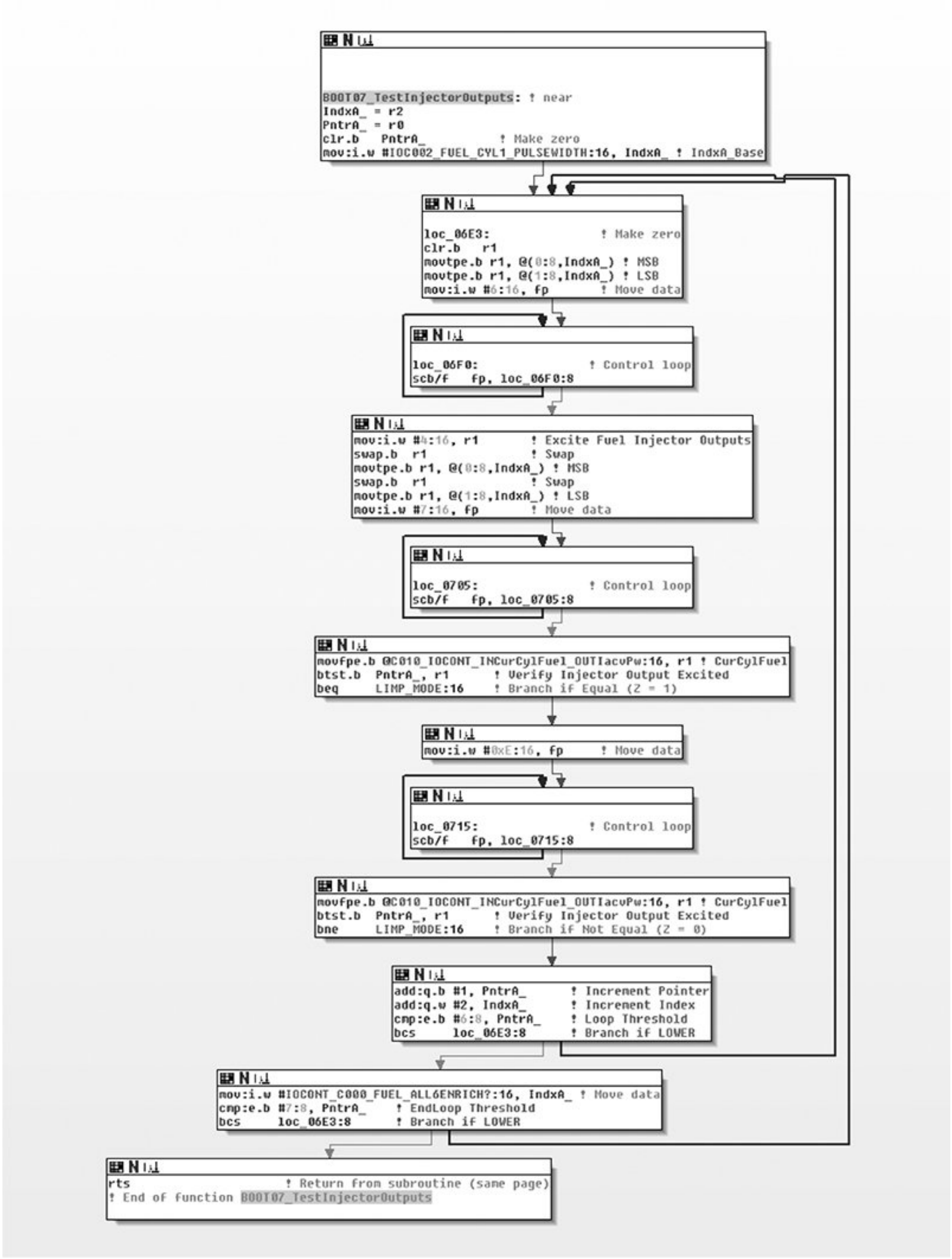
```
ACKNOWLEDGE:             ! TEST & WAIT for Clear Serial Send Buffer
btst.b  #7:16, @REG_SCI_SSR:16
beq     ACKNOWLEDGE:8    ! Branch if Equal (Z = 1)
```

```
mov:g.b #0xAC:8, @REG_SCI_TRN_BUF:16 ! '¼' ! Send "AC" over SCI to indicate command processed
bclr.b  #7:16, @REG_SCI_SSR:16 ! Clear Serial Send Buffer
```

```
loc_8284:                ! Return from subroutine (same page)
rts
! End of function NVRAM
```

100.00% (-113,-4) (1066,318) 00008205 00008205: NVRAM+5

Figure 6-17: IDA diagram showing a custom-written routine for NVRAM real-time programming



```
BOOT07_TestInjectorOutputs: ! near
IndxA_ = r2
PntrA_ = r0
clr.b   PntrA_          ! Make zero
mov:i.w #IOC002_FUEL_CYL1_PULSEWIDTH:16, IndxA_ ! IndxA_Base
```

```
loc_06E3:                     ! Make zero
clr.b   r1
movtpe.b r1, @(0:8,IndxA_) ! MSB
movtpe.b r1, @(1:8,IndxA_) ! LSB
mov:i.w #6:16, fp          ! Move data
```

```
loc_06F0:                    ! Control loop
scb/f    fp, loc_06F0:8
```

```
mov:i.w #4:16, r1        ! Excite Fuel Injector Outputs
swap.b  r1               ! Swap
movtpe.b r1, @(0:8,IndxA_) ! MSB
swap.b  r1               ! Swap
movtpe.b r1, @(1:8,IndxA_) ! LSB
mov:i.w #7:16, fp        ! Move data
```

```
loc_0705:                    ! Control loop
scb/f    fp, loc_0705:8
```

```
movfpe.b @C010_IOCONT_INCurCylFuel_OUTIacvPw:16, r1 ! CurCylFuel
btst.b  PntrA_, r1       ! Verify Injector Output Excited
beq     LIMP_MODE:16     ! Branch if Equal (Z = 1)
```

```
mov:i.w #0xE:16, fp      ! Move data
```

```
loc_0715:                    ! Control loop
scb/f    fp, loc_0715:8
```

```
movfpe.b @C010_IOCONT_INCurCylFuel_OUTIacvPw:16, r1 ! CurCylFuel
btst.b  PntrA_, r1       ! Verify Injector Output Excited
bne     LIMP_MODE:16     ! Branch if Not Equal (Z = 0)
```

```
add:q.b #1, PntrA_       ! Increment Pointer
add:q.w #2, IndxA_       ! Increment Index
cmp:e.b #6:8, PntrA_     ! Loop Threshold
bcs     loc_06E3:8       ! Branch if LOWER
```

```
mov:i.w #IOCONT_C000_FUEL_ALL6ENRICH?:16, IndxA_ ! Move data
cmp:e.b #7:8, PntrA_     ! EndLoop Threshold
bcs     loc_06E3:8       ! Branch if LOWER
```

```
rts                          ! Return from subroutine (same page)
! End of function BOOT07_TestInjectorOutputs
```

Figure 6-18: IDA diagram of code for checking fuel injectors on NSX ECU

Summary

Because hacking on the ECU often involves processors that are smaller than those used in more powerful modern devices, such as cell phones, the tools used for reversing the firmware differ for each target. By using a combination of techniques, such as data visualization to locate tables, and by reversing the firmware directly, you can identify the areas you’re interested in modifying. The methods discussed in this chapter are techniques commonly used by performance tuners to adjust how a vehicle handles fuel efficiency. All can be used to unlock features hidden in the code of your vehicle. We’ll look at performance tuning in more detail in Chapter 13.