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

Chapter 7. BUILDING AND USING ECU TEST BENCHES

An ECU test bench, like the one shown in Figure 7-1, consists of an ECU, a power supply, an optional power switch, and an OBD-II connector. You can also add an IC or other CAN-related systems for testing, but just building a basic ECU test bench is a great way to learn the CAN bus and how to create custom tools. In this chapter, we'll walk step by step through the process of building a test bench for development and testing.

The Basic ECU Test Bench

The most basic test bench is the device that you want to target and a power supply. When you give an ECU the proper amount of power, you can start performing tests on its inputs and communications. For example, Figure 7-1 shows a basic test bench containing a PC power supply and an ECU.

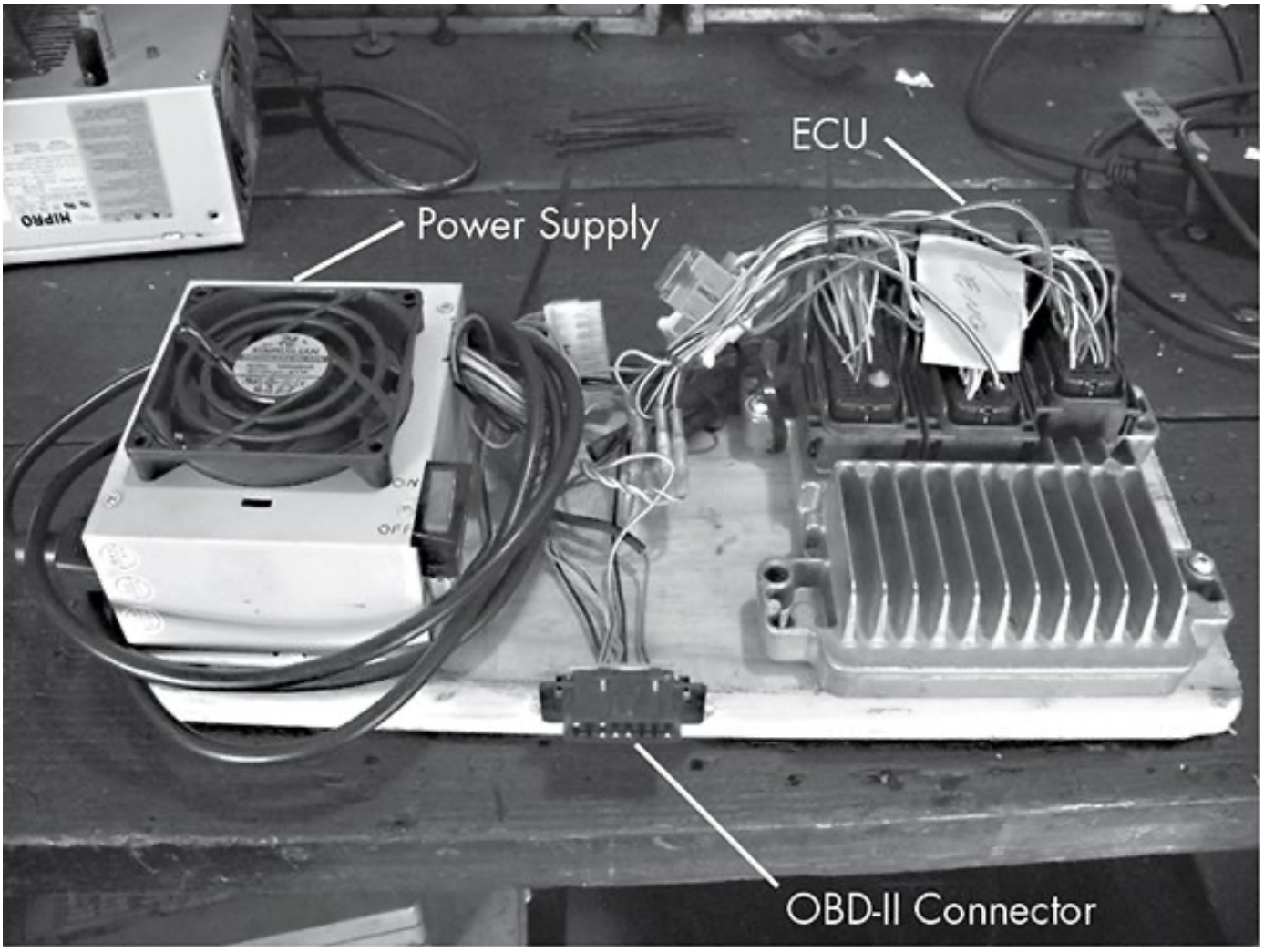

Figure 7-1: A simple ECU test bench

However, you'll often want to at least add some components or ports to make the test bench easier to use and operate. To make it easier to turn the device on and off, you can add a

switch to the power supply. An OBD port allows for specialized mechanics tools to communicate with the vehicle's network. In order for that OBD port to fully function, we need to expose the vehicle's network wires from the ECU to the OBD port.

Finding an ECU

One place to find an ECU is, of course, at the junkyard. You'll typically find the ECU behind a car's radio in the center console or behind the glove box. If you're having trouble finding it, try using the massive wiring harness to trace back to the ECU. When pulling one out yourself (it should cost only about $150), be sure to pull it from a vehicle that supports CAN. You can use a reference website such as *http://www.autoterraweb.com/aboutcan.html* to help you identify a target vehicle. Also, make sure you leave at least a pigtail's worth of wiring when you remove the ECU; this will make it easier to wire up later.

If you're not comfortable pulling devices out of junked cars, you can order an ECU online at a site like *car-part.com*. The cost will be a bit higher because you're paying for someone else to get the part and ship it to you. Be sure that the ECU you buy includes the wire bundles.

NOTE

One downside to buying an ECU online is that it may be difficult to acquire parts from the same car if you need multiple parts. For instance, you may need both the body control module (BCM) and the ECU because you want to include keys and the immobilizer is in the BCM. In this case, if you mix and match from two different vehicles, the vehicle won't "start" properly.

Instead of harvesting or buying a used ECU, you could also use a prebuilt simulator, like the ECUsim 2000 by ScanTool (see Figure 7-2). A simulator like ECUsim will cost around $200 per protocol and will support only OBD/UDS communications. Simulators can generate faults and MIL lights, and they include fault knobs for changing common vehicle parameters, such as speed. Unless you're building an application that uses only UDS packets, however, a simulator probably isn't the way to go.

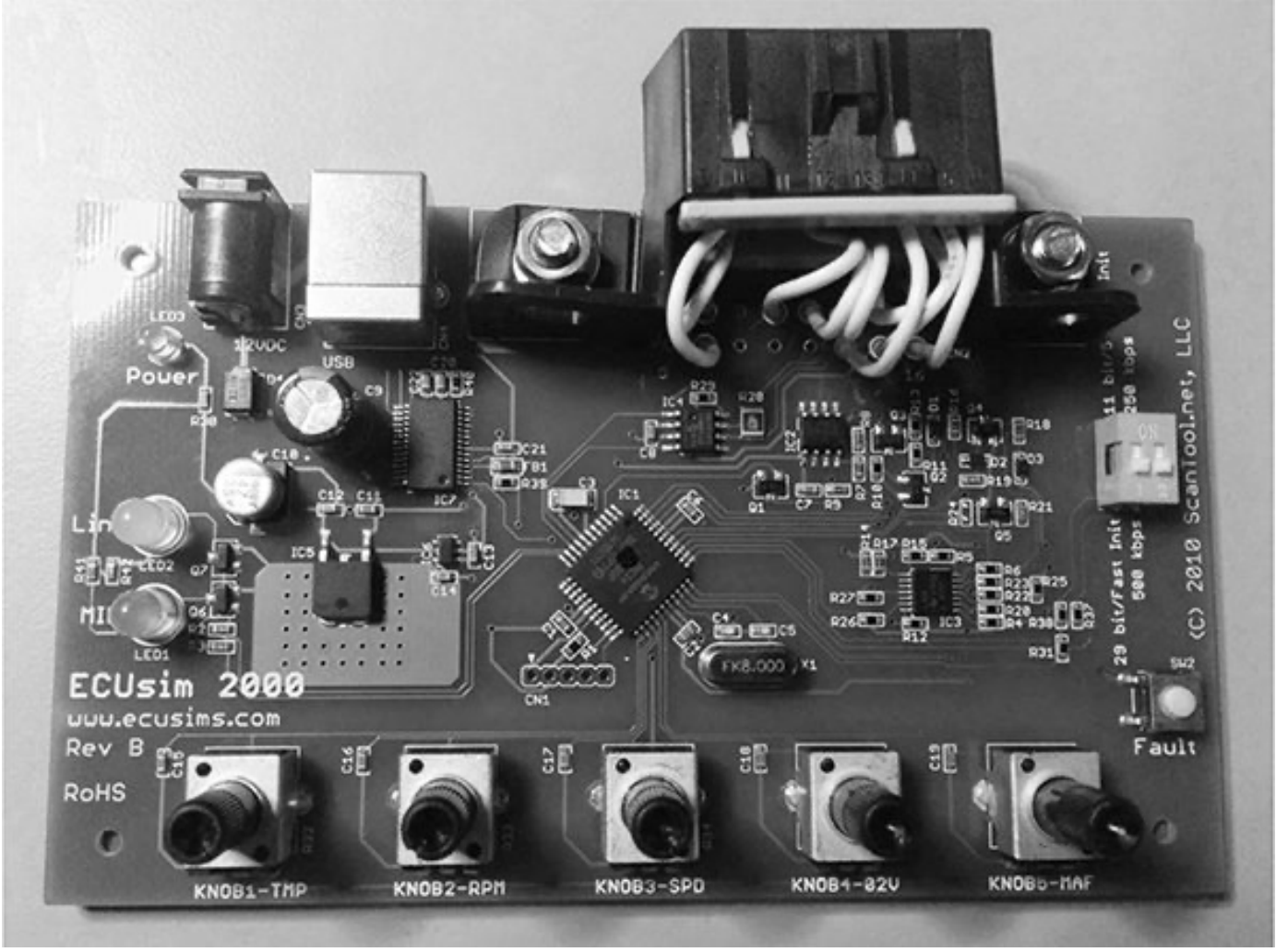

Figure 7-2: ECUsim OBD simulator

Dissecting the ECU Wiring

Once you have all of the parts, you'll need to find the ECU's wiring diagram to determine which wires you need to connect in order to get it to work. Visit a website such as ALLDATA (*http://www.alldata.com/*) or Mitchell 1 (*http://mitchell1.com/main/*) to get a complete wiring diagram. You'll find that off-the-shelf service manuals will sometimes have wiring diagrams, but they're often incomplete and contain only common repair areas.

Wiring diagrams aren't always easy to read, mainly because some combine numerous small components (see Figure 7-3). Try to mentally break down each component to get a better idea of which wires to focus on.

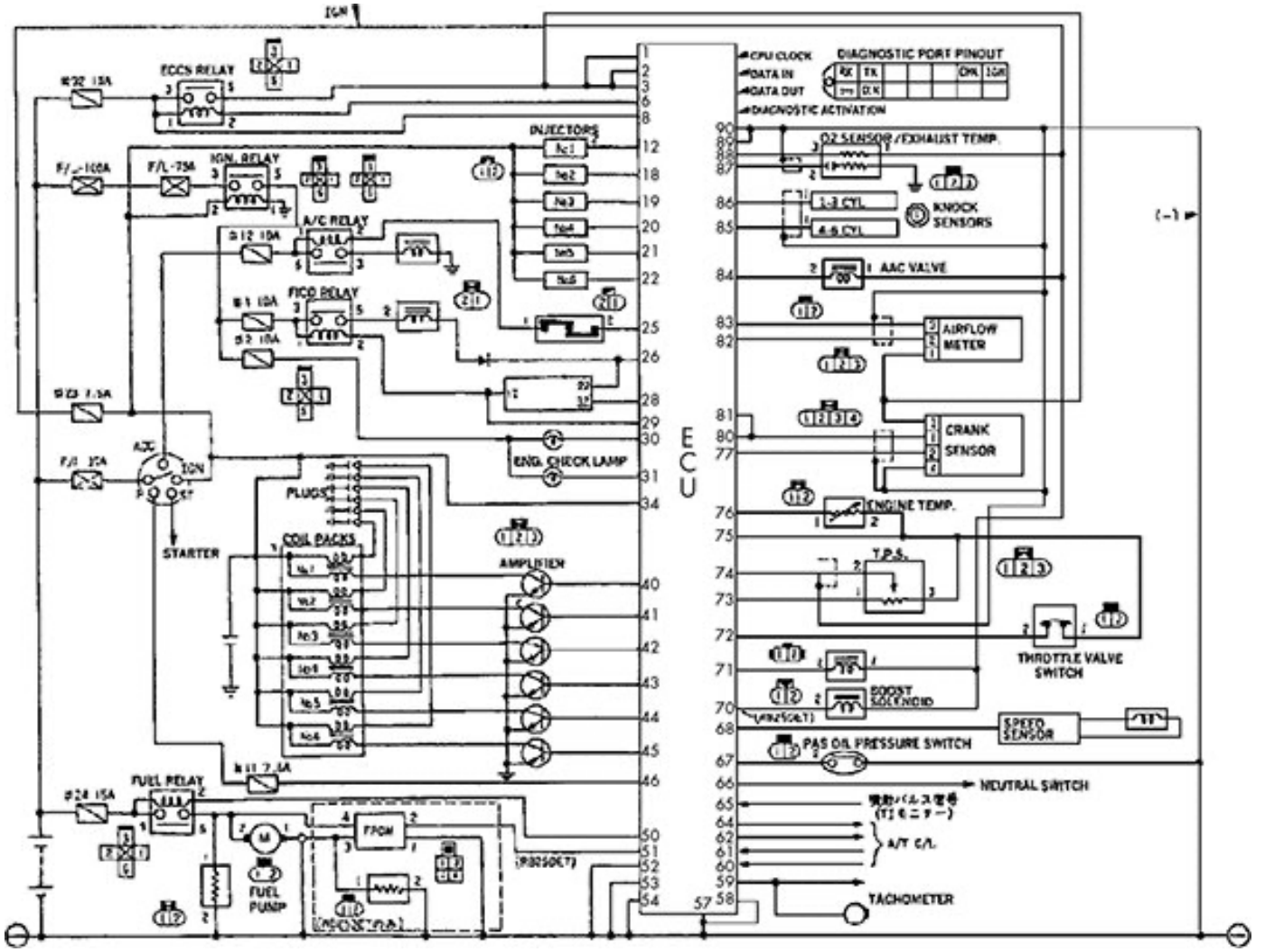

Figure 7-3: Example of an ECU wiring diagram

Pinouts

You can get pinouts for the ECUs on several different vehicles from *http://www.innovatemotorsports.com/resources/ecu_pinout.php* and from commercial resources like ALLDATA and

Mitchell 1. Books like the Chilton auto repair manuals include block diagrams, but you'll find that they typically cover only the most common repair components, not the entire ECU.

Block Diagrams

Block diagrams are often easier to read than wiring diagrams that show all components on the same sheet. Block diagrams usually show the wiring for only one component and offer a higher-level overview of the main components, whereas schematics show all the circuitry details. Some block diagrams also include a legend showing which connector block the diagram refers to and the connectors on that module; you'll typically find these in the corner of the block diagram (see Table 7-1).

Table 7-1: Example Connector Legend

CONN ID	Pin count	Color
C1	68	WH
C2	68	L-GY
C3	68	M-GY
C4	12	BK

The legend should give the connector number, its number pin count, and the color. For instance, the line C1 = 68 WH in Table 7-1 means that the C1 connector has 68 pins and is white. L-GY probably means light gray, and so on. A

connector number like C2-55 refers to connector 2, pin 55. The connectors usually have a number on the first and last pin in the row.

Wiring Things Up

Once you have information on the connector's wiring, it's time to wire it up. Wire the CAN to the proper ports on the connector, as discussed in "OBD-II Connector Pinout Maps" on page 31. When you provide power—a power supply from an old PC should suffice—and add a CAN sniffer, you should see packets. You can use just a simple OBD-II scan tool that you can pick up at any automotive store. If you have everything wired correctly, the scan tool should be able to identify the vehicle, assuming that your test bench includes the main ECU.

NOTE

Your MIL, or engine light, will most likely be reported as on *by the scan tool/ECU.*

If you've wired everything but you still don't see packets on your CAN bus, you may be missing termination. To address this problem, start by adding a 120-ohm resistor, as a CAN bus has 120-ohm resistors at each end of the bus. If that doesn't work, add a second resistor. The maximum missing resistance should be 240 ohms. If the bus still isn't working,

then recheck your wires and try again.

NOTE

A lot of components communicate with the ECU in a simple manner, either via set digital signals or through analog signals. Analog signals are easy to simulate with a potentiometer and you can often tie a 1 kilohm potentiometer to the engine temp and fuel lines to control them.

Building a More Advanced Test Bench

If you're ready to take your car hacking research further, consider building a more advanced ECU test bench, like the one shown in Figure 7-4.

This unit combines an ECU with a BCM because it also has the original keys to start the vehicle. Notice that the optional IC has two 1 kilohm potentiometers, or variable resistors, on the lower left side, both of which are tied to the engine temperature and fuel lines. We use these potentiometers to generate sensor signals, as discussed in the following section. This particular test bench also includes a small MCU that allows you to simulate sending crankshaft and camshaft signals to the ECU.

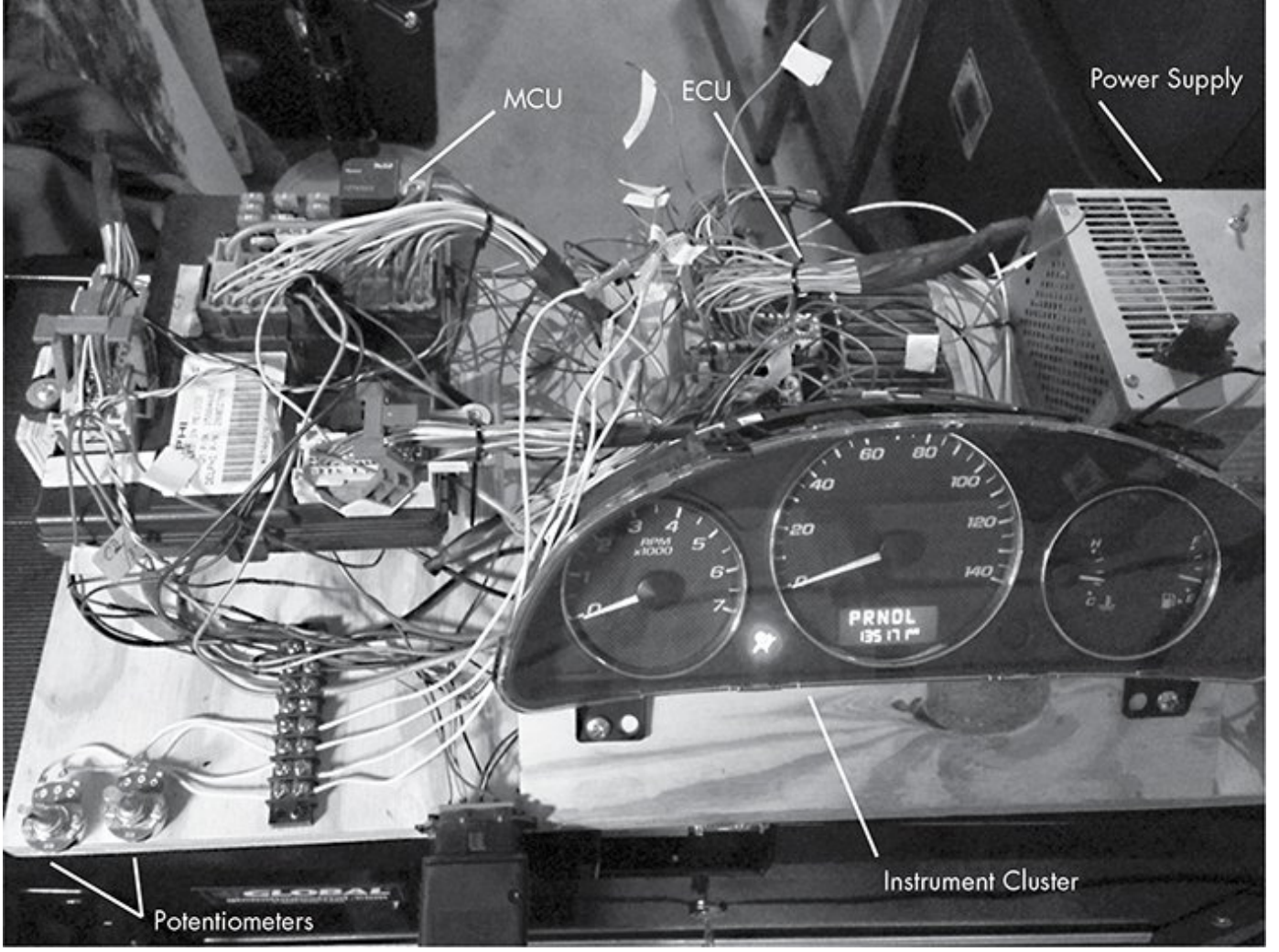

Figure 7-4: More complex test bench

A more complex unit like the one in Figure 7-4 makes it trivial to determine CAN traffic: just load a sniffer, adjust the knob, and watch for the packets to change. If you know which wires you're targeting and the type of input they take, you can easily fake signals from most components.

Simulating Sensor Signals

As I mentioned, you can use the potentiometers in this setup to simulate various vehicle sensors, including the following:

- Coolant temperature sensor

• Fuel sensor

• Oxygen sensors, which detect post-combustion oxygen in the exhaust

• Throttle position, which is probably already a potentiometer in the actual vehicle

• Pressure sensors

If your goal is to generate more complex or digital signals, use a small microcontroller, such as an Arduino, or a Raspberry Pi.

For our test bench, we also want to control the RPMs and/or speedometer needle. In order to do this, we need a little background on how the ECU measures speed.

Hall Effect Sensors

Hall effect sensors are often used to sense engine speed and crankshaft position (CKP) and to generate digital signals. In Figure 7-5, the Hall effect sensor uses a shutter wheel, or a wheel with gaps in it, to measure the rotation speed. The gallium arsenate crystal changes its conductivity when exposed to a magnetic field. As the shutter wheel spins, the crystal detects the magnet and sends a pulse when not blocked by the wheel. By measuring the frequency of pulses, you can derive the vehicle speed.

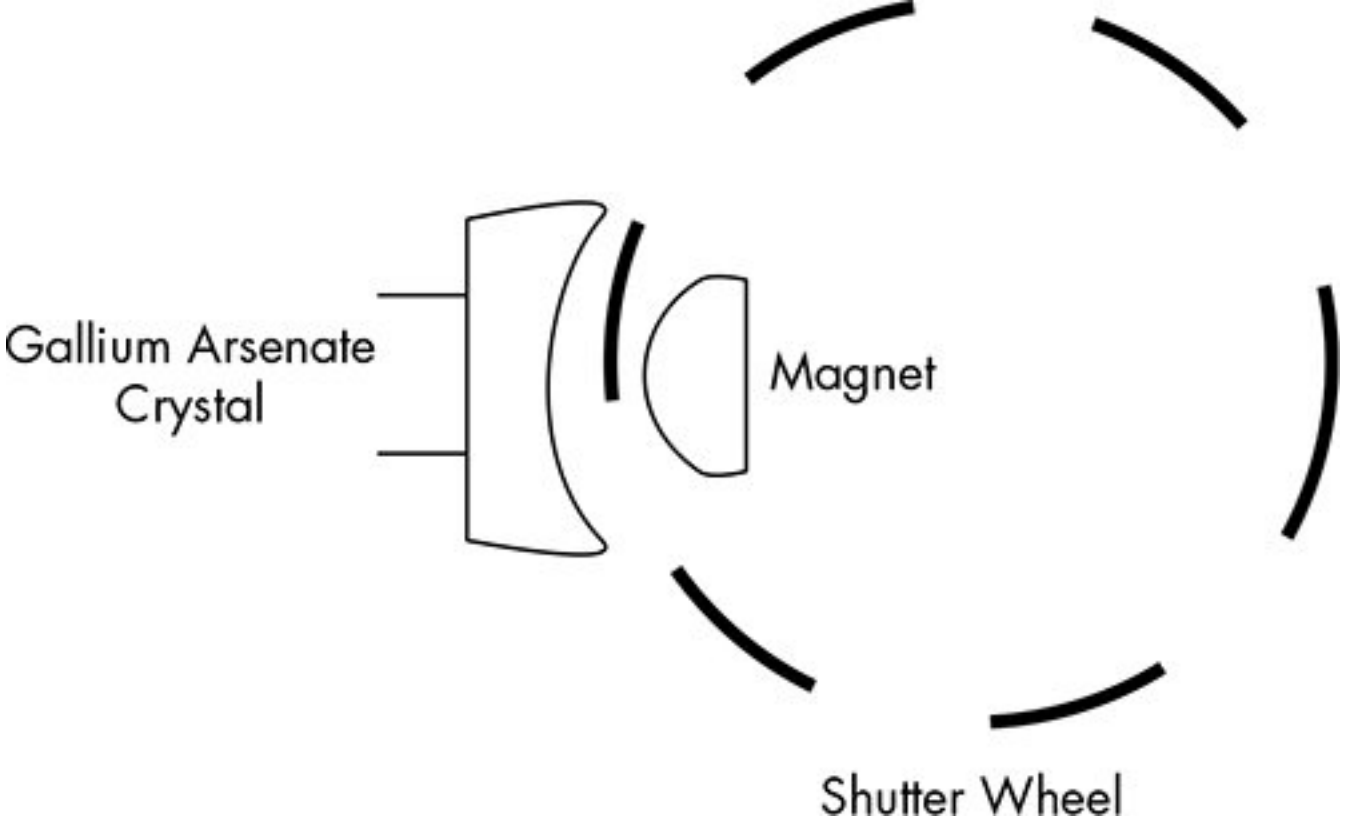

Figure 7-5: Shutter wheel diagram for Hall effect sensor

You can also use the camshaft timing sprocket to measure speed. When you look at the camshaft timing sprocket, the magnet is on the side of the wheel (see Figure 7-6).

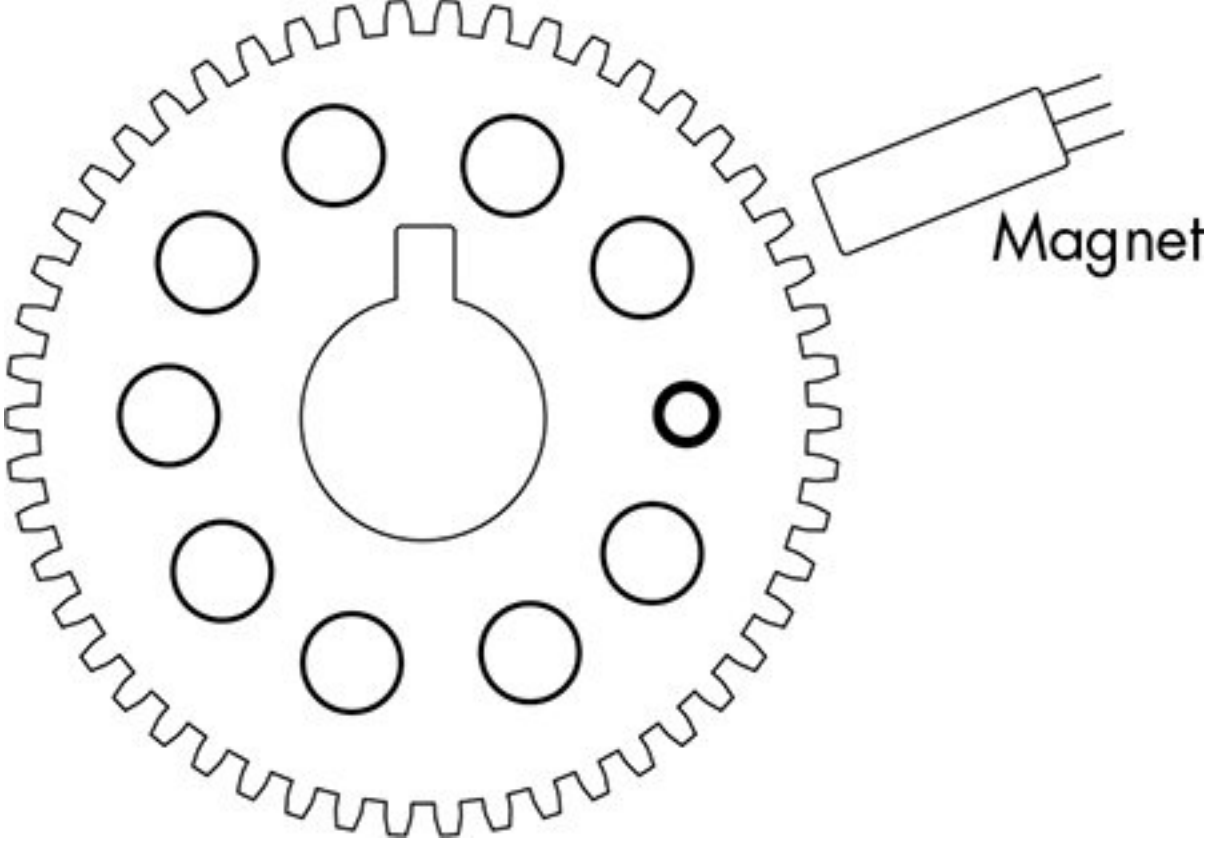

Figure 7-6: Camshaft timing sprocket

Using a scope on the signal wire shows that the Hall effect sensor produces a square wave. Typically, there are three wires on the camshaft sensor: power, ground, and sensor. Power is usually 12V, but the signal wire typically operates at 5V back to the ECM. Camshaft sensors also come as optical sensors, which work in a similar fashion except an LED is on one side and a photocell is on the other.

You can gauge full rotation timing with a missing tooth called a *trigger wheel* or with a timing mark. It's important to know when the camshaft has made a full rotation. An inductive camshaft sensor produces a sine wave and will often have a missing tooth to detect full rotation.

Figure 7-7 shows the camshaft sensor repeating approximately every 2 milliseconds. The jump or a gap you see in the wave at around the 40-millisecond mark occurs when the missing tooth is reached. The location of that gap marks the point at which the camshaft has completed a full rotation. In order to fake these camshaft signals into the ECU test bench, you'd need to write a small sketch for your microcontroller. When writing microcontroller code to mimic these sensors, it's important to know what type of sensor your vehicle uses so that you'll know whether to use a digital or analog output when faking the teeth.

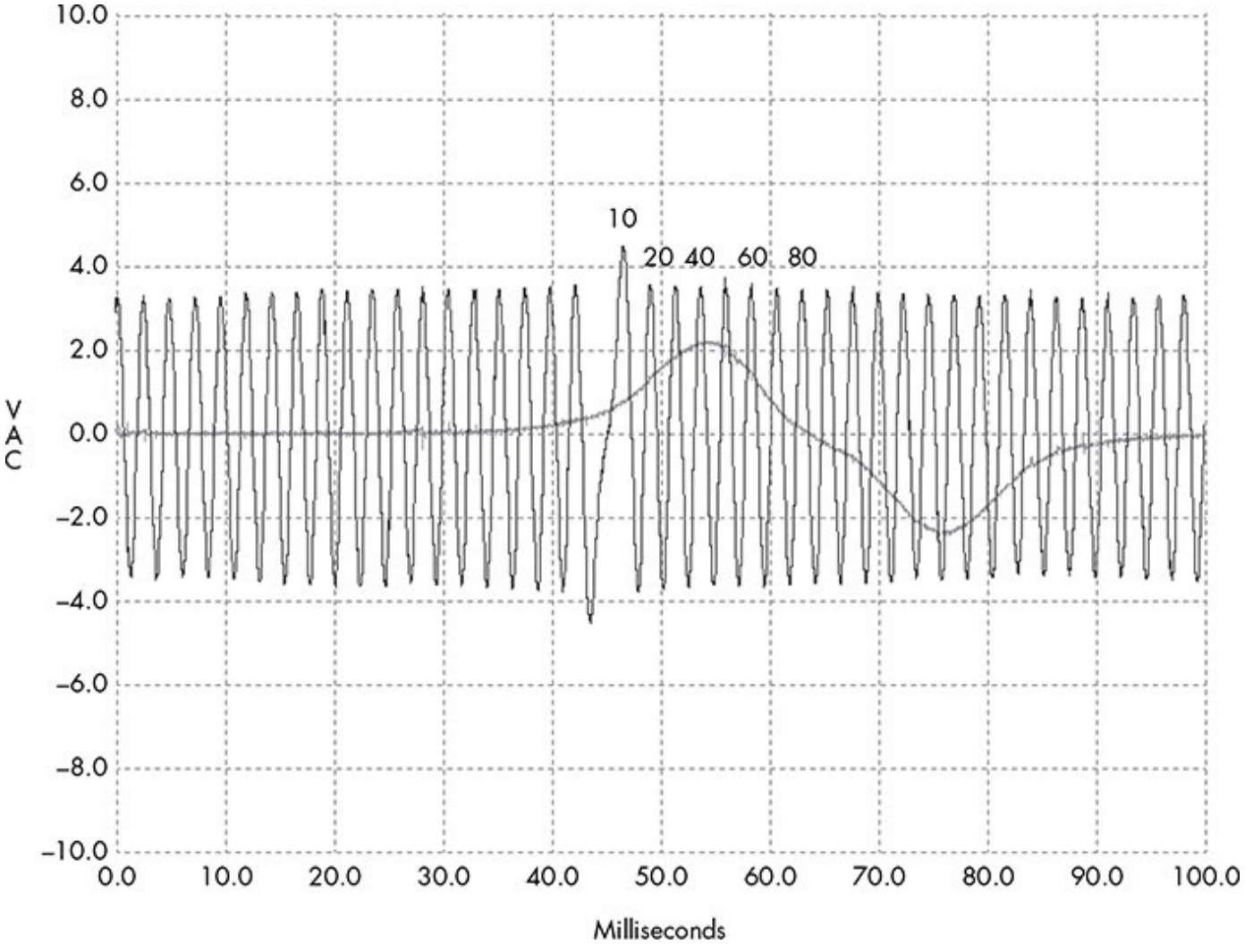

Figure 7-7: Camshaft sensor signals under a scope

Simulating Vehicle Speed

Now, we'll build a test bench to simulate vehicle speed. We'll use this test bench together with the IC shown in Figure 7-4 to pull a vehicle's VIN via the OBD-II connector. This will give us the exact year, make, model, and engine type of the vehicle. (We looked at how to do this manually in "Unified Diagnostic Services" on page 54.) Table 7-2 shows the results.

Table 7-2: Vehicle Information

VIN	Model	Year	Make	Body	Eng
1G1ZT53826F109149	Malibu	2006	Chevrolet	Sedan 4Door	3.5L V6 OHV 12V

Once we know a vehicle's year of manufacture and engine type, we can fetch the wiring diagram to determine which of the ECU wires control the engine speed (see Figure 7-8). Then, we can send simulated speed data to the ECU in order to measure effects. Using wiring diagrams to simulate real engine behavior can make it easy to identify target signals on the CAN bus.

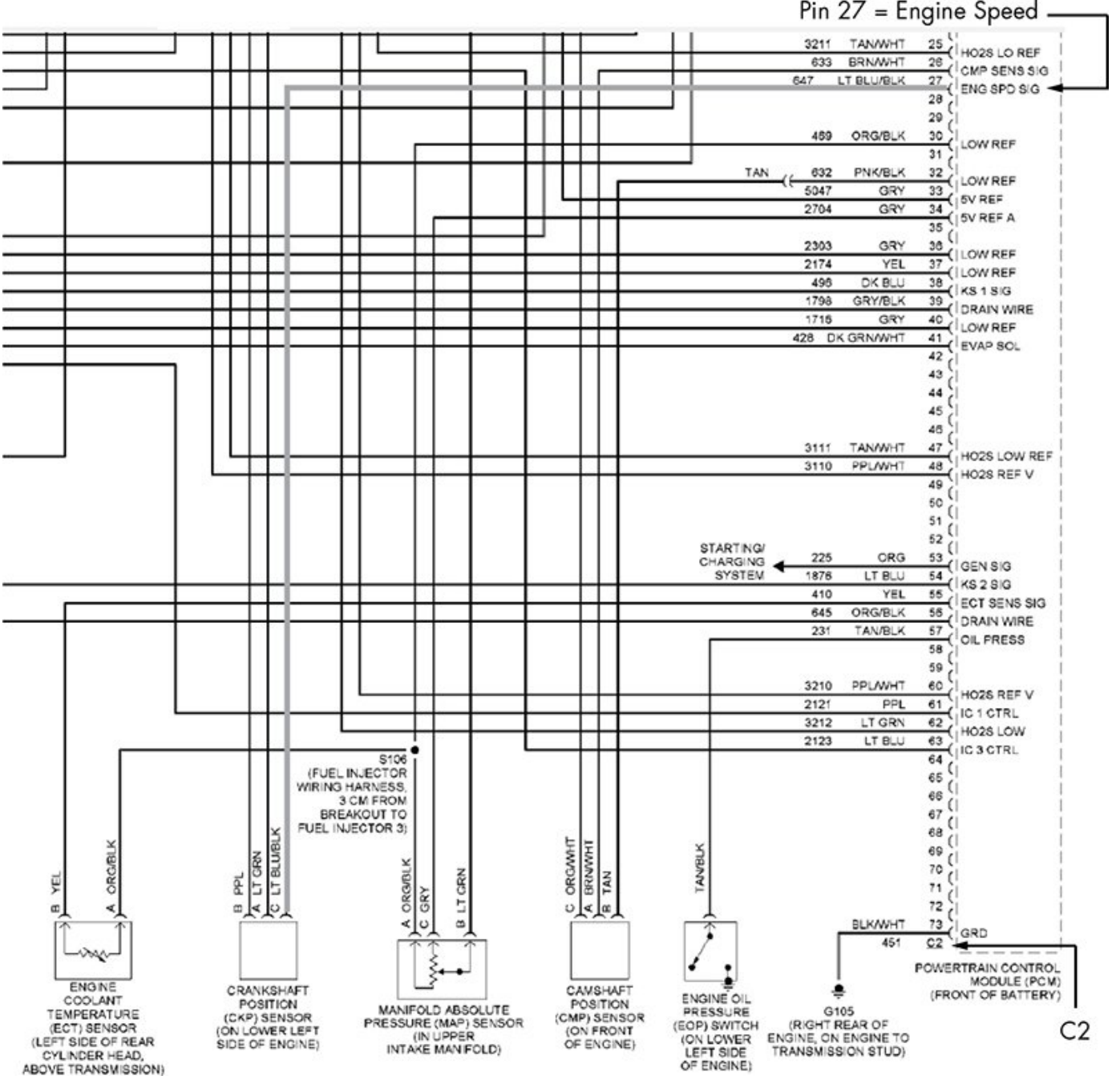

Figure 7-8: Wiring diagram showing the engine speed pin

The wiring diagram in Figure 7-8 shows how you can trace the wire from the CKP sensor so that connector C2, pin 27 receives the engine speed from the crankshaft sensor. Having identified this pin in the wiring diagram, we locate the corresponding wire on the ECU. We can connect this wire to any digital IO pin on an Arduino. In this example, we'll use pin

2 and then add a potentiometer to A0 to control the speed of the CKP sensor's "teeth" going to the ECM. Pin 2 will send output to C2, pin 27.

In order to simulate engine speed sent from the CKP sensor, we code up an Arduino sketch to send high and low pulses with a delay interval mapped to the potentiometer position (see Listing 7-1).

```
int ENG_SPD_PIN = 2;
long interval = 500;
long previousMicros = 0;
int state = LOW;

// the setup routine runs once when you press reset
void setup() {
  pinMode(ENG_SPD_PIN, OUTPUT);
}

// the loop routine repeats forever
void loop() {
  unsigned long currentMicros = micros();

  // read the input on analog pin 0
  int sensorValue = analogRead(A0);
  interval = map(sensorValue, 0, 1023, 0, 3000);
```

```
  if(currentMicros - previousMicros > interval) {
    previousMicros = currentMicros;

    if (state == LOW)
      state = HIGH;
    else
      state = LOW;

    if (interval == 0)
      state = LOW;  // turning the pot all the way down turns it
"off"

    digitalWrite(ENG_SPD_PIN, state);
  }
}
```

Listing 7-1: Arduino sketch designed to simulate engine speed

Now, we upload this sketch to the Arduino, power up the test bench, and when we turn the knob on the potentiometer, the RPM dial moves on the IC. In Figure 7-9, the second line of the cansniffer traffic shows bytes 2 and 3—0x0B and 0x89—changing as we rotate the potentiometer knob for Arbitration ID 0x110 (the column labeled *ID*).

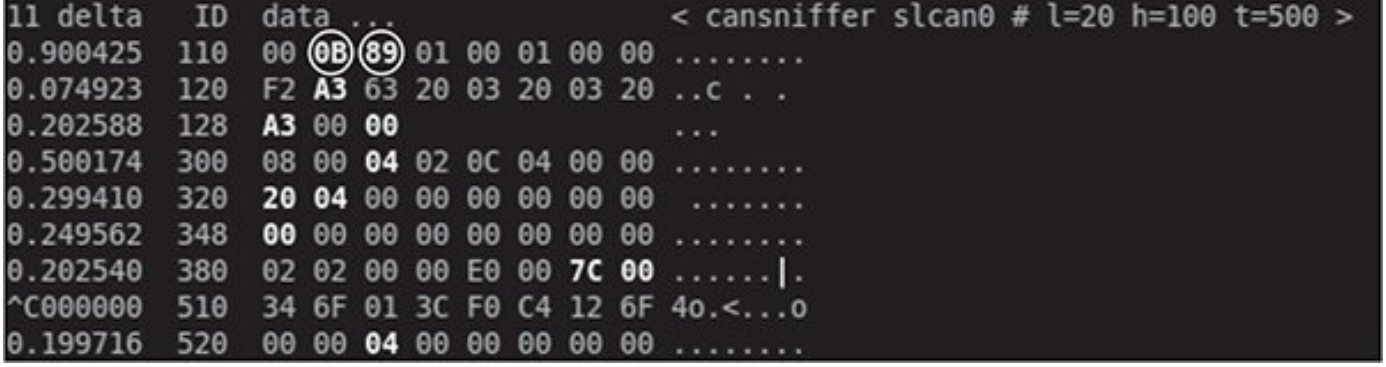

Figure 7-9: cansniffer identifying RPMs

NOTE

0x0B and 0x89 don't directly translate into the RPMs; rather, they're shorthand. In other words, if you're going to 1000 RPMs, you won't see the hex for 1000. When you query an engine for RPMs, the algorithm to convert these two bytes into RPMs is commonly the following:

$$\frac{(A \times 256) + B}{4}$$

A is the first byte and B *is the second byte. If you apply that algorithm to what's shown in Figure 7-9 (converted from hex to decimal), you get this:*

$$\frac{(11 \times 256) + 137}{4} = 738.25 \text{ RPMs}$$

You can simplify this method to taking 0xB89, which is 2953 in decimal form. When you divide this by 4, you get 738.25 RPMs.

When this screenshot was taken, the needle was idling a bit below the 1 on the RPM gauge, so that's probably the same algorithm. (Sometimes you'll find that the values in the true CAN packets don't always match the algorithms used by off-the-shelf diagnostic tools using the UDS service, but it's nice when they do.)

To verify that arbitration ID 0x110 with bytes 2 and 3 controls the RPM, we'll send our own custom packet. By flooding the bus with a loop that sends the following, we'll peg the needle at max RPMs.

$ **cansend slcan0 110#00ffff3500380000**

While this method works and, once connected, takes only a few seconds to identify the CAN packet responsible for RPMs, there are still some visible issues. Every so often a CAN signal shows up that resets the values to 00 00 and stops the speedometer from moving. So while the ECM is fairly certain the crankshaft is spinning, it's detecting a problem and attempting to reset.

You can use the ISO-TP tools discussed in Chapter 3 to pull data. In two different terminals, we can check whether there was a diagnostic code. (You can also use a scan tool.)

In one terminal, enter the following:

$ **isotpsniffer -s 7df -d 7e8 slcan0**

And in another terminal, send this command:

$ **echo "03" | isotpsend -s 7DF -d 7E8 slcan0**

You should see this output in the first terminal:

slcan0 7DF [1] 03 - '.'
slcan0 7E8 [6] 43 02 00 68 C1 07 - 'C..h..'

Looks like we have a DTC set. Querying PID 0x03 returned a 4-byte DTC (0x0068C107). The first two bytes make up the standard DTC (0x00 0x68). This converts to P0068, which the Chilton manual refers to as “throttle body airflow performance.” A quick Google search will let you know that this is just a generic error code that results from a discrepancy between what the PCM thinks is going on and what data it’s getting from the intake manifold. If we wanted to spoof that data as well, we’d need to spoof three additional sensors: the MAF sensor, the throttle position, and the manifold air pressure (MAP). Fixing these may not actually fix our problem, though. The PCM may continue to think the vehicle is running smoothly, but unless you really care about fudging all the data, you may be able to find other ways to trick the signals you want out of the PCM without having to be immune to triggering DTC faults.

If you don’t want to use an Arduino to send signals, you can also buy a signal generator. A professional one will cost at

least $150, but you can also get one from SparkFun for around $50 (*http://www.sparkfun.com/products/11394/*). Another great alternative is the JimStim for Megasquirt. This can be purchased as a kit or fully assembled for $90 from DIYAutoTune (*http://www.diyautotune.com/catalog/jimstim-15-megasquirt-stimulator-wheel-simulator-assembled-p-178.html*).

Summary

In this chapter you learned how to build an ECU test bench as an affordable solution to safe vehicle security testing. We went over where you can get parts for building a test bench and how to read wiring diagrams so you know how to hook those parts up. You also learned how to build a more advanced test bench that can simulate engine signals, in order to trick components into thinking the vehicle is present.

Building a test bench can be a time-consuming process during your initial research, but it will pay off in the end. Not only is it safer to do your testing on a test bench, but these units are also great for training and can be transported to where you need them.