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

Chapter 2. BUS PROTOCOLS

In this chapter, we'll discuss the different bus protocols common in vehicle communications. Your vehicle may have only one of these, or if it was built earlier than 2000, it may have none.

Bus protocols govern the transfer of packets through the network of your vehicle. Several networks and hundreds of sensors communicate on these bus systems, sending messages that control how the vehicle behaves and what information the network knows at any given time.

Each manufacturer decides which bus and which protocols make the most sense for its vehicle. One protocol, the CAN bus, exists in a standard location on all vehicles: on the OBD-II connector. That said, the packets themselves that travel over a vehicle's CAN bus aren't standardized.

Vehicle-critical communication, such as RPM management and braking, happens on high-speed bus lines, while noncritical communication, such as door lock and A/C control, happens on mid- to low-speed bus lines.

We'll detail the different buses and protocols you may run across on your vehicle. To determine the bus lines for your specific vehicle, check its OBD-II pinout online.

The CAN Bus

CAN is a simple protocol used in manufacturing and in the automobile industry. Modern vehicles are full of little embedded systems and electronic control units (ECUs) that can communicate using the CAN protocol. CAN has been a standard on US cars and light trucks since 1996, but it wasn't made mandatory until 2008 (2001 for European vehicles). If your car is older than 1996, it still may have CAN, but you'll need to check.

CAN runs on two wires: CAN high (CANH) and CAN low (CANL). CAN uses *differential signaling* (with the exception of low-speed CAN, discussed in "The GMLAN Bus" on page 20), which means that when a signal comes in, CAN raises the voltage on one line and drops the other line an equal amount (see Figure 2-1). Differential signaling is used in environments that must be fault tolerant to noise, such as in automotive systems and manufacturing.

Figure 2-1: CAN differential signaling

Figure 2-1 shows a signal captured using a PicoScope, which listens to both CANH (darker lines at the top of the graph) and CANL (lighter lines at the bottom of the graph). Notice that when a bit is transmitted on the CAN bus, the signal will simultaneously broadcast both 1V higher and lower. The sensors and ECUs have a transceiver that checks to ensure both signals are triggered; if they are not, the transceiver rejects the packet as noise.

The two twisted-pair wires make up the bus and require the bus to be terminated on each end. There's a 120-ohm resistor across both wires on the termination ends. If the module isn't on the end of the bus, it doesn't have to worry about termination. As someone who may tap into the lines, the only time you'll need to worry about termination is if you remove a terminating device in order to sniff the wires.

The OBD-II Connector

Many vehicles come equipped with an OBD-II connector, also known as the *diagnostic link connector (DLC)*, which communicates with the vehicle's internal network. You'll usually find this connector under the steering column or hidden elsewhere on the dash in a relatively accessible place. You may have to hunt around for it, but its outline looks similar to that in Figure 2-2.

Figure 2-2: Possible locations of the OBD-II connector

In some vehicles, you'll find these connectors behind small access panels. They'll typically be either black or white. Some are easy to access, and others are tucked up under the plastic. Search and you shall find!

Finding CAN Connections

CAN is easy to find when hunting through cables because its resting voltage is 2.5V. When a signal comes in, it'll add or subtract 1V (3.5V or 1.5V). CAN wires run through the vehicle and connect between the ECUs and other sensors, and they're always in dual-wire pairs. If you hook up a multimeter and check the voltage of wires in your vehicle, you'll find that they'll be at rest at 2.5V or fluctuating by 1V. If you find a wire transmitting at 2.5V, it's almost certainly CAN.

You should find the CANH and CANL connections on pins 6 and 14 of your OBD-II connector, as shown in Figure 2-3.

Figure 2-3: CAN pins cable view on the OBD-II connector

In the figure, pins 6 and 14 are for standard high-speed CAN lines (HS-CAN). Mid-speed and low-speed communications happen on other pins. Some cars use CAN for the mid-speed (MS-CAN) and low-speed (LS-CAN), but many vehicles use different protocols for these communications.

You’ll find that not all buses are exposed via the OBD-II connector. You can use wiring diagrams to help locate additional “internal” bus lines.

CAN Bus Packet Layout

There are two types of CAN packets: *standard* and *extended*. Extended packets are like standard ones but with a larger space to hold IDs.

Standard Packets

Each CAN bus packet contains four key elements:

Arbitration ID The arbitration ID is a broadcast message that identifies the ID of the device trying to communicate, though any one device can send multiple arbitration IDs. If two CAN packets are sent along the bus at the same time, the one with the lower arbitration ID wins.

Identifier extension (IDE) This bit is always 0 for standard CAN.

Data length code (DLC) This is the size of the data, which ranges from 0 to 8 bytes.

Data This is the data itself. The maximum size of the data carried by a standard CAN bus packet can be up to 8 bytes, but some systems force 8 bytes by padding out the packet.

Figure 2-4 shows the format of standard CAN packets.

Figure 2-4: Format of standard CAN packets

Because CAN bus packets are broadcast, all controllers on the same network see *every* packet, kind of like UDP on Ethernet networks. The packets don't carry information about which controller (or attacker) sent what. Because any device can see and transmit packets, it's trivial for any device on the bus to simulate any other device.

Extended Packets

Extended packets are like standard ones, except that they can be chained together to create longer IDs. Extended packets are designed to fit inside standard CAN formatting in order to maintain backward compatibility. So if a sensor doesn't have support for extended packets, it won't break if another packet transmits extended CAN packets on the same network.

Standard packets also differ from extended ones in their use of flags. When looking at extended packets in a network dump, you'll see that unlike standard packets, extended packets use substitute remote request (SRR) in place of the remote transmission request (RTR) with SSR set to 1. They'll also have the IDE set to 1, and their packets will have an 18-bit identifier, which is the second part of the standard 11-bit identifier. There are additional CAN-style protocols that are specific to some manufacturers, and they're also backward compatible with standard CAN in much the same way as extended CAN.

The ISO-TP Protocol

ISO 15765-2, also known as *ISO-TP*, is a standard for sending packets over the CAN bus that extends the 8-byte CAN limit to support up to 4095 bytes by chaining CAN packets together. The most common use of ISO-TP is for diagnostics (see "Unified Diagnostic Services" on page 54) and KWP messages (an alternative protocol to CAN), but it can also be used any time large amounts of data need to be transferred over CAN. The can-utils program includes isotptun, a proof-of-concept tunneling tool for SocketCAN that allows two devices to tunnel IP over CAN. (For a detailed explanation of how to install and use can-utils, see "Setting Up can-utils to Connect to CAN Devices" on page 36.)

In order to encapsulate ISO-TP into CAN, the first byte is used for extended addressing, leaving only 7 bytes for data per packet. Sending lots of information over ISO-TP can easily flood the bus, so be careful when using this standard for large transfers on an active bus.

The CANopen Protocol

Another example of extending the CAN protocol is the CANopen protocol. CANopen breaks down the 11-bit identifier to a 4-bit function code and 7-bit node ID—a combination known as a *communication object identifier (COB-ID)*. A broadcast message on this system has 0x for both the function code and the node ID. CANopen is seen more in industrial settings than it is in automotive ones.

If you see a bunch of arbitration IDs of 0x0, you've found a good indicator that the system is using CANopen for communications. CANopen is very similar to normal CAN but has a defined structure around the arbitration IDs. For example, heartbeat messages are in the format of 0x700 + node ID. CANopen networks are slightly easier to reverse and document than standard CAN bus.

The GMLAN Bus

GMLAN is a CAN bus implementation by General Motors. It's based on ISO 15765-2 ISO-TP, just like UDS (see "Unified Diagnostic Services" on page 54). The GMLAN bus consists of a single-wire low-speed and a dual-wire high-speed bus. The low-speed bus, a single-wire CAN bus that operates at 33.33Kbps with a maximum of 32 nodes, was adopted in an attempt to lower the cost of communication and wiring. It's used to transport noncritical information for things like the infotainment center, HVAC controls, door locks, immobilizers, and so on. In contrast, the high-speed bus runs at 500Kbps with a maximum of 16 nodes. Nodes in a GMLAN network relate to the sensors on that bus.

The SAE J1850 Protocol

The SAE J1850 protocol was originally adopted in 1994 and can still be found in some of today's vehicles, for example some General Motors and Chrysler vehicles. These bus systems are older and slower than CAN but cheaper to implement.

There are two types of J1850 protocols: pulse width modulation (PWM) and variable pulse width (VPW). Figure 2-5 shows where to find PWM pins on the OBD-II connector. VPW uses only pin 2.

Figure 2-5: PWM pins cable view

The speed is grouped into three classes: A, B, and C. The 10.4Kbps speeds of PWM and VPW are considered class A, which means they're devices marketed exclusively for use in business, industrial, and commercial environments. (The 10.4Kbps J1850 VPW bus meets the automotive industry's requirements for low-radiating emissions.) Class B devices are marketed for use anywhere, including residential environments and have a second SAE standard implementation that can communicate at 100Kbps, but it's slightly more expensive. The final implementation can operate at up to 1Mbps, and it's used in class C devices. As you might expect, this third implementation is the most expensive, and it's used primarily in real-time critical systems and media networks.

The PWM Protocol

PWM uses differential signaling on pins 2 and 10 and is mainly used by Ford. It operates with a high voltage of 5V and at 41.6Kbps, and it uses dual-wire differential signaling, like CAN.

PMW has a fixed-bit signal, so a 1 is always a high signal and a 0 is always a low signal. Other than that, the communication protocol is identical to that of VPW. The differences are the speed, voltage, and number of wires used to make up the bus.

The VPW Protocol

VPW, a single-wire bus system, uses only pin 2 and is typically used by General Motors and Chrysler. VPW has a high voltage of 7V and a speed of 10.4Kbps.

When compared with CAN, there are some key differences in the way VPW interprets data. For one, because VPW uses time-dependent signaling, receiving 1 bit isn't determined by just a high potential on the bus. The bit must remain either high or low for a set amount of time in order to be considered a single 1 bit or a 0 bit. Pulling the bus to a high position will put it at around 7V, while sending a low signal will put it to ground or near-ground levels. This bus also is at a resting, or nontransmission, stage at a near-ground level (up to 3V).

VPW packets use the format in Figure 2-6.

Figure 2-6: VPW Format

The data section is a set size—always 11 bits followed by a 1-bit CRC validity check. Table 2-1 shows the meaning of the header bits.

Table 2-1: Meaning of Header Bits

Header bits	**Meaning**	**Notes**
PPP	Message priority	000 = Highest, 111 = Lowest
H	Header size	0 = 3 bytes, 1 = single byte
K	In-frame response	0 = Required, 1 = Not allowed
Y	Addressing mode	0 = Functional, 1 = Physical
ZZ	Message type	Will vary based on how K and Y are set

In-frame response (IFR) data may follow immediately after this message. Normally, an end-of-data (EOD) signal consisting of 200µs-long low-potential signal would occur just after the CRC, and if IFR data is included, it'll start immediately after the EOD. If IFR isn't being used, the EOD will extend to 280µs, causing an end-of-frame (EOF) signal.

The Keyword Protocol and ISO 9141-2

The Keyword Protocol 2000 (ISO 14230), also known as *KWP2000*, uses pin 7 and is common in US vehicles made after 2003. Messages sent using KWP2000 may contain up to 255 bytes.

The KWP2000 protocol has two variations that differ mainly in baud initialization. The variations are:

- ISO 14230-4 KWP (5-baud init, 10.4 Kbaud)
- ISO 14230-4 KWP (fast init, 10.4 Kbaud)

ISO 9141-2, or K-Line, is a variation of KWP2000 seen most often in European vehicles. K-Line uses pin 7 and, optionally, pin 15, as shown in Figure 2-7. K-Line is a UART protocol similar to serial. UARTs use start bits and may include a parity bit and a stop bit. (If you've ever set up a modem, you should recognize this terminology.)

Figure 2-7: KWP K-Line pins cable view

Figure 2-8 shows the protocol's packet layout. Unlike CAN packets, K-Line packets have a source (transmitter) and a destination (receiver) address. K-Line can use the same or a similar parameter ID (PID) request structure as CAN. (For more on PIDs, see "Unified Diagnostic Services" on page 54.)

Figure 2-8: KWP K-Line packet layout

The Local Interconnect Network Protocol

The *Local Interconnect Network (LIN)* is the cheapest of the vehicle protocols. It was designed to complement CAN. It has no arbitration or priority code; instead, a single master node does all the transmission.

LIN can support up to 16 slave nodes that primarily just listen to the master node. They do need to respond on occasion, but that's not their main function. Often the LIN master node is connected to a CAN bus.

The maximum speed of LIN is 20Kbps. LIN is a single-wire bus that operates at 12V. You won't see LIN broken out to the OBD connector, but it's often used instead of direct CAN packets to handle controls to simple devices, so be aware of its existence.

A LIN message frame includes a header, which is always sent by the master, and a response section, which may be sent by master or slave (see Figure 2-9).

Figure 2-9: LIN format

The SYNC field is used for clock synchroniziation. The ID represents the message contents—that is, the type of data being transmitted. The ID can contain up to 64 possibilities. ID 60 and 61 are used to carry diagnostic information.

When reading diagnostic information, the master sends with ID 60 and the slave responds with ID 61. All 8 bytes are used in diagnostics. The first byte is called the node address for diagnostics (NAD). The first half of the byte range (that is, 1–127) is defined for ISO-compliant diagnostics, while 128–255 can be specific to that device.

The MOST Protocol

The *Media Oriented Systems Transport (MOST) protocol* is designed for multimedia devices. Typically, MOST is laid out in a ring topology, or virtual star, that supports a maximum of 64 MOST devices. One MOST device acts as the timing master, which continuously feeds frames into the ring.

MOST runs at approximately 23 Mbaud and supports up to 15 uncompressed CD quality audio or MPEG1 audio/video channels. A separate control channel runs at 768 Kbaud and sends configuration messages to the MOST devices.

MOST comes in three speeds: MOST25, MOST50, and MOST150. Standard MOST, or MOST25, runs on plastic optical fiber (POF). Transmission is done through the red light wavelength at 650 nm using an LED. A similar protocol, MOST50, doubles the bandwidth and increases the frame length to 1025 bits. MOST50 traffic is usually transported on unshielded twisted-pair (UTP) cables instead of optical fiber. Finally, MOST150 implements Ethernet and increases the frame rate to 3072 bits or 150Mbps—approximately six times the bandwidth of MOST25.

Each MOST frame has three channels:

Synchronous Streamed data (audio/video)

Asynchronous Packet distributed data (TCP/IP)

Control Control and low-speed data (HMI)

In addition to a timing master, a MOST network master automatically assigns addresses to devices, which allows for a kind of plug-and-play structure. Another unique feature of MOST is that, unlike other buses, it routes packets through separate inport and outport ports.

MOST Network Layers

Unless your goal is to hack a car's video or audio stream, the MOST protocol may not be all that interesting to you. That said, MOST does allow access to the in-vehicle microphone or cell system, as well as traffic information that's likely to be of interest to malware authors.

Figure 2-10 shows how MOST is divided up amongst the seven layers of the Open Systems Interconnection (OSI) model that standardizes communication over networks. If you're familiar with other media-based networking protocols, then MOST may look familiar.

Figure 2-10: MOST divided into the seven layers of the OSI model. The OSI layers are in the right column.

MOST Control Blocks

In MOST25, a block consists of 16 frames. A frame is 512 bits and looks like the illustration in Figure 2-11.

Figure 2-11: MOST25 frame

Synchronous data contains 6 to 15 quadlets (each quadlet is 4 bytes), and asynchronous data contains 0 to 9 quadlets. A control frame is 2 bytes, but after combining a full block, or 16 frames, you end up with 32 bytes of control data.

An assembled control block is laid out as shown in Figure 2-12.

Figure 2-12: Assembled control block layout

The data area contains the FblockID, InstID, FktID, OP Type, Tel ID, Tel Len, and 12 bytes of data. FblockIDs are the core component IDs, or function blocks. For example, an FblockID of 0x52 might be the navigation system. InstID is the instance of the function block. There can be more than one core function, such as having two CD changes. InstID differentiates which core to talk to. FktID is used to query higher-level function blocks. For instance, a FktID of 0x0 queries a list of function IDs supported by the function block. OP Type is the type of operation to perform, get, set, increment, decrement, and so forth. The Tel ID and Len are the type of telegram and length, respectively. Telegram types represent a single transfer or a multipacket transfer and the length of the telegram itself.

MOST50 has a similar layout to MOST25 but with a larger data section. MOST150 provides two additional channels: Ethernet and Isochronous. Ethernet works like normal TCP/IP and Appletalk setups. Isochronous has three mechanisms: burst mode, constant rate, and packet streaming.

Hacking MOST

MOST can be hacked from a device that already supports it, such as through a vehicle's infotainment unit or via an onboard MOST controller. The Linux-based project most4linux provides a kernel driver for MOST PCI devices and, as of this writing, supports Siemens CT SE 2 and OASIS Silicon Systems or SMSC PCI cards. The most4linux driver allows for user-space communication over the MOST network and links to the Advanced Linux Sound Architecture (ALSA) framework to read and write audio data. At the moment, most4linux should be considered alpha quality, but it includes some example utilities that you may be able to build upon, namely:

most_aplay Plays a *.wav* file

ctrl_tx Sends a broadcast control message and checks status

sync_tx Constantly transmits

sync_rx Constantly receives

The current most4linux driver was written for 2.6 Linux kernels, so you may have your work cut out for you if you want to make a generic sniffer. MOST is rather expensive to implement, so a generic sniffer won't be cheap.

The FlexRay Bus

FlexRay is a high-speed bus that can communicate at speeds of up to 10Mbps. It's geared for time-sensitive communication, such as drive-by-wire, steer-by-wire, brake-by-wire, and so on. FlexRay is more expensive to implement than CAN, so most implementations use FlexRay for high-end systems, CAN for midrange, and LIN for low-cost devices.

Hardware

FlexRay uses twisted-pair wiring but can also support a dual-channel setup, which can increase fault tolerance and bandwidth. However, most FlexRay implementations use only a single pair of wiring similar to CAN bus implementations.

Network Topology

FlexRay supports a standard bus topology, like CAN bus, where many ECUs run off a twisted-pair bus. It also supports star topology, like Ethernet, that can run longer segments. When implemented in the star topology, a FlexRay hub is a central, active FlexRay device that talks to the other nodes. In a bus layout, FlexRay requires proper resistor termination, as in a standard CAN bus. The bus and star topologies can be combined to create a hybrid layout if desired.

Implementation

When creating a FlexRay network, the manufacturer must tell the devices about the network setup. Recall that in a CAN network each device just needs to know the baud rate and which IDs it cares about (if any). In a bus layout, only one device can talk on the bus at a time. In the case of the CAN bus, the order of who talks first on a collision is determined by the arbitration ID.

In contrast, when FlexRay is configured to talk on a bus, it uses something called a *time division multiple access (TDMA)* scheme to guarantee determinism: the rate is always the same (deterministic), and the system relies on the transmitters to fill in the data as the packets pass down the wire, similar to the way cellular networks like GSM operate. FlexRay devices don’t automatically detect the network or addresses on the network, so they must have that information programed in at manufacturing time.

While this static addressing approach cuts down on cost during manufacturing, it can be tricky for a testing device to participate on the bus without knowing how the network is configured, as a device added to your FlexRay network won't know what data is designed to go into which slots. To address this problem, specific data exchange formats, such as the Field Bus Exchange Format (FIBEX), were designed during the development of FlexRay.

FIBEX is an XML format used to describe FlexRay, as well as CAN, LIN, and MOST network setups. FIBEX topology maps record the ECUs and how they are connected via channels, and they can implement gateways to determine the routing behavior between buses. These maps can also include all the signals and how they're meant to be interpreted.

FIBEX data is used during firmware compile time and allows developers to reference the known network signals in their code; the compiler handles all the placement and configuration. To view a FIBEX, download FIBEX Explorer from *http://sourceforge.net/projects/fibexplorer/*.

FlexRay Cycles

A FlexRay cycle can be viewed as a packet. The length of each cycle is determined at design time and should consist of four parts, as shown in Figure 2-13.

Figure 2-13: Four parts of a FlexRay cycle

The static segment contains reserved slots for data that always represent the same meaning. The dynamic segment slots contain data that can have different representations. The symbol window is used by the network for signaling, and the idle segment (quiet time) is used for synchronization.

The smallest unit of time on FlexRay is called a *macrotick*, which is typically one millisecond. All nodes are time synced, and they trigger their macrotick data at the same time.

The static section of a FlexRay cycle contains a set amount of slots to store data, kind of like empty train cars. When an ECU needs to update a static data unit, it fills in its defined slot or car; every ECU knows which car is defined for it. This system works because all of the participants on a FlexRay bus are time synchronized.

The dynamic section is split up into minislots, typically one macrotick long. The dynamic section is usually used for less important, intermittent data, such as internal air temperature. As a minislot passes, an ECU may choose to fill the minislots with data. If all the minislots are full, the ECU must wait for the next cycle.

In Figure 2-14, the FlexRay cycles are represented as train cars. Transmitters responsible for filling in information for static slots do so when the cycle passes, but dynamic slots are filled in on a first-come, first-served basis. All train cars are the same size and represent the time deterministic properties of FlexRay.

Figure 2-14: FlexRay train representing cycles

The symbol window isn’t normally used directly by most FlexRay devices, which means that when thinking like a hacker, you should definitely mess with this section. FlexRay clusters work in states that are controlled by the FlexRay state manager. According to AUTOSAR 4.2.1 Standard, these states are as follows: ready, wake-up, start-up, halt-req, online, online-passive, keyslot-only, and low-number-of-coldstarters.

While most states are obvious, some need further explanation. Specifically, online is the normal communication state, while online-passive should only occur when there are synchronization errors. In online-passive mode, no data is sent or received. Keyslot-only means that data can be transmitted only in the key slots. Low-number-of-coldstarters means that the bus is still operating in full communication mode but is relying on the sync frames only.

There are additional operational states, too, such as config, sleep, receive only, and standby.

Packet Layout

The actual packet that FlexRay uses contains several fields and fits into the cycle in the static or dynamic slot (see Figure 2-15).

Figure 2-15: FlexRay packet layout

The status bits are:

- Reserved bit
- Payload preamble indicator
- NULL frame indicator
- Sync frame indicator
- Startup frame indicator

The frame ID is the slot the packet should be transmitted in when used for static slots. When the packet is destined for a dynamic slot (1–2047), the frame ID represents the priority of this packet. If two packets have the same signal, then the one with the highest priority wins. Payload length is the number in words (2 bytes), and it can be up to 127 words in length, which means that a FlexRay packet can carry 254 bytes of data—more than 30 times that of a CAN packet. Header CRC should be obvious, and the cycle count is used as a communication counter that increments each time a communication cycle starts.

One really neat thing about static slots is that an ECU can read earlier static slots and output a value based on those inputs in the same cycle. For instance, say you have a component that needs to know the position of each wheel before it can output any needed adjustments. If the first four slots in a static cycle contain each wheel position, the calibration ECU can read them and still have time to fill in a later slot with any adjustments.

Sniffing a FlexRay Network

As of this writing, Linux doesn't have official support for FlexRay, but there are some patches from various manufacturers that add support to certain kernels and architectures. (Linux has FlexCAN support, but FlexCAN is a CAN bus network inspired by FlexRay.)

At this time, there are no standard open source tools for sniffing a FlexRay network. If you need a generic tool to sniff

FlexRay traffic, you currently have to go with a proprietary product that'll cost a lot. If you want to monitor a FlexRay network without a FIBEX file, you'll at *least* need to know the baud rate of the bus. Ideally, you'll also know the cycle length (in milliseconds) and, if possible, the size of the cluster partitioning (static-to-dynamic ratio). Technically, a FlexRay cluster can have up to 1048 configurations with 74 parameters. You'll find the approach to identifying these parameters detailed in the paper "Automatic Parameter Identification in FlexRay based Automotive Communication Networks" (IEEE, 2006) by Eric Armengaud, Andreas Steininger, and Martin Horauer.

When spoofing packets on a FlexRay network with two channels, you need to simultaneously spoof both. Also, you'll encounter FlexRay implementations called *Bus Guardian* that are designed to prevent flooding or monopolization of the bus by any one device. Bus Guardian works at the hardware level via a pin on the FlexRay chip typically called *Bus Guardian Enable (BGE)*. This pin is often marked as optional, but the Bus Guardian can drive this pin too high to disable a misbehaving device.

Automotive Ethernet

Because MOST and FlexRay are expensive and losing support (the FlexRay consortium appears to have disbanded), most newer vehicles are moving to Ethernet.

Ethernet implementations vary, but they're basically the same as what you'd find in a standard computer network. Often, CAN packets are encapsulated as UDP, and audio is transported as voice over IP (VoIP). Ethernet can transmit data at speeds up to 10Gbps, using nonproprietary protocols and any chosen topology.

While there's no common standard for CAN traffic, manufacturers are starting to use the IEEE 802.1AS Audio Video Bridging (AVB) standard. This standard supports quality of service (QoS) and traffic shaping, and it uses time-synchronized UDP packets. In order to achieve this synchronization, the nodes follow a *best master clock* algorithm to determine which node is to be the timing master. The master node will normally sync with an outside timing source, such as GPS or (worst case) an on-board oscillator. The master syncs with the other nodes by sending timed packets (10 milliseconds), the slave responds with a *delay request*, and the time offset is calculated from that exchange.

From a researcher's perspective, the only challenge with vehicle Ethernet lies in figuring out how to talk to the Ethernet. You may need to make or buy a custom cable to communicate with vehicle Ethernet cables because they won't look like the standard twisted-pair cables that you'd find in a networking closet. Typically, a connector will just be wires like the ones you find connected to an ECU. Don't expect the connectors to have their own plug, but if they do, it won't look like an RJ-45 connector. Some exposed connectors are actually round, as shown in Figure 2-16.

Figure 2-16: Round Ethernet connectors

OBD-II Connector Pinout Maps

The remaining pins in the OBD-II pinout are manufacturer specific. Mappings vary by manufacturer, and these are just guidelines. Your pinout could differ depending on your make and model. For example, Figure 2-17 shows a General Motors pinout.

Figure 2-17: Complete OBD pinout cable view for a General Motors vehicle

Notice that the OBD connector can have more than one CAN line, such as a low-speed line (LS-CAN) or a mid-speed one (MS-CAN). Low-speed operates around 33Kbps, mid-speed is around 128Kbps, and high-speed (HS-CAN) is around 500Kbps.

Often you’ll use a DB9-to-OBDII connector when connecting your sniffer to your vehicle’s OBD-II connector. Figure 2-

18 shows the plug view, not that of the cable.

Figure 2-18: Typical DB9 connector plug view. An asterisk (*) means that the pin is optional. A DB9 adapter can have as few as three pins connected.

This pinout is a common pinout in the United Kingdom, and if you're making a cable yourself, this one will be the easiest to use. However, some sniffers, such as many Arduino shields, expect the US-style DB9 connector (see Figure 2-19).

Figure 2-19: US-style DB9 connector, plug view

The US version has more features and gives you more access to other OBD connectors besides just CAN. Luckily, power is pin 9 on both style connectors, so you shouldn’t fry your sniffer if you happen to grab the wrong cable. Some sniffers, such as CANtact, have jumpers that you can set depending on which style cable you’re using.

The OBD-III Standard

OBD-III is a rather controversial evolution of the OBD-II standard. OBD-II was originally designed to be compliant with emissions testing (at least from the regulators' perspective), but now that the powertrain control module (PCM) knows whether a vehicle is within guidelines, we're still left with the inconvenience of the vehicle owner having to go for testing every other year. The OBD-III standard allows the PCM to communicate its status remotely without the owner's interaction. This communication is typically accomplished through a roadside transponder, but cell phones and satellite communications work as well.

The California Air Resources Board (CARB) began testing roadside readers for OBD-III in 1994 and is capable of reading vehicle data from eight lanes of traffic traveling at 100 miles per hour. If a fault is detected in the system, it'll transmit the diagnostic trouble codes (DTC) and vehicle identification numbers (VIN) to a nearby transponder (see "Diagnostic Trouble Codes" on page 52). The idea is to have the system report that pollutants are entering the atmosphere without having to wait up to two years for an emissions check.

Most implementations of OBD-III are manufacturer specific. The vehicle phones home to the manufacturer with faults and then contacts the owner to inform them of the need for repairs. As you might imagine, this system has some obvious legal questions that still need to be answered, including the risk of mass surveillance of private property. Certainly, there's lots of room for abuses by law enforcement, including speed traps, tracking, immobilization, and so on.

Some submitted request for proposals to integrate OBD-III into vehicles claim to use transponders to store the following information:

- Date and time of current query
- Date and time of last query
- VIN
- Status, such as "OK," "Trouble," or "No response"
- Stored codes (DTCs)
- Receiver station number

It's important to note that even if OBD-III sends only DTC and VIN, it's trivial to add additional metadata, such as location, time, and history of the vehicle passing the transponder. For the most part, OBD-III is the bogeyman under the bed. As of this writing, it has yet to be deployed with a transponder approach, although phone-home systems such as OnStar are being deployed to notify the car dealer of various security or safety issues.

Summary

When working on your target vehicle, you may run into a number of different buses and protocols. When you do, examine the pins that your OBD-II connector uses for your particular vehicle to help you determine what tools you'll need and what to expect when reversing your vehicle's network.

I've focused in this chapter on easily accessible buses via the OBD-II connector, but you should also look at your vehicle wiring diagrams to determine where to find other bus lines between sensors. Not all bus lines are exposed via the OBD-II connector, and when looking for a certain packet, it may be easier to locate the module and bus lines leaving a specific module in order to reverse a particular packet. (See Chapter 7 for details on how to read wiring diagrams.)