Innovate Motorsports
OT-1b/2 SDK

Version 1.3

Disclaimer: Information in this document is being **provided as-is** without any warranty/guarantee of any kind. Innovate Motorsports has taken all reasonable measures to ensure the quality, reliability, and accuracy of the information in this document. But if you corrupt an ECU, melt the casing off your computer, or experience some other terrible calamity, it is not our problem. Feel free to let us know, we might enjoy laughing at your misfortune, but please do not sue us. You have been warned!

Table of Contents

Revision History:

Rev	Date	Author	Description
1.0	11/20/09	jjf	Initial version
1.1	12/02/09	jjf	Structure typo fixes in Setup Mode
1.2	12/13/09	jjf	Updated Expert Modes to match version 1.02 Added ‘a’ to Setup Mode command list Added appendix on determining PID availability
1.3	12/28/09	jjf	Added standard IMS input values to App A Fixed type in ISO Export ‘O’ command

Introduction

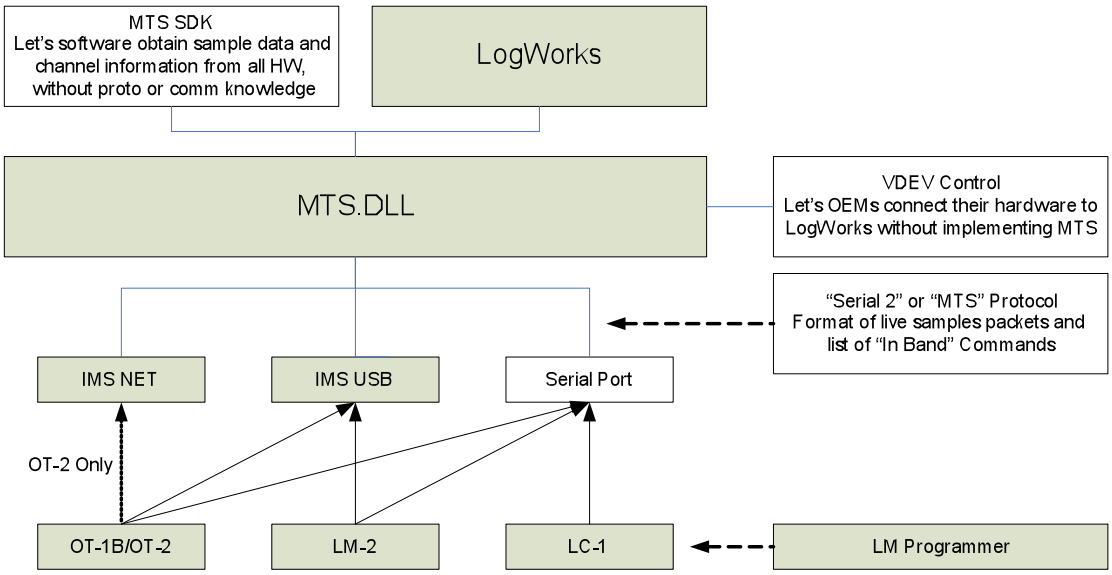
Welcome to the world of OT-1b/2 Programming! Hopefully, this document and the accompanying sample applications should make the process trivial… or not…

Seriously, our policy with regards to the OT-1b/2 SDK is full disclosure and an emphasis on 3rd party support. If there is a question that is not answered here, just ask. Likewise, if there is a feature that would make the OT-1b/2 work better (or work at all!) for your application, let us know. We can always be reached at support@innovatemotorsports.com.

Further, the enclosed is not covered by any form of Non Disclosure Agreement, so feel free to share information with other developers. But again, the policy here is disclosure. So, if you find a 'back door' which is useful to you, it is best to let us know. That way, we will not accidentally 'close it' in future firmware updates, etc.

SDK Scope

Because Innovate has released various "SDKs" and specifications over the years, it is probably worth covering where this SDK fits in the overall scheme of things. This will also introduce you to some long standing Innovate 'buzz words'. Here is (roughly) the state of public and not so public information just prior to this SDK:



At the lowest 'public' level in the chart above is the "Serial 2" or "MTS" Protocol specification. Our original product, the LM-1, uses a slightly different serial protocol, but all other devices, including the OT-1b and OT-2, automatically convert that protocol to

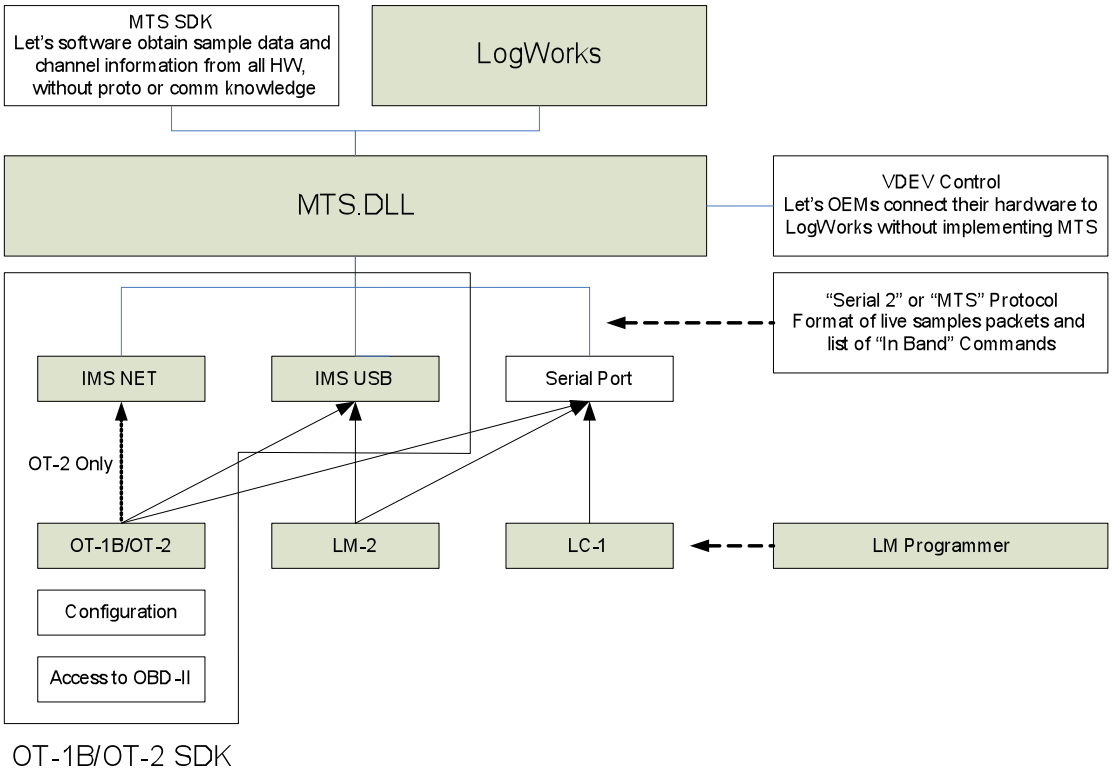
MTS when they see it. So, if you are only connecting through an OT-1b/2, you should never have to worry about the older format.

Although a substantial number of 3rd party developers have made good use of the public information above, several limitations have gradually emerged. First, just understanding the basic data packet structure does not give developers direct access to our newer 'transport layers' (IMS USB and IMS NET). Applications that use our high level SDK (MTS SDK) have continued to pick up new transports and devices for 'free'. For example, the 'sample app' included with that SDK has not been recompiled since it was first released, but the app 'understands' an LM-2 via USB, and can connect wirelessly to an OT-2, even though both those products (and transports) came much later.

However, developers who are not working in Windows, cannot utilize ActiveX, or simply need a more seamless connection to our hardware, have been left a bit behind. The devices still 'speak' MTS, but the developers do not now how to properly 'listen', or access the data stream.

Second, up to now, device configuration has (mostly) been closed (accessed by LM Programmer only, even LogWorks, our own data acquisition application, does not directly configure hardware). We have given out bits and pieces of information on how to configure individual devices, but have, in general, been reluctant to release this information. To be frank, part of the reluctance has been pragmatic. It is possible to put some of the devices into strange, non-working configurations. And, if the product stops working, the manufacturer, not the 3rd party vendor, gets the support call. But part of the reason has been complexity. The MTS protocol *does* have a mechanism for getting and setting configuration information for specific devices in a chain, but it was not properly implemented in the first MTS devices deployed. And those devices, in particular the LC-1, were deployed in substantial quantities, for more than a year, before the problem was ever discovered.

These original mistakes, combined with the need to provide proper legacy support, have made configuration management inside our own software products pretty complicated. There has been some resistance to exposing that sort of complexity to outside developers, but, clearly, some applications really warrant configuration control. So enter our new diagram:



As the outline on the diagram above shows, this SDK covers, generically, how to communicate with Innovate devices via either our universal USB driver or our network protocol. This information can be applied to devices besides the OT-1b and OT-2, such as the LM-2.

In addition to this generic information, this SDK provides information that is *specific* to the OT-1b and OT-2. Namely, how to configure those two particular devices (all the features exposed by LM Programmer, plus a few more) and, perhaps more interestingly, access to very low OBD-II services for fairly direct interaction with vehicle ECUs (Electronic Control Units).

Prerequisites

We tried to create the MTS SDK with the application programmer in mind. But, because of the subjects involved, this SDK presumes some system level programming experience. A solid understanding of the C programming language will be very helpful in understanding the examples and explanations.

In individual subjects additional background knowledge can also be very helpful. For example, a basic understanding of Windows or BSD 'sockets' will make the section on IMS Net much easier to understand. However, we have tried to add some introduction links to each subject where possible.

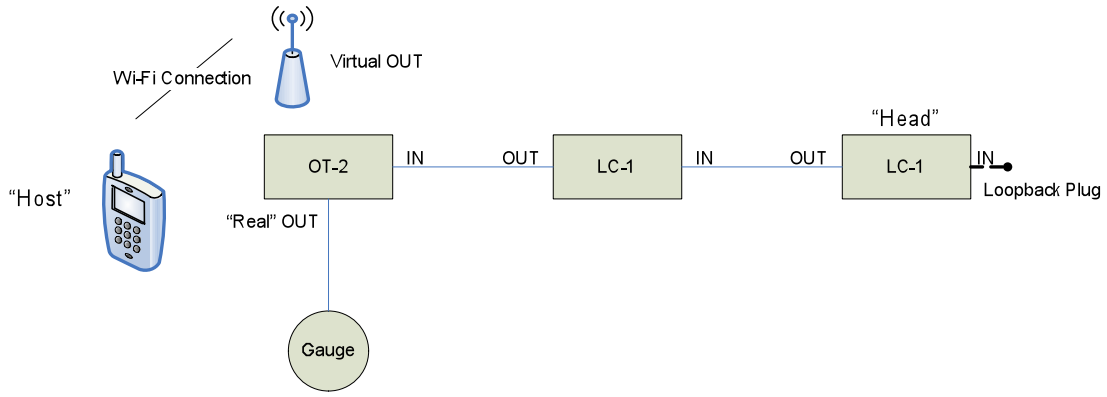
MTS Basics (or "Serial 2 revisited")

Regardless of how you physically connect to an OT-1B/2, its default behavior is to continuously send a stream of MTS data packets. We often refer to this as "in band" communication. This is also the *only* way to get information from any other MTS devices chained to the OT-1b/2, so some review of the Serial 2 protocol is in order. This protocol is covered in the verbosely named "Innovate Serial Protocol 2 (ISP2) specification (preliminary)". That document, as well as the only slightly more tersely named "Serial2 Protocol Supplement" are possibly worth reading (simply email support@innovate-tech.com and request them). But they should be taken with a grain of salt. Some features are not implemented in all devices, and the OT-1b/2, when accessed via USB/Wi-Fi, 'breaks' a couple of the documented rules (which were written with serial port communication in mind).

The following, combined with the source samples will, hopefully, give you most, if not all, the information typically required.

Basic Scheme

The basic scheme of MTS is not *too* complicated. With the exception of the host (that's generally you!), all devices have two serial ports. The "Serial IN" is for talking to devices earlier in the chain. The "Serial OUT" is for talking to devices later in the chain (or, if it is the last device, the host). Even though the terms, "IN" and "OUT" imply a direction, it is important to understand that each is a full duplex RS-232 port that the devices use to talk to each other. Even if you are communicating with the device via USB or Wi-Fi you are, in an MTS sense, 'talking' to the "Serial OUT" port of what is, from your point of view, the last device in the chain. Examine this simple (and reasonably common) setup:



The user has two LC-1 wideband controllers, one for each 'bank' in a large engine, an OT-2 for obtaining some basic information from the vehicle's ECU (like RPM), and one of our gauges for monitoring air fuel measurements from the LC-1's in real time.

When our host connects to the OT-2 via Wi-Fi, it is virtually connecting to the physical serial OUT on the device. That is, it starts seeing the same MTS data that is streaming to the gauge. But it does *not* see the gauge, or any other devices chained from the OT-2's serial OUT on.

Because of the way the protocol works, everyone (including the host) only has access to data generated *ahead* of them in the chain. Anything *behind* you is invisible to you.

So, from the point of view of our host (presumably an application running on the retro looking 'smart phone' above), the gauge simply does not exist. We can put our finger over it and see a straight line chain, with us at the left side of it.

At the other end of our chain from our host we have our chain's "Head" unit. How does that particular LC-1 know that it is the head unit? Simple, at power-up each device (but not the host) sends an "H" to its serial IN port. If the device hears an echo, it knows that it is the one at the end of a chain with the loop back plug.

There has been some confusion about loop back plugs (sometimes called "Arnold", for the sake of a pun too painful to repeat here). User's seem confused about rather the plugs are necessary or not. The bottom line is that a working MTS chain must always have a loopback at the end. The reason that it seems like it is sometimes not necessary is that most new MTS devices (like the OT-1b/2) self terminate when nothing is plugged into them. Only the earlier MTS devices, like the LC-1, actually require a physical plug.

Once a unit has determined that it is 'Head', it begins to start sending data packets, each containing what information it has (in this case, an AFR measurement) to the device's serial OUT connection. Data packets are **always generated once every 81.92 milliseconds**.

This is important because it means that the MTS stream does not just represent data values, but also represents a timeline. As to 'why 81.92 milliseconds?' the answer is that automotive science suggests that a sample rate of at least 12 Hz is very good for the types of measurements we normally do with our equipment. But dividing 1 with 12 gives 83 and 1/3 milliseconds (lots of 3333…) But if you take an 8 MHz clock and divide it by 65536 (a 16 bit counter), you get 81.92 milliseconds. Care to guess at the clock rate of the original LM-1 or the size of the counter registers used to generate the MTS sample rate in it?!

So, now we have a 'timeline' (or sample clock) where each packet represents a 'tick', a measurement from the first LC-1, and a history lesson, but what happens next? To understand you need to remember that devices 'see' data from all the devices that proceed them (it arrives on their own serial IN port). So, our second LC-1, just after the 'head', 'sees' the MTS packets transmitted by the first one.

Not only does it see them, the second LC-1 specifically has to pass them on, transmitting them on its own serial OUT connector, for the data to ever reach our host. But, it does not have to pass on *exactly* what it hears!

What the second LC-1 does is to alter the packet header before passing it on. It increases the 'length' field, so that the packet the next unit in the chain (our OT-2) sees is larger – by the size of the data that the second LC-1 itself has to provide (again, like the first LC-1, just one AFR measurement). Then, after altering and relaying the header, the second LC-1 relays the rest of the original packet it receives unchanged. Finally, it sends its own data which, because of the altered header, is now properly part of the packet which the next device in line (the OT-2) receives.

For most of the life of an MTS chain, this is what occurs. The head generates a data packet, and each device in the chain expands the 'size' portion of the header, and then tags its individual sample data at the end of the packet it is passing on, until the ever growing packet reaches the host, where it is typically recorded or displayed.

In our illustration above, this is data flowing from right to left, over and over and over. But there are two other basic cases; 'Commands' and 'Queries'. Both are single characters sent from devices downstream (the left side of our diagram) to devices upstream (the right side of our diagram). That is, they start by flowing in the opposite direction from the data packets described above.

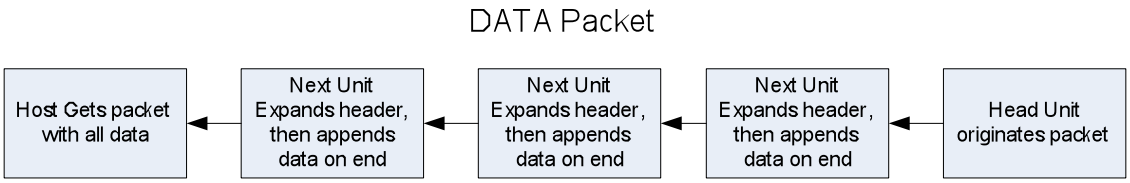
Commands are easy. They are received on each device's serial OUT port, acted on (if applicable), and then passed on, unchanged, to the device's serial IN port. Take the case of 'c', the calibrate command.

Either our host, or our gauge, can transmit it. It is received by the OT-2 on either the 'real' or 'virtual' serial out port. The OT-2 has no need for calibration, so it does nothing, but it passes the command on by transmitting it via its serial IN port.

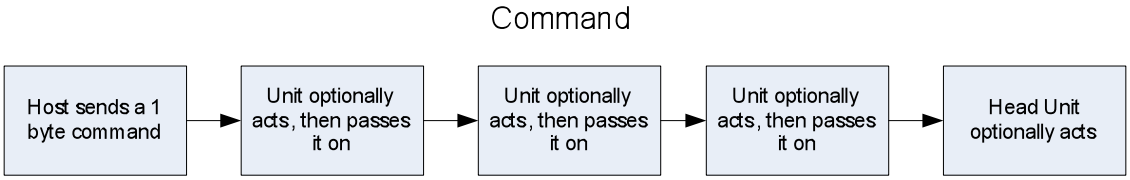
Next, an LC-1 receives the command, and it does have a need to calibrate. So it starts free air calibration, and then passes the command on to the next device upstream. That LC-1, in turn, also starts free air calibration. But, because it is the 'head' unit, it does not bother passing the command on to its serial IN port.

Queries are sort of a combination of these two cases. They are one byte, with the high bit set, that are generated by the host. Unlike commands, they are not immediately acted on by devices in the chain, but are passed on. The Head unit is the first unit to take action. Instead of a data packet, it generates a response packet, with its own response to the query. That packet then flows down the chain just like a data packet, with each device altering the size of the header and adding its own response information to the end of the packet, so that the packet is a collection of responses by the time it reaches the host.

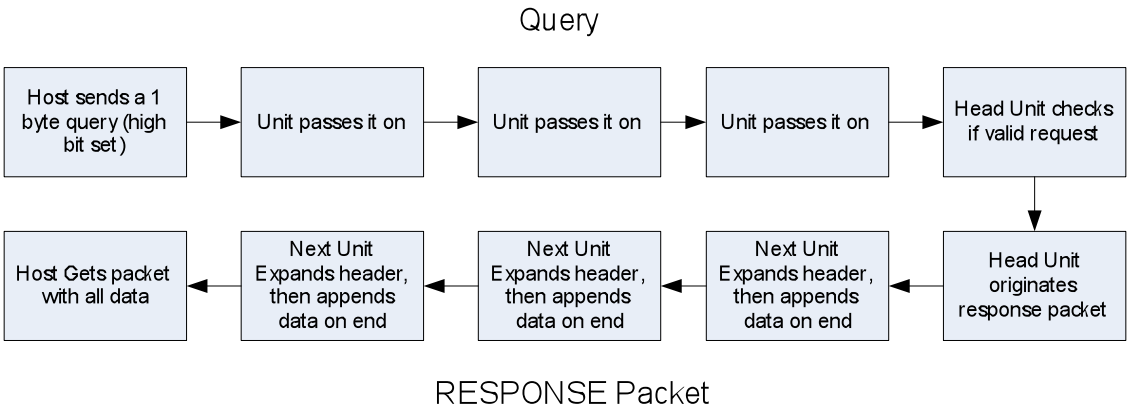
So, to recap, we have three basic cases. First, typical sample data flow:



Second, ‘Commands’:



Third, and finally, the more complicated ‘Query’:



So, for MTS “In Band” communication (normal), the Host receives two kinds of packets:

- Data
- Response

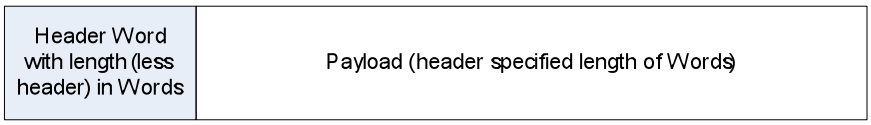
And transmits two types of one byte instructions:

- Commands
- Queries

Not exactly simple, but not ASN encoded H.324 connection negotiation either. Now that you have the big picture, let’s review the technical details.

Packets in General

As covered above, in 'in band' (normal) MTS communication, hosts receive two types of packets, data, and query responses. The packets are almost identical (they are distinguished by one bit in the header). The basic structure is:



Note that "Word" means two bytes (16 bits), and they are sent in "Big Endian" order, that is, 0x1234 would be sent in two consecutive bytes, '0x12', then '0x34'. For better or worse, there is no checksum or other form of error checking.

In the Serial 2 document, the header looks a bit daunting:

Header Word

Word	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 3	Bit 1	Bit 0
0	1	R	1	D/S	HF	X	1	B7	1	B6	B5	B4	B3	B2	B1	B0

But it really breaks down into just a few simple things. First, there are some fixed values:

Word	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 3	Bit 1	Bit 0
0	1	R	1	D/S	HF	X	1	B7	1	B6	B5	B4	B3	B2	B1	B0

These uniquely identify the header from any other in band data sent. When you start monitoring the stream of bytes you can begin by AND'ing each byte with 0xA2 and seeing if the result is 0xA2. If it is, you might have synced to the first byte of a packet. If you AND the next byte with 0x80 and get 0x80, you can be sure. This test should not pass at any other point in the data stream except the beginning of a packet.

Next, we have the length (in words) of what is to follow:

Word	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 3	Bit 1	Bit 0
0	1	R	1	D/S	HF	X	1	B7	1	B6	B5	B4	B3	B2	B1	B0

This is one byte, so the maximum payload size is 255 words, or 510 bytes. One oddness is that, because of the fixed values, the size is split and has to be reassembled. But, if you count off received data, this reconstructed length will lead you to the next header.

The last piece of information that we generally care about is the packet type (remember, we have two):

Word	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 3	Bit 1	Bit 0
0	1	R	1	D/S	HF	X	1	B7	1	B6	B5	B4	B3	B2	B1	B0

This is all handled by bit 12. A 1 indicates that it is a DATA packet, a 0 indicates that it is a RESPONSE packet.

This leaves three other bits:

Word	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 3	Bit 1	Bit 0
0	1	R	1	D/S	HF	X	1	B7	1	B6	B5	B4	B3	B2	B1	B0

Of these, only bit 14 might be of interest. It indicates that someone in the chain is recording. This is how our gauges know when to blink the 'recording' light. Bits 10 and 11 should be ignored.

Now that we can sync to the stream and grab packets, let's look at what is inside them.

Data Packets

In discussing the basic MTS scheme, we referred a lot to 'devices', but when analyzing a data packet, it is much easier to think in terms of what we commonly refer to as 'channels'. A channel would be a single sensor reading.

Devices can add more than one channel to the data packet (though the firmware in many of the devices, and our LogWorks software assumes that the *total* count of channels in the packet will not exceed 32). But, while an LC-1 adds 1 channel, an LMA-3 adds 5, an OT-1b/2 adds from 1-16, and an LM-2 can add up to a whopping 23, the channels added will always be one of three types:

- An Aux Channel
- A 'new' lambda/AFR channel
- An 'old' (LM-1) lambda/AFR channel

Aux Channels

An Aux Channel is the simplest, as we can see from the Serial 2 Specification:

Word	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 3	Bit 1	Bit 0
N	0	0	D12	D11	D10	D9	D8	D7	0	D6	D5	D3	D3	D2	D1	D0

It has room for 13 bits of data, but **all Innovate Devices only use 10 bits**. The reason is legacy/historical. Notice that there is no indication of data size in the one word (two byte) payload. The first MTS devices sent 10 bit data, and sent it in bits D0 through D9. If a device now used the upper bits, the host would have no easy way of knowing which channels require which scale. So, for an Aux Channel, 0 (0x000) is always the minimum, 1023 (0x3FF) is always the maximum.

Note that, unlike the header word, the MSB in both bytes of this word are 0. This is to facilitate sync/packet parsing, but it does require that the fixed 0 bit has to be removed and the 10 bit value properly combined for use.

'New' Lambda/AFR Channels

A 'new' lambda/AFR Channel is a bit more complicated:

Word	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 3	Bit 1	Bit 0
0	0	1	0	F2	F1	F0	1	AF 7	0	AF 6	AF 5	AF 4	AF 3	AF 2	AF 1	AF 0
1	0	0	L12	L11	L10	L9	L8	L7	0	L6	L5	L4	L3	L2	L1	L0

The first thing to note is bit 14 of the first word. Unlike an Aux Channel, which is always 0, in this channel type, it is always 1. This signals that the channel requires an additional word. Like the Aux Channel, the high bits are 0 in every byte, to help distinguish it from the header word.

Again, the idea is to facilitate parsing, high bits, think header, counting off channels, bit 14 set, grab another word for this channel… But, again, this also requires that values sometimes be recombined. Ultimately, the channel contains three basic pieces of information. First, a value:

Word	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 3	Bit 1	Bit 0
0	0	1	0	F2	F1	F0	1	AF 7	0	AF 6	AF 5	AF 4	AF 3	AF 2	AF 1	AF 0
1	0	0	L12	L11	L10	L9	L8	L7	0	L6	L5	L4	L3	L2	L1	L0

Although our software often clips this value to 10 bits, the devices themselves do, in fact, send a full 12 bits of information. However, depending on the circumstances, the value represents different things (lambda, percentage of O2, and error code, etc.). So the packet contains information about what type of information is being sent:

Word	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 3	Bit 1	Bit 0
0	0	1	0	F2	F1	F0	1	AF 7	0	AF 6	AF 5	AF 4	AF 3	AF 2	AF 1	AF 0
1	0	0	L12	L11	L10	L9	L8	L7	0	L6	L5	L4	L3	L2	L1	L0

These three bits allow for seven different 'functions', or data meanings:

000	Lambda valid, lambda value contains lambda reading in .001 lambda increments Offset by .5 lambda (0x000 = .5 lambda, 0x3FF = 1.523 lambda)
001	Lambda value contains O2 level in 1/10%
010	Free air calibration in progress, Lambda data not valid
011	Need Free air Calibration Request, Lambda data not valid
100	Warming up, Lambda value is temp in 1/10% of operating temp.
101	Heater Calibration, Lambda value contains calibration countdown.
110	Error code in Lambda value
111	reserved

If you don't want to parse all possible states, you can simply go on 000 means valid data, not 000 means invalid data. Last we have the information needed to approximate AFR from the actual lambda reading:

Word	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 3	Bit 1	Bit 0
0	0	1	0	F2	F1	F0	1	AF 7	0	AF 6	AF 5	AF 4	AF 3	AF 2	AF 1	AF 0
1	0	0	L12	L11	L10	L9	L8	L7	0	L6	L5	L4	L3	L2	L1	L0

This is the 'Air Fuel Multiplier', scaled by 10. In other words, 147 really represents 14.7. You would multiply this number by the lambda measurement to get approximate AFR for the fuel currently specified.

Although you probably have figured all this out, we get asked how to calculate the proper values from this packet a lot, so, at the risk of repeating, the basic measurements are:

```
Assemble "L"
Assemble "AF"
Assemble "F"

if (F == 0) // Valid
{
      float lambda = (L * 0.001) + 0.500;
      float afr = lambda * AF / 10;
}
```

'Old' (LM-1) Lambda/AFR Channels

The old lambda channel (sent only by the LM-1) is the most complicated, and also the most troublesome for parsing. Although all the sample applications properly parse this packet, frankly, if you are writing your own parser you might do well to simply ignore it, unless you know that you will have to deal with connected LM-1's.

Word	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 3	Bit 1	Bit 0
0	LM1	R	0	F2	F1	F0	0	AF 7	0	AF 6	AF 5	AF 4	AF 3	AF 2	AF 1	AF 0
1	0	0	L12	L11	L10	L9	L8	L7	0	L6	L5	L4	L3	L2	L1	L0
2	0	0	mb 2	mb 1	mb 0	bv9	bv8	bv7	0	bv6	Bv5	bv4	bv4	bv2	bv1	bv0

3	0	0	0	0	0	Ax19	Ax18	Ax17	0	Ax16	Ax15	Ax14	Ax13	Ax12	Ax11	Ax10
4	0	0	0	0	0	Ax29	Ax28	Ax27	0	Ax26	Ax25	Ax24	Ax23	Ax22	Ax21	Ax20
5	0	0	0	0	0	Ax39	Ax38	Ax37	0	Ax36	Ax35	Ax34	Ax33	Ax32	Ax31	Ax30
6	0	0	0	0	0	Ax49	Ax48	Ax47	0	Ax46	Ax45	Ax44	Ax43	Ax42	Ax41	Ax40
7	0	0	0	0	0	Ax59	Ax58	Ax57	0	Ax56	Ax55	Ax54	Ax53	Ax52	Ax51	Ax50

Although the Serial 2 specification documents this as 7 words, the last five (3-7) are Aux Channels, and can be treated as such. Word 2 contains battery voltage (albeit in a complex formula) and can also generally be ignored.

This leaves the first two words. The good news is that they contain the same three values (L, F, and AF) as the 'new' lambda channel. And, better still, the meaning is the same. The bad news is that bit 15 is set in the first word, and bit 14 may or may not be set, eliminating our ability to find header bytes or count off channels simply. In any event, the LM-1 channel would have to identified so the battery information would not be inadvertently treated as an Aux Channel.

Response Packets

If you value your sanity (and the stability of the devices in the MTS chain), you will only ever send two queries:[1]

- 0xCE to obtain device names
- 0xF3 to obtain device types

Yes, there are other documented queries, and they look interesting, but these are the only two that we use in our software, so they are the only ones consistently and correctly implemented in all devices.

The response packet generated by each is very similar:

Header Word with length (less header) in Words	Word indicating type of response	Device Response Always 4 Words (8 Bytes)	Repeat for number of devices responding

The header we covered above. The "Response Type" word is also pretty simple. It is the Query (0xCE or 0xF3), placed in a word which follows the same rules as an Aux Channel in a Data Packet. That is, the high bit of both bytes is 0:

Word	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 3	Bit 1	Bit 0

[1] For the one exception to this, see "Special Considerations for Network/Wi-Fi" later in this document.

N	0	0	0	0	0	0	0	D7	0	D6	D5	D3	D3	D2	D1	D0

This leaves the 4 Words (sometimes easier to think of as 8 bytes in this case) that each device provides in the response. The response to the 0xCE query you can probably guess. The 8 bytes contain the Device's ASCII name, padded to 8 bytes with zeros. Since users can alter the names, and the default names are the type of device, we use this name information to provide somewhat descriptive default channel names in our software. The response to 0xF3, query device types, is also 8 bytes, and contains the following information:

Byte	0-1	2-5	6	7
Description	Firmware Version	Identifier	CPU	Channels/Flags

The first two bytes contain the firmware version, encoded in nibbles (Big Endian). For example, 1.23a would be 0x12 0x3A. The last nibble should be ignored, since it represents type of build (beta, manufacturing, update dld file, etc.). However, you will want to pay attention to the upper three nibbles. This SDK requires version 1.02 (0x102n) or higher to work. All OT-2 units meet this requirement, but OT-1b units may require firmware update.

The next four bytes are a unique device identifier. This would be "OT1B" or "OT2 " (note the trailing space character) for the two devices covered by this SDK. Again, if you are connected to an OT-2 via Wi-Fi, this is not really something you have to check. But if you are connected via USB, then you definitely need to check, because you could be connected to something completely different, like an LM-2.

The CPU byte is a number that represents the CPU and clock speed of the device. This is only of interest to LM Programmer for firmware updates, so I have no idea why it is included here, so do not ask.

The last byte is of real interest, but its meaning is device specific. For devices which have fixed numbers of channels, like the LC-1 and the TC-4, this byte carries some hardware related flags. In devices that have variable numbers of channels, like the OT-1b/2 and LM-2, this byte gives information on how many channels that the device is currently adding to data packets. For the OT-1, OT-1B, and OT-2, using type byte this is simple. The byte is a binary count of Aux Channels currently being generated.

So, if you are only interested in OT-1b/2 channels, it is not too complicated. You can get the device type, check this byte, and know that the last n channels in every MTS data packet are the channels you are interested in.

If you are specifically interested in channels from other devices, it gets more complicated. You need to know how many channels the fixed count devices each produce, and you need to understand how the LM-2 uses this byte, which is a bit more complicated because it can provide variable numbers of both Aux and Lambda channels.

At the time of this writing this would be:

Device	Channels
LC-1	(fixed) 1 Lambda
LM-1	(fixed) 1 Lambda (old) + 5 Aux
LMA-3	(fixed) 5 Aux
DL-32	(fixed) 5 Aux
SSI-4	(fixed) 4 Aux
TC-4	(fixed) 4 Aux
OT-1	(variable) Simple Count, 1-16 in last byte of Type Response
OT-1B	(variable) Simple Count, 1-16 in last byte of Type Response
OT-2	(variable) Simple Count, 1-16 in last byte of Type Response
LM-2	(variable) Count (See Below) in last byte of Type Response

* The LM-2 always provides 1 Lambda channel minimum. In addition, the last Type Response byte has the following meaning, bit 7 set, +1 Lambda (2nd sensor), bit 6 set, +1 Aux (RPM), bit 5 set, +4 Aux (Analog Inputs), bits 0-4, count of additional Aux (OBD-II, can be 0-16)

Important: Remember, because of the way MTS works, the device closest to the host is the last response in the packet, the device furthest away the first. So, for either a USB or network connection, the safe procedure to make sure that the rest of the features in this SDK are available would be:

1. Sync to the MTS stream (find the header and start parsing packets)
2. Send a Type query (0xF3)
3. Check that the last device in the response packet is either an “OT1B” or “OT2 ”
4. Check that (FirmwareVersion & 0xFFF0) >= 0x1020

Commands

Although the OT-1b and the OT-2 both only respond to one in band MTS commands, there are some others that you might want to send to the chain:

‘c’ - Calibrate (useful for calibrating LC-1s)
‘R’ - Start Recording (works with LM-1, LM-2, and DL-32)
‘r’ - Stop Recording
‘e’ - ‘Erase’

This literally erases the log memory in an LM-1, but with an LM-2 or DL-32, it only forces the next ‘R’ command to create a new file on the SD card.

If you are reading this SDK, then there is one command that you will probably want to use for sure, because the OT-1b/2 will respond to it:

‘S’ - Enter Setup Mode…

Get to the Good Stuff!

OK, you've waded though more stuff on in band MTS communication than any sane person would ever want to know. Just think of all those wasted brain cells! So now it is time to start getting some payoff.

You've synced to the MTS data stream, you are parsing packets, and you have even gotten snazzy and checked device types and names (or, you do not care and are ignoring everything that the OT-1b/2 is currently spitting at you). Now, you send one byte:

'S'

Congratulations, you have just boldly gone where third party developers have never gone before (at least with support and documentation), you have entered "Setup Mode".

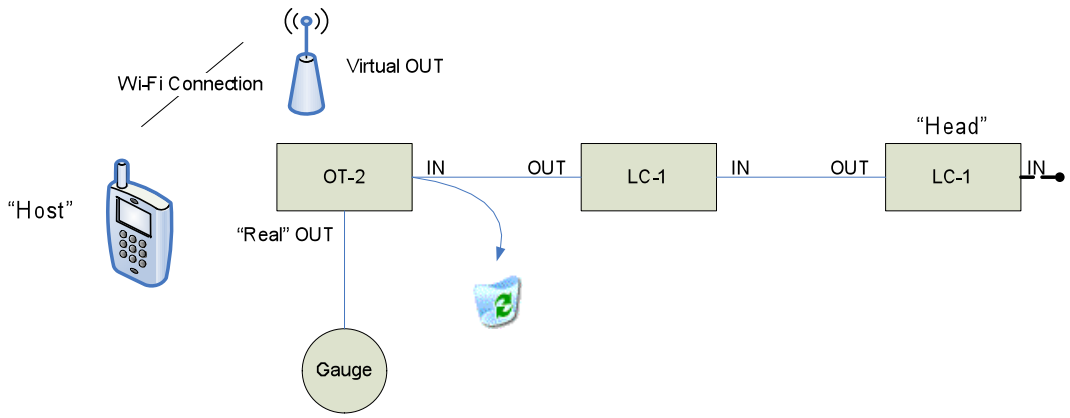
This command is mentioned in the Serial 2 protocol specification, but there are no real details. So…

Setup Mode

Normally, devices can only enter this mode if they are 'head', that is, nothing chained before them. This caveat is because of problems mentioned previously with large numbers of units in the field which do not properly handle in-chain configuration related commands. However, the OT-1b and OT-2 are special in that, when connected to via USB or Wi-Fi (OT-2 only), they can enter Setup Mode at any time. This is because there are no devices between them and the host to potentially get confused.

When you enter Setup Mode, three things happen immediately.

1. Any data being received from MTS devices upstream, starts being discarded



This means that we are no longer receiving packets from the LC-1s in the example above. In addition, the gauge is no longer receiving information. So, unless you have a compelling reason to do so, or you know that no other devices are attached to the OT-2, you do not want to linger in Setup Mode. This relates to the next thing that happens…

2. The OT-1b/2 starts a 'Watchdog' timer

Because the units are designed to be setup and left in a vehicle, and because Setup Mode potentially stops expected behavior with other devices (see #1 above), the firmware takes steps to insure that you are in Setup Mode on purpose. Namely, it times out and exits after 10 seconds of inactivity (or more than 2 seconds of delay between bytes on a multi-byte command see below).

Setup mode is also exited automatically if the firmware detects that the connection (USB or Wi-Fi) has been severed. The point behind this is that even if the host software crashes (or is rudely and suddenly terminated by the operating system, as sometimes occurs on handheld platforms like Android and the iPhone), the unit will restore itself to a normal state after a reasonable delay and without user intervention.

It is mentioned here because it can be a gotcha when debugging. You might be single step debugging, become briefly distracted by the sheer elegance of your code, and then suddenly find yourself receiving wholly unexpected responses. You look down, and find that the MTS light on the device itself has started blinking again…

Seriously, as long as we remain aware of it, we have not found this to be much of a problem in our own debugging. But some platforms can take more than 10 seconds just to transition from a break point to single step debugging (especially in some embedded development). So, if you find yourself stumped, and unable to debug your problem because of the watch dog timer, please let us know and we will help you work around it.

If you simply *must* stay in Setup Mode, even though you have gaps in needed commands of 10 seconds or more, you can periodically send the dummy command 0xFF (no response) to reset the watchdog timer.

3. The OT-1b/2 sends a 'welcome to setup mode' message

This message is 15 bytes long and, frankly, the last 9 bytes are essentially gibberish for. They contain a lot of conditional information that LM Programmer uses for options presentation and firmware updates. So, for the purposes of this SDK, let's call them reserved for Innovate use. This makes our 15 byte message as follows:

Byte	0-1	2-5	6-14
Description	Firmware Version	Identifier	Bytes you do not need to know about and which I do not want to document

The Firmware Version and Identifier should be exactly the same as the response that you got for the in band Query Device Types (0xF3). This can be a handy way to 'sync' when entering Setup Mode.

Since data packets have been streaming, you may have an indeterminate amount of data buffered up. You can send the 'S' command, and then parse the data stream until you find the expected Firmware Version and Identifier. Discard 9 more bytes and you are ready to begin using Setup Mode Commands.

Setup Mode Commands

Note: Many of the commands below assume you have a basic knowledge of OBD-II. It is beyond the scope of this SDK to fully cover this, but a good background article can be found here:

http://www.dakota-truck.net/OBD2/obd2.html

And, although I am loathe to cite it, Wikipedia has some pretty good general material on the subject (though, since anyone can edit it, that can easily change):

http://en.wikipedia.org/wiki/OBD-II#OBD-II

That out of the way, back to the subject at hand; unlike in-band mode, Setup Mode is not packeted, and commands and responses are variable in length. There is no terminator or delimiter (<cr>, <lf>, etc.), bytes are processed as they are received.

Important: In the following list, commands and/or responses are often documented as pseudo C structures. In all cases, the structures are **byte packed** (no padding or alignment bytes) and values are stored in **little endian** (yes, 'in band' communication is all big endian, but Setup Mode is device specific and the OT-1b/2 is little endian).

Also note, U8, U16, U32, S8, S16, etc. represent unsigned and signed values of the given number of bits, again in little endian. No, the world does not need yet another set of standard data types, but we often share code between firmware and host software, so we have standardized on these.

Similarly, while the example applications generally use defines for various return values, the following explanations will typically include absolute numbers. This is simply a matter of preference. I, personally, like to be able to use this sort of reference to immediately interpret what I am seeing. Using defines adds an extra level of complexity, since the matching header files, etc. have to be referenced as well.

Command ‘S’ - Get Setup Mode Header

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘S’
            }

Responds:   15 bytes
            {
                  U8 VersionH;      // Firmware version MSB
                  U8 VersionL;      // Firmware version LSB
                  U8 Identifier[4]; // Unique identifier
                  U8 Reserved[9];   // Don’t ask!
            }

Notes:      This is exactly the same response you receive automatically
            when you entered the mode.
```

Command ‘s’ – Exit Setup Mode

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘s’
            }

Responds:   None

Notes:      This command returns you to in-band mode, and disables the
            watchdog timer.
```

Command ‘j’ – Get OBD-II Connection Status

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘j’
            }

Responds:   33 bytes
            {
                  U8 Status;        // Connection Status
                                    // 0 = Not Connected
                                    // 1 = Connected CAN
                                    // 2 = Connected J1850 PWM
                                    // 3 = Connected J1850 VPW
                                    // 4 = Connected KWP2000
                                    // 5 = Connected ISO
                                    // 0xFF = Powerdown mode
                  U32 PidMasks[8];  // These 8 32 bit values represent
                                    // responses to the standard
                                    // 00, 20, 40, etc. mode 1 PID
                                    // requests
            }
```

Notes: This is a case where the abstraction level in the firmware is not complete. The unit does not directly poll PIDs. Instead, you designated one of about 100 ‘normalized PIDs’ (see the ‘c’ command and the separate section). But, in terms of availability of normalized PIDs, the host has to check for the associated ECU pid in the table returned by this command (see Appendix B), or ask about a given ‘normalized PID’ using the ‘a’ command below.

Command ‘a’ – Check availability of a ‘normalized’ PID

```
Sent:       3 bytes
            {
                  U8 Cmd;           // ‘a’
                  U16 Pid;          // Normalized PID to check
            }

Responds:   1 byte
            {
                  U8 Status;        // PID Status
                                    // 0 = Not Currently Available
                                    // 1 = Available
            }
```

Notes: Although this command is here for completeness, it is not a very efficient way to see if a particular normalized PID is available. It is generally faster to check for the associated ECU pid against the PID masks returned by the ‘j’ command (see above). See Appendix B for an example of how to do this.

Command ‘k’ – Get OBD-II Diagnostic Trouble Codes (DTCs)

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘k’
            }

Responds:   33 bytes
            {
                  U8 Count;         // Number of DTCs returned, or
                                    // 0xFF for none
                  U16 DTCs[16];     // DTCs
            }
```

Notes: This list should be open ended, but is fixed at 16 max for legacy reasons (compatibility with the original OT-1). The DTCs are in their original J1979 format.

Command ‘K’ – Clear All OBD-II Diagnostic Trouble Codes (DTCs)

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘K’
            }

Responds:   1 byte
            {
                  U8 Result;        // 1=success, 0=failure
            }
```

Notes: It should be noted that ‘success’ only means that the request has been submitted to the ECU. Rather or not the codes were, in fact, all cleared is something you have to ask the ECU (see ‘k’ above).

Command ‘l’ – Get Last OBD-II Loop Time

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘l’
            }

Responds:   2 bytes
            {
                  U16 Time;         // Time in mS of last OBD-II thread
                                    // ‘loop’, or 0 if not connected
            }
```

Notes: When you enter Setup Mode, the thread which polls the ECU for values to send out via MTS keeps running. Each time the thread has grabbed all the PIDs requested, it tracks time. This command gets the last time measured. We use this to intelligently configure the device. For example, our preference may be to grab RPM + VSS, but if we set that configuration and the loop is too slow for our purposes (ISO 9141 perhaps?) we might degrade to RPM only. The measurement is just one iteration, and will vary from cycle to cycle, but it gives an approximation of performance for the current configuration.

Command ‘v’ – Get Vehicle VIN number

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘v’
            }

Responds:   18 bytes
            {
                  U8 Count;         // char count (norm. 17)
                                    // or 0xFF for failure
                  U8 Vin[17];       // Vehicle VIN
            }
```

Notes: Not all vehicles support this mode 9 info type (2), but if the vehicle does, it is a nice way to make your software vehicle specific aware.

Command ‘t’ – Get Basic Emissions Status

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘t’
            }

Responds:   5 bytes
            {
                  U8 Count;         // char count (norm. 4)
                                    // or 0xFF for failure
                  U32 Status;       // This is simply the 4 bytes
                                    // from Mode 1, PID 1

            }
```

Notes: This PID (mode 1 pid 1) just gives a simple overview of the emissions tests and their current status. Of interest if you live in California.

Command ‘c’ – Get Device Configuration

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘c’
            }

Responds:   36 bytes
            {
                  U8 Channels;      // Number of channels (16 max)
                  U8 Protocol;      // OBD-II protocol
                                    // 0 = automatic
                                    // 1 = can
                                    // 2 = pwm
                                    // 3 = vpw
                                    // 4 = kwp
                                    // 5 = iso
                  U16 NormPid[16];  // Table of ‘normalized’ PIDs
                                    // to scan
                  U16 Flags;        // Priority flags
            }
```

Notes: This controls how the ECU is connected to, what channels are scanned and put in MTS data packets, and how the values are scanned. It should be noted that these are not ECU pids, but abstracted, or ‘normalized’ PIDs. This is to accommodate the limits of 10 bit channels, etc. in MTS. See the separate section on understanding this configuration for more information.

Command ‘C’ – Set Device Configuration

```
Sent:       37 bytes
            {
                  U8 Cmd;           // ‘C’
                  U8 Channels;      // Number of channels (16 max)
                  U8 Protocol;      // OBD-II protocol
                                    // 0 = automatic
                                    // 1 = can
                                    // 2 = pwm
                                    // 3 = vpw
                                    // 4 = kwp
                                    // 5 = iso
                  U16 NormPid[16];  // Table of ‘normalized’ PIDs
                                    // to scan
                  U16 Flags;        // Priority flags
            }

Responds:   1 byte
            {
                  U8 Result;        // Should always be 0xD
            }
```

Notes: This writes a new configuration in the device’s flash memory (see ‘c’ above). WE STRONGLY RECOMMEND NOT USING THIS COMMAND UNLESS ABSOLUTELY NECESSARY!!! Instead, consider using the ‘M’ command (see below), which will temporarily change the configuration (the original configuration is restored when your connection to the unit is terminated).

Command ‘M’ – Set ‘My’ (temporary) Device Configuration

```
Sent:       37 bytes
            {
                  U8 Cmd;           // ‘M’
                  U8 Channels;      // Number of channels (16 max)
                  U8 Protocol;      // OBD-II protocol
                                    // 0 = automatic
                                    // 1 = can
                                    // 2 = pwm
                                    // 3 = vpw
                                    // 4 = kwp
                                    // 5 = iso
                  U16 NormPid[16];  // Table of ‘normalized’ PIDs
                                    // to scan
                  U16 Flags;        // Priority flags
            }

Responds:   1 byte
            {
                  U8 Result;        // Should always be 0xD
            }
```

Notes: This has alters the device configuration (see ‘c’ above), just like the ‘C’ command (see above). The difference is that it does not alter the settings permanently, but only for the duration of the current USB or Wi-Fi connection to the device. This lets applications do whatever they want, but then settings return automatically to the user’s programmed preferences when you exit/disconnect. WE STRONGLY RECOMMEND USING THIS COMMAND INSTEAD OF ‘C’ UNLESS ABSOLUTELY NECESSARY!!!

Command ‘m’ – Clear ‘My’ (temporary) Device Configuration

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘m’
            }

Responds:   1 byte
            {
                  U8 Result;        // should always be 0xD
            }
```

Notes: This restores the current device settings (see ‘c’ above) to those stored in flash memory (see ‘C’ above). This happens automatically when a connection is terminated.

Command ‘w’ – Get Wi-Fi Settings

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘w’
            }

Responds:   16 bytes
            {
                  U8 HWAddr[6];     // MAC Address
                  U32 IPAddr;       // IP Address (in NETWORK ORDER!)
                  U32 IPMask;       // IP Mask (in NETWORK ORDER!)
            }

Notes:      On an OT-1B, this will return all zeros.
```

Command ‘e’ – Enter an Expert Mode

```
Sent:       2 bytes
            {
                  U8 Cmd;           // ‘e’
                  U8 Mode;          // 1 = can
                                    // 2 = j1850
                                    // 3 = iso
                                    // 4 = iso raw
            }

Responds:   1 byte
            {
                  U8 Result;        // Should match ‘Mode’ requested
            }

Notes:      Expert Modes are ‘sub modes’ to Setup Mode. Because they
            allow you to perform very low level OBD-II operations, you
            have to pick the mode that matches your desired electrical
            protocol. You would use 1 for all can modes (std, ext.), 2
            for j1850 (pwm or vpw), and 3 for K line based standards
            (iso 9141 or kwp2000). Mode 4 is for vendor specific K line
            based protocols (ex. MUT, SSM, etc.) One important note is
            that this command can, potentially, take a LONG time to
            respond. See the section on Expert Mode for more details.
```

Command 0xFF – Don't hang up on me!

Sent: 1 bytes

```
{
      U8 Cmd;           // 0xFF
}
```

Responds: None

Notes: This command merely gives you a way to thwart the Setup Mode watchdog timer. In general, you should avoid even needing it. But, here it is, just in case…

Understanding OT-1b/2 Device Configuration

To fully understand the Device Configuration (which can be obtained with the ‘c’ command, and set with the ‘C’ and ‘M’ commands documented in the previous section), we need to take a step back and look at some MTS/OBD-II conflicts.

To start with, different OBD-II PIDs return different size values, often with huge ranges. But, unless you jumped over the long and tedious first sections, you know that MTS Aux Channels are limited to 10 bits (0-1023) for full range.

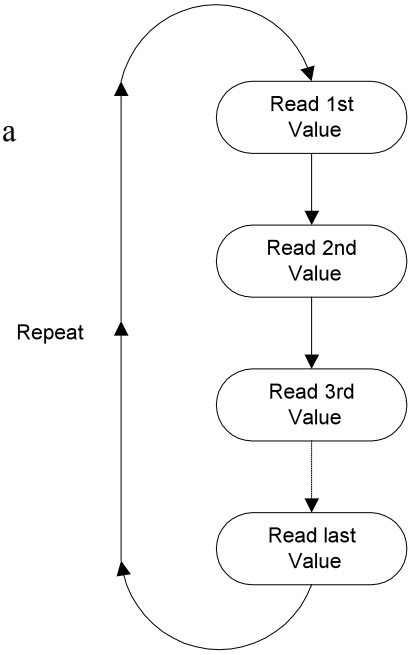
A second problem is that ECU response times vary wildly, but the MTS ‘timeline’ keeps ticking at 81.92 milliseconds. We concluded that it was too much to expect the user to directly deal with both these problems, so we addressed the first one by creating a layer of abstraction between the user and actual ECU PIDs. We call these ‘virtual’ PIDs, ‘normalized’ PIDs, because they have been scaled to fit into our 10 bit samples and typical scales for certain values. A table of these ‘Normalized PIDs’, along with their ranges and associated ECU mode 1 PID can be found in Appendix A.

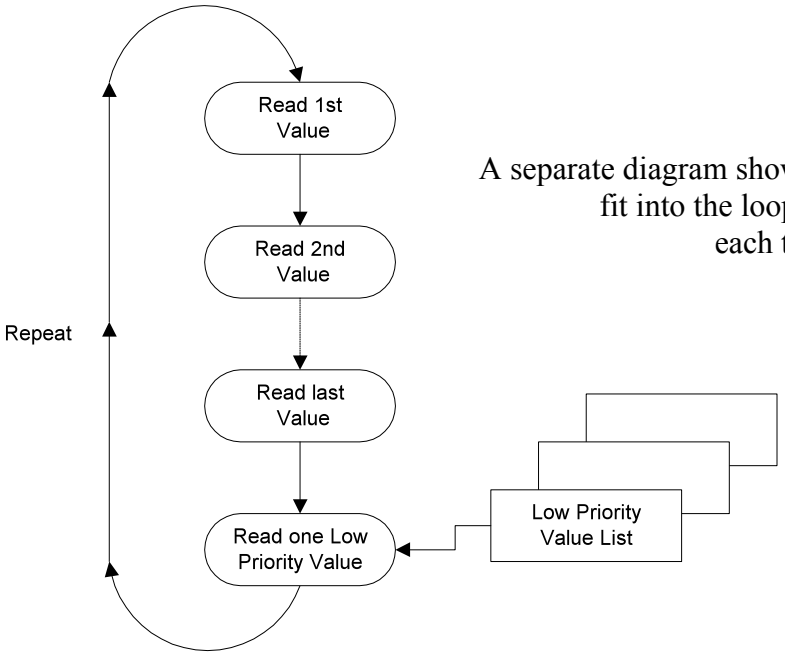
The problem of ECU performance we partially address by giving the user some tools. For example, the Vehicle led blinks at the same scale that we use for the MTS light, so users can get a visual indication on how the polling speed of their selected normalized PIDs compares to the fixed MTS sample rate. Software can make a similar comparison using the ‘l’ (Get Last OBD-II loop time) command. We also use the PID mask returned from the connection status command (‘j’) to determine which normalized PIDs to present to the user.

Last, we give the user a simple priority mechanism. Individual channels can be normal priority – polled once each loop, or low priority. Only one low priority channel is polled each time through the loop. So, if the ECU supports 4 channels at acceptable rate, 3 can be used for critical values, and the 4th shared for multiple low rate channels, like various temperature readings.

Some simple diagrams of this scheme can be found in the OT-x manuals. Normal priority channels are read one after another in a loop.

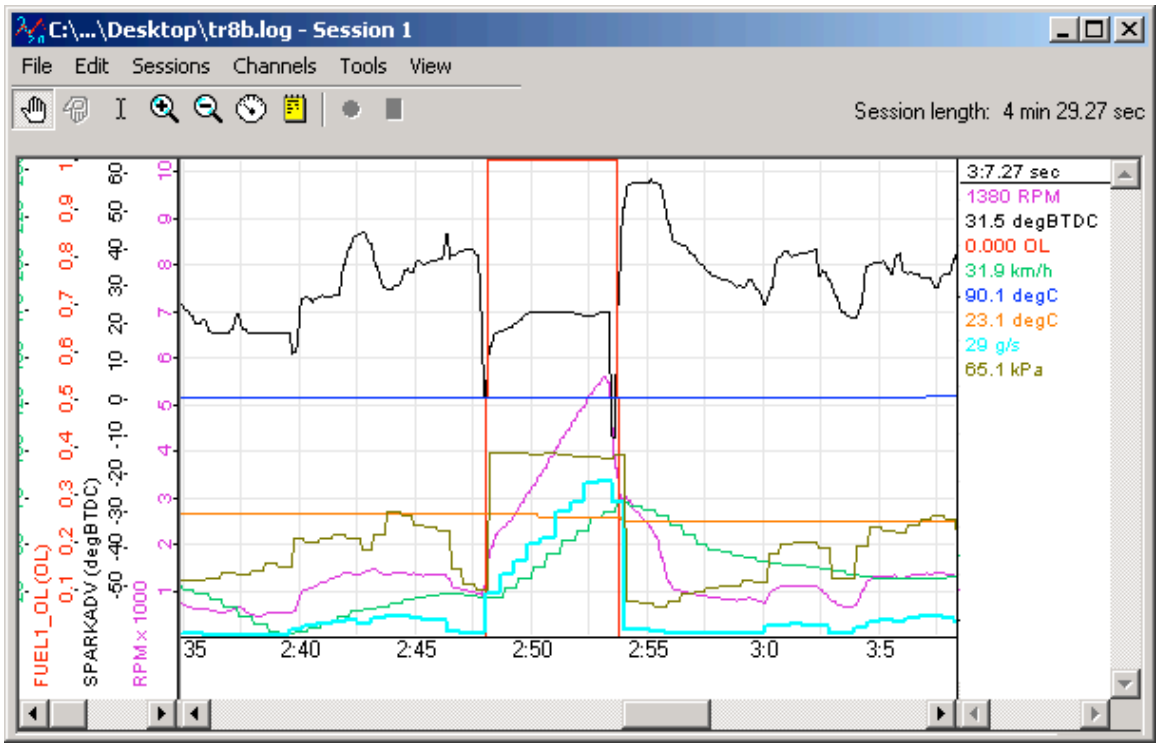
The more channels you add, the slower the loop.





A separate diagram shows how low priority channels fit into the loop. Only one is read each time around.

The manual then shows this scheme in actual practice with a log from a Chevy Suburban:



The ECU can typically keep up at MTS rates with about 3-4 channels. Using the ‘low priority’ mechanism lets some critical channels, like spark advance and RPM, be tracked at higher resolution (normal priority), while lower priority measurements are allowed to get ‘steppy’ (low priority).

Now that we understand ‘Normalized PIDs’ and ‘Priority’, let’s look at the response to the ‘Get Configuration’ (‘c’) command again:

```
Responds:   36 bytes
            {
                  U8 Channels;       // Number of channels (16 max)
                  U8 Protocol;       // OBD-II protocol
                                     // 0 = automatic
                                     // 1 = can
                                     // 2 = pwm
                                     // 3 = vpw
                                     // 4 = kwp
                                     // 5 = iso
                  U16 NormPid[16];   // Table of 'normalized' PIDs
                                     // to scan
                  U16 Flags;         // Priority flags
            }
```

The first value, ‘Channels’, is both the number of MTS Aux Channels that will be generated and the number of Normalized PIDs that will be scanned (it is a one to one relationship).

‘Protocol’ controls how the ECU will be communicated with. If you do not know what these mean, best to leave it on automatic!

‘NormPid’ is a table of Normalized PIDs. These are the items that will be polled for each channel. So, if ‘Channels’ is set to 2, the first two entries will be used, ‘Channels’ at 16, all entries will be used.

It is your responsibility to make sure that the normalized PID selected can be supported by the ECU (yes, a little ugly). However, if you select a ‘bad’ Normalized PID, do not worry, the firmware will skip it and just return 0 for that channel in the MTS Data Packet. You can inquire about the availability of a specific normalized PID using the ‘a’ command in Setup Mode. Or, by checking the PID masks returned by the ‘j’ command (see Appendix B).

The ‘Flags’ field is where low priority can be selected. One bit per channel. A 1 = low priority, a 0 = normal priority. Bit 0 controls the first channel (NormPid[0]), bit 1 controls the second channel (NormPid[1]), and so on.

In Setup Mode, the OBD-II scan thread keeps running, so if you only change channels or priority, the ‘M’ (or ‘C’) command is very quick, with the MTS stream changing immediately. If you change the protocol, the connection to the ECU will be terminated and a new one attempted. This can take as much as 20 seconds.

Benefits of In Band Mode

The reasons for Innovate using MTS In Band Mode are pretty clear, it makes OBD-II compatible with our existing product line and all of our software. But your objectives are probably quite different, so you may wonder if you should bother with MTS data packets and normalized PIDs at all.

This is a reasonable question. There is some complexity, and some real limitations, like fitting everything into 10 bit samples. But there are some benefits as well.

First, there is basically the same reason that motivates us, it lets you concurrently sample OBD-II data and our other instruments. Our lambda instruments are, demonstrably, the fastest and most accurate that you can buy, and MTS in band support lets you sample their output digitally.

Second, if your OBD-II needs are relatively simple, just using normalized PIDs and reading the resulting MTS Data Packets may be a lot simpler than learning the underlying protocols so that you can use them directly.

The third and final advantage is a bit more subtle. Using an MTS data stream offloads the responsibility of tracking time. For some applications, consistent and steady sampling is a must, but circumstances can conspire against achieving that on the host. For example, some platforms and environments will intermittently freeze, as processing power is yielded to other activities.

A less obvious culprit is connectivity. In computer terms, the MTS data stream is very slow. By comparison, Wi-Fi is very fast. One would assume that there is no problem of time. But, in networking, you have to make a choice between immediacy and reliability. The OT-2 uses “TCP” as the protocol for connectivity (see the section on connectivity and transports later in this document). The advantage to TCP is that delivery is guaranteed. That is, data is not lost. The downside is that when network errors occur, and they do occasionally occur (1% is common on a typical wirless network), then there is often a pause of 200 mS or more before the host’s network stack attempts to retry.

If you are using MTS data packets, this is a non-issue, the lost packets are resent. But if you are polling the ECU yourself, then this timeout/retry gap becomes a gap where you have no sampled data.

The point of this is not to pressure you into using MTS data streams. But merely to point out that there are tradeoffs depending on how you use the instrument. Some applications absolutely need a much more direct, ECU aware, approach than the default behavior of the OT-1b/2. For those applications, there are…

Expert Modes

As mentioned earlier, Expert Modes are entered using the 'e' command from Setup Mode. They are specialized 'sub modes', each dedicated to one of the main three physical transports for OBD-II. Mode 1 is for CAN protocols, Mode 2 is for J1850 protocols, and Mode 3 is for ISO (K line based) protocols.

Note: If you plan on using the Expert Modes, you really should know more than me about OBD-II protocols and standards. Frankly, that is not all that hard a bar to clear, but it is high enough that there is no reason to mess around with tutorials and online articles, you really need to go to the standards themselves. The two primary sources are the Society of Automotive Engineers (SAE):

http://www.sae.org

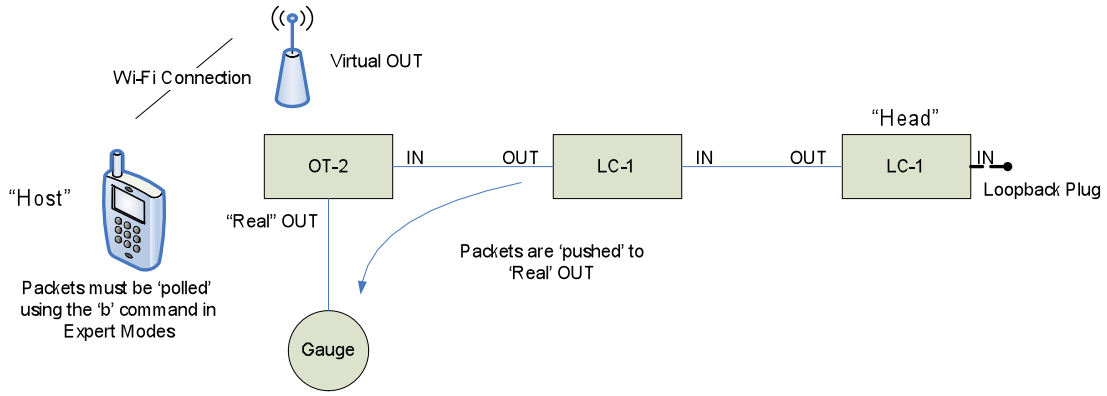
The International Organization for Standardization (ISO), and the International Electrotechnical Commission (IEC). Standards from the later two can normally be obtained through the American National Standards Institute (ANSI):

http://www.ansi.org/

Overview

Assuming you are already an OBD master, we can go over some basics that all three Expert Modes share. First, like the Setup Mode, there is a watchdog timer running. 10 seconds without activity and you get the boot. Like Setup Mode, you can prolong the agony by periodically sending a null command (0xFF). However, unlike Setup Mode, you will get a one byte response (also 0xFF). The reason for this is that the Expert Modes often return variable length results, so you want to be careful to stay in perfect sync.

Another significant difference between the Expert Modes and Setup Mode is that, if additional MTS compatible devices are attached to the OT-1b/2 Serial IN, MTS packets will flow to the OT-1b/2 Serial OUT. However, the channels associated with the OT-1b/2 will be '0' (since the unit is not polling OBD-II):



Unlike In Band mode, MTS packets are not 'pushed' to the virtual OUT connection in Expert Modes. You can poll the last MTS packet passed with the 'b' command which is available in all expert modes.

All three modes offer very similar commands, but the parameters are tweaked to match the physical requirements of the underlying protocol. In general, all modes offer a way to:

- Set Data Rate
- Send, and optionally receive a specified amount
- Just Receive
- Set filters on received data
- Control the Vehicle LED on the OT-1b/2
- Exit back to Setup Mode

There are some specialized commands for several of the protocols, but this is the basics. At first glance you might wonder why there is both a Send/Receive and a Receive only. One reason is timing. Depending on the host operating system (uncontrolled yields) or transport (bit errors on a network), the Host may not be able to meet the timing requirements for transactions for a given protocol. On the other hand, some applications require passive monitoring of other activity on the bus.

It also should be clear from this basic command structure that the interface is fundamentally half duplex. That is, you are either sending, or specifically listening to the OBD-II interface in question. And, when you are listening, you are listening for either a period of time or an expected number of responses.

There are some limitations to this sort of interface model. However, the primarily reason that it was selected was for best performance under as many network conditions as possible. A full duplex model would involve data tricking in both directions at variable rates. This sort of communication model is most likely to run into timing and throughput problems in a TCP connection. Half duplex, in the form of command/response, helps eliminate problems such as 'delayed ACK' (see the section on special considerations for Wi-Fi later in this document for more details).

Special Considerations Entering Expert Modes

Before entering an Expert Mode, the OT-1b/2 firmware cleanly suspends the thread that normally does OBD-II polling, this includes making sure that all OBD-II hardware lines are in a reasonable state. If the unit is already connected to an ECU, this process is reasonably quick. However, if the unit is set to automatic, and no ECU connection has yet been established, it can take a long time. The response to the 'e' command may not come for 10 seconds or more (don't worry, it does not count against your watchdog time).

For some applications, this may be acceptable. However, in many it can make for an unacceptable user experience.

Depending on your exact requirements, you could consider one of the following workarounds. First, you might consider not entering an expert mode until a connection to an ECU has been established. You could then use the result from Get Connection Status ('j') to decide which mode to use.

This can be a nice approach, but it will not work if either:

a. The ECU uses a protocol we do not understand (but you do)
b. Our code is not compatible with the ECU

And yes, obviously, b. cases do exist. In fact, that is one of the cases that expert mode can be particularly useful. In these cases, you might consider a different approach and configure the unit for something other than automatic. Ideally you would set it to a protocol that is fast (CAN or J1850pwm), but which you know is not supported. In this case entry into expert mode will be reasonably quick, and the normal OBD-II foreground loop will not have mucked up the water, so to speak, with a lot of intermittent connectivity to the ECU.

But, regardless of rather you get into an expert mode by waiting, or a workaround, you still have to use it, so…

Can

The general flow with Can Expert Mode (‘e’<1>), is:

1. Select a rate (250 Kbit or 500 Kbit)
2. Set up filters for the CAN identifiers that you are interested in
3. Output a packet and (optionally) wait for a designated number of response packets

Can mode is unique in that there are actually two types of filters. Normal ‘pass’ filters (see the ‘F’ and ‘f’ commands below), which simply identify packets to be passed on and Handshake filters (see the ‘E’ and ‘e’ commands below). Handshake filters instruct the OT-1b/2 to automatically deal with ISO 15765 flow control responses for a given identifier.

The export mode commands to accomplish these tasks are:

Command ‘R’ – Set the Protocol Rate

```
Sent:       2 bytes
            {
                  U8 Cmd;           // 'R'
                  U8 Rate;          // 1 = 250 Kbit
                                    // 2 = 500 Kbit
            }

Responds:   1 byte
            {
                  U8 Result;        // Should match 'Rate' requested
            }

Notes:      The default rate when you enter this Expert Mode is 250K
            bit. The two rates provided are the two that are covered in
            the ISO specification. However, if non standard rates would
            be useful, let us know.
```

Command ‘r’ – Get the Protocol Rate

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘r’
            }

Responds:   1 byte
            {
                  U8 Result;        // Current rate
                                    // 1 = 250 Kbit
                                    // 2 = 500 Kbit
            }
```

Notes: The default rate when you enter this Expert Mode is 250K bit. The two rates provided are the two that are covered in the ISO specification. However, if non standard rates would be useful, let us know.

Command ‘F’ – Add a Pass Filter

```
Sent:       9 bytes
            {
                  U8 Cmd;           // ‘F’
                  U32 Identifier;   // Identifier to pass on
                  U32 Mask;         // AND mask, 0xFFFFFFFF=exact match
                                    // 0x00000000=all pass
            }

Responds:   1 byte
            {
                  U8 Result;        // Number of filters in use
            }
```

Notes: Reminder, the U32 values above are in little endian!

These filters are just that, filters. Messages that match them are passed on, as received, and applied to the ‘replies’ count in O and I commands (see below). For automatic flow control, see the ‘E’ command below. It should be noted that if a packet passes one of these filters, it is not tested against the handshake filters. So, typically, you would use one type of filter or the other, not both for the same expected message.

Command ‘f’ – Clear all Pass Filters

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘f’
            }

Responds:   1 byte
            {
                  U8 Result;        // Should be 0
            }
```

Notes: Clear all pass filters. New filters are added with ‘F’ (above).

Command ‘E’ – Add a Handshake Filter

```
Sent:       9 bytes
            {
                  U8 Cmd;           // ‘E’
                  U32 Identifier;   // Identifier to pass on/FC
                  U32 HSAddress;    // Identifier to send FC messages
                                    // to
            }

Responds:   1 byte
            {
                  U8 Result;        // Number of handshake filters in
                                    // use
            }
```

Notes: Reminder, the U32 values above are in little endian!

These filters do two things. First, they specify and identifier to pass on. Second, matching identifier messages are checked for ISO 15765 flow control messages. If a multipart message is detected, a flow control message is sent to the identifier specified in HSAddress. Although standard OBD-II messages use a -8 offset for standard CAN messages and a BYTE swap (src/dest) for extended CAN messages, this is not honored in vehicle specific applications.

Command ‘e’ – Clear all Handshake Filters

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘e’
            }

Responds:   1 byte
            {
                  U8 Result;        // Should be 0
            }
```

Notes: Clear all handshake filters. New filters are added with ‘E’ (above).

Command ‘O’ – Output Data and (optionally) look for input

```
Sent:       Variable
            {
                  U8 Cmd;           // ‘O’
                  U8 Timeout;       // Timeout (in mS) for responses
                  U8 Replies;       // Expected number of replies
                                    // Can exceed 8
                  U8 ExtFlag;       // 0=std ID, 1=ext. ID
                  U32 ID;           // Can Message ID
                  U8 Len;           // Data length 1-8 (sets DLC)
                  U8 Data[Len];     // Data Bytes to Send
            }

Responds:   Variable (1 byte minimum)
            {
                  // 0 to ‘Replies’ of
                  {
                        U8 ExtFlag; // 0=std ID, 1=ext. ID
                        U32 ID;     // Can Message ID
                        U8 Len;     // Data length (from rcv’ed DLC)
                        U8 Data[Len];     // Received Data
                  }
                  U8 End;           // Always 0xFF
            }
```

Notes: Replies is CAN messages.

If replies is set to 0, the message will be sent, but the 0xFF will immediately be returned, regardless of ‘Timeout’.

Timeout is for all messages, not per message. If replies is set, the 0xFF will be returned after either a) the desired number of replies is received or b) this time is reached.

Timeout cannot be set to 0, doing do will default to a timeout of 50 mS.

Command ‘I’ – Look for Input Data

```
Sent:       3 bytes
            {
                  U8 Cmd;           // ‘I’
                  U8 Timeout;       // Timeout (in mS) for responses
                  U8 Replies;       // Expected number of replies
                                    // Can exceed 8
            }

Responds:   Variable (1 byte minimum)
            {
                  // 0 to replies of
                  {
                        U8 ExtFlag; // 0=std ID, 1=ext. ID
                        U32 ID;     // Can Message ID
                        U8 Len;     // Data length (from rcv’ed DLC)
                        U8 Data[Len];     // Received Data
                  }
                  U8 End;           // Always 0xFF
            }
```

Notes: Replies is ‘Can Messages’.

Timeout is for all messages, not per message. If replies is set, the 0xFF will be returned after either a) the desired number of replies is received or b) this time is reached.

Timeout cannot be 0. If a zero is submitted, a value of 1 is used instead.

Command ‘L’ – Set the Vehicle LED State

```
Sent:       2 bytes
            {
                  U8 Cmd;           // ‘L’
                  U8 State;         // 0 = off
                                    // 1 = on
            }

Responds:   1 byte
            {
                  U8 Result;        // Should match ‘State’ requested
            }
```

Notes: Because we have conditioned the user to expect the vehicle light to light or blink, you might want to follow the same model. You may also find this useful for debugging.

Command ‘b’ – Get Last MTS Packet passed

```
Sent:       1 bytes
            {
                  U8 Cmd;           // ‘b’
            }

Responds:   Variable (2 bytes minimum)
            {
                  U8 Len;           // Length of data to follow
                  U8 Sequence;      // Sequence Counter, incremented
                                    // each time an MTS packet is
                                    // passed
                  // Optional
                  {
                        U8 Packet[Len-1]; // MTS Packet (if any)
                  }
            }

            Notes:      If there is no MTS hardware attacked, you will
            get ‘0x01 0x00’ (1 byte, sequence 0). Otherwise, you will
            get a changing sequence, followed by a raw MTS packet.
```

Command ‘s’ – Exit Expert Mode

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘s’
            }

Responds:   None

Notes:      This command returns you to Setup Mode.
```

Command 0xFF – Don’t hang up on me!

```
Sent:       1 bytes
            {
                  U8 Cmd;           // 0xFF
            }

Responds:   1 byte
            {
                  U8 Result;        // should be 0xFF
            }

Notes:      This command merely gives you a way to thwart the Expert
            Mode watchdog timer. In general, you should avoid even
            needing it. But, here it is, just in case…
```

J1850

The general flow with J1850 Expert Mode (‘e’<2>), is:

1. Select a ‘rate’ (PWM or VPW, really selecting rate, voltage and encoding)
2. Setup filters for the J1850 packets you are interested in
3. Output a packet and (optionally) wait for a designated number of response packets

The export mode commands to accomplish these tasks are:

Command ‘R’ – Set the Protocol Rate

```
Sent:       2 bytes
            {
                  U8 Cmd;           // ‘R’
                  U8 Rate;          // 1 = VPW
                                    // 2 = PWM
            }

Responds:   1 byte
            {
                  U8 Result;        // Should match ‘Rate’ requested
            }

Notes:      J1850 has no default rate so you must set one. In addition,
            this command CAN FAIL (result does not match request). The
            reason is that the command checks if the required line(s)
            ever go idle. If they don’t, the command is failed, because
            attempting to transmit would hang.
```

Command ‘r’ – Get the Protocol Rate

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘r’
            }

Responds:   1 byte
            {
                  U8 Result;        // Current rate
                                    // 0 = none
                                    // 1 = VPW
                                    // 2 = PWM
            }

Notes:      There is no default rate for J1850 and Set Rate (‘R’) can
            fail (see above).
```

Command ‘F’ – Add a Pass Filter

```
Sent:       7 bytes
            {
                  U8 Cmd;           // ‘F’
                  U8 Header;        // Header to pass on
                  U8 Destination;   // Destination address to pass on
                  U8 Source;        // Source to pass on
                  U8 HMask;         // Header AND mask (0xFF = exact)
                  U8 DMask;         // Destination AND mask
                  U8 SMask;         // Source AND mask
            }

Responds:   1 byte
            {
                  U8 Result;        // Number of filters in use
            }

Notes:      These filters are just that, filters. Messages that match
            them are passed on, as received, and applied to the
            ‘replies’ count in O and I commands (see below).
```

Command ‘f’ – Clear all Pass Filters

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘f’
            }

Responds:   1 byte
            {
                  U8 Result;        // Should be 0
            }

Notes:      Clear all pass filters. New filters are added with ‘F’
            (above).
```

Command ‘O’ – Output Data and (optionally) look for input

```
Sent:       Variable
            {
                  U8 Cmd;           // ‘O’
                  U8 Timeout;       // Timeout (in mS) for responses
                  U8 Replies;       // Expected number of replies
                                    // 16 max
                  U8 Len;           // Data length 1-32 (32 must
                                    // include CRC)
                                    // If the high bit is set, the
                                    // firmware will calculate and
                                    // append a CRC byte for you
                  U8 Data[(Len&0x7F)];    // Data Bytes to Send
            }

Responds:   Variable (1 byte minimum)
            {
                  // 0 to ‘Replies’ of
                  {
                        U8 Len;     // Bits 0-6:
                                    // Data length of packet
                                    // 32 bytes maximum, CRC is
                                    // included in Len and Data
                                    // (below)
                                    //
                                    // Bit 7:
                                    // CRC test, 1=failed
                        U8 Data[(Len&0x7F)];    // Received Data
                  }
                  U8 End;           // Always 0xFF
            }
```

Notes: Replies is for J1850 packets.

If replies is set to 0, the message will be sent, but the 0xFF will immediately be returned, regardless of ‘Timeout’.

Timeout is per message. If replies is set, the 0xFF will be returned after either a) the desired number of replies is received or b) this time is reached without a new reply.

Timeout cannot be set to 0, doing do will default to a timeout of 100 mS.

Command 'I' – Look for Input Data

```
Sent:       3 bytes
            {
                  U8 Cmd;           // 'I'
                  U8 Timeout;       // Timeout (in mS) for responses
                  U8 Replies;       // Expected number of replies
                                    // Up to 16
            }

Responds:   Variable (1 byte minimum)
            {
                  // 0 to 'Replies' of
                  {
                        U8 Len;     // Bits 0-6:
                                    // Data length of packet
                                    // 32 bytes maximum, CRC is
                                    // included in Len and Data
                                    // (below)
                                    //
                                    // Bit 7:
                                    // CRC test, 1=failed
                        U8 Data[(Len&0x7F)];    // Received Data
                  }
                  U8 End;           // Always 0xFF
            }
```

Notes: Replies is for J1850 packets.

Timeout is per message. If replies is set, the 0xFF will be returned after either a) the desired number of replies is received or b) this time is reached without a new reply.

Timeout cannot be set to 0, doing do will default to a timeout of 100 mS.

Command 'L' – Set the Vehicle LED State

```
Sent:       2 bytes
            {
                  U8 Cmd;           // 'L'
                  U8 State;         // 0 = off
                                    // 1 = on
            }

Responds:   1 byte
            {
                  U8 Result;        // Should match 'State' requested
            }
```

Notes: Because we have conditioned the user to expect the vehicle light to light or blink, you might want to follow the same model. You may also find this useful for debugging.

Command ‘b’ – Get Last MTS Packet passed

```
Sent:       1 bytes
            {
                  U8 Cmd;           // ‘b’
            }

Responds:   Variable (2 bytes minimum)
            {
                  U8 Len;           // Length of data to follow
                  U8 Sequence;      // Sequence Counter, incremented
                                    // each time an MTS packet is
                                    // passed
                  // Optional
                  {
                        U8 Packet[Len-1]; // MTS Packet (if any)
                  }
            }

            Notes:      If there is no MTS hardware attacked, you will
            get ‘0x01 0x00’ (1 byte, sequence 0). Otherwise, you will
            get a changing sequence, followed by a raw MTS packet.
```

Command ‘s’ – Exit Expert Mode

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘s’
            }

Responds:   None

Notes:      This command returns you to Setup Mode.
```

Command 0xFF – Don’t hang up on me!

```
Sent:       1 bytes
            {
                  U8 Cmd;           // 0xFF
            }

Responds:   1 byte
            {
                  U8 Result;        // should be 0xFF
            }

Notes:      This command merely gives you a way to thwart the Expert
            Mode watchdog timer. In general, you should avoid even
            needing it. But, here it is, just in case…
```

ISO

The general flow with ISO Expert Mode (‘e’<3>), is:

1. Select a protocol (ISO 9141 or KWP 2000)
2. Setup filters for the messages you are interested in
3. Output some bytes and (optionally) wait for a designated number of response bytes

The export mode commands to accomplish these tasks are:

Command ‘R’ – Set the Protocol Rate

```
Sent:       2 bytes
            {
                  U8 Cmd;           // ‘R’
                  U8 Rate;          // 1=ISO 9141
                                    // 2=KWP 2000
            }

Responds:   1 byte
            {
                  U8 Result;        // Should match ‘Rate’ requested
            }

Notes:      Default is ISO 9141.
```

Command ‘r’ – Get the Protocol Rate

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘r’
            }

Responds:   1 byte
            {
                  U8 Result;        // Current rate
                                    // 1=ISO 9141
                                    // 2=KWP 2000
            }

Notes:      The default rate when you enter this Expert Mode is ISO
            9141.
```

Command ‘F’ – Add a Pass Filter

```
Sent:       7 bytes
            {
                  U8 Cmd;           // ‘F’
                  U8 Header;        // Header to pass on
                  U8 Destination;   // Destination address to pass on
                  U8 Source;        // Source to pass on
                  U8 HMask;         // Header AND mask (0xFF = exact)
                  U8 DMask;         // Destination AND mask
                  U8 SMask;         // Source AND mask
            }

Responds:   1 byte
            {
                  U8 Result;        // Number of filters in use
            }
```

Notes: These filters are just that, filters. Messages that match them are passed on, as received, and applied to the ‘replies’ count in O and I commands (see below).

Command ‘f’ – Clear all Pass Filters

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘f’
            }

Responds:   1 byte
            {
                  U8 Result;        // Should be 0
            }
```

Notes: Clear all pass filters. New filters are added with ‘F’ (above).

Command ‘O’ – Output Data and (optionally) look for input

```
Sent:       Variable
            {
                  U8 Cmd;           // ‘O’
                  U8 Timeout;       // Timeout (in mS) for responses
                                    // (P2)
                  U8 Replies;       // Expected number of replies
                  U8 DataLen;       // Expected Data length of replies
                                    // 1-7 (0 will be treated as 7)
                  U8 ByteDelay;     // Time to delay (in mS) between
                                    // each byte sent
                  U8 Len;           // Data length (32 Max)
                                    // If len & 0x80, a checksum is
                                    // added for you
                  U8 Data[Len];     // Data Bytes to Send
            }

Responds:   Variable (1 byte minimum)
            {
                  // Optional
                  {
                        U8 Len;     // Data length of data received
                                    // Bit 7 set = checksum failure
                        U8 Data[Len];     // Received Data
                  }
                  U8 End;           // Always 0xFF
            }
```

Notes: If replies is set to 0, the message will be sent, but the 0xFF will immediately be returned, regardless of ‘Timeout’.

Timeout is basically “P2”. If replies is set, the 0xFF will be returned after either a) the desired number of bytes is received or b) this time is reached without a received packet.

Timeout cannot be set to 0, doing do will default to a timeout of 150 mS.

DataLen is the expected number of J1979 data bytes in responses (1-7). You really want to set this. If you do not, then the unit has to wait for timers to expire, because ISO protocols do not nec. have packet length fields. If you set it to 0, then, depending on the protocol, you may have to wait for full timeout per response (slooow).

ByteDelay can be set to zero, but the ISO 9141 and KWP 2000 protocols require 5 (we’ve found that 6 helps with older Japanese vehicles). This is the transmit spacing for bytes.

Command 'I' – Look for Input Data

```
Sent:       3 bytes
            {
                  U8 Cmd;           // 'I'
                  U8 Timeout;       // Timeout (in mS) for responses
                  U8 DataLen;       // Expected data bytes in replies
                                    // 1-7, (0 will be treated as 7)
                  U8 Replies;       // Expected number of replies
            }

Responds:   Variable (1 byte minimum)
            {
                  // Optional
                  {
                        U8 Len;     // Data length of packet
                                    // Bit 7 set indicates csum failure
                        U8 Data[Len];     // Received Data
                  }
                  U8 End;           // Always 0xFF
            }
```

Notes: Timeout is basically P2. If replies is set, the 0xFF will be returned after either a) the desired number of bytes is received or b) this time is reached without a received byte.

Timeout cannot be set to 0, doing do will default to a timeout of 150 mS.

DataLen is the expected number of J1979 data bytes in responses (1-7). You really want to set this. If you do not, then the unit has to wait for timers to expire, because ISO protocols do not nec. have packet length fields. If you set it to 0, then, depending on the protocol, you may have to wait for full timeout per response (slooow).

ByteDelay can be set to zero, but the ISO 9141 and KWP 2000 protocols require 5 (we've found that 6 helps with older Japanese vehicles).

Command ‘5’ – Perform standard 5 Baud Initialization

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘5’
            }

Responds:   1 byte
            {
                  U8 Result;        // 0=failed
                                    // 1=ISO 9141 found
                                    // 2=KWP 2000 found
            }
```

Notes: This sends a 5 baud initialization, and then handles ‘Key Word’ exchanges to determine protocol. It must be used with care because, prior to entering this expert mode, the initialization may have already occurred. Doing this twice, too close together, can really confuse some ECUs.

Command ‘q’ – Perform standard ‘fast’ KWP-2000 initialization

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘f’
            }

Responds:   1 byte
            {
                  U8 Result;        // 0=failed
                                    // 1=KWP 2000 found
            }
```

Notes: This performs a pulse and initialization exchange to start KWP 2000. It must be used with care because, prior to entering this expert mode, the initialization may have already occurred. Doing this twice, too close together, can really confuse some ECUs.

Command ‘L’ – Set the Vehicle LED State

```
Sent:       2 bytes
            {
                  U8 Cmd;           // ‘L’
                  U8 State;         // 0 = off
                                    // 1 = on
            }

Responds:   1 byte
            {
                  U8 Result;        // Should match ‘State’ requested
            }
```

Notes: Because we have conditioned the user to expect the vehicle light to light or blink, you might want to follow the same model. You may also find this useful for debugging.

Command ‘b’ – Get Last MTS Packet passed

```
Sent:       1 bytes
            {
                  U8 Cmd;           // ‘b’
            }

Responds:   Variable (2 bytes minimum)
            {
                  U8 Len;           // Length of data to follow
                  U8 Sequence;      // Sequence Counter, incremented
                                    // each time an MTS packet is
                                    // passed
                  // Optional
                  {
                        U8 Packet[Len-1]; // MTS Packet (if any)
                  }
            }
```

Notes: If there is no MTS hardware attacked, you will get ‘0x01 0x00’ (1 byte, sequence 0). Otherwise, you will get a changing sequence, followed by a raw MTS packet.

Command ‘s’ – Exit Expert Mode

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘s’
            }

Responds:   None
```

Notes: This command returns you to Setup Mode.

Command 0xFF – Don't hang up on me!

```
Sent:       1 bytes
            {
                  U8 Cmd;           // 0xFF
            }

Responds:   1 byte
            {
                  U8 Result;        // should be 0xFF
            }

Notes:      This command merely gives you a way to thwart the Expert
            Mode watchdog timer. In general, you should avoid even
            needing it. But, here it is, just in case…
```

ISO Raw

The general flow with ISO Raw Expert Mode (‘e’<4>), is:

1. Select a baud rate
2. Output bytes and optionally receive bytes

ISO Raw mode is different in that there is no understanding of packets or filters. It is intended for cases where less conventional ISO 9141/KWP 2000 packets/communication are being used or for alternate vehicle protocols, like MUT and SSM. The export mode commands to accomplish these tasks are:

Command ‘R’ – Set the Protocol Rate

```
Sent:       2 bytes
            {
                  U8 Cmd;           // 'R'
                  U8 Rate;          // 1=19200
                                    // 2=15625
                                    // 3=10400
                                    // 4=9600
                                    // 5=4800
            }

Responds:   1 byte
            {
                  U8 Result;        // Should match 'Rate' requested
            }

Notes:      ISO has a default rate of 10400 (3), which is correct for
            ISO 9141 and KWP2000.
```

Command ‘r’ – Get the Protocol Rate

```
Sent:       1 byte
            {
                  U8 Cmd;           // 'r'
            }

Responds:   1 byte
            {
                  U8 Result;        // Current rate
                                    // 1=19200
                                    // 2=15625
                                    // 3=10400
                                    // 4=9600
                                    // 5=4800
            }

Notes:      The default rate when you enter this Expert Mode is 10400.
```

Command ‘O’ – Output Data and (optionally) look for input

```
Sent:       Variable
            {
                  U8 Cmd;           // ‘O’
                  U8 Timeout;       // Timeout (in mS) for responses
                  U8 Replies;       // Expected number of replies
                  U8 ByteDelay;     // Time to delay (in mS) between
                                    // each byte sent
                  U8 Len;           // Data length (32 Max)
                  U8 Data[Len];     // Data Bytes to Send
            }

Responds:   Variable (1 byte minimum)
            {
                  // Optional
                  {
                        U8 Len;     // Data length of data received
                        U8 Data[Len];     // Received Data
                  }
                  U8 End;           // Always 0xFF
            }
```

Notes: Replies is for BYTES, not packets. This expert mode has no understanding of packets, it leaves that entirely up to you. Unlike J1850, which will calculate CRC for you, checksum bytes, etc., are entirely your responsibility (CRC is a pain, but we figured you can add!)

If replies is set to 0, the message will be sent, but the 0xFF will immediately be returned, regardless of ‘Timeout’.

Timeout is per byte received. If replies is set, the 0xFF will be returned after either a) the desired number of bytes is received or b) this time is reached without a received byte.

Timeout cannot be set to 0, doing do will default to a timeout of 150 mS.

ByteDelay can be set to zero, but the ISO 9141 and KWP 2000 protocols require 5 (we’ve found that 6 helps with older Japanese vehicles).

Command 'I' – Look for Input Data

```
Sent:       3 bytes
            {
                  U8 Cmd;           // 'I'
                  U8 Timeout;       // Timeout (in mS) for responses
                  U8 Replies;       // Expected number of replies
            }

Responds:   Variable (1 byte minimum)
            {
                  // Optional
                  {
                        U8 Len;     // Data length of packet
                        U8 Data[Len];     // Received Data
                  }
                  U8 End;           // Always 0xFF
            }
```

Notes: Replies is for BYTES, not packets. This expert mode has no understanding of packets, it leaves that entirely up to you. Unlike J1850, which will calculate CRC for you, checksum bytes, etc., are entirely your responsibility (CRC is a pain, but we figured you can add!)

Timeout is per byte received. If replies is set, the 0xFF will be returned after either a) the desired number of bytes is received or b) this time is reached without a received byte.

Timeout cannot be set to 0, doing do will default to a timeout of 150 mS.

ByteDelay can be set to zero, but the ISO 9141 and KWP 2000 protocols require 5 (we've found that 6 helps with older Japanese vehicles).

Command ‘5’ – Perform standard 5 Baud Initialization

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘5’
            }

Responds:   1 byte
            {
                  U8 Result;        // 0=failed
                                    // 1=ISO 9141 found
                                    // 2=KWP 2000 found
            }
```

Notes: This sends a 5 baud initialization, and then handles ‘Key Word’ exchanges to determine protocol. It must be used with care because, prior to entering this expert mode, the initialization may have already occurred. Doing this twice, too close together, can really confuse some ECUs.

Command ‘q’ – Perform standard ‘fast’ KWP-2000 initialization

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘f’
            }

Responds:   1 byte
            {
                  U8 Result;        // 0=failed
                                    // 1=KWP 2000 found
            }
```

Notes: This performs a pulse and initialization exchange to start KWP 2000. It must be used with care because, prior to entering this expert mode, the initialization may have already occurred. Doing this twice, too close together, can really confuse some ECUs.

Command ‘0’ – Send a 00h at 5 baud on the K line

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘0’
            }

Responds:   1 byte
            {
                  U8 Result;        // always 1
            }
```

Notes: The description says it all. This command, combined with the weird 15625 baud rate is nec. to communicate with an ECU via MUT.

Command ‘L’ – Set the Vehicle LED State

```
Sent:       2 bytes
            {
                  U8 Cmd;           // ‘L’
                  U8 State;         // 0 = off
                                    // 1 = on
            }

Responds:   1 byte
            {
                  U8 Result;        // Should match ‘State’ requested
            }
```

Notes: Because we have conditioned the user to expect the vehicle light to light or blink, you might want to follow the same model. You may also find this useful for debugging.

Command ‘b’ – Get Last MTS Packet passed

```
Sent:       1 bytes
            {
                  U8 Cmd;           // ‘b’
            }

Responds:   Variable (2 bytes minimum)
            {
                  U8 Len;           // Length of data to follow
                  U8 Sequence;      // Sequence Counter, incremented
                                    // each time an MTS packet is
                                    // passed
                  // Optional
                  {
                        U8 Packet[Len-1]; // MTS Packet (if any)
                  }
            }

            Notes:      If there is no MTS hardware attacked, you will
            get ‘0x01 0x00’ (1 byte, sequence 0). Otherwise, you will
            get a changing sequence, followed by a raw MTS packet.
```

Command ‘s’ – Exit Expert Mode

```
Sent:       1 byte
            {
                  U8 Cmd;           // ‘s’
            }

Responds:   None

Notes:      This command returns you to Setup Mode.
```

Command 0xFF – Don’t hang up on me!

```
Sent:       1 bytes
            {
                  U8 Cmd;           // 0xFF
            }

Responds:   1 byte
            {
                  U8 Result;        // should be 0xFF
            }

Notes:      This command merely gives you a way to thwart the Expert
            Mode watchdog timer. In general, you should avoid even
            needing it. But, here it is, just in case…
```

But How Do I Connect?!

If you have reached this point, you now have a wealth of information. You know how to read MTS packets, configure the OT-1b/2, and even go mucking directly with an ECU. But you are missing a critical piece of information, getting connected to the OT-1b/2 in the first place!

This is not an oversight, all of the information that you have learned to this point is 'transport independent'. That is, it applies regardless of how you connect to the unit. But, to get anything done, you will need to connect!

Note: The information that follows is not really specific to the OT-1b/2. It can be used with other devices, like the LM-2, and will be applicable with future devices.

Innovate Transports

MTS is now available over three types of connections, RS-232, USB (IMS USB), and network (IMS NET). However, on the OT-1b/2, the RS-232 'Serial Out' connector is limited to 'in band' MTS communication only. That is, data packets flow out of it, and it accepts in-band commands and queries, but it will not accept commands to go into Setup or Expert modes.

Since this SDK is primarily focused on using those modes, we will limit our discussion to the other two protocols.

IMS USB

It is beyond the scope of this document to fully cover USB. However a good primer (from an embedded systems engineer) can be found here:

http://www.computer-solutions.co.uk/info/Embedded_tutorials/usb_tutorial.htm

A reasonable question would be why does IMS USB exist at all? After all, a common solution to this is to use a USB->Serial chip. Connect to it through your existing serial interface, and have it appear on the computer as a serial port, which all the serial port aware software can use as before.

Actually, we tried this, with the original OT-1. Our hope was that this would make device drivers for all platforms, etc., someone else's problem (we are big fans of delegating grief). However, although we used a very popular and reputable USB/Serial chip, it was a serious support headache.

Precisely because the approach is popular, it is common to have other applications that are installing the same, or similar drivers, but not always correctly. Another reoccurring problem is that Windows sometimes sees the MTS in band serial stream as a 'Microsoft

Tablet', via serial plug-and-play. This requires fiddling with the Windows device manager to rectify. Perhaps one of the most frustrating problems is that Windows can accumulate vast numbers of non existent serial ports and, although there is no obvious way to delete them, having COM ports over 256 can cause unpredictable behavior when the ports are later opened.

On top of everything else, there was a question of performance. Serial ports have extra driver overhead on the Windows side, and this overall approach still limits the device to serial port speeds at the hardware end. This difference is glaring in something more data intensive, like firmware update. The original OT-1 takes about 10 times longer to perform a firmware update over USB than an LM-2, even though the later has about 6 times as much flash memory to fill.

These support problems, combined with the nominal performance, are why, starting with the LM-2, we went to native USB support on the device and our own USB drivers for Windows. Based on the dramatically lower USB related support incidents with that produce, we elected to update the OT-1 (OT-1b), and the new OT-2 to all utilize native USB, and our common USB driver, as well.

Technical Details

- The OT-1b, OT-2, and LM-2 are all native, USB 1.1, full speed devices
- Each product reports a unique product ID via USB
- But all use an identical USB interface, namely, the MTS protocol over one 'bulk endpoint'
- We currently only provide device drivers for the Windows platforms
- All Innovate devices use the same USB Driver
- The driver is used to access the bulk pipe via 'Device IO Control' calls

IMS USB under Windows

The recommended method for accessing Innovate hardware via our USB driver under Windows is to utilize a dynamic link library called "imsusb.dll", included with the source samples of this SDK.

To use the library from C/C++ you will need to:

- Include the header file imsusb.h in your source
- Link the library file imsusb.lib into your project

Note: The DLL entry points are declared "extern C" and use the standard Windows API call stacks, so it should be accessible from most other programming languages for the platform. However, Visual C++ is the only development environment that we have tested.

The interface is byte oriented, and only involves a few calls:

Discovery

```
BOOL imsusbCanConnect(BOOL *inuse); // OUT: indicates if the reason for
                                    // failure is that the device is in
                                    // use
```

Check if a device is available.
Returns: TRUE if a USB device can be connected to, FALSE if not. If the result is FALSE, inuse can be used to determine if there is no device present, or if a device is present but currently in use.

Open/Close

```
BOOL imsusbConnect();
```

Connect to the IMS USB Device.
Returns: TRUE if successful, FALSE on failure.

```
void imsusbClose();
```

Release the previously connected IMS USB device.

Receive Data

```
BOOL imsusbIsByte();
```

Check if a byte is ready to be received.
Returns: TRUE if a data byte is ready, FALSE if no data is ready.

```
BYTE imsusbGetByte();
```

Read a byte from device.
Returns: byte read from USB, will block until a byte is received. If USB terminates abnormally, returns 0.

```
int imsusbGetBytes(int len,         // Number of bytes to read
                   BYTE *data)      // OUT: buffer for read data
```

Read multiple bytes from USB.
Returns: Number of bytes read or -1 for error.

Send Data

```
BOOL imsusbPutByte(BYTE b);
```

Send a byte to the device.
Returns: TRUE on success, FALSE on failure

```
int imsusbPutBytes(int len,          // Number of bytes to write
                   BYTE *data)       // Data to write
```

Write bytes to the device.
Returns: Number of bytes written or -1 on error.

Note: The reason that we recommend using the DLL over direct driver access is that the DLL automatically handles usage conflicts and abnormal application terminations.

Other Platforms

We are actively looking at supporting other platforms. In addition, we are happy to provide additional technical information to support third party development of compatible drivers. Contact support@innovatemotorsports.com for more information.

IMS NET (Wi-Fi)

This section assumes some familiarity with network programming and concepts. It is beyond the scope of this document to cover this subject, but there are many, many, many tutorials on the web. I am reluctant to link to any particular one because each platform/development environment has its own peculiarities.

The OT-2 is the first Innovate product to natively support networking (via Wi-Fi). The rest of this section explains some concepts, like discovery, that are intended for long term support of future Innovate network based products and possible future changes to the existing OT-2. Of course, we would prefer that you conform to these practices, but we also understand that sometimes you need to get things working. So, if you are familiar with network programming and are willing to deal with conflicts as they arise down the road, here are the basic details for OT-2:

Wi-Fi: 802.11b/g

Provides Ad Hoc Network:
- Channel 6
- SSID “INNOVATE_xxxx” (xxxx is unique per unit)

IP Address: 10.3.2.1
- Expects MTS connection via TCP on port 0xC001 (49153)

IP Mask: 255.255.255.0

DHCP server: Provided (rudimentary)

So no host network configuration should normally be necessary, aside from connecting to the ad hoc network above

OK, now that the down and dirty basics are out of the way, it is time to make a pitch for a little added complexity. Instead of hard coding for the above IP address/port, we recommend using a 'Discovery Method' to find the unit and connect to it.

Discovery basically consists of:

1. Create a UDP (user datagram) socket
2. Set it with broadcast options
3. Broadcast a discovery packet
4. Listen for a response
5. Open a TCP connection using the IP address and port provided in the discovery response

There are two advantages to this. First, it allows us to change the IP address and port used without breaking your software. Second, although the unit will only accept one TCP connection for MTS data at a time, it will always respond to discovery inquiries. If it is currently being used, it reports that in the response. This lets you distinguish between failures because of usage conflict and literal 'device not there' problems.

For discovery, we use port 0x1936, which is the 'well known port' for a protocol called ArtNet. The discovery packet and response is structured so as not to confuse any actual ArtNet devices which might be present:

```
typedef unsigned char U8;
typedef unsigned short U16;
typedef unsigned long U32;

// Protocol ID
#define IMSNET_PROTO_ID "IMS Net"

// Interface Version
#define IMSNET_VERSION (1)

// UDP Discovery Port
#define IMSNET_DISCOVERY (0x1936)

// OpCodes
#define IMSNET_OPCODE_POLL (0x4000)
#define IMSNET_OPCODE_POLLREPLY (0x4100)

// Flags
#define IMSNET_FLAG_INUSE (0x1)

// Poll Structures
#pragma pack(push)
#pragma pack(1)         // must be byte aligned
```

```
// All in BIG Endian !!!!
typedef struct {
      U8 ProtoID[8];    // protocol ID = "IMS Net"
      U16 OpCode;       // == IMSNET_OPCODE_POLL
      U8 VersionH;      // 0
      U8 VersionL;      // protocol version, set to IMSNET_VERSION
} IMSNET_POLL;          // HOST BROADCAST to find unit

typedef struct {
      U8 ProtoID[8];    // protocol ID = "IMS Net"
      U16 OpCode;       // == IMSNET_OPCODE_POLLREPLY
      U8 VersionH;      // 0
      U8 VersionL;      // protocol version, set to IMSNET_VERSION
      U32 Address;      // IP Address
      U16 Port;         // Port for service
      U16 Flags;        // See Flags above
      U32 Info;         // When IMSNET_FLAG_INUSE, IP Address of user
} IMSNET_POLLREPLY;     // DEVICE REPLY to discovery broadcast

#pragma pack(pop)
```

See the sample projects, which all use discovery, for further information.

Special Considerations for Network/Wi-Fi

Although, in general, network programming with the OT-2 is straightforward, there are some special considerations worth mentioning. Given the modest rate and volume of MTS data, and the high bit rate of Wi-Fi in comparison, data throughput would not normally be thought of as a potential problem. However, because of the way that TCP works on many systems, it is.

Performance

The primary issue is that network stacks generally make some effort to minimize network traffic. They often do this in a couple of ways that directly conflict with typical operation of the OT-2. First, systems often let small packets accumulate before actually sending anything. The method used is often some variant of something called the "Nagle Algorithm".

This works well at minimizing the number of small payload packets on the network, but it adds unnecessary delays to the delivery of small commands, like those used in the different modes of the OT-2. In other words, by default, you might send a 'j' command to get connection status, but your computer's network stack may decide to wait 200 mS before actually sending it, on the chance that you might have more information to send, which could all be put into one, larger packet.

Fortunately, this sort of buffering can normally be turned off. Search your operating system reference for TCP_NODELAY. It is usually a "socket option" that is applied at the "protocol level". With the Nagle algorithm turned off, bytes are transmitted right away.

However, another source of delay may remain. Again, in order to avoid small packets, many network stacks (including Windows') use something called "Delayed Ack". In a TCP packet, both data and handshaking information can be included. When the Windows stack receives data it typically waits a brief interval before acknowledging the receipt of the data. It does this in case the Windows side user has something to send in response. If it does, the stack knows it could combine the data and the acknowledgement in one packet, hence saving network bandwidth.

The problem with this is the sending side must wait for the acknowledgement before sending more data. Because of delayed acknowledgement, data ends up coming in larger chunks from the device side. From a network traffic point of view, this is a good thing. But if you want to update your onscreen gauges, etc. quickly, then you do not want MTS data packets, which are only about 12 Hz to begin with, coming in 2-3 at a time. This slows your UI update to roughly ¼ of a second.

No data is lost, so this is not a problem for logging, but it can diminish the user experience. There is not generally an easy way to turn this behavior off in the stack, but

you can circumvent it another way. In our own Windows software, we send the bogus MTS Query 0xFF each time we receive an in band packet. These queries are discarded by the OT-2, but permit our acknowledgement to go out briskly and data to flow at roughly packet rate.

Debugging

Debugging Wi-Fi traffic, particularly between the OT-2 and something like a smart phone, can sometimes be a problem. There are many free applications that monitor wired network traffic, but some of the more popular ones, like Wireshark (http://www.wireshark.org/) do not generally monitor Wi-Fi traffic without additional external hardware.

Microsoft offers a free Network Monitor, version 3.3, which can be downloaded here:

http://www.microsoft.com/downloads/details.aspx?familyid=983B941D-06CB-4658-B7F6-3088333D062F&displaylang=en

Network Monitor will capture Wi-Fi traffic, *provided* that the computer running it has a Wi-Fi chipset that the program can put in promiscuous mode.

If you do not have access to suitable hardware, one technique that worked well here has been to rely on simulators/emulators for handheld devices for network debugging. Generally, programs like Wireshark (available for Mac and PC) *will* capture wireless network traffic to and from the specific machine the network monitor is running on. By running on a device simulator on a desktop machine, you can get access to network capture even when no suitable Wi-Fi capture hardware is available.

Samples

They say that a line of code is worth a thousand words. I suppose that depends on the code. However, it is generally helpful to have working examples. With that in mind we have prepared the following:

Sample Name	Platform(s)/Language	Description
OTConsole	Windows/C	This is a command line application that demonstrates basic functionality of all SDK features, including expert modes. Currently, this is the only sample that supports both IMS NET and IMS USB
O-JUCE	Windows, OSX, Linux/C++	This is a simple graphical application that is based on the JUCE multi-platform application framework from Raw Material Software. Although it does not utilize more advanced SDK features, it does compile and run on all platforms. Currently, it is IMS NET only.
OTMobile	iPhone/Objective C	This is similar to O-JUCE, but written natively for iPhone. It is IMS NET only, and will remain so as long as the iPhone has no USB port.

All samples can be built with freely available tools. For Windows, we used Visual C++ Express (2008, not the beta of 2010), which is available here:

http://www.microsoft.com/express/vc/

For Mac OS-X and iPhone we used X-Code and the iPhone SDK, which can be downloaded here:

http://developer.apple.com/iphone/

For Linux we used Ubuntu, which can be downloaded here:

http://www.ubuntu.com

Further information on JUCE can be found here:

http://www.rawmaterialsoftware.com/juce.php

For the latest information on building and samples, see the document “buildme.txt” in the root of the samples folder.

Appendix A: Normalized PIDs

This table gives names, descriptions, units, and ranges to scale MTS samples (0-1023) to. For the ECU mode 1 PID associated with each normalized PID, see the table after this one.

```
typedef struct {
        char    name[26];
        char description[32];
        char  units[12];
        double min;
        double max;
} _NORM_PID;

static _NORM_PID NormPids[] = {
//      Short Name      Description                     Units           Min     Max
        "OBD_None",     "None",                         "Volts",        0.0,    5.0,
        "OBD_RPM",      "Engine RPM",                   "RPM",          0.0,    10230.0,
        "OBD_TP",       "Throttle Position(abs)",       "%",            0.0,    100.0,
        "OBD_LOAD_PCT","Engine Load(calc)",             "%",            0.0,    100.0,
        "OBD_SPARKADV","Timing Advance(cyl1)",          "degBTDC",      -64.0,  63.5,
        "OBD_MAF",      "Mass Air Flow",                "g/s",          0.0,    655.35,
        "OBD_MAP",      "Manifold Abs. Presure",        "kPa",          0.0,    255.0,
        "OBD_VSS",      "Vehicle Speed Sensor",         "km/h",         0.0,    255.0,
        "OBD_ECT",      "Engine Coolant Temp",          "degC",         -40.0,  215.0,
        "OBD_IAT",      "Intake Air Temp",              "degC",         -40.0,  215.0,
        "OBD_PTO_STAT","PTO Status",                    "PTO",          0.0,    1.0,
        "OBD_FUEL1_OL","Fuel Sys1 Open Loop",           "OL",           0.0,    1.0,
        "OBD_FUEL2_OL","Fuel Sys2 Open Loop",           "OL",           0.0,    1.0,
        "OBD_SHRTFT1", "Short Term Fuel Trim 1",        "%",            -100.0, 99.22,
        "OBD_LONGFT1", "Long Term Fuel Trim 1",         "%",            -100.0, 99.22,
        "OBD_SHRTFT2", "Short Term Fuel Trim 2",        "%",            -100.0, 99.22,
        "OBD_LONGFT2", "Long Term Fuel Trim 2",         "%",            -100.0, 99.22,
        "OBD_SHRTFT3", "Short Term Fuel Trim 3",        "%",            -100.0, 99.22,
        "OBD_LONGFT3", "Long Term Fuel Trim 3",         "%",            -100.0, 99.22,
        "OBD_SHRTFT4", "Short Term Fuel Trim 4",        "%",            -100.0, 99.22,
        "OBD_LONGFT4", "Long Term Fuel Trim 4",         "%",            -100.0, 99.22,
        "OBD_FRP",      "Fuel Rail Pressure",           "kPa",          0.0,    765.0,
        "OBD_FRP_MED", "Fuel Rail Pressure",            "kPa",          0.0,    5177.27,
        "OBD_FRP_HIGH","Fuel Rail Pressure",            "kPa",          0.0,    655350.0,
        "OBD_EQ_RAT",  "Commanded Equiv. Ratio",        "lambda",       0.0,    1.999,
        "OBD_LOAD_ABS","Absolute Load Value",           "%",            0.0,    802.75,
        "OBD_EGR_PCT", "Commanded EGR",                 "%",            0.0,    100.0,
        "OBD_EGR_ERR", "EGR Error",                     "%",            -100.0, 99.22,
        "OBD_TP_R",     "Throttle Position(rel)",       "%",            0.0,    100.0,
        "OBD_TP_B",     "Throttle Position B(abs)",     "%",            0.0,    100.0,
        "OBD_TP_C",     "Throttle Position C(abs)",     "%",            0.0,    100.0,
        "OBD_APP_D",    "Acc. Pedal Position D",        "%",            0.0,    100.0,
        "OBD_APP_E",    "Acc. Pedal Position D",        "%",            0.0,    100.0,
        "OBD_APP_F",    "Acc. Pedal Position D",        "%",            0.0,    100.0,
        "OBD_TAC_PCT", "Commanded Throttle",            "%",            0.0,    100.0,
        "OBD_EVAP_PCT","Commanded Evap. Purge",         "%",            0.0,    100.0,
        "OBD_EVAP_VP", "Evap. Vapor Pressure",          "Pa",           -8192.0, 8191.0,
        "OBD_AIR_UPS", "Secondary Air DNS",             "UPS",          0.0,    1.0,
        "OBD_AIR_DNS", "Secondary Air DNS",             "DNS",          0.0,    1.0,
        "OBD_AIR_OFF", "Secondary Air DNS",             "OFF",          0.0,    1.0,
        "OBD_FLI",      "Fuel Level Indicator",         "%",            0.0,    100.0,
        "OBD_BARO",     "Barometric Pressure",          "kPa",          0.0,    255.0,
        "OBD_AAT",      "Ambient Air Temp",             "degC",         -40.0,  215.0,
        "OBD_VPWR",     "Control Module Voltage",       "Volts",        0.0,    65.535,
        "OBD_MIL",      "Malfunction Indicator Lamp",   "MIL",          0.0,    1.0,
        "OBD_DTC_CNT", "DTC Count",                     "DTCs",         0.0,    1023.0,
        "OBD_MIL_DIST","Distance MIL active",           "km",           0.0,    65535.0,
        "OBD_MIL_TIME","Hours MIL active",              "hours",        0.0,    1023.0,
        "OBD_CLR_DIST","Distance MIL clear",            "km",           0.0,    65535.0,
        "OBD_WARM_UPS","Warm Ups MIL clear",            "WUs",          0.0,    1023.0,
        "OBD_RUNTM",    "Run Time",                     "mins",         0.0,    1023.0,
```

```
        "OBD_O2S11",    "O2 Sensor(NB) 1-1",          "Volts",       0.0,    1.275,
        "OBD_SHRTFT11","O2 Fuel Trim 1-1",            "%",           -100.0, 99.22,
        "OBD_O2S12",    "O2 Sensor(NB) 1-2",          "Volts",       0.0,    1.275,
        "OBD_SHRTFT12","O2 Fuel Trim 1-2",            "%",           -100.0, 99.22,
        "OBD_O2S21",    "O2 Sensor(NB) 2-1",          "Volts",       0.0,    1.275,
        "OBD_SHRTFT21","O2 Fuel Trim 2-1",            "%",           -100.0, 99.22,
        "OBD_O2S22",    "O2 Sensor(NB) 2-2",          "Volts",       0.0,    1.275,
        "OBD_SHRTFT22","O2 Fuel Trim 2-2",            "%",           -100.0, 99.22,
        "OBD_O2S31",    "O2 Sensor(NB) 3-1",          "Volts",       0.0,    1.275,
        "OBD_SHRTFT31","O2 Fuel Trim 3-1",            "%",           -100.0, 99.22,
        "OBD_O2S32",    "O2 Sensor(NB) 3-2",          "Volts",       0.0,    1.275,
        "OBD_SHRTFT32","O2 Fuel Trim 3-2",            "%",           -100.0, 99.22,
        "OBD_O2S41",    "O2 Sensor(NB) 4-1",          "Volts",       0.0,    1.275,
        "OBD_SHRTFT41","O2 Fuel Trim 4-1",            "%",           -100.0, 99.22,
        "OBD_O2S42",    "O2 Sensor(NB) 4-2",          "Volts",       0.0,    1.275,
        "OBD_SHRTFT42","O2 Fuel Trim 4-2",            "%",           -100.0, 99.22,
        "OBD_EQ_RAT11","WideO2 Equiv-Ratio 1-1",      "lambda",      0.0,    1.999,
        "OBD_WO2S11",  "WideO2 Voltage 1-1",          "Volts",       0.0,    7.999,
        "OBD_EQ_RAT12","WideO2 Equiv-Ratio 1-2",      "lambda",      0.0,    1.999,
        "OBD_WO2S12",  "WideO2 Voltage 1-2",          "Volts",       0.0,    7.999,
        "OBD_EQ_RAT21","WideO2 Equiv-Ratio 2-1",      "lambda",      0.0,    1.999,
        "OBD_WO2S21",  "WideO2 Voltage 2-1",          "Volts",       0.0,    7.999,
        "OBD_EQ_RAT22","WideO2 Equiv-Ratio 2-2",      "lambda",      0.0,    1.999,
        "OBD_WO2S22",  "WideO2 Voltage 2-2",          "Volts",       0.0,    7.999,
        "OBD_EQ_RAT31","WideO2 Equiv-Ratio 3-1",      "lambda",      0.0,    1.999,
        "OBD_WO2S31",  "WideO2 Voltage 3-1",          "Volts",       0.0,    7.999,
        "OBD_EQ_RAT32","WideO2 Equiv-Ratio 3-2",      "lambda",      0.0,    1.999,
        "OBD_WO2S32",  "WideO2 Voltage 3-2",          "Volts",       0.0,    7.999,
        "OBD_EQ_RAT41","WideO2 Equiv-Ratio 4-1",      "lambda",      0.0,    1.999,
        "OBD_WO2S41",  "WideO2 Voltage 4-1",          "Volts",       0.0,    7.999,
        "OBD_EQ_RAT42","WideO2 Equiv-Ratio 4-2",      "lambda",      0.0,    1.999,
        "OBD_WO2S42",  "WideO2 Voltage 4-2",          "Volts",       0.0,    7.999,
        "OBD_WBEQ_RAT11","WB-O2 Equiv-Ratio 1-1",     "lambda",      0.0,    1.999,
        "OBD_WBO2S11", "WB-O2 Voltage 1-1",           "mA",          -128.0, 127.996,
        "OBD_WBEQ_RAT12","WB-O2 Equiv-Ratio 1-2",     "lambda",      0.0,    1.999,
        "OBD_WBO2S12", "WB-O2 Voltage 1-2",           "mA",          -128.0, 127.996,
        "OBD_WBEQ_RAT21","WB-O2 Equiv-Ratio 2-1",     "lambda",      0.0,    1.999,
        "OBD_WBO2S21", "WB-O2 Voltage 2-1",           "mA",          -128.0, 127.996,
        "OBD_WBEQ_RAT22","WB-O2 Equiv-Ratio 2-2",     "lambda",      0.0,    1.999,
        "OBD_WBO2S22", "WB-O2 Voltage 2-2",           "mA",          -128.0, 127.996,
        "OBD_WBEQ_RAT31","WB-O2 Equiv-Ratio 3-1",     "lambda",      0.0,    1.999,
        "OBD_WBO2S31", "WB-O2 Voltage 3-1",           "mA",          -128.0, 127.996,
        "OBD_WBEQ_RAT32","WB-O2 Equiv-Ratio 3-2",     "lambda",      0.0,    1.999,
        "OBD_WBO2S32", "WB-O2 Voltage 3-2",           "mA",          -128.0, 127.996,
        "OBD_WBEQ_RAT41","WB-O2 Equiv-Ratio 4-1",     "lambda",      0.0,    1.999,
        "OBD_WBO2S41", "WB-O2 Voltage 4-1",           "mA",          -128.0, 127.996,
        "OBD_WBEQ_RAT42","WB-O2 Equiv-Ratio 4-2",     "lambda",      0.0,    1.999,
        "OBD_WBO2S42", "WB-O2 Voltage 4-2",           "mA",          -128.0, 127.996,
        "OBD_CATEMP11","Catalyst Temp 1-1",           "degC",        -40.0,  6513.5,
        "OBD_CATEMP21","Catalyst Temp 2-1",           "degC",        -40.0,  6513.5,
        "OBD_CATEMP12","Catalyst Temp 1-2",           "degC",        -40.0,  6513.5,
        "OBD_CATEMP22","Catalyst Temp 2-2",           "degC",        -40.0,  6513.5,
        "OBD_RPM2",    "Engine RPM",                  "RPM",         0.0,    20460.0,
};
```

Common non-OBD IMS Inputs:

```
static _NORM_PID IMSPids[] = {
//      Short Name      Description                   Units          Min     Max
        "O2",           "Wideband Measurement",       "lambda",      0.5,    1.523,
        "RPM",          "Engine RPM",                 "RPM",         0.0,    10230.0,
        "RPM2",         "Engine RPM",                 "RPM",         0.0,    20460.0,
        "FREQ",         "Frequency",                  "Hz",          0.0,    1000.0,
        "DWELL",        "Dwell",                      "%",           0.0,    100.0,
        "EGT",          "EGT",                        "degC",        0.0,    1093.0,
        "CHT",          "CHT",                        "degC",        0.0,    300.0,
        "SIDE2",        "Side Force 2G",              "g",           -2.0,   2.0,
        "SIDE1",        "Side Force 1G",              "g",           -1.0,   1.0,
        "SIDE25",       "Side Force .25G",            "g",           -0.25,  0.25,
```

```
        "TIMING",       "Ignition Timing",          "deg",          -10.0,  50.0,
        "MAP3BA",       "MAP 3Ba",                  "PSIa",         0.0,    44.1,
        "MAP1BA",       "MAP 1Ba",                  "PSIa",         0.0,    14.7,
        "MAP3BG",       "MAP 3Bg",                  "PSIg",         -14.7,  29.4,
        "MAP1BG",       "MAP 1Bg",                  "PSIg",         -14.7,  0.0,
        "ACC2",         "Acceleration 2G",          "g",            -2.0,   2.0,
        "ACC1",         "Acceleration 1G",          "g",            -1.0,   1.0,
        "ACC25",        "Acceleration .25G",        "g",            -0.25,  0.25,
        "AUX",          "Aux. Input Volts",         "Volt",         0.0,    5.0,
        "AUXP",         "Aux. Input Percentage",    "%",            0.0,    100.0,
};
```

Mapping normalized PIDs to ECU mode one PIDs:

```
typedef struct {
        unsigned char ecuPid;
        char description[32];
} _ECU_PID;

static _ECU_PID EcuPidMap[] = {
//      PID     Description
        0,      "None - Inactive",
        0xC,    "RPM - Engine RPM",
        0x11,   "TP - Throttle Pos.(abs)",
        0x04,   "LOAD_PCT - Eng. Load(calc)",
        0x0E,   "SPARKADV - Timing Advance",
        0x10,   "MAF - Mass Air Flow",
        0x0B,   "MAP - Manifold Abs. Pres.",
        0x0D,   "VSS - Vehicle Speed Sensor",
        0x05,   "ECT - Engine Coolant Temp",
        0x0F,   "IAT - Intake Air Temp",
        0x1E,   "PTO_STAT - PTO Status",
        0x03,   "FUEL1_OL - Fuel1 Open Loop",
        0x03,   "FUEL2_OL - Fuel2 Open Loop",
        0x06,   "SHRTFT1 - Short Fuel Trim 1",
        0x07,   "LONGFT1 - Long Fuel Trim 1",
        0x08,   "SHRTFT2 - Short Fuel Trim 2",
        0x09,   "LONGFT2 - Long Fuel Trim 2",
        0x06,   "SHRTFT3 - Short Fuel Trim 3",
        0x07,   "LONGFT3 - Long Fuel Trim 3",
        0x08,   "SHRTFT4 - Short Fuel Trim 4",
        0x09,   "LONGFT4 - Long Fuel Trim 4",
        0x0A,   "FRP - Fuel Rail Pressure",
        0x22,   "FRP_MED - Fuel Rail Pres.",
        0x23,   "FRP_HIGH - Fuel Rail Pres.",
        0x44,   "EQ_RAT - Cmd. Equiv. Ratio",
        0x43,   "LOAD_ABS - Abs Load Value",
        0x2C,   "EGR_PCT - Cmd. EGR",
        0x2D,   "EGR_ERR - EGR Error",
        0x45,   "TP_R - Throttle Pos.(rel)",
        0x47,   "TP_B - Throttle Pos. B(abs)",
        0x48,   "TP_C - Throttle Pos. C(abs)",
        0x49,   "APP_D - Acc. Pedal Pos. D",
        0x4A,   "APP_E - Acc. Pedal Pos. D",
        0x4B,   "APP_F - Acc. Pedal Pos. D",
        0x4C,   "TAC_PCT - Cmd. Throttle",
        0x2E,   "EVAP_PCT - Cmd. Evap. Purge",
        0x32,   "EVAP_VP - Evap. Vapor Pres.",
        0x12,   "AIR_UPS - Secondary Air DNS",
        0x12,   "AIR_DNS - Secondary Air DNS",
        0x12,   "AIR_OFF - Secondary Air DNS",
        0x2F,   "FLI - Fuel Level Indicator",
        0x33,   "BARO - Barometric Pres",
        0x46,   "AAT - Ambient Air Temp",
        0x42,   "VPWR - Control Module Volts",
        0x01,   "MIL - Malfunction Ind. Lamp",
        0x01,   "DTC_CNT - DTC Count",
        0x21,   "MIL_DIST - Dist. MIL active",
        0x4D,   "MIL_TIME - Hours MIL active",
```

```
        0x31,   "CLR_DIST - Dist. MIL clear",
        0x30,   "WARM_UPS - Since MIL clear",
        0x1F,   "RUNTM - Run Time",
        0x14,   "O2S11 - O2 Sensor(NB) 1-1",
        0x14,   "SHRTFT11 - O2 Fuel Trim 1-1",
        0x15,   "O2S12 - O2 Sensor(NB) 1-2",
        0x15,   "SHRTFT12 - O2 Fuel Trim 1-2",
        0x16,   "O2S21 - O2 Sensor(NB) 2-1",
        0x16,   "SHRTFT21 - O2 Fuel Trim 2-1",
        0x17,   "O2S22 - O2 Sensor(NB) 2-2",
        0x17,   "SHRTFT22 - O2 Fuel Trim 2-2",
        0x18,   "O2S31 - O2 Sensor(NB) 3-1",
        0x18,   "SHRTFT31 - O2 Fuel Trim 3-1",
        0x19,   "O2S32 - O2 Sensor(NB) 3-2",
        0x19,   "SHRTFT32 - O2 Fuel Trim 3-2",
        0x1A,   "O2S41 - O2 Sensor(NB) 4-1",
        0x1A,   "SHRTFT41 - O2 Fuel Trim 4-1",
        0x1B,   "O2S42 - O2 Sensor(NB) 4-2",
        0x1B,   "SHRTFT42 - O2 Fuel Trim 4-2",
        0x24,   "EQ_RAT11 - WideO2 Eq-Rat 1-1",
        0x24,   "WO2S11 - WideO2 Voltage 1-1",
        0x25,   "EQ_RAT12 - WideO2 Eq-Rat 1-2",
        0x25,   "WO2S12 - WideO2 Voltage 1-2",
        0x26,   "EQ_RAT21 - WideO2 Eq-Rat 2-1",
        0x26,   "WO2S21 - WideO2 Voltage 2-1",
        0x27,   "EQ_RAT22 - WideO2 Eq-Rat 2-2",
        0x27,   "WO2S22 - WideO2 Voltage 2-2",
        0x28,   "EQ_RAT31 - WideO2 Eq-Rat 3-1",
        0x28,   "WO2S31 - WideO2 Voltage 3-1",
        0x29,   "EQ_RAT32 - WideO2 Eq-Rat 3-2",
        0x29,   "WO2S32 - WideO2 Voltage 3-2",
        0x2A,   "EQ_RAT41 - WideO2 Eq-Rat 4-1",
        0x2A,   "WO2S41 - WideO2 Voltage 4-1",
        0x2B,   "EQ_RAT42 - WideO2 Eq-Rat 4-2",
        0x2B,   "WO2S42 - WideO2 Voltage 4-2",
        0x34,   "WBEQ_RAT11-WBO2 Eq-Rat 1-1",
        0x34,   "WBO2S11-WBO2 Voltage 1-1",
        0x35,   "WBEQ_RAT12-WBO2 Eq-Rat 1-2",
        0x35,   "WBO2S12-WBO2 Voltage 1-2",
        0x36,   "WBEQ_RAT21-WBO2 Eq-Rat 2-1",
        0x36,   "WBO2S21-WBO2 Voltage 2-1",
        0x37,   "WBEQ_RAT22-WBO2 Eq-Rat 2-2",
        0x37,   "WBO2S22-WBO2 Voltage 2-2",
        0x38,   "WBEQ_RAT31-WBO2 Eq-Rat 3-1",
        0x38,   "WBO2S31-WBO2 Voltage 3-1",
        0x39,   "WBEQ_RAT32-WBO2 Eq-Rat 3-2",
        0x39,   "WBO2S32-WBO2 Voltage 3-2",
        0x3A,   "WBEQ_RAT41-WBO2 Eq-Rat 4-1",
        0x3A,   "WBO2S41-WBO2 Voltage 4-1",
        0x3B,   "WBEQ_RAT42-WBO2 Eq-Rat 4-2",
        0x3B,   "WBO2S42-WBO2 Voltage 4-2",
        0x3C,   "CATEMP11 - Catalyst Temp 1-1",
        0x3D,   "CATEMP21 - Catalyst Temp 2-1",
        0x3E,   "CATEMP12 - Catalyst Temp 1-2",
        0x3F,   "CATEMP22 - Catalyst Temp 2-2",
        0xC,    "RPM2 - Extended Range RPM",
};
```

Appendix B – Determining Normalized PID Availability

Although it is possible to use the ‘a’ command in Setup Mode to determine if a normalized PID is available, asking 100+ times before presenting a list is a bit inefficient. An alternate approach is to check for the availability of the associated ECU PID yourself, using the ‘PID Masks’ returned in the Setup Mode ‘j’ (connection status) command.

The PID masks are 8, 32 bit ‘bit fields’. The MSB represents the lowest PID for that field, the LSB the highest. So, in the first 32 bit mask, bit 31 = ECU PID 1, bit 0 = ECU PID 0x20.

In the previous appendix we have a table of ECU pids and normalized PID names (EcuPidMap). So, with the 8 PID masks returned from Connection Status (‘j’), we can determine if a normalized PID is available without using the ‘a’ command, which tests only one normalized PID at a time.

The routines would look something like this:

```
// Test an ECU pid against the ObdiPidMask
// 1 2 3 4 5 6 7 8....0x20
// 0x21 0x22..........0x40
int TestEcuPid(U8 ecupid)
{
      U8 index;
      U8 bit;

      ecupid--;
      index = ecupid / 0x20;
      bit = ecupid % 0x20;

      if (PidMask[index] & (0x80000000 >> bit))
            return 1;

      return 0;
}

int TestNormPid(U16 normpid)
{
      // ‘none’ is always available
      if (! normpid)
            return 1;

      // Out of range?
      if (normpid & 0xFF00)
            return 0;

      return (TestEcuPid(EcuPidMap[normpid].ecuPid));
}
```