Data Refresh Rates from OBD-II over Bluetooth or USB

Summary

Manufacturers of On Board Diagnostic (OBD) devices for cars, and the associated software, may advertise rates of data refresh that can never be achieved by users in practice. The rate-limiting steps in the fastest OBD systems appear to be (1) gateway modules in modern cars and (2) matching in unregulated features of the SPP profile used in communication between a Bluetooth OBD interface (which is plugged into the car OBD port) and the software (which is installed in a host / display computer). For example, the OBDLink MX+ system is advertised at a maximum of ~100 PIDs/sec in Win10 and Android, but in reality a modern-car owner will be lucky to get 30 OEM PIDs/sec; and 7-12 PIDs/sec is more common. SAE PIDs may be a little faster.

Rates vary between cars, depending on data bus speed and limitations interposed in any gateway module. Almost every car made since 2022 has advanced driver assistance systems (ADAS) that rely on an enormously increased flow of data (sensor readings) across the CAN bus, and therefore uses a gateway module to protect these systems from overloading by OBD data requests. This will restrict OBD data refresh rates, and in practice it cannot be altered.

This article lists some factors affecting OBD data rates, gives examples of real rates achieved, warns about different methods used to state refresh rates, explores the factors likely to be rate-limiting in the process, and explains why consumers can not yet predict the rates they will achieve before system purchase. USB devices are an alternative for those who want faster OBD data refresh rates without ‘luck of the draw’ Bluetooth SPP profile matching; but there are other inconveniences, and data refresh rates are still typically far below those advertised. Mention is also made of methods including J2534 readers that avoid some limitations of OBD designs to achieve much higher data rates. Unfortunately, these newer systems are also more expensive.

Introduction

Some users of OBD data are interested in obtaining sub-second readings from multiple sensors in their cars. For example, a car at 100 kph travels over 27 metres per second, so it is reasonable for a person testing brake function to be interested in readings from multiple sensors associated with brake function, at frequencies far higher than once per second. A modern car bus will carry signals from these sensors at very short durations, to be used in the electronic control of braking. The OBD port is connected to the data bus that carries these signals. Can these signals from multiple sensors (PID readings) be logged through the OBD port at intervals below 0.1 second, using an OBD system ‘affordable’ to a casual user? Probably not, as shown below.

Current OBD readers inject requests/acknowledgements, and receive responses from Electronic Control Units (ECUs), via a data bus in the vehicle. More requests or faster refresh means more work for the relevant ECU, and more traffic on the bus. Increased data traffic may compromise safety if the reader is used while moving, unless both the OBD reader and the car are designed to eliminate such risk (which is never guaranteed). CAN bus systems (multiplexed Controller Area Network used for OBD in all cars since 2008) resolve conflicts based on message ID/priority. Some manufacturers dispense with requests to share data over CAN between ECU nodes.

But remember that the system implemented for traffic on the CAN bus(es) is not necessarily the OBDII protocol, even if the CAN bus(es) interface (through a gateway module and DLC3 port) with a diagnostic system that follows OBDII. The OBDII protocol is by definition a request and response protocol; so there will be some lag, even with a CAN bus that allows multiple requests per transmission. A gateway module can implement manufacturer-specific higher-level protocols that assign a low priority to OBD requests, but this is rarely divulged. Such protocols could make it safer to use OBD while driving, at the expense of data refresh rate. The J2534 API mentioned later in this article can listen, without the need for requests, if PID data are ‘broadcast’ on the bus.

Factors Affecting OBD Data Refresh Rates

1. The maximum data rate from the OBD port can not exceed the data rate of the vehicle bus to which it is connected. A 500 kbaud 11 bit CAN bus should be able to deliver 125 bit frames (PID results) at the rate of 4,000 frames per second. Designers aim to keep a bus below 30% load, which would still be >1,000 PIDs/sec. Older cars may have a much slower bus for OBD.

2. Modern cars have in the order of 1,000 sensors reporting PID readings. Most are wired directly to the most relevant ECU. Traffic on the CAN bus(es) comprises requests and responses, or timed broadcasts, of data needed by other ECUs. Recent Toyota ECUs are said to issue no requests, and rely on timed or event-driven supply of shared data. Some PIDs are updated every 0.01 sec, others are much slower. The Toyota Hybrid Control ECU alone lists more than 300 PIDs. Response times of each ECU to OBD requests are not specified.

3. Though ISO 15765 standards (for the ISO-TP CAN transport protocol mandated in all USA cars since 2008) have much on timing, it is not obvious how this translates into data refresh rates.

4. The rate at which an interface (OBD reader) can query the CAN bus and receive a reply through the OBD port about a specific PID is much lower than the frame rate handled by a modern CAN bus. Car manufacturers can add elements such as OBD gateway modules that limit OBD data rates or priorities. These details are rarely made public.

5. OBD readers vary in price, quality and speed. All require software in a host computer to convert and display the binary data in a human-readable form. Most can read only SAE PIDs. User-defined 'custom PIDs' may be available, but these require information about addresses and formulae, which most current car manufacturers keep secret.

6. The OBDLink interfaces are regarded as being among the fastest 'affordable' devices. They are sold in a system including an OBD app (based on OCTech software like OBD Fusion and TouchScan) and in some cases access to PIDs from some car manufacturers (OEM PIDs). OBDLink MX+ is advertised at 'maximum ~100 PIDs/sec', but a modern-car owner using a Bluetooth android or iOS host device will be lucky to get 30 OEM PIDs/sec from an MX+, and 7-12 OEM PIDs/sec is more common. Wired (USB) devices can be faster, BLE typically slower.

7. Some OBD apps (*eg* OBDwiz, OBDLink, OBD Fusion, Torque Pro or Harry's Lap Timer) report the rate at which they receive refreshed PIDs, while connected to the OBD interface. Most apps do not detail how they estimate refresh rates (but some use SAE PIDs accessed for background fuel rate calculations, even when the engine is off).

8. Gauges may have smoothing, so they are not a reliable indicator of the refresh rate being achieved. Instead, look at the rate data changes in a (csv file) log of selected PIDs. Be wary of apps that use interpolation to give the appearance of faster data refresh rates.

9. Every PID counts, so if you are using a dashboard or diagnostic screen with PIDs, it will reduce the rate at which PIDs can be refreshed in a log. Go back to the app home screen.

10. Only log the PIDs you really need, for fastest refresh of those PIDs. Turn off background fuel rate calculations etc if you do not need the PIDs they use.

11. Using PIDs from multiple ECUs may slow things down (I experienced this with faster J2534, but not with slower OBD, using two ECUs delivering data through a gateway module).

12. In OBD app settings, choose to log whenever readouts change; or use a short Sample Time. Also use Fast Polling and short Dwell Time (and use a reader that supports these).

13. The same parameter can be reported from multiple ECUs. Some apps have additional settings like 'use first responding ECU for each PID' to speed things up. From the way PIDs are selected, I think OBDLink only polls a single ECU for each selected PID.

14. Some people argue that the maximum rate is limited by the speed of the device used to host the OBD app (typically a phone or tablet), or by the Bluetooth implementation in the host device. Noise in the wireless spectrum can certainly slow wireless data rates. More on those issues later.

Real Rates (results from repeated tests vary slightly, reflecting variation between replicates)
I tried all of this (except turning off the fuel rate calculations) using OBDLink MX+ (r2.0, Firmware 5.7.1) and the OBDLink app (v5.31.0) with a Samsung Tab S2 tablet dedicated to OBD use, so nothing else using the Bluetooth and no phone or wi-fi or background apps. The car was a 2019 RAV4 hybrid (AXAH54), which uses ISO 15765 (500 kbaud 11 bit CAN bus). When set to log every 0.05 seconds, this combination only gave updates every 0.6 - 0.8 sec from 5 PIDs in one (brake) ECU; ie about 7 OEM PIDs/sec. Brake PIDs are high priority. The app reported 10-11 PIDs/sec. With 7 PIDs from 2 ECUs (Toyota brake and hybrid control) it updated about once per second; ie still 7 PIDs/sec. It also achieved only 7 OEM PIDs/sec using 'Trigger on PID frame', which cycles as fast as the app can read the specified set of PIDs.

In a CAT S41 phone, when 7 OBD SAE PIDs were logged, the fastest (like ICE rpm) changed about every 0.3 seconds (but others like time since start changed only every 2-3 seconds). Let's go with the fastest PIDs and say ~23 SAE PIDs/sec. In Torque lite on that phone, the fastest SAE PIDs also changed about every 0.3 seconds.

This was far below the rate advertised for the MX+ system, so I checked what the apps reported when not logging:

- OBDLink on a Samsung Tab S2 tablet (Android 6→7, 1.4-1.8 GHz octa-core, 3 GB LPDDR3 933MHz 2 ch RAM, BT4.1) dedicated to OBD use, with only MX+ connected: reported 13-15 PIDs/sec (apparently from SAE PIDs used in the fuel rate calculation).
- OBDLink on a CAT S41 phone (Android 7→8, 1.6-2.3 GHz octa-core, 3 GB LPDDR3 933MHz 1 ch RAM, BT4.1) with wi-fi disabled, and only MX+ connected: reported 16 PIDs/sec. This rose to 18 PIDs/sec with phone reception off (flight mode + BT).
- OBDwiz on a HP Pro-book running Win10 Pro 64 bit (Ver 21H2 Build 19044.2130) at maximum performance with no wi-fi, no other programs running, and only MX+ connected: reported 17.6 PIDs/sec. (This program seems unstable: if OBDWiz drops connection then reconnects, it shows 90-125 PIDs/sec. If the computer sleeps then wakes, it may show 2000 PIDs/sec!)
- OBDLink software will not talk to other branded dongles, so I tried Torque Pro on my CAT S41. The rate for MX+ varied between connections, averaging from 22-42 PIDs/sec whereas Panlong (a generic cheapie) averaged 27 PIDs/sec.

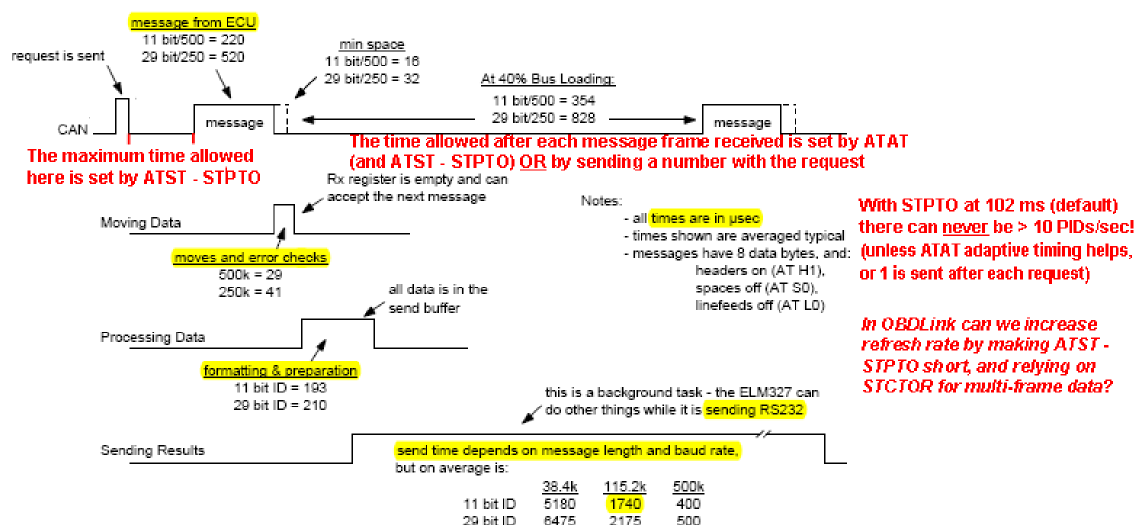
What About Those Specs?

The MX+ was released in November 2018, adding iOS compatibility to the MX Bluetooth which was available from at least 2014, with description and specs unchanged since 2016. MX+ uses Bluetooth (BT) 3.0, and is clearly advertised for (and predominantly sold to) car drivers. For example, the advertisements highlight "legislated OBD-II protocols" "vehicle coverage", "your vehicle's battery", and "PIDs/sec" (not bits or bytes, but PIDs). We are never told what vehicles, software and host devices were used to validate the (PC and Android) rate claims first advertised in 2014-2016. But OBDLink (when pressed) acknowledges that the advertised "MAXIMUM PARAMETER ID (PID) RATE of ~100 PIDs/second for PC & Android" was obtained using an "ECU simulator", and does not reflect the rate that customers will experience.

In any linear process, one step is rate-limiting. Sure, another step may be almost as slow, but if we can identify the slowest steps (one-by-one) we can aim to make them faster, or give up and declare an honest speed for the process. Yes, OBD devices do some things in parallel (like read data from the car and transmit converted data over Bluetooth), but under typical conditions one of these process streams is rate-limiting. OBDLink MX+ (STN2255, Rev J board) includes an STN2120-like OBD interpreter in a modified dsPIC33EP512GM604 DSP including UART from Microchip Technology, and a BT33LT-G Bluetooth module from Amp'ed RF Technology.

BT3.0 with optional HS can reach 24 million bytes per second (Mbps) and should achieve at least 2 Mbps to meet the BT3.0 EDR specification; but the BT33LT-G module, as implemented in MX+, has data throughput up to 200 kbps. Much slower than users might guess (< BT1.2)

but even allowing for overheads and packet retransmission caused by spectrum noise, this is probably not limiting much below noise levels sufficient to trigger BT connection failures. OBDwiz reports MX+ as an 115,200 baud device. I interpret this as the MX+ UART transmit speed; with a single transmit channel it equals 115,200 bps. It does not matter that the CAN bus at 500 kbaud and the BT module at 200 kbps are capable of higher speeds, the UART will be rate-limiting. But at 125 bits / PID frame, 115.2 kbps allows 921 PIDs/sec (less overheads). The ELM327 manual (p 74) seems to indicate that using an 11/500 CAN bus and 115.2 kbps RS232 modem should result in a total (request, receipt, processing and transmission) message time of less than 5 ms; ie 200 PIDs/sec. OBDLink says that STN is faster, so the MX+ hardware should not be rate-limiting. But OBD interface manufacturers tend to set slooow speeds in hardware and firmware, and reinforce these in software, so the devices will be suitable for the slowest supported vehicle protocol. Could these settings be rate-limiting in use with a fast bus and computer? Probably. See the figure below.



It can be dangerous to shorten timeouts, because the vehicle bus may expect a response to a frame that is never received. Then it may resend such frames and (if multiple PIDs are involved) eventually overload the bus. This might not be so bad when parked, but it is obviously to be avoided when driving. And some PID responses need several frames (VIN numbers are so long they need 4 CAN frames, more in older protocols). Nevertheless, CAN systems tend to be quite robust, with several error detecting methods. And OBDLink devices use some extra commands like STCTOR with default timeouts for fc and cf, which implies that they are smart enough to use the PCI byte in CAN responses to know if there are multiple frames. ATE0, ATS0, ATH0, ATL0, ATAT2 and STCTOR fc50, cf50 did not improve OBDLink. As an experiment I added the command STPTO 8 to Interface Initialization for MX+. This made a big difference:

Raw OBD SAE Data Refresh Rates Achieved Using the OBDLink MX+ System

	OBDLink in Android 7-8				OBDwiz in Win10			
Initialization	**None**		**STPTO 8**		**None**		**STPTO 8**	
Trigger	**0.05 sec**	**PID Frame**	**0.05 sec**	**PID Frame**	**0.05 sec**	**PID Frame**	**0.05 sec**	**PID Frame**
5 SAE PIDs	15 PIDs/sec	**18** PIDs/sec	26 PIDs/sec	**30** PIDs/sec	NT	**16** PIDs/sec	NT	**28** PIDs/sec
ICE rpm only	11 Hz	15 Hz	≥20 Hz	24 Hz	NT	13 Hz	NT	22 Hz

Rates were determined using an AXAH54 RAV4 with 11/500 CAN bus, high-priority SAE PIDs, OBDLink MX+, current versions of OBDLink software with Fuel Calculation off, Dashboard off; and OBD-dedicated computers (no phone, wi-fi, background programs, power-saving, competing Bluetooth etc).

Rates were determined from log files after allowing adaptive timeouts to settle.

Initialization commands like STPTO are highly experimental and may be dangerous to use without advice from OBDLink.

Dashboard gauges worked as usual. When logging five OEM braking PIDs via MX+, STPTO8 seemed counter-productive but STPTO15 increased the rate from ~9 to ~14 PIDs/sec. The OBDLink app reported ~24 PIDs/sec with STPTO15 and no logging. Unexpectedly, the generic Panlong reader also responded to STPTO (an STN command, similar to ATST in ELM327). Clearly STPTO can be rate-limiting (in a 2019 RAV4HV), but it may be unwise to alter it.

What About OBD Software?

Different editions of Harry's Lap Timer are reported to give 65-120 Hz (single PID) rates from MX Bluetooth with data supplied from the same "ECU simulator". It is useful to read the Engine Data document from this developer for a better understanding of different ways of expressing PID rates, and ways of optimising OBD software. Some OBD apps allow users to assign PID request priority. Unless a higher-layer protocol has other priority measures, this may increase the danger of overloading the vehicle bus with multiple data requests while driving.

OBD programs (apps) also differ in their (usually unspecified and sometimes dubious) refresh rate calculation methods, so their results may not be comparable. 'Maximum rate' is a nonsense, but logs allow calculation of a meaningful 'average rate'. Developers will generally not reveal whether their software exploits known methods to increase OBD data refresh rates, eg:
1. Use ISO 15765 CAN ability to receive multiple requests, and so handle some PIDs in parallel.
2. Use the CAN PCI byte with STCTOR to avoid waiting for time-out after the final frame.
3. Use physical addressing to all ECUs (including all OEM PIDs).
4. Use an adaptive time-out algorithm that achieves much better performance than ELM's.

As users we do not know. Unless the software exploits all available techniques to maximize OBD data throughput, it is irrelevant to specify a rate at which an OBD reader alone can process data from a 'simulator'. Manufacturers are arguably at fault for slow defaults that apply even under fast bus protocols. The ELM327 command protocol for OBD-RS232 interfaces (which was introduced in 2005 and is maintained for compatibility) and the RS232 standard (which specifies <20 kbps) were not designed for high rates. But some modern RS232 chips deliver 250 kbps, and shortcomings can be reduced by software. No mater how fast are the car bus, the OBD reader and the display computer, we can only see data refreshed at the rate of the slowest link, which in the OBDLink MX+ system is evidently the software (including its SPP implementation - see below). The OBDLink app has a good user interface, it is safe and stable, it is the only app with terrific OEM data access bought with the MX+; but (in a 2019 RAV4HV) it is slow.

As expected, OBD Fusion (v5.4.0, also from OCTech) behaved like OBDLink. So I tried (in CAT S41) HLT Buddy, which polled 156 SAE PIDs in 18 requests. It reported up to 46 Hz raw (≈ PIDs/sec) / 13 Hz Gross or Net (≈ Full Channel Sets/sec) with MX+, but reported rates were unstable. Instability and inability to use OEM PIDs are useless for some purposes; but just considering raw refresh rates for a moment, and taken with results from Torque Pro and OBDLink given above, it is clear that software (including SPP implementation) is rate-limiting.

What About the Host Device (Display Computer)?

Prsnikt on the OBDLink forum reports >45 PIDs/sec from MX+ in OBDLink on SonyZ and Nokia6 Androids and a Benz. Another source using a Samsung Galaxy Tab A or iPad Air with OBD Fusion and a 2011 Infiniti G37 reported 42 PIDs/sec from MX+. Faster than I have seen, but there are too many variables to be sure of the cause. Perhaps gateway modules between the CAN bus and the OBD port slow the data rate in some car models. That is the problem with secret systems - the purchasers cannot know and manufacturers blame each other with impunity.

I don't think that Android host device CPU speed and RAM are likely to be limiting for several reasons: Firstly, I used the Simple System Monitor app to look at these factors before and during OBDLink app operation in my Samsung Tab S2. CPU usage did not change from ~20% (with

few ‘over load’ conditions), the OBDLink app used only ~130-170 MB RAM when idle or logging, and total RAM usage with all system overheads remained below 50%.

As a further test of the “get a faster phone” hypothesis, I tested the OBDLink MX+ system in an OPPO A74 5G (Android 11, 1.8-2.0 GHz octa-core, 6+2 GB LPDDR4X 2133MHz 2 ch RAM, BT5.1). Dedicated to OBD use, it behaved like the Android 7-8 systems tested above (~15 SAE PIDs/sec with no initialization, fuel calculation off, logging triggered by PID change). Moreover, rates of 65-120 Hz from MX Bluetooth were achieved using various editions of HLT in a humble Nexus 4 (Android 5, 1.5 GHz quad-core 32-bit, 2GB LPDDR2 533 MHz 2 ch RAM, BT4) and a CAN 11/500 “ECU simulator” from OBDLink.

Results are sometimes posted showing different rates in different phones, but we are rarely given enough information to know if they are relevant to real-world usage. For example, OBDLink Co. posted results “done with an ECU simulator that is much faster than any vehicle we have ever seen, using the standard OBD-II request/response format” whatever that is.

They were asked:

1. Is BT33 Bluetooth (BT3)? What is 24H? *
2. What was the bus / ECU protocol?
3. Were the rates measured in the OBDLink app?
4. What PIDS were used (SAE? OEM?)
5. Were these average raw PIDs/sec?
6. Were the rates measured from a log file, or from a (secret?) algorithm in the software?
7. Did OBDLink ever replicate these tests, or repeat them on another phone of the same model? The few in common are sometimes very different from those posted elsewhere (insert below).

Device	BT33	24H
BLU C5L	28	31
BLU Studio G3	50	57
Fire Tablet	22	25
Tab 4	37	41
Tab A	36	42
LG G5	45	43
Huawei P8	57	59
Galaxy S9	47	48
Note 5	27	37
Pixel 1	39	46
KTE KT 107	40	47
Galasy Tab S2	5	3
iPhone Xr	23	40

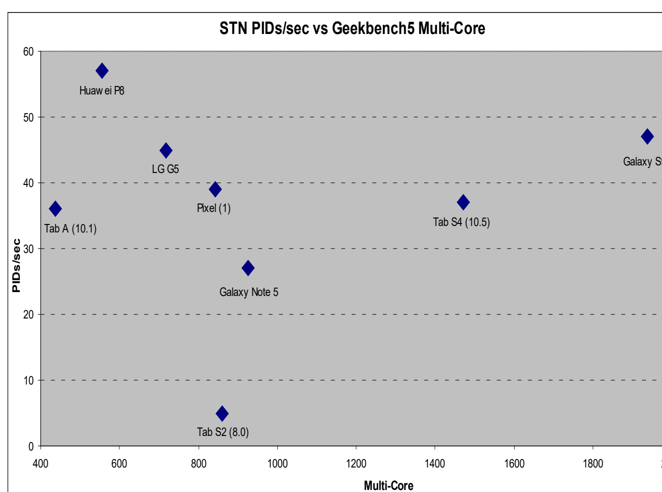
8. Any idea why the Huawei P8 performed so well in your tests with MX+? It was a 2015 phone (Android 4→6, 1.5-2.0 GHz octa-core, 3 GB LPDDR3 1600 MHz 2 ch RAM, BT4.1); not so different in specs from the worst-performing 2015 Samsung Tab S2† (T713: Android 6→7, Qualcomm 1.4-1.8 GHz octa-core, 3 GB LPDDR3 933 MHz 2 ch RAM, BT4.1).
9. Did your analysis reveal any spec(s) predictive of performance? I tried to find some relationship between the STN scores and Geekbench (GB5 multi-core) scores without joy (attached, excluding devices where I found no benchmarks or too many variants to guess what you tested). Obviously most customers who find that their phone is rate-limiting will not be too happy with the advice: “Buy at least 10 current high-quality Android devices, install ODBLink on each, test them in your car, and use whichever gives highest data refresh results.”

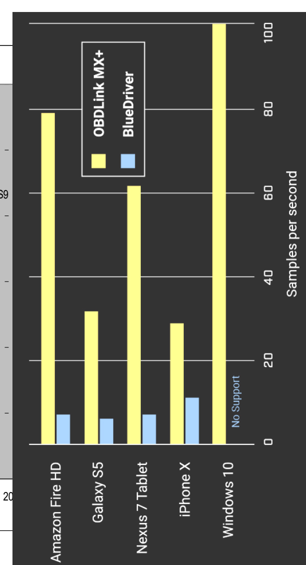
These questions were never answered.

* I think now that BT33 and 24H are different BT chips.

† From my results, the OBDLink data must be from use of a broken Tab S2.

The results also seem strange because my Tab S2 with stock-standard Samsung-supplied Android 7 (in a gen5 RAV4 with 11/500 CAN bus) gives a much higher PID refresh rate.





The BT 3.0 standard specifies (or implies) a much higher data rate than the MX+ UART can deliver. But manufacturers evidently claim compliance based on use of a BT module which may comply in the lower BT stack, but be far below the specified rate as implemented in the marketed device. The rate may be lower still for the BT profile used in the data transfer process.

The SPP profile (and the underlying RFCOMM and L2CAP protocols) allow serial port emulation so that the OBDLink app (or other OBD app) can receive data from MX+ in an RS232 format. The profiles seem to be even less stringently regulated than the core Bluetooth standard. For example, SPP specifies that data rates up to 128 kbps can be used, but support for higher rates is optional. Of course, allowing for SPP overheads, the payload (app level) data rate must be less than the BT module rate, which in MX+ is only 200 kbs. Up to 1900 kbps via SPP has been claimed between matched devices. But 8-10 kbps has been measured, depending on the phone and SPP module. Things like flow control, packet size, sending frequency, environmental interference and matching of SPP module brands and profile settings in the two devices can make a big difference to the data rate achieved between two devices. Large interactions of phone x BT transmission chip are evident in the table above, even for two Amp'ed chips. Also note general superiority of (BT5.2-compliant?) BT24H with 300 kbps throughput over (BT3.0-compliant?) BT33LT-G with 200 kbps throughput (from 8 MHz CPU).

At 8-10 kbps via SPP, suddenly we are 'sailing very close to the wind': allowing for the full cycle of request, receipt, processing, transmission and acknowledgement; the user is likely to experience below 60 PIDs/sec. If different phones implement the SPP profile differently, and/or different apps use it with different efficiency, we can begin to understand why one or the other may become rate-limiting. Obviously there might be an interaction, depending how well the 'optional' features of the profile are matched in the software and host device. Can we test directly how efficiently any particular app or phone uses or implements SPP (or how well they match MX+ SPP settings) and therefore predict the data rate to be expected in PIDs/sec? Not that I have understood. It would be very helpful to have this insight, to 'match' a host device to the OBD reader and its software, in order to achieve something close to the advertised data rate.

As an aside, BLE has a lower 1-2 Mbps theoretical transmission rate (for data packets up to 20 Bytes, in bursts), and BLE apps generally use a variant of (slower) GATT. Protocol limitations typically result in payload data rates of 60-220 kbps depending on optimisation of the exchange strategy for the data type.

Is There a Faster Solution?

In the absence of the required information about BT SPP (or BLE GATT) implementations, those seeking high data rates might consider a USB-OTG OBD reader. For example, OBDLink SX or EX, or a cheaper version may support OEM PIDs through compatible software. The devices are advertised by OBDLink (and click-bait reviewers) at "max rate ~200 PIDs/second"; and claimed to deliver ~170 SAE PIDs/sec from one real car (2011 Infiniti G37) through OBD Fusion (≈ OBDLink app). OEM PIDs may be slower, and there may be limits in CAN-OBD gateway modules in other cars. USB2 Hi-Speed capacity of 480 Mbps is not limiting; even allowing for inefficiency of small data packages and overheads such as protocols, latency and flow control. USB devices avoid limitations of SPP and security of wireless ODB. But with some USB OBD readers, the RS232 stage is limited to 51.7 kbps; and allowing for overheads, requests etc, this may give below 200 PIDs/sec. Data transfer rates from 3 - 3000 kbaud are claimed for FTDI 232R or 230X (RS232-USB) chips, depending on software and other hardware.

The STN1130 (modified PIC24HJ128GP502?) UART default is 115.2 kbps but it is claimed to provide 500 kbps or more for faster data refresh, if the OBD app can support it. I was unable to connect OBDwiz if the SX Baud rate was changed. OBDLink and Torque Pro apps have settings for Baud rate, but showed no effect on reported average PID refresh rate when ST Baud was changed (to 500k or 2000k) using STNterm. PID rate is apparently limited by other car / device /

app factors (*eg* how fast data is generated / computed / moved to and from buffers). Logs and the 'benchmark' graphic showed opposite effects in Torque Pro: I do not understand the graphic.

The SX reader draws less power from the car than Bluetooth units in active (8 mA) or sleep (0.16 mA) mode. SX is said to draw 3 mA from the sleeping host device, but I measured ~80 mA (with OBDLink in automatic connection mode, even if in the background and not connected to SX) from both Samsung S2 and Lenovo M8. Android USB power out generally cannot be turned off without rooting (if at all). Host device batteries have much less capacity than car batteries, and would typically be flat very quickly from 80 mA of phantom drain. Some USB peripherals (*eg* mice, but not the SX reader) power off when the host device sleeps. Some host device manufacturers use "cold standby" with VBUS turned off, or an OTG time-out. Perhaps it can be controlled through kernel settings in muic.h (see below). The SX reader requires both USB and OBD power to function. Those who use a device that remains in the car may find that manual intervention (to unplug and replug the SX USB cable) is essential. Or the phantom drain can be stopped using manual SX connection in the OBDLink App and/or an intermediate device (see below). Those who unplug the host device (*eg* their phone) after each drive will be unconcerned.

From the 'community' forum, the SX has various problems for many users. It has undergone many revisions. In my hands, the OBDLink (or Fusion) app was slow, and after 5.31.0 would not install on a Tab S2 running Samsung Android 7. OEM add-ons included in an MX+ price are limited to that reader, and must be purchased again for SX (at the current higher price). Even if you do not re-purchase, the OBDLink app wastes your time loading vehicle data and shows the OEM PIDs under Vehicle Editor, but they will not work in Dashboards etc. In contrast, those purchased through OBD Fusion are simply licensed to one app (play) store account.

Reported refresh rate is higher in Torque Pro, but alas few of the included Prius PIDs work with a RAV4 hybrid, the Toyota plug-in adds nothing, and shortest logging interval is 0.1 sec.

A cable is not as neat as Bluetooth connection, and it may hit your knee unless you use a right-angle adaptor. But you can probably conceal the cable easily, and connection should be very stable. A Windows laptop needs drivers from FTDI. In my experience OBDwiz is slow, 6-22 PIDs/sec with SX. Android host devices require data exchange with an OBD reader via OTG (most made since 2015 can do this). To supply power to an Android host device while driving and have the OBD reader sleep when not driving requires at least an OTG charge / data splitter (accessory charging adaptor). Whether it works (and the resistance required between ID/4 and Gnd/5 pins) varies between host devices. None of three adaptors that I tested was effective (unless modified as described below) for my Tab S2, S41 or M8. USB C host devices should handle pass-through power more reliably, though the hubs are expensive. With SX rev K / 4.2:

PID refresh rates from stock-standard Toyota RAV4 (AXAH54) and Samsung Tab S2 (SM-T713):

	Default	500 kbps (to 2 Mbps in OBDLink)	STPTO 8 (15)
OBDLink reported (SAE)	22 PIDs/sec	22 PIDs/sec	**55** (39) PIDs/sec
Torque Pro reported (SAE)	70-80* PIDs/sec	No benefit, except benchmark	No benefit
Logging* (10 TP braking PIDs)	≤ 100 PIDs/sec	≤ **20** PIDs/sec	nt

* – fuel calculation

PID refresh rates from stock-standard Toyota RAV4 (AXAH54) and CAT S41 (+/- Aeroplane mode):

	Default	Benchmark (500 kbps)	STPTO 8 (15)
OBDLink reported (SAE)	22 PIDs/sec	na	**55** (35) PIDs/sec
Torque Pro reported (SAE)	80 PIDs /sec	~ 160 (≤ **250**) PIDs/sec	No benefit

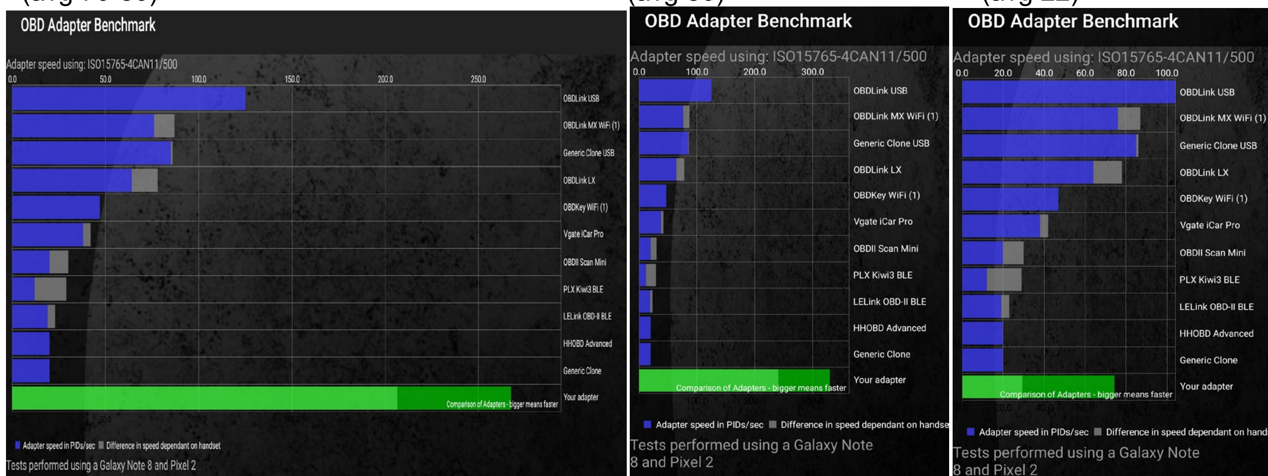
HLT Buddy reported 60-70 Hz raw / 16-20 Hz Gross with SX.

So, even with OBD-USB devices:

1. The software is rate-limiting, with OBDLink (OCTech) software far slower than Torque Pro or HLT; at least in the 2019 Rav4HV, where a gateway module limits OBD data rates (see below).
2. There is variation in functionality between host devices, and some of this variation appears to depend on features that you can not test, or determine from device specs.
3. Data refresh rates are generally far below those advertised, but USB is faster than BT.

Torque Pro 'Benchmark' Graphics (and reported average PIDs/sec):

Tab S2, SX at 500 kbps (avg 70-80)	CAT S41, SX at 500 kbps (avg 80)	CAT S41, MX+ (avg 22)



I don't know what the green bars represent in these graphs, or why they differ from the speed reported under 'adapter status'. Perhaps the solid green bar is 'maximum' SAE PID speed achieved? Could the translucent green bar be speed from some hypothetical host device? Or a using a higher baud rate? Or without (gateway) ECU limitation on OBD? Could Torque estimate this from direct CAN frame rate? What a shame it is not documented.

Tab S2 USB Power Solution and Failed Lenovo M8 USB Power 'Solution'

USB-OTG normally allows either Android 'host' device charging (open circuit between micro-USB pins 4 & 5) or data transfer between 'host' and connected 'slave' devices while powering the 'slave' from the 'host' (short circuit between micro-USB pins 4 & 5). However, some manufacturers, in some device models have implemented various OTG (muic) modes, depending on resistance value between pins 4 & 5. Some modes (designed for hubs or docks) allow 'host' charging while data is transferred. Unfortunately the standard for this is not respected, and the capability (and resistance values) are not disclosed in device specifications. Sometimes they can be found in xda-forums or elsewhere, where enthusiasts may have explored the firmware codes or just tested various resistors to see if anything worked. USB-C behaves differently.

Using OBDLink (Fusion) and Automate Apps, and starting with a "Micro USB to OTG Charging Cable" (Y cable) that included a junction box, I was able to modify the ID connection as shown below to achieve: (i) OTG input from the SX OBD reader while driving; (ii) charging of the host device while driving; (iii) negligible phantom power from the host device; (iv) automatic start of the host device and connection of the reader when ignition is turned on; (v) automatic sleep of both host device and reader when ignition is turned off. In position SWI with 121 kΩ resistance between ID and Ground pins, the Tab S2 enters ADC_UNIVERSAL_MMDOCK mode. In this mode, it charges from the micro-USB port and simultaneously provides OTG data, but it does not send power from the tablet to the SX reader when external power is cut (there must be a diode in the circuit, but other Y cables or hubs may not do this). The charge rate (from a car lighter to USB adapter) is low, but it seems to more than keep up with OBD usage.

If OBDLink is set to automatically connect, the minimal Automate 'flow' for operation in the Samsung Tab S2 without manual intervention (provided the Obdlink App is in the foreground) is: Power On > Pause 10 seconds (while OBD connects) > Simulate Touch 'Dashboard'; Power Off > Android Back (to OBDLink home screen). The Android device should sleep from the OBDLink home screen, but SX will not sleep. Any 'communication error' message in the App is resolved when SX is re-powered. For unknown reasons, it may be necessary to go through app communication settings afresh after each physical disconnection; but after that communication seems solid. The required "pause" time for OBD connection varied between starts, for reasons I do not understand.

For anyone still looking into this again, the correct value for the Tab S2 SM-T713 (gts28vewifi) for OTG + Charge with the stock kernel (tested on LineageOS 16 and 17 for me) is 121k. This will set the device to a cable type of "UNIVERSAL_MMDOCK" which allows OTG as well as fast charging, up to 2.1A. Found in muic.h in the samsung kernel source:

Information from solidxsnake13224 at:
https://forum.xda-developers.com/t/usb-otg-charge.3588019/

Code:

```
/* MUIC ADC table */
typedef enum {
    ADC_GND             = 0x00,
    ADC_SEND_END            = 0x01, /* 0x00001 2K ohm */
    ADC_REMOTE_S11          = 0x0c, /* 0x01100 20.5K ohm */
    ADC_REMOTE_S12          = 0x0d, /* 0x01101 24.07K ohm */
    ADC_RESERVED_VZW        = 0x0e, /* 0x01110 28.7K ohm */
    ADC_INCOMPATIBLE_VZW        = 0x0f, /* 0x01111 34K ohm */
    ADC_SMARTDOCK           = 0x10, /* 0x10000 40.2K ohm */
    ADC_HMT             = 0x11, /* 0x10001 49.9K ohm */
    ADC_AUDIODOCK           = 0x12, /* 0x10010 64.9K ohm */
    ADC_USB_LANHUB          = 0x13, /* 0x10011 80.07K ohm */
    ADC_CHARGING_CABLE      = 0x14, /* 0x10100 102K ohm */
[B] ADC_UNIVERSAL_MMDOCK        = 0x15, /* 0x10101 121K ohm */[/B]
    ADC_UART_CABLE          = 0x16, /* 0x10110 150K ohm */
    ADC_CEA936ATYPE1_CHG        = 0x17, /* 0x10111 200K ohm */
    ADC_JIG_USB_OFF         = 0x18, /* 0x11000 255K ohm */
    ADC_JIG_USB_ON          = 0x19, /* 0x11001 301K ohm */
    ADC_DESKDOCK            = 0x1a, /* 0x11010 365K ohm */
    ADC_CEA936ATYPE2_CHG        = 0x1b, /* 0x11011 442K ohm */
    ADC_JIG_UART_OFF        = 0x1c, /* 0x11100 523K ohm */
    ADC_JIG_UART_ON         = 0x1d, /* 0x11101 619K ohm */
    ADC_AUDIOMODE_W_REMOTE  = 0x1e, /* 0x11110 1000K ohm */
    ADC_OPEN            = 0x1f,
    ADC_OPEN_219            = 0xfb, /* ADC open or 219.3K ohm */
    ADC_219             = 0xfc, /* ADC open or 219.3K ohm */

    ADC_UNDEFINED           = 0xfd, /* Undefied range */
    ADC_DONTCARE            = 0xfe, /* ADC don't care for MHL */
    ADC_ERROR           = 0xff, /* ADC value read error */
} muic_adc_t;
```

4 3 2 1
Type A (F)
OBD
12345
Micro-B (F)
Power
Unsolder wire from 'Tablet' terminal 4 on board.
121 kΩ
Add resistor shown in blue between 'Tablet' wires 4 & 5.
Tablet
54321
Micro-B (M)

Unfortunately, later versions of the OBDLink App (which bring useful new features like warnings, alerts, and faster loading on App start) have a nag screen on attempted connection to an enhanced network, which further complicates automation (accomplished below).

The resistance values used to signal docking modes (if such options are implemented) vary between Android devices (and possibly between operating systems if the devices are rooted). Lenovo M8 is reported to use 10-18 kΩ. I tested 15 kΩ, which worked (with some unexpected effects). The USB device was detected only while power was supplied through the Y cable. This is a good thing as it blocks USB phantom power use (SX drew 80 mA with the car ignition on but 0 mA with ignition off in this set up). It also gives an option for Automate (used below). The M8 retained a 'Plugged In' status after power connection, even when power was removed, until another OTG cable and device was plugged in. Battery saver and doze will not work, and the device may use excessive (CPU *etc*) power even when unplugged and with screen asleep. Turn

off the device power-on LED (in device Settings), and allow the screen to dim when power is on (in Developer Options). The screen still times-out without user input, but it no longer wakes when power is reconnected. Monitoring of Android-reported power status avails not, requiring a different approach. Automate also provides several Blocks relevant to the Lenovo "Plugged In" flaw. Sadly, the Lenono M8 has a quirk of high battery use in docked mode (whenever this cable is engaged); and this cannot be overcome using Automate or ADB shell.

I installed the Automate App and legacy apk, and used settings for ADB shell command and privileged access (explore Automate Settings and Documentation; shell command `adb tcpip 5555` will have to be sent from PC after each Android reboot). I left 'time fix' alone as it caused problems. Here is the 'flow' tried for operation without manual intervention of OBDLink App & SX in the Lenovo M8 (Android 10 TB-8505F_S30116_230914_BMP) provided the OBDLink App is pre-loaded with manual connection in settings and running (at least power-restricted in the background). This Flow works ONLY with the Y cable (NOT with the SX connected directly to the tablet).

Flow beginning > When USB device attached (When changed) *{as power sensing is confused in dock mode}* >

YES > Device doze mode set state (Deactivate) *{deactivates doze}* > Shell command privileged (`dumpsys battery reset`) *{in M8 with Y cable enables the device battery gauge and disables the battery-saver}* > Keep device awake (CPU and screen & Awake screen) *{wakes the screen}* > Start app ... OBDLink *{ensures it is in the foreground}* > Delay 2s > Interact touch Click 50 88 *{connects to SX }* > Delay 1s > Interact touch Click 72 68 *{closes the enhanced network connection nag}* > Delay 8s *{while OBD modules load}* > Interact touch Click 50 32 *{goes to dashboard}* > Back to 'When USB device attached (changed)'.

NO > Delay 1s > Interact touch Click 50 75 *{closes the disconnect nag, SX has disconnected and can sleep}* > Interact Back OBDLink *{goes to the OBDLink home screen}* > Delay 1s > Interact Home *{optional, to background the app}* > Keep device awake (Allow sleep) *{important to allow the device screen to sleep again}* > Shell command privileged `dumpsys battery unplugged` > Device doze mode set state (Force activate) *{these 2 blocks overcome the Lenovo "Plugged in" flaw and maximize device battery life}* > Back to 'When USB device attached (changed)'.

This flow is far more complicated than that described above for the Tab S2. Sadly, even using Flight Mode and everything in M8 device Settings that seemed likely to reduce battery drain when asleep, there was about 32% device battery use per day with the Y cable in. This drops to <2% per day without the Y cable. It is a Lenove quirk that cannot be overcome using Automate or ADB commands. Some of the Delay times might need to be altered for other vehicles or Android versions. X and Y values (from top left) for simulated clicks are found using Automate > gesture record. The Flow may get confused if ignition is turned off too quickly for completion. Sometimes OBDLink gets confused (*eg* 'connects' but shows zero values for all PIDs) and has to be restarted. Users with fewer OBD set-up constraints can simplify the flow accordingly.

The only 'solution' to the Lenovo "Plugged In" flaw is to run on battery (not the Y cable). Automate (set for unrestricted background activity) uses a little power to watch for a trigger. This may be a problem if your device battery is old or the vehicle sits for long periods. In that case, it may be best to stop Automate. The OBDLink App in automatic connection mode also watches for a trigger (reader connection), so the same would apply. The micro-USB plug can be removed if there is any phantom power leakage from the Android device; or to connect a charger. If `dumpsys battery unplugged` is active, the battery still charges and discharges; it just fails to report. Charge display then will need `dumpsys battery reset`. Any of these requires user intervention.

Android 10?: Same Effect

I have not found an OBD solution capable of reading Toyota OEM PIDs (sensor outputs) at fast refresh rates (> 60 PIDs/sec) on an 8" tablet that fits a RAV4 console. This is exacerbated by failure of OBDLink and OBD Fusion v5.32 to install under Android v7 in the Samsung Tab S2, and the preference of tablet manufacturers for larger sizes. (See below for TabS2 custom ROMs.)

Therefore, I tested a Lenovo M8 HD (a low-priced tablet generally rated higher than similarly-priced Amazon Fire HD8, but below the Samsung Tab S2 in Geekbench5 multi-core scores: 543 vs 860). Lenovo M8 HD wi-fi tablet: ('2nd gen' TB-8505F / ZA5G0036AU, 2019, Android 9-10, Media Tek 2.0 GHz quad-core, 2 GB LPDDR3 933MHz 1 ch RAM, 32 GB eMMC, BT5). There was no 1st gen (maybe the tab E8), but there are later versions. The cheaper Lenovo M8 (with Android 10) generally performed slightly below the flagship Samsung Tab S2 (with Android 7) in OBD tests with MX+; but with SX, M8 gave higher rates for HLT and possibly for OBDLink STPTO (see safety caution above).

App-reported avg PIDs/sec:
<u>OBDLink</u>

MX+: 12
STPTO8: 21
STPTO15: 19

SX: 21
500 baud: 21
2000 baud: 22
STPTO8: 57
STPTO15: 41

<u>Torque Pro</u>– fuel calculation

MX+: 22

SX: 81
SX benchmark: 205

<u>HLT Buddy</u>

MX+: 47/13

SX: 74/20

Again the software was rate-limiting, and USB was faster. There was some software x device interaction. M8 was within 2-3 PIDs/sec of S2 (and lacks an effective power solution). Rates with SX were marginal to log multiple OEM PIDs (only available in the OBDLink = OBD Fusion app) at sub-second intervals.

Custom ROMs

Installing LineageOS 17.1 (Android 10) on the TabS2 allowed OBDLink 5.34 to load, but it was no faster. Installing a custom ROM is not for the faint-hearted: some features (like camera and phone use) are lost. Even with a nandroid backup it can be one-way to a bricked device.

However this test did reveal that SX performs better than previously seen in MX+ with STPTO:

STPTO	OBDLink Info (SAE / MAF)	Log* (fast OEM PIDs)
102 (def)	23	20
15	39	30
12	45	34
10	48	34
8	55	37

* Fuel calculation off, no gauges active, log on PID change, 'fast polling' and 0 dwell. Fast PIDs (motor torque or engine rpm) changed at every log point. STPTO5 caused communication errors. STPTO8 gave some strange logs (possibly false negatives from too short time allowed between OBD request and response), and maybe rough running.

OCTech seems conservative about adaptive timeouts. Using an OBDLink dashboard or fuel calculation causes a further big reduction in logged rates.

Even using a very short time-out, this is < 20% of the advertised "maximum rate" for SX. Maybe enough for useful sub-second logging of a few chosen OEM PIDs (but see safety caution above).

Other Cars

In a 2006 Honda Civic Vti (29/500 CAN bus), MX+ gave 75 SAE PIDs/sec and SX gave 160 SAE PIDs/sec as reported by OBDLink 5.34 in a Tab S2, irrespective of STPTO. OBD data rate and the STPTO effect have something to do with the vehicle: probably the gateway module (ECU) in a 2019 RAV4HV! Most modern cars use a gateway between CAN bus(es) and OBD port, so owners can anticipate such effects. Almost every car made since 2022 (i) has advanced driver assistance systems (ADAS) that rely on an enormously increased flow of data (sensor readings) across the CAN bus, and therefore (ii) uses a gateway module to protect these systems from overloading by OBD data requests. The programming of gateway ECUs is proprietary.

OBD Refresh Rates Reported by Apps from a 2006 Honda Civic Vti (29/500 CAN bus):

Android device		SAE PIDs/sec from OBD Fusion 5.4 :: OBDLink 5.34*		
	Android (GB5m)	Panlong BT	MX+ BT	SX USB
Lenono Tab M8	10 (~780)	23 :: -	50 :: 55	200 :: 200
Samsung Tab S2	10 Lin. (~850)	37 :: -	68 :: 75	160 :: 160
CAT S41 (phone)	8 (~750)	40 :: -	70 :: 65	180 :: 170
OPPO A74 (phone)	11 (~1400)	45 :: -	75 :: 75	200 :: 180

* BT rates fluctuated a lot so highest levels seen frequently are given. Fast Polling was on.

With the 2006 Honda Civic (probably no speed-limiting gateway): (1) Torque Pro was slower than OCTech apps; (2) There was a rough correlation between price of Android host device and OBD data rate from BT readers, but not a USB reader; (3) There was no relationship to GB5m score; (4) MX+ gave much a higher data rate than generic Panlong. In contrast, the difference was small with a (slow) gateway ECU between CAN bus and OBD reader in the 2019 RAV4HV.

Why use RS232 or UART at all?

In the modern era, there seems to be no good reason to use the (slow, old) RS232 / UART communication protocol at all in OBD systems. The conversions used at present (for wide compatibility, and familiarity of coding?) are something like:
CAN ↔ OBD Port ↔ RS232 (SPP) ↔ USB or Wireless (Bluetooth) ↔ RS232 (SPP) ↔ OBD software.

RS232 made sense in the long-past days of serial modems, but not today when faster methods (CAN, USB, wireless or ethernet) are used for data transmission. Interposing RS232 just adds conversions into the stream, with all the speed losses associated with buffering, overheads, and format conversion. It is possible to get the OBDLink SX closer to the advertised data refresh rate under Windows by tweaking settings of the virtual serial (com) port used with the FTDI chip The optimum settings probably depend on both the rate of OBD data flow into the FTDI chip (number of PIDs polled) and how the OBD program requests data from the USB buffer. There are no such tweaks for Android, so users are limited to the settings made by the app developer.

It is time that manufacturers of OBD interface devices and OBD software dispensed with these unnecessary conversions for faster readout of the vastly increased number of sensors that report through the OBD port in modern cars. It would be faster, simpler and cheaper. Perhaps it will take a new entrant to the OBD space to realize the opportunity in such an inevitable evolution.

Why use Requests for Shared Sensor Data?

A reading that traverses several modules may lag, irrespective of the update rate for the PID. Gateway modules can affect lag and refresh rate of OBD data, variably between CAN nodes.

Logging indicates that CAN data received by the gateway is transmitted to the DLC3 (OBD) port. Given that Toyota CAN data is shared between ECUs without requests, why use requests at all? It should be enough to listen on the bus, using appropriate filters for the required data. It is probably necessary to establish CAN ID and equation for each item (which may, or may not, be the same as the PID used in OBDII). Using data sent by the original ECU should result in the fastest refresh rate and minimum lag for shared sensor readings (check logs), with no increase in

bus traffic from the DLC3 reader. The gateway could be bypassed if it restricted or delayed CAN data to DLC3, but this would require another access to the CAN bus(es) and seems unnecessary. Something similar is already used in Autosport applications, with known sensor CAN IDs.

Is J2534 the solution?

OBD-II (SAE J1979, including a J1962 data link connector) became mandatory in most countries between 1996 and 2006, aimed at testing for compliance with emissions standards. By 2004, it became mandatory in the USA and Europe to provide for programming of drive-train ECUs through an interface specified in SAE J2534, which accessed the vehicle CAN data bus through the same J1962 connector. By 2008, ISO 15765 CAN became the mandatory data transport protocol, and the same J1962 pins provided CAN data via OBD or J2534. Many car makers used proprietary software through J2534 for diagnostics, and provided OBD access to the same parameters. J2534 standards continue to evolve (*eg* to accommodate faster communication protocols). They are presently windows-centric, but the API-DLL match has been ported to other computers. SAE J1979-2 (OBDonUDS) will be mandatory in the USA from 2027. SAE J1979-3 (ZEVonUDS) is intended for use with 'zero emission' cars. New scan tools may then be required.

As cars incorporated more electronic sensors and data for purposes that affected safety, it became important to protect the data bus from overloading and interference with these functions. Data needed in other ECUs are typically 'broadcast' on the CAN bus with no need for requests. But OBD-II is explicitly a 'request and response protocol', so as electronic complexity increased, manufacturers introduced gateway ECUs between the CAN bus and OBD reader, which limited OBD data rates and thus protected against overloading with OBD data. In contrast, software using a J2534 interface can 'listen' for broadcast CAN data, without increasing traffic on the data bus. Thus it need not be throttled for safety by a gateway ECU. Which data are broadcast and effects on J2534 vs OBD are platform-specific, but the 2019 RAV4HV gateway evidently slows OBD data rates rather than rates of CAN data to DLC3 (J1962 or OBD port) *per se*.

Whereas OBD tools perform many data conversions within the reader, J2534 tools typically use the (much faster) host computer for more of this work. PC programs like Blue-J, Techstream and ECU Hacker access CAN bus(es) through a J2534 tool. Some J2534 devices use fast microprocessors (currently up to 1 GHz, 32-64 bit) with direct high-speed USB access, and can be much faster than OBDLink readers. But they may be more expensive, software-dependent, have no sleep mode, and/or allow ECU programming which is best left to professionals.

For example, Techstream uses a J2534 interface via DCL3 to the car CAN bus and ECUs. It is a very powerful diagnostic and ECU re-flashing program designed for use by Toyota technicians. Although there is a reasonably good help system, it can seem quite ponderous to use for simple operations. With interfaces that I have tested, connection to the vehicle then to a selected ECU is very slow, compared to OBDLink tools. But once connected, Techstream is able to acquire live data at a rate higher than OBD tools (especially as recent Toyota gateway ECUs restrict OBD data rates). With a Vista-era computer and cheap/nasty Xhorse-clone (mini-VCI), it reports a fastest refresh rate of 100 - 150 ms for single parameters. But depending on the ECU and PIDs selected, it can deliver well over 200 OEM PIDs/sec (with a "sweet zone" often between 20 and 100 parameters). Newer J2534 interfaces may be substantially faster (Tactrix OP2 with a mere 72 MHz processor is reported to achieve >1,000 PIDs/sec). In Techstream, recording (set-up under the Function menu while in a data screen) uses a proprietary format. It can export to a tab-separated format (though the file is labelled csv). The unusual mm:ss.### time code can be converted to seconds (with 1 ms resolution) through a spreadsheet formula (in XL, format as a number, then paste special multiply by 86400). Graphed and Logged refresh rates can be faster than the stated Live Data rates. Dual ECU data refresh much more slowly and will not export.

Interestingly, Techstream offers 'generic OBD' access to three drive-train ECUs via J2534. The refresh rate for the same parameter was much lower in OBD mode than OEM mode (graphed at

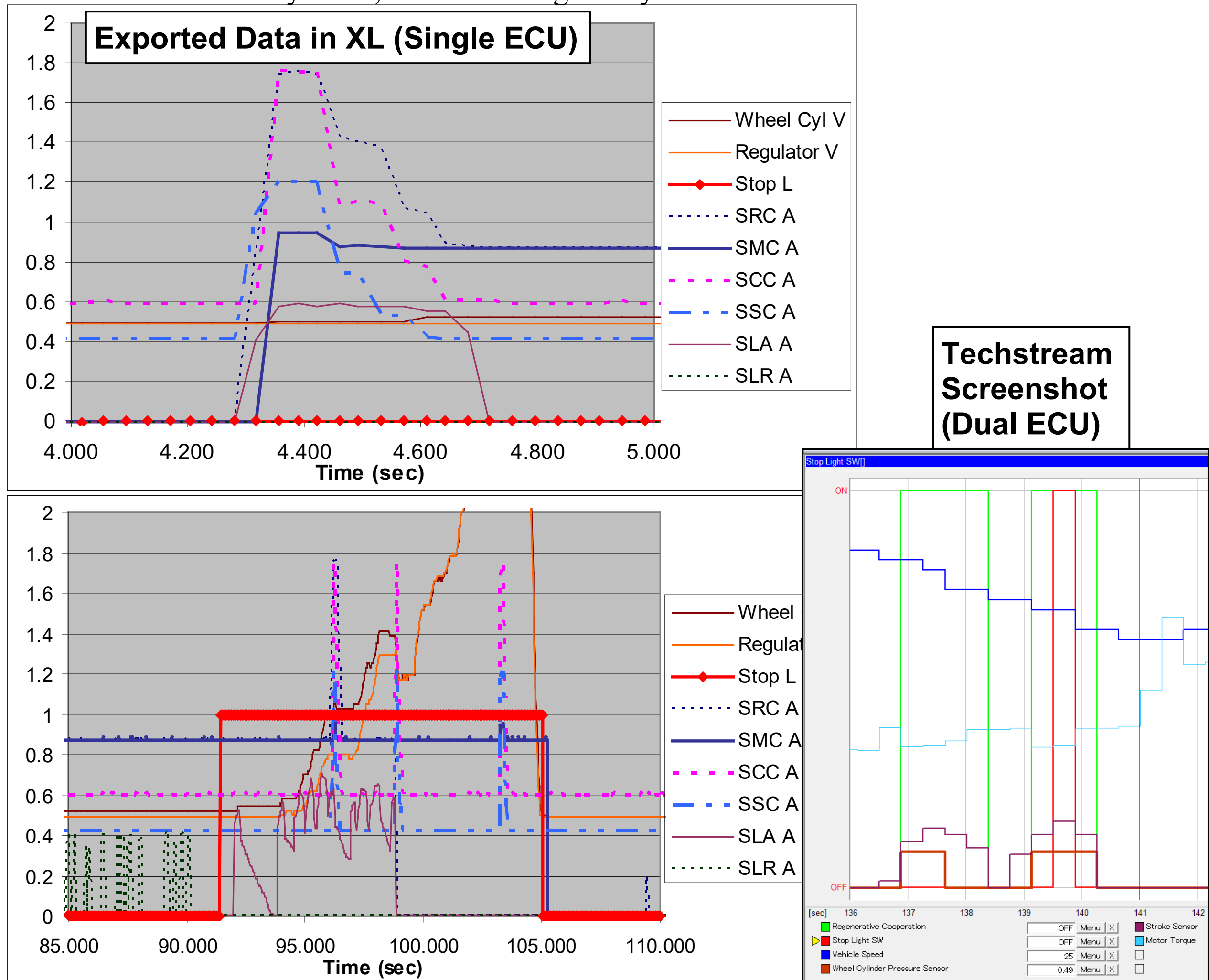
5/sec vs 22/sec for engine RPM, using a cheap Tactrix OP2 clone in Vista or Win10 computers).

Techstream is licensed by Toyota at prices pitched to professionals who can pass on the costs and divide them between multiple customers. Sellers of cloned J2534 interfaces commonly provide pirated versions of Techstream software. Some of these pirated versions are virus-infested, incomplete or flawed. It is unwise to use them, especially for functions that can damage the vehicle or its components. Similarly, it is probably unwise to use (cheaper) cloned J2534 interfaces for such functions. Toyota has endorsed Drew J2534 interfaces (Mongoose, TS3-ETH).

However, the cost of professional tools and subscriptions may be hard to justify for a car owner interested in only a sub-set of functions such as diagnostics. For whatever reasons, manufacturers have not provided (cheaper) versions of the interfaces or the software with only those functions most sought by car owners. That creates a market for (and is presumably the lure of) the cloned J2534 devices and pirated software.

Ideally, slick software like OBD Fusion might work with OEM PIDs accessed via J2534 (to avoid the gateway speed limitation and/or bus overload potential of OBD). J2534 Fusion? It should be possible for a company like OBD solutions to manufacture a J2534 interface with the quality and features of OBDLink devices (power off, automatic connection, simultaneous access to PIDs from multiple ECUs, affordability). Or a diagnostic-only genuine Tactrix?

As an example of what can be achieved, the graphs below show some parameters logged using Techstream with a cheap J2534 cable (Tactrix OP2 clone). Logged data refresh rate for 25 parameters from the brake ECU of AXAH54 was ~37 ms, or >670 PIDs/sec. Gentle braking can activate various solenoids that send some pressure to the wheel cylinders (plus regenerative braking in this hybrid car) without illuminating the stop lights. It is much less revealing to examine such effects by OBD, when the car gateway limits OBD to ~20 PIDs/sec.



OBD Gateway

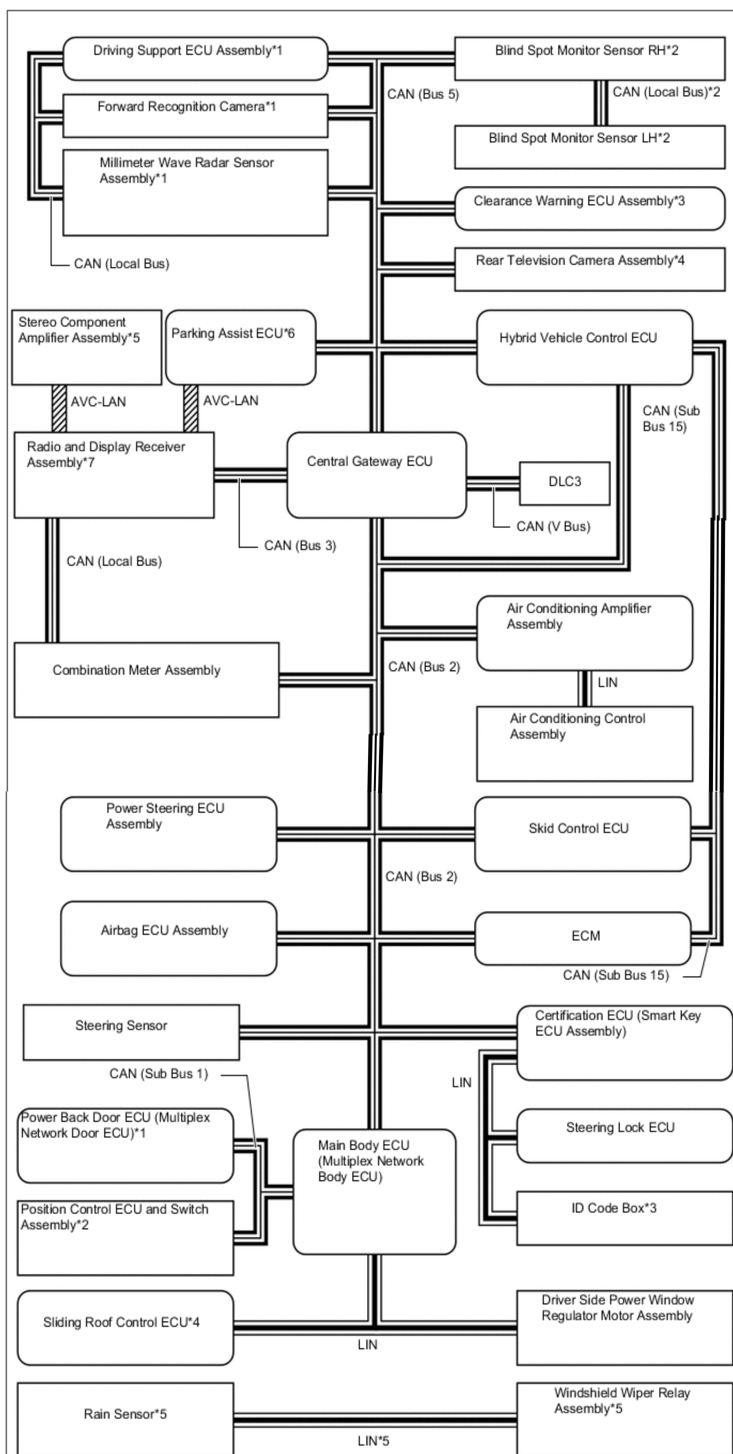
Example of a gateway ECU between OBD port (DLC3) and car CAN bus:

Shown is RAV4HV AVA44 multiplex communication system diagram.

An OBD reader adds a CAN node through DLC3: the ISO15765 default ID for a tester is 7DF for 11-bit CAN, or 18DB33F1 for 29-bit CAN. A gateway ECU can recognise requests from a scan tool.

In the 2019 RAV4HV, the central gateway limits OBD (request and response) data rate to ~20 OEM PIDs/sec, but it does not seem to limit J2534 data rate through the same DCL3 connector and CAN pins.

One possibility is that the gateway actually limits request rates from DCL3, rather than rates of data from the car CAN buses to DCL3. A related possibility is that a busy gateway simply gives very low priority to (OBD) requests.



SPECULATIONS ABOUT DATA RATES

These speculations are based on what I have seen with a 2019 RAV4HV (AXAH54), using an SX reader with OBDLink under Android 10, or Tactrix OP2 clone with Techstream v13.30.018 under Vista (performance of v18.00.008 in Win10 was similar).

Car makers vary in what ECUs they install and the programming of ECUs. These details are proprietary (secret), so some effects are sure to be brand and model-specific.

'Security' gateways require owners to use a compatible scan tool, then pay & jump through security hoops online, to access their own car data or clear fault codes. The bad guys will simply bypass the gateway.

OBD 'request and response' data

The OBD protocol is 'request and response', but recent CAN protocols allow both requests and responses to be concatenated, so 20 PIDs/sec can be <40 CAN frames/sec. The central gateway ECU limits OBD data rates to about 20 PIDs/sec, even though the CAN bus can deliver >1,000 PIDs/sec. The OBD data rate can be increased by about 50% by decreasing STPTO, so the rate is controlled partly by timeouts, but altering STPTO may not be safe, and the cap is imposed mostly in other ways. This is presumably a safeguard against overloading the CAN bus with OBD data. Also note that the fact that OBD system can 'refresh' data (complete a request and response cycle) at a specified rate does not mean that any particular sensor or ECU actually updates the data at that rate.

The central gateway ECU presumably collates SAE OBD data (there is no SAE ECU) from the engine or other ECUs. OEM data comes from various ECUs, via the gateway ECU. Although OEM diagnostics are not regulated in the OBD standards, most car makers find it convenient to use the same protocols for both regulated (SAE) and OEM data. The OBDLink (≈ Fusion) app can communicate through DCL3 and the gateway ECU with one CAN 'network' in addition to SAE data. 'Network A' appears to be much like CAN bus 2 in the RAV4HV AVA44 multiplex communication system diagram above.

Some ECUs connect to multiple CAN buses or networks. Some parameters (such as engine RPM or vehicle speed) are used in, and can be read from, various ECUs. There may be a difference in processing time within each ECU, so the parameter value read in one 'OBD request batch' from several ECUs can vary. The ECU that receives the original sensor data should have least lag, unless the gateway imposes differential lags on ECUs (or responds faster to SAE PID requests).

J2534 'diagnostic' data

Whether, and how, car makers implement diagnostics are not regulated in the J2534 standard. As implemented in Techstream software, a J2534 interface allows much faster refresh rates for OEM parameters from a single ECU. The paradigm is different from OBD data. The refresh rate is specified in ms for the full set of selected data, rather than PIDs/sec. These are of course interchangeable for any specified number of parameters (PIDs), but (unlike OBD) the refresh interval in ms via J2534 often does not increase with added parameters. This means that the refresh rate in PIDs/sec often increases with added parameters. Techstream offers a dual-ECU mode, but it refreshes much slower, and can not export recorded data to a spreadsheet.

The refresh rate depends on both the particular ECU and parameters of interest. It appears to operate in a batch mode with rate determined by the slowest-reporting sensor in the specified list of parameters (and eventually by the number of listed parameters). For example, engine RPM alone may only be refreshed every 50-100 ms but many additional parameters can be listed at the same time, resulting in an overall rate of >1,000 PIDs/sec. OBD might poll engine RPM at a higher rate, but does the ECU actually update readings at the higher rate reported by OBD? Check logs. Direct sensor readings are likely fastest (*eg* for a tachometer to show engine RPM).

The J2534 'diagnostic' mode must increase traffic on the CAN bus, to the extent that data batches are requested and reported in addition to traffic already on the bus. Perhaps the data rate is not throttled by the gateway ECU when a few parameters are monitored from a single ECU, as the load is not problematic?

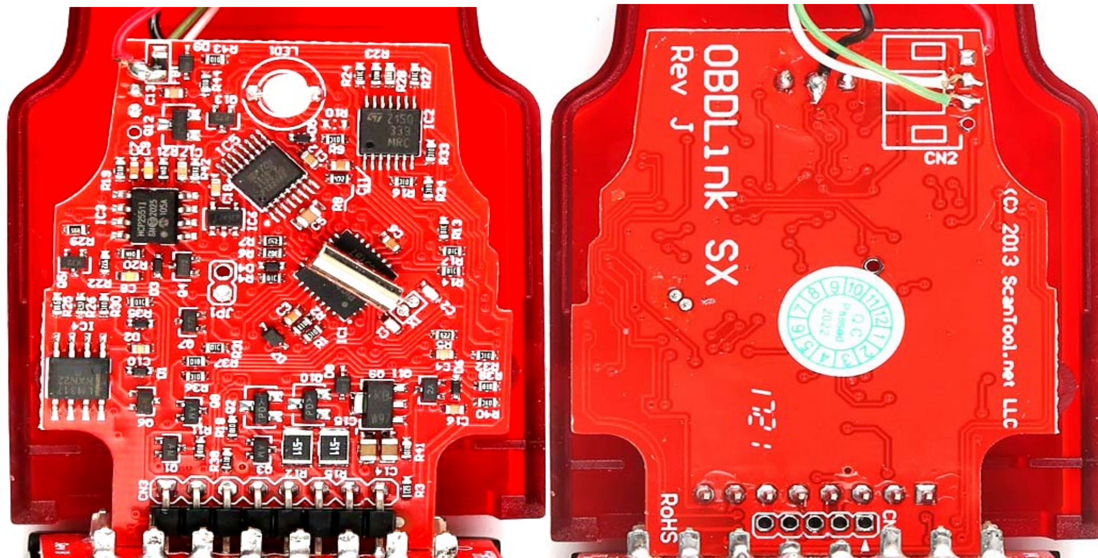
'Broadcast' data

Toyota apparently 'broadcasts' over the CAN bus without requests (ie timed or event-driven broadcasts) any sensor data needed by multiple ECUs. This must include all PIDs that appear under multiple ECUs, and those used in either car display. To reduce unnecessary data traffic on the bus, those parameters not needed by any other ECU may not be broadcast. I can not be sure of this without reverse engineering. As an aside, CAN cable breakage could interfere with use of shared data by a recipient ECU, without interfering with use of the same data in the source ECU.

If a user is interested in only 'broadcast' parameters, it should be possible for a (J2534?) reader and software to filter CAN bus traffic for the parameters of interest, at whatever rate they are updated onto the bus, with no increase in load on the bus. Any combination of hardware and software should suffice that can, at speed, read CAN data, filter for specified OEM PIDs, apply the associated formulae for human-readable results, and display/log these results. This seems ideal for fast diagnostics, with no risk while driving the vehicle.

For now, no commercial app/dongle combination seems able to read broadcast data for multiple PIDs. Safety aside, the best compromise may be to use (slower) OBD for a slick user interface, and change to J2534 (Techstream for Toyota) when higher data speed is needed (from one ECU).

SOME OBD OR J2534 READERS

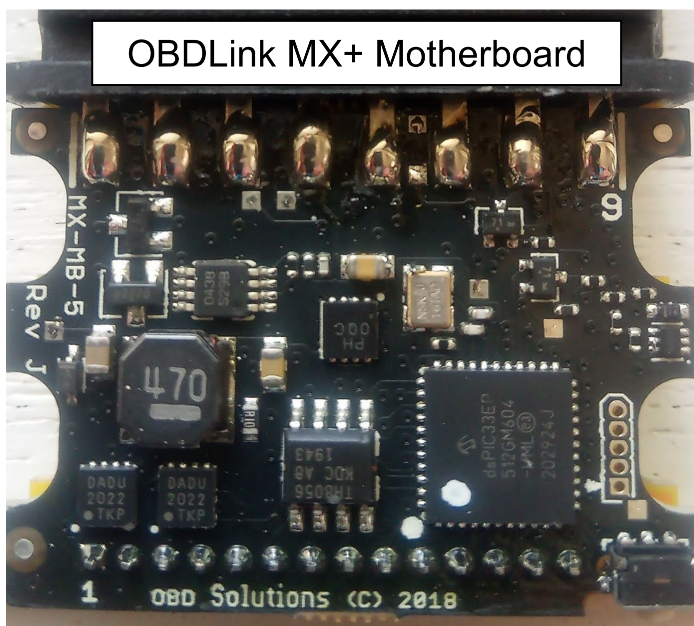


OBDLink SX (early Rev J variant) Pictures from Google Images

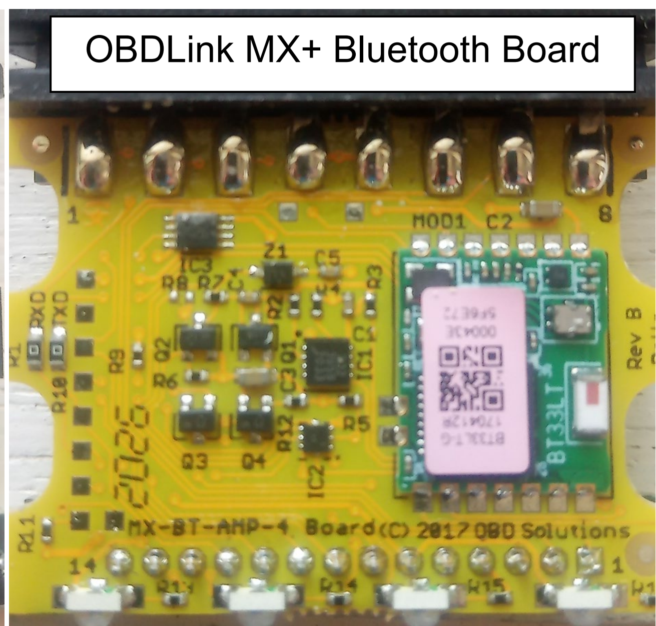
Roles of some chips on the board:

- LM317 is an adjustable voltage regulator.
- MCP2551 is an ISO-11898 compliant CAN signal transceiver. It converts the signals generated by a CAN controller (via the STN chip) to signals suitable for transmission over a physical CAN bus; and converts the data stream from the CAN bus to levels that STN needs.
- (ST / LM)339 is a quad differential comparator, used to convert K Line, L Line and J1850 line signals into a digital format suitable for the STN chip.
- STNxxxx is an OBD to UART (~RS232) interpreter, modified from a PIC chip. It provides bi-directional half-duplex communication with the car OBD port (16 MHz?, 16 bit).
- FTDI chips convert UART to USB (and vice-versa), requiring a driver in the connected host device.

The OBD software in a host computer provides processor and CAN controller functions: sending and reading messages and displaying human-readable results.

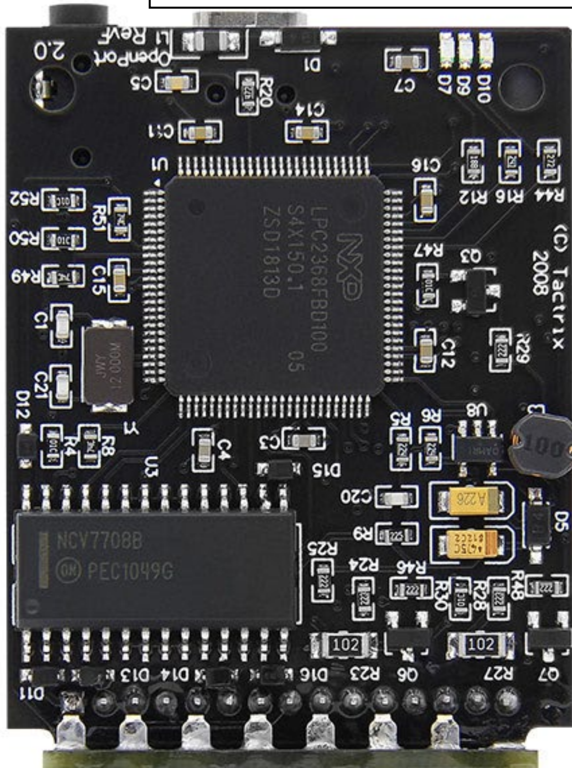


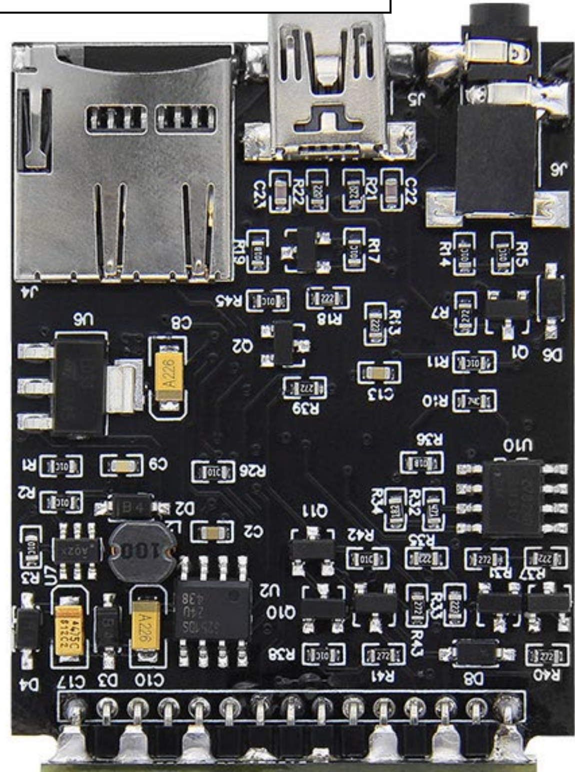
OBDLink MX+ Motherboard



OBDLink MX+ Bluetooth Board

Tactrix OP2 Clone (J2534) Pictures from OBD365

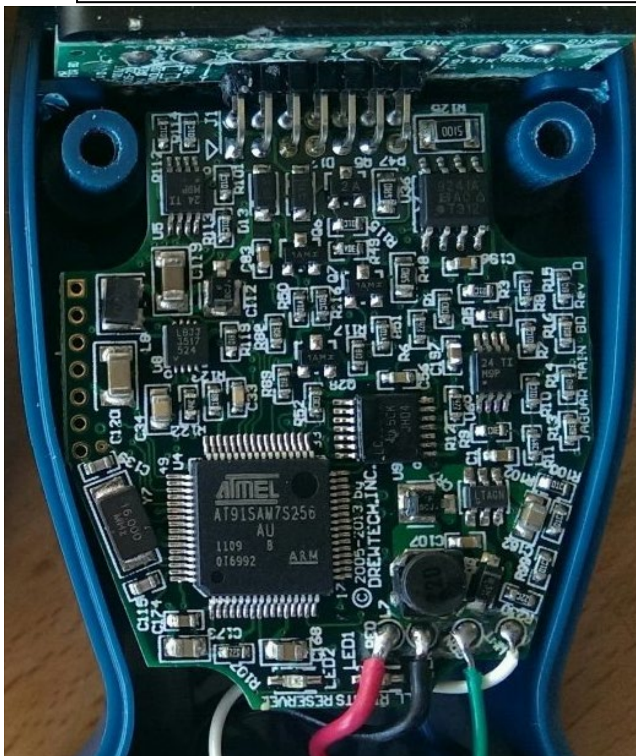


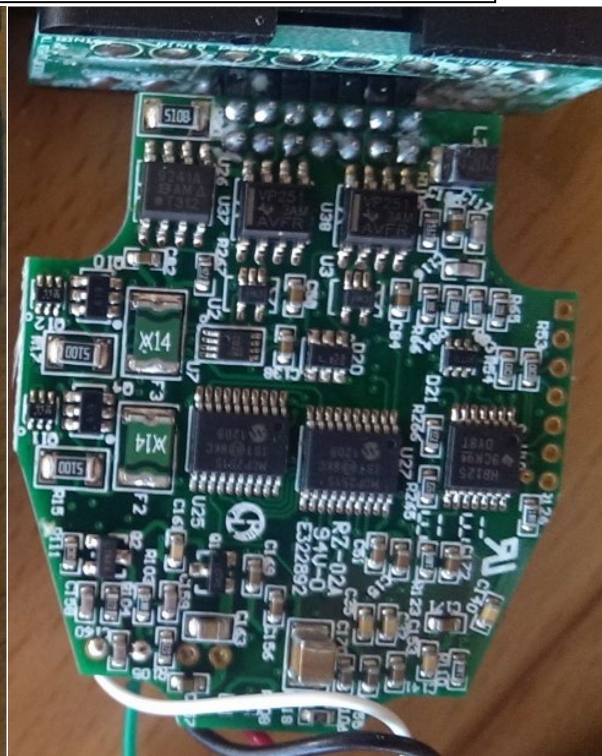


- LM317 is an adjustable voltage regulator.
- 6251DS is a CAN signal transceiver.
- NCV7708B is a switching multiple load driver (probably only used for ECU flashing).
- LPC2368FBD100 is an ARM7 TDIMI-S microprocessor (72 MHz, 512kb flash RAM / 58kb internal RAM, 32-64 bit) with integrated CAN and USB support.
- Some components (including SD card reader and mini-jack) are not needed for ECU diagnostics.

The J2534 software in a host computer provides processor and CAN controller functions.

Drew Mongoose Pro (J2534) Pictures from OBD365





OBDLink Sleep

This is relevant for both (a) car battery drain when SX is left in the OBD port between drives and (b) Android host device battery drain while SX remains connected to the USB-OTG port. Despite claims in OBDLink advertising and FRPM, the SX reader does not sleep (at least not while physically connected to USB-OTG with the OBDLink App in automatic connection mode).

To see what is going on, you will need to use an Android USB terminal. I used Serial USB Terminal 1.57 by Kai Morich (from the Google Play Store), installed on my phone rather than the tablet with the OBDLink App. The Terminal App needs some changes under Settings to work with SX and display as expected:

Baud rate: 115200. Receive Newline: CR. It may help to set Menu ⋮ > Data > Log (On) before use, to record the results in a file; or you can save the session later. You can also change the save location (from Android > Data > de.kai... > files) if you want to access files readily under Android.

The App must find and connect to the SX FTDI chip (via the OTG-USB cable). Then you can send:
ATI to see the ELM version.
AT@1 (or **STDI**) to see the device description.
ATZ if a reboot/reset of SX is needed (*eg* after sending new parameters).
STSLCS to see current ‘powersave’ settings.
STSLLT to see last sleep and wake triggers (since the last reset).
STSLUIT *sec* to set UART sleep time (in seconds).
STSLUWP *min*, *max* to set UART wake pulse width in microseconds. Increase ***min*** (above 15) and reduce ***max*** (below 30000) for more noise rejection. Default is 0-0 (any UART signal >20 ns).
STSLVL *sleep*, *wakeup* to change whether SX sleeps if the car battery drops too low (on or off).
STSLVLS *<|> volts|0xhhh, sec* to configure voltage level sleep trigger (default is <13.00, 3600 meaning below 13V for 3600 seconds).
STSLU *sleep, wakeup* to turn UART sleep/wakeup triggers on/off.
STSLVG *on|off* to turn voltage change wakeup trigger on/off.
STSLVGW *[+|-]volts, ms* to configure voltage change wakeup trigger (default is 0.20, 1000 meaning voltage changing by 0.2V in any direction, with one second between the samples).

If the response is the ELM version, the SX was asleep (it is woken by the first command, but does not echo or respond normally to that first command). If the response is ?, the command is not recognised (it was probably a typo). If changes are made, send ATZ then STSLCS to check that they ‘stuck’.

Defaults were:

```
CTRL MODE:  NATIVE
PWR_CTRL:   LOW PWR = LOW
UART SLEEP: ON,  600 s
UART WAKE:  ON,  0-0 us
EXT INPUT:  HIGH = SLEEP
EXT SLEEP:  OFF, HIGH FOR 3000 ms
EXT WAKE:   ON,  LOW FOR 2000 ms
VL SLEEP:   OFF, <13.00V FOR 36000 s
VL WAKE:    OFF, >13.20V FOR 1 s
VCHG WAKE:  OFF, 0.20V IN 1000 ms
```

This gave no sleep as reported by STSLLT. From what follows, it seems that this was due to OBDLink app in automatic connection mode on the tablet in the past; and sleep was just very slow (10 min) to start without the OBDLink App while the phone was awake and connected via the terminal App.

I tried `UART SLEEP: ON,   60 s.` With the phone Terminal connected, this at least gave SX sleep (UART signal) after the short (1 min) set delay, but also gave wake (UART signal). From the terminal responses and LED status, this wake is not from noise but rather due to the signal from the Terminal itself (unfortunately, time of triggers is not reported).

But with the OBDLink App 6.14.0.110 running in automatic connection mode on the Lenovo Tab M8 (even in the background and disconnected from the reader) SX sleep is blocked. If the OBDLink App is closed fully, SX sleeps when the android tablet sleeps (sometimes with no apparent delay). Presumably the App regularly checks for an OBD reader, and this activity keeps the SX reader awake, even if no connection is made. This may vary between host devices, depending *eg* on background app restrictions and doze implementation. With a meter in the line (and no App running), SX waited for the UART inactivity trigger. Using my phone (with no OBDLink app), SX waited for the UART inactivity trigger (with or without Terminal running).

To do this, the OBDLink App must send poll signals through the USB data wires, and the SX UART must sense these signals (and thus remain awake), even though there is no power through the USB power wires, and Android as well as the OBDLink App does not 'see' the SX unit on the USB cable. This is a little strange, but it is certainly possible. I think the OBDLink App developer can, and should, correct it. There is no reason for the App to poll until Android 'sees' a USB device. It is a waste of energy. But until this is corrected, we have to find another solution.

The solution (with OBDLink 6.14.0.110 and OBDLink SX 2 r4.2 firmware 5.10.1) is to use manual connection of the App to the SX reader (under Preferences > Communications). It needs additional 'Delay' and 'Interact touch Click' blocks in the Automate Flow to connect SX 'manually' (as in the Flow given above). Probably the 'power-on pulse' to the USB wakes SX (upon which the Automate Flow runs the path for USB device on). SX looses USB power and sleeps when the car 'ignition' is switched off (and Automate runs the path for USB device off).

Provided the SX LED has not been turned off in the OBDLink App, it is on when awake (red/green when dis/connected to the car OBD network) and off when asleep. Green flickers during OBD data transmission. As the SX LED is off in sleep mode, there seems no reason to inactivate unless it annoys.

USB phantom current dropped from 80 mA to 4 mA on SX sleep. Any change to the in-line amp meter setting triggered SX wake (it was probably seen as UART activity). Likewise, connection of my phone USB triggered SX wake. Sleep followed after the set time with no more UART activity. All phantom current from the M8 USB port to the USB device is blocked when power is off to the Y cable described above. SX does not need USB phantom current to keep settings or wake on a UART trigger when power is restored to the Y cable.

I also changed to `VL SLEEP:    ON, <13.00V FOR 600 s` to protect the car battery if for some reason SX continued to draw power from the OBD port during a long period in Park. I assume this will work without USB power, if ever needed (not tested). The car system voltage should not drop below 13v for 10 min while driving (+/- my car's LFP battery).

It may be wise to check 'powersave' settings (using STSLCS) after any firmware update.

Some cars use timed shut-down of various electrical circuits after the 'ignition' is switched off, to protect their battery. You may be able to see this in the LED of a device attached to the OBD port, or in an OEM PID for phantom battery drain. Some Android devices or accessory cables stop USB-OTG phantom power use (as for the Y cable mentioned above). These are useful back-ups, but it is preferable for devices like OBD readers to cut their own power use in 'sleep' mode.

As of 2026, the SX is no longer manufactured (though some new units are still in the retail pipeline). Similar considerations may apply to other USB units such as the OBDLink EX model.