



Harry's LapTimer

Documentation v1.14



Introduction and Scope

This paper is part of LapTimer's documentation. It covers all available editions LapTimer comes in – both for iOS and Android. In case functionality or wording differs, the document marks the respective sections using an iOS  or an Android  icon. For historical reasons, most snapshots are iOS pictures. However, as both lines of apps converge over time and will show only minor differences, pictures are not doubled in general.

For further documentation <http://www.gps-laptimer.de/documentation> is the first address for everything.

Although the aspects described in this document go far beyond the depth and detail you will see for an app, it misses other areas completely. So far now, please consider it as a series of technical papers.

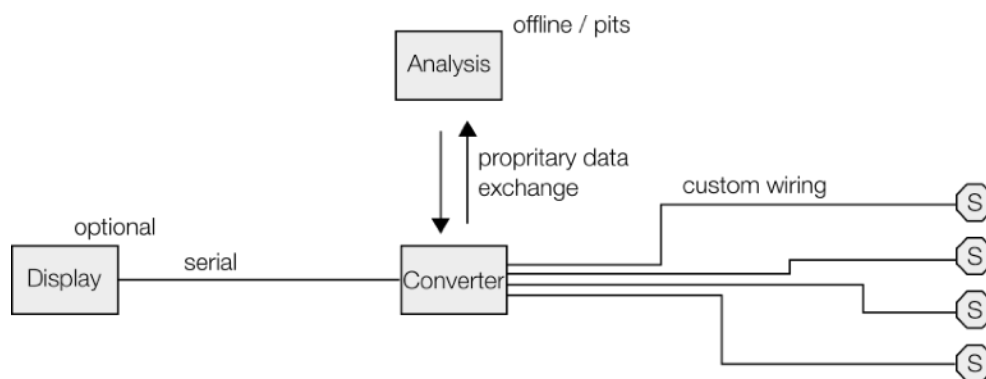
Engine Data (OBD)

Overview

Adding engine data (or more general, telemetry data) to LapTimer recordings will allow more effective data analysis and exciting additions to videos overlaid. Available for motor sport event TV broadcasts, access to this type of telemetry data is available for all LapTimer users today.

Access to engine or other sensors available in your vehicle will require some additional hardware. It is responsible to retrieve data from the vehicle and distribute it to your smartphone.

In former times, this has always been a big investment and labor intense task. Without a standardized infrastructure available in the car, it started with installing sensors. These are sensors measuring voltage matching pressures, speeds, fill level, or sensors measuring contacts digitally, or Hall effect sensors used to measure frequencies like engine revolutions. This analog or digital inputs needed to be feed into a converter providing the data serialized to an optional on board display unit, and store it permanently for later analysis (logging). These converters were expensive and limited in connectivity. Wiring all that stuff required days of work even with a small set of sensors (like rpm, speed, throttle, brake) added.

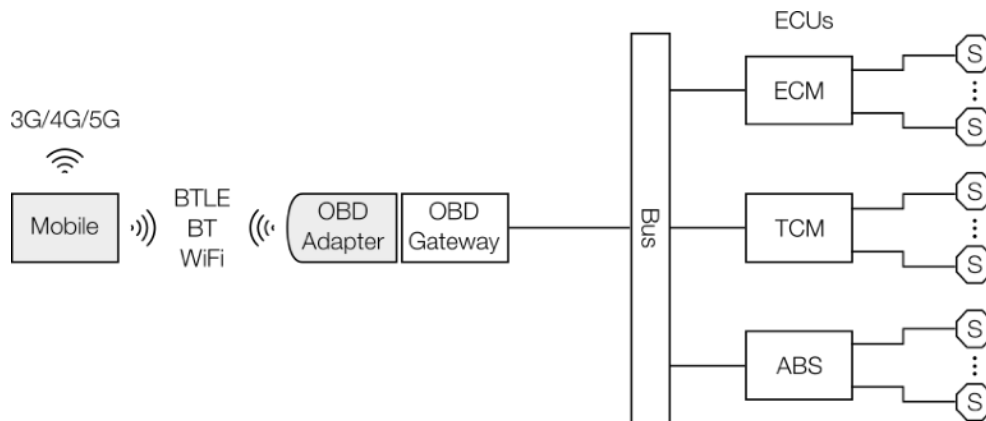


Accessing Vehicle Data Directly (All Units are Custom)

Practically, this meant engine data has not been available for 95% of all track drivers.

This type of configuration is still used by professional teams and cars / motorbikes not derived from road legal vehicles directly. But for most of us, the setup is using the standardized OBD-II port today. Introduced to allow manufacturer-independent emission testing and error analysis in workshops, it makes an inexpensive solution for real time measurements too. Due to its roots, it does neither provide the speed or flexibility of a custom setup, but it comes at 1 to 2% of the investment! OBD-II has been mandatory for US cars since 1996 and for European cars since 2001. NB: any OBD systems before OBD-II were manufacturer specific and cannot be used for our purpose (missing hardware, compatibility and communication speed).

To have a common understanding for the remainder of the section, let us have a look into the next chart. The main difference between a custom solution and OBD-II is that it uses the existing vehicle infrastructure. All we need to access engine data is a small OBD-II adapter connecting the smartphone to the interface / plug available already. From here, sensor data is retrieved through the vehicle's OBD gateway, one or more car bus systems, several control units, and hundreds of stock sensors built into your car. Reality is by far more complex than shown in the chart. A real car has not only an incredible amount of sensors, units, buses, and connection types. It will include different priority buses, routers, and additional connections propagated from the past. Furthermore, the vehicle network is an area in automotive developing extremely fast. We can expect to see completely different structures in the future!



Accessing Vehicle Data using OBD-II (Mobile and OBD Adapter are Custom)

Usually, the smartphone is connected to the OBD adapter using some wireless technology. The options available are Bluetooth using the serial protocol (BT SPP), Bluetooth low energy, or WiFi. Wired solutions (mostly USB serial lines) have become rare and are not supported currently. The different wireless connection technologies have their pros and cons:

Bluetooth SPP is the best solution if available for your smartphone. It allows the connection of other Bluetooth accessories at the same time, has sufficient bandwidth, and is easy to use and stable. For iOS 🍏, Bluetooth SPP can be used by Made for iPhone certified adapters only. This reduces available options significantly.

Bluetooth Low Energy (BTLE) is even easier to connect but provides low bandwidth. In case communication between smartphone and adapter is not optimized beyond the standards discussed below, BTLE is becoming the bottleneck for engine data access - at least for fast car busses. BT LE can be used for both Android 🌿 and iOS 🍏.

WiFi is available for all smartphones and the fastest solution. But there are two disadvantages: WiFi connections tend to drop when used with mobiles in changing locations. From our support channel we know this is a frequent issue. Furthermore, a smartphone can connect to one WiFi at a time only. So an action cam cannot be controlled by LapTimer at the same time.

This is communication on technical level. The app and the OBD adapter use some communication protocol to talk to each other. The best known protocol is ELM 327 introduced by [ELM Electronics](#) in 2005. It defines a lingo allowing an app to ask for parameters (named channels) like engine rpm and receive an answer once the adapter has retrieved the data from the bus and sensors. The ELM 327 lingo is purely about communication between app and adapter. While ELM 327 is the ad hoc standard, there are several “proprietary” application level protocols used by adapters too. Additional protocols supported by LapTimer are GoPoint’s binary format or the more general protocol used by Autosport Labs (RaceCapture devices). We may add more proprietary protocols in the future but have changed our policy to allow 3rd parties to provide their own LapTimer plug in instead. See our developer program ([HarrysLua](#)) for this.

The OBD adapter is plugged into the OBD II socket installed somewhere under the steering wheel / driver side. The OBD gateway provides some electrical decoupling and allows access to various lower level bus types standardized by OBD II. We will come back to bus types when discussing options to make communication faster.

The amount of data available from the gateway depends on the car manufacturer, the model, and model year. There is an unhealthy trend in automotive to restrict data to a small set mandatory for emission testing. A typical car will provide around two dozen parameters over OBD II only while it has hundreds available. A typical sample is oil temperature. It is not mandatory for emission testing but is standardized by OBD II. Ford engines deliver this parameter, while e.g. Porsche keeps it as a secret.

The bus (in reality, a car will have several busses and bus types) replaces the custom wiring we have seen above. It is the transport level that allows control units and gateways to exchange data across the system. All modern cars use some kind of CAN bus (ISO 15765-4). The adoption of CAN started around 2006. Any car starting 2008 will feature a CAN based bus. Older protocols supported by OBD II are SAE J1850, ISO 9141-2, and ISO 14230-4 KWP in various versions. There is a significant speed difference between CAN and non-CAN bus types. While one can pull 50-150 parameters per second using CAN, the old protocols will rarely deliver more than 10. In case you have a non-CAN bus vehicle, the bottleneck for update rates will always be the bus, not the adapter or the wireless connection technology used.

Once a compatible OBD adapter is connected to LapTimer, racing relevant channels (see table below) will be retrieved and stored like GPS and acceleration data. This is done for all LapTimer editions including Rookie. Opposed to this, display of data depends on the LapTimer edition: Petrolhead will add a gauge to video overlays if engine data is available, furthermore, some additional charts are available. GrandPrix adds an extra in depth display of engine data ([Engine View](#)) as well as further charts for analysis plus a second gauge in video overlays.

Engine channels predefined (as of LapTimer v22) are listed below. Channels standardized by OBD II (but not necessarily available for every vehicle, see above) have a mode \$01 or mode \$09 parameter ID (PID). The combination from mode and PID is passed to OBD adapter and ECUs when asking for a parameter. Some channels are calculated. This means the value is derived from other channels and e.g. the vehicle specification. All modes and PIDs are given in hex characters and a leading \$ here. LapTimer will display metric or imperial units for countries like the U.S. In the table below, we show metric units as used by the OBD standard and LapTimer's database.

Channel	OBD Mode PID	Unit	Frequency	Comments
Vehicle Identification Number	VIN \$09\$02	17 chars	once	
OBD Standard	OBDSUP \$01\$1C	-	once	
Bus Type	-	-	once	retrieved from adapter
Engine Type	\$01\$01	-	once	1 = Otto, 2 = Diesel
Fuel Level	FLI \$01\$2F	%	low	
Battery	-	V	low	retrieved from adapter
Engine Coolant Temperature	ECT \$01\$05	°C	low	
Engine Oil Temperature	EOT \$01\$5C	°C	low	
Intake Air Temperature	IAT \$01\$0F	°C	low	
Catalyst Temperature	CATT \$01\$3C	°C	low	
Mass Air Flow	MAF \$01\$10	g/s	medium	
Manifold Air Pressure	MAP \$01\$0B	kPa	high	
Engine Revolutions per Minute	RPM \$01\$0C	1/min	high	
Throttle Position	TPS \$01\$11	%	high	alternatives \$47-\$4B
Wheel Speed	VSS \$01\$0D	km/h	high	
Fuel Consumption	calculated	l/100 km	-	mpg for imperial units
Power Developed	calculated	kW	-	
Torque Developed	calculated	Nm	-	
Gear	calculated	-	-	uses vehicle definitions

Standard Channels Table

Other channels are supported by LapTimer but do not have a standard OBD mapping and will not be used in case they are not set. To set such a channel, users can provide so called custom PIDs specific to their vehicle. As an example, for Fords, the mode \$22 PID \$3201 parameter will deliver the current steering angle. As an alternative, these channels can be set using non-OBD sensor input using LapTimer scripting (see Developer Program / [HarrysLua](#)).

Channel	Unit	Comments
Odometer	Kilometer	
Gear	-	-1 for rear, 0 neutral etc.
Oil Pressure	kPa	
Brake Pressure	kPa	
Wheel Speed Rear Left	km/h	all of these need to be set or not
Wheel Speed Rear Right	km/h	
Wheel Speed Front Left	km/h	
Wheel Speed Front Right	km/h	
Yaw Rate	°/s	
Steering Wheel Angle	°	turned clockwise: positive value
Steering Wheel Rate	°/s	

Extended Channels Table

For more informations on adapters and technologies, please visit www.gps-laptimer.de and navigate to LapTimer's Android and iOS compatibility / accessories pages. It has an always up to date list of compatible adapters and recommendations.

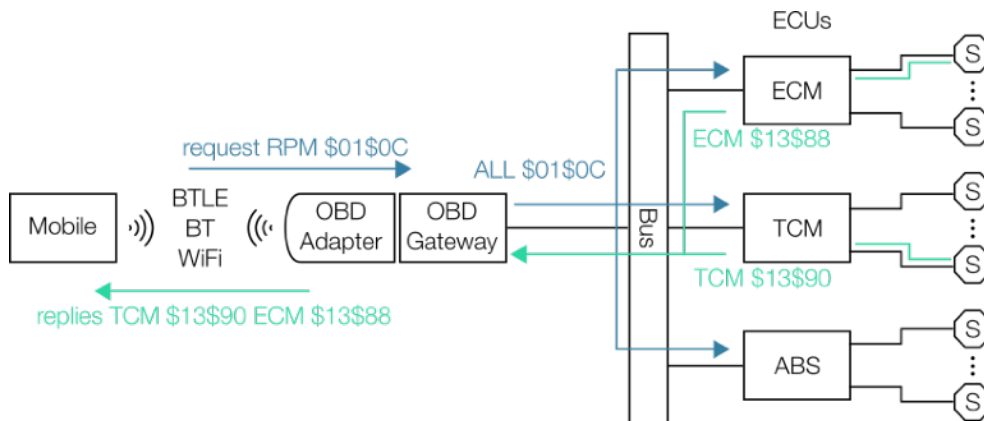
The following sections give additional technical insights. They are targeted to technical oriented readers only.

Optimizing Update Rates

Many operations with adapters and car busses are serialized somehow. To get a high speed for channel updates, we need to make every operation as fast as possible, perform only operations necessary, and try to do things in parallel if feasible. The ELM327 protocol is not well designed to achieve high rates. The primary reason for this is, that an app needs to poll for new parameters constantly. Instead of streaming data that has changed, an app needs to ask for a parameter, wait, receive the result, process it, and continue with the next parameter. Often, the adapter will send data that has actually not changed.

But LapTimer's ELM parser is an extremely powerful, flexible, and feature rich solution. It went through several redesigns and refinement cycles. It utilizes all known technics to make engine data transfer through OBD II ports as fast as possible. In fact, we believe it is by far the fastest solution in the market. This section will describe the concepts and levers available. It will give you insights in how you can tune the system for your car and adapter. It will show where it doesn't make sense to tweak the system too.

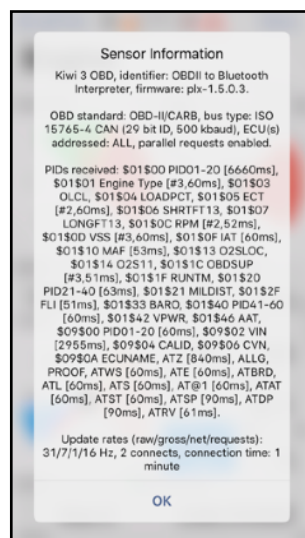
To understand the overall process going on for each dialog, please have a look into the sketch below. It shows what happens when the app requests the engine's RPM channel. The request is broadcasted to all ECUs in the system here, with the Engine Control Module and the Transmission Control Module answering (with slightly different values measured). The whole cycle contributes two PIDs to the update rate counter.



Full Cycle Requesting Channel RPM with two ECUs Answering

There are four different update rate notions in LapTimer, they are measured in Hz (number per second): **raw**, **gross**, **net**, and **requests**. The **raw** rate is the rate at which individual pieces of data are coming in. For engine data, this is the number of channel updates we get per second. In OBD terms, this is the **number of PIDs per second**. The rate shown to users by default is the **gross** rate. For engine data, this is the **number of full channel sets we receive per second**. The net rate is similar to the gross rate, but it does not count repeated channel sets. So if an OBD adapter updates data from sensors only once per second but can deliver 5 full sets per second to LapTimer, it will have a gross rate of 5 Hz, but only 1 Hz net. For most adapters, net rate is the same as the gross rate. The **request** update rate is the frequency at which LapTimer asks the adapter for something. We will use the different notions when discussing the levers to improve update rates.

In case you are curious about the rates achieved with your configuration, there are several options you can choose from. In [Administration ▶ Sensor List](#) you will see the **gross** update rate and all update rates in case you have changed LapTimer's Display option to **Expert**. In the detailed views on GPS and OBD ([GPS View](#) and [Engine View](#)), you will see the respective sensor's rate. Touch the **Rate** gadget to toggle between **gross** rate, high precision **gross** rate, and raw rate (**PID** rate).



Full Details on an ELM327 Compatible OBD Adapter

Finally, most sensors in [Sensor List](#) come with an in depth detail view ([Info](#)). The snapshot above is showing a sample. Details start with information on the adapter itself. Next, we see the car connected implements OBD-II as defined by the CARB standard. The bus type used is a modern 29bit CAN bus running at 500baud. This is about the fastest bus you find currently. Later, we will see that this bus types has features we can use to get drastically increased update rates.

Another important information is, that the adapter is configured to ask [ALL](#) ECUs available for replies, and that requests are issued in parallel. These are two levers to improve performance too.

The list following is showing all the parameters reported as available by the adapter. They follow the common form

`$<MODE>$<PID> <NAME> [#<REPLIES>, <TIME>ms]`

The `<MODE>` and `<PID>` placeholders are the hex representation of the mode / service and parameter ID of this entry. An extensive and well maintained list of known PIDs is available here: https://en.wikipedia.org/wiki/OBD-II_PIDs. The `<NAME>` is an abbreviation for the OBD parameter / channel references by LapTimer. In case the item is not only available, but actually used by LapTimer, we get some statistics for this parameter in brackets. In case more than one ECU sends a reply when requesting the parameter, the number is given by `<REPLIES>`. The more important item is the average reply time `<TIME>` seen for this parameter. Given in milliseconds, it can be as low as 10 or 20 for a fast CAN bus and a fast wireless connection, and 100 and more for slow busses. `<TIME>` is the time needed for the full request cycle shown in [Full Cycle Requesting Channel RPM with 2 ECUs Answering](#). There are some complex requests like those for PIDs available, which will take a very long time. Typically, these are requests issued only once.

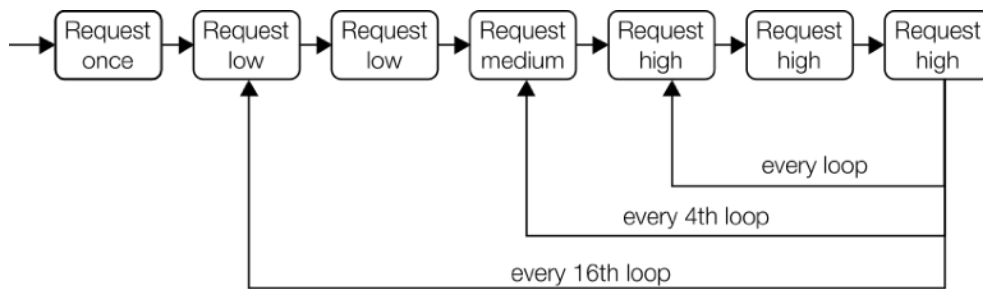
Entries starting with "AT" are from ELM specific initialization commands and can be ignored.

The last section is giving us the rates in detail. In the sample shown, the adapter delivers 31 PIDs per second. This is kind of next to the limit achievable with a Bluetooth Low Energy adapter. This 31 PIDs allow LapTimer to built 7 complete engine data entries, i.e. a gross rate of 7 Hz. This value compares directly to the rates we see for GPS adapters: 1, 5, 10, 20 Hz. As the car idles, the net rate is lower currently. Once we play with the throttle, it will go up to 7 Hz too. Finally, LapTimer is issuing 17 requests each second. The reason this differs from the raw rate is because we have [parallel request enabled](#) and because several parameters requested are returned more than once. More on this below.

The number of connects shown can be used to understand whether the adapter suffers from intermittent disconnects. This counter will go up by one every time the adapter re-connects completely. You will see an update when sending the app to background and pull it forward again too. We did this when creating the snapshot. The duration of the last connection is shown as the last item. Disconnects during operation will reduce the update rate significantly. The whole initialization (10 seconds and more) will be repeated...

Adjusting Request Frequencies

Engine data parameters have different volatilities. As an example, the fuel level or coolant temperature will not change fast, while turbo boost (MAP) or engine revolutions per minute will change frequently. This behavior can be used to use available bandwidth and processing capacity for channels worth requesting frequently. The schema LapTimer uses is assigning one of the frequencies *once*, *low*, *medium* and *high* to every channel. A channel marked *once* will be requested only once and the result is stored permanently. Other requests are issued regularly. Those marked *high* will be issued most often:



Prioritizing High Frequency Requests by Loop Selection

In case frequencies are defined matching their volatility, we will get a higher gross update without losing relevant information. To recap the introduction on the different rate notions: the gross rate is actually the number of loops performed per second. Both short / high frequency loops and long / low frequency loops are going into this measure. The remaining question for the frequency topic is if this is something we can customize? The answer is yes, at least for custom PIDs (i.e. requests added by the user) and in case you set up your own parser customization using [HarrysLua](#) scripting. While there are no dedicated settings available to change the frequency for standard parameters, you can define a custom PID with standard settings (mode \$01/\$09) and customized frequency, which will replace the standard channel setting as a side effect. See section [Custom PIDs](#) for the details.

Skipping Channels

Questions whether it is possible to skip channels are quite frequent. Once you realize “the more you ask, the slower the overall update rate gets” this is the consequent next step. Like for the former section, LapTimer has optimized this topic already. To start with, LapTimer focusses on racing relevant channels only. There is a lot more you can ask an OBD adapter than what is listed in the [Standard Channels Table](#). So LapTimer is pretty picky here. Next, most vehicles do not deliver all channels in scope. To not request something we do not get an answer for anyway, LapTimer asks the OBD adapter for parameters supported initially. In addition, in case a parameter is not delivered although it should, it is disabled after 3 unsuccessful attempts and will be skipped in further request loops.

Assuming we find a channel we want to skip on top of this, what is the effect? This depends a lot on the frequency set. Looking into an example, we have a car answering parameters ECT, IAT, CATT, MAF, RPM, TPS, VSS at 20 ms in average. From the Standard Channels Table you can see the temperatures ECT, IAT, CATT are requested at low, MAF at medium, and RPM, TPS, VSS at high frequencies. This results in this repeated looping:

Loop #	ECT	IAT	CATT	MAF	RPM	TPS	VSS	Time
1	X	X	X	X	X	X	X	140 ms
2	-	-	-	-	X	X	X	60 ms
3	-	-	-	-	X	X	X	60 ms
4	-	-	-	-	X	X	X	60 ms

Loop #	ECT	IAT	CATT	MAF	RPM	TPS	VSS	Time
5	-	-	-	X	X	X	X	80 ms
6	-	-	-	-	X	X	X	60 ms
7	-	-	-	-	X	X	X	60 ms
8	-	-	-	-	X	X	X	60 ms
9	-	-	-	X	X	X	X	80 ms
10	-	-	-	-	X	X	X	60 ms
11	-	-	-	-	X	X	X	60 ms
12	-	-	-	-	X	X	X	60 ms
13	-	-	-	X	X	X	X	80 ms
14	-	-	-	-	X	X	X	60 ms
15	-	-	-	-	X	X	X	60 ms
16	-	-	-	-	X	X	X	60 ms
								1100 ms

Because of the 20 ms reply time, we get 50 PIDs per second, i.e. a raw rate of 50 Hz. For the gross rate, we get 16 sets of data (with changing update frequencies) in 1.1 seconds, which mean 14.5 Hz (14.5 sets per second). We compare this to the following scenarios: skip a low, a medium, or a high frequency channel.

Scenario	Raw Rate	Gross Rate	Improvement
Standard	50 Hz	14.5455 Hz	-
Skip CATT (low)	50 Hz	14.8148 Hz	1.9 %
Skip MAF (medium)	50 Hz	15.6863 Hz	7.8 %
Skip VSS (high)	50 Hz	20.5128 Hz	41.0 %

This means skipping one of the temperature channels makes no sense, skipping a medium frequency channel neither. Skipping a high frequency channel however has a significant impact, especially in case the amount of high frequency channels is low already. But what happens when skipping a channel LapTimer retrieves usually? For the temperatures, it means you limit your options to look for engine problems and will not get alerts in case one of the channels exceeds warning and error thresholds. This may be acceptable but has little to no effect as seen above. Removing MAF or MAP removes the option to calculate power, torque, fuel consumption. This will not be acceptable for most users and add little improvement. For RPM and TPS, the negative consequences are the same like for MAF and MAP, RPM is required for gear calculation in addition. So removing these is probably not acceptable either, although update rate improvements are high. The remaining skip candidate is VSS. Removing VSS has the big improvement effect shown above. In addition, LapTimer will replace VSS (wheel speed) by GPS speed to not lose functionality. But there is certainly a reason VSS is included by default: gear calculations are more precise (when shifting) using wheel speed, and wheel speed is a lot better to analyze performance on track compared to GPS speed. Wheel speed is reacting faster to changes and will not miss min and max values like GPS tends to do.

To skip a channel, LapTimer has an [Expert Setting](#) named [Exclude PIDs](#). It needs to be a hex character sequence with a multiple of 4 length. Each 4 character portion is a mode / PID combination. We do not have dollar signs here.

To skip CATT and VSS, set [Exclude PIDs](#) to

013C010D

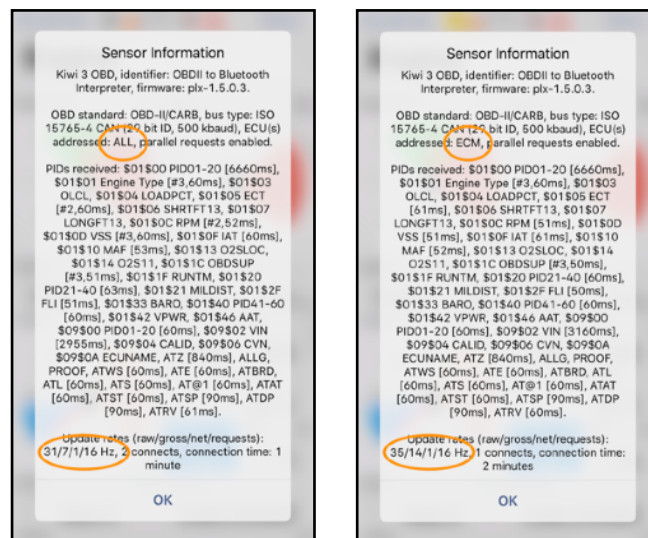
in [Administration](#) ▶ [Settings](#) ▶ [Expert Settings](#) ▶ [OBD Tweaks](#).

Addressing Specific ECUs (v22)

By default, OBD adapters will broadcast any request to the vehicle network. It is well possible more than one ECU will deliver a result and return it to the app. A sample is shown in [Full Cycle Requesting Channel RPM with 2 ECUs Answering](#) where the engine revolutions per minute are returned by both the [Engine Control Module](#) (ECM or PCM) and the [Transmission Control Module](#) (TCM). Broadcasted requests are named “functional calls” in a bus infrastructure. While broadcasting is convenient (we do not need to know to which ECU to talk), we are mostly not interested in receiving the same piece of data multiple times.

In case we know the ECU to talk to, it is possible to direct a request directly to that ECU. This type of requests are named “physical calls” because they address a physical unit. The effect on performance is influenced positively: all units involved in the full request cycle need to process less data. Bandwidth on the vehicle's bus and wireless connection between smartphone and adapter is not wasted and can be used for a higher rate.

LapTimer has an [Expert Setting](#) to use physical calls instead of functional calls. The setting is a global switch and does not allow a “by parameter” configuration. As most parameters we are interested in are available from the [Engine Control Module](#), this will be sufficient for most users. The effect will depend a lot on the setup, the type of wireless connection, and the bus type. The snapshots below show the same adapter using functional calls (ALL) and physical calls (ECM).



Sensor Results Using Functional (Left) and Physical (Right) Requests

Looking into the results, we see the raw rate is slightly faster (raw rate 31 Hz ▶ 35 Hz). But the main effect is that the raw throughput is used a lot better to receive full sets of engine data ([gross rate 7 Hz ▶ 14 Hz](#)). But please be warned, your mileage may vary a lot. In case your main ECU does not provide all parameters required by LapTimer, the option cannot be used.

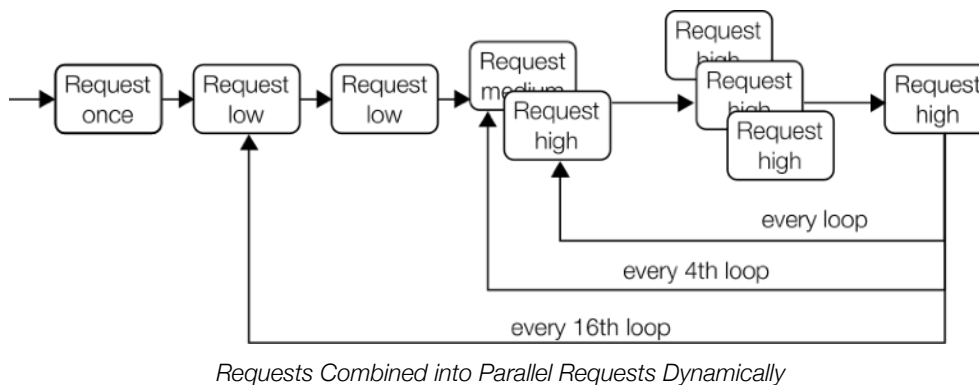
Selecting the ECU address can be difficult because physical ECU addresses are not standardized for all bus types. The place to select it is in [Administration](#) ▶ [Settings](#) ▶ [Expert Settings](#) ▶ [OBD Tweaks](#) / [ECUs Addressed](#). Except for the very old ISO 9141-2 bus, LapTimer is coming with some defaults. They can be used by entering one of the symbolic names [ALL](#), [ECM](#) or [PCM](#), [TCM](#), or [ABS](#). These defaults will work reliably for CAN bus types only. In case the symbolic name doesn't work, you need to enter a three byte hex value like

C110F1

for ISO 14230-4 KWP busses. The 10 is the actually address (for this bus type) you need to get your hands on. But just start by trying [ECM](#) instead of [ALL](#). Please give some time or even reset the adapter to make this new setting take effect.

Parallel Requests

From the discussion in [Adjusting Request Frequencies](#) it is obvious, that serialization of requests and replies is the major bottleneck in ELM 327 dialogs. So the question is, if there are options to parallelize the request / response pattern. In fact, there is an option to do this in case you are lucky and have a CAN bus type in your car. CAN allows issuing several request at a time, allowing the system to perform some operations in parallel. This pattern is repeated in the ELM 327 lingo and utilized by LapTimer. What actually happens is displayed in the diagram below. Packing multiple requests into one is implemented as a highly dynamic and versatile algorithm in LapTimer. It depend on whether two or more planned requests can be combined (compatible answering behavior) and if the combined requests and replies fit into the buffer sizes available. Effectively we get processing loops like shown in the next figure:



Once parallel requests are enabled, the effective raw rate is nearly doubled. Especially when combined with selecting a physical ECU (which allows easier composition of requests), the highest performance possible can be achieved.

In LapTimer, parallel requests are enabled by default for CAN bus types. They are always disabled for non-CAN busses. In case you are facing issues with parallel request, you can disable them completely using [Administration](#) ▶ [Settings](#) ▶ [Expert Settings](#) ▶ [OBD Tweaks / Disable Parallel Requests](#).

Adaptive Timing

The last feature we want to discuss here is the so called “Adaptive Timing”. This concept is a standard feature of ELM 327 adapters but improved by LapTimer. The background story is a requirement from asynchronous processing.

“Asynchronous” means one party sends a message to another but does not know when and whether and how many replies it is getting. This is exactly the OBD adapter situation: it sends a requests to the bus and needs to wait for an answer. Once it gets an answer, it needs to decide if this is everything or if another reply is coming in later, which needs to be joined with the first answer. In case no answer is coming in, it needs to decide at some point in time that it makes no sense to wait any further. Any time passed since the last answer is lost time in this case-not good if you want to process as many loops as possible.

The ELM 327 chip tries to make a valid guess on how many replies will come in. In addition, it has a timeout setting which stops any waiting. This mechanisms however do not seem to work perfectly. So instead of delegating this type of decision,

LapTimer monitors the system and starts telling the adapter how many answers are expected after a short learning phase. Compared to ELM's built-in mechanism, LapTimer achieves approximately 30% better performance - which is a lot.

For old bus types and for some ELM knock offs, LapTimer's hinting will not work well however. This problem has been seen for early BMW models in particular. In case you see dropping connections or extremely volatile update rates, try disabling LapTimer's [Adaptive Timing](#).

Here are the options available in [Administration](#) ▶ [Settings](#) ▶ [Expert Settings](#) ▶ [OBD Tweaks](#) / [Adaptive Timing](#):

[Optimized](#): use LapTimer's Adaptive timing, the fastest option available and the default

[Enabled](#): use ELM's built in [Adaptive Timing](#), highest compatibility but the slowest option

[Aggressive](#): use a more aggressive version of ELM's Adaptive Timing

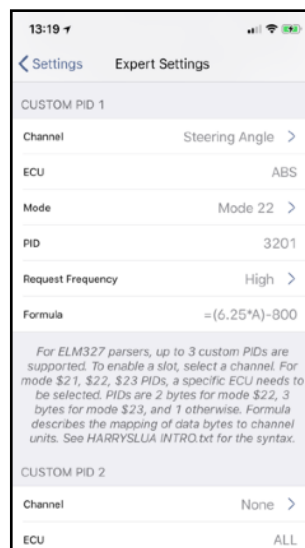
Custom PIDs (v22)

As written in [Engine Data Overview](#), only a small fraction of data available on a vehicles bus is accessible using parameter IDs defined by the OBD II standard. Even parameters standardized by OBD II are not necessarily mapped to the prepared slots but hidden by the car manufacturer. So this is an in-transparent and difficult to handle topic.

For some manufacturers, there are known service IDs utilized by the car manufacturer to access hidden parameters. LapTimer tries to make these parameters accessible by offering the option to add custom PIDs. Properly defined, this custom PIDs can be used to feed data into both standard (see [Standard Channels Table](#)) and extended channels (see [Extended Channels Table](#)) and make them usable by LapTimer. As no official and public documentation is available, this feature needs to be considered experimental. We appreciate any hint from our user base and we are happy to extend the Custom PID feature in case it is not powerful enough to access your car's data.

To understand how a known custom PID can be utilized, let us have a look into an example: for some Ford cars, the steering wheel angle is available as a mode \$22 parameter. Ford uses 2 byte PIDs which is \$3201 for the angle. Most custom PIDs are not answered when requested using functional addressing / OBD mode. Instead a physical ECU needs to be addressed. For this custom PID, it is the ABS ECU which provides the parameter. Finally, the reply is a one byte value "A", which can be mapped to an degree angle by calculating $(6.25 \cdot A) - 800$.

This is how it is entered into LapTimer's Expert Settings:



Custom PID in Expert Settings (Sample)

Channels available for selection are those listed in the [Standard](#) and [Extended Channels](#) tables. In case you select a channel defined already, the existing definition is replaced by the custom PID definition. In case the Custom PID should be disabled, please select [None](#) as channel.

Information on [ECU](#) addressing can be found in section [Addressing Specific ECUs](#). For this PID, we use the symbolic name [ABS](#) which will be mapped to the actual bus specific address required.

Modes can be one of the two standard modes \$01 and \$09, or one of the extended modes \$21, \$22, and \$23. In case other modes are required, please let us know.

While standard parameters always use one byte for the PID, custom PIDs can use an arbitrary number. To signal the width in bytes, please always precede PIDs with the necessary number of zero digits. For \$3201 we do not need this because it is two bytes wide already. If e.g. a custom PID requires a three bytes PID and the PID is \$FF, you need to enter 0000FF into the [PID](#) row.

Steering wheel angles change fast, so we select a [High](#) frequency for request looping (see section [Adjusting Request Frequencies](#)).

[Formula](#) is the expression to map bytes included in an answer to the value LapTimer should process. Let us have a deeper look into the syntax of this formulas. Before we start, let us re-check what we need to do: we need to map a one byte value to the [Extended Channel Steering Angle](#) channel which expects a degree value (see the table again). This formula is $(6.25 \cdot A) - 800$. A byte comes with values between 0 and 255. So the result for the formula when setting A to the min value 0 is -800 (degree). When setting A to 255, we get $(6.25 \cdot 255) - 800$ which evaluates to 793.75 degree. When entering 128 (which is the middle value), we get 0 degree (i.e. driving straight). As a result, the formula evaluates roughly to the range $\pm 800^\circ$ (which is more than 2 full turns to each side).

The syntax for the formula will be familiar to all C programmers:

```
formula           = [ "=" ] conditionalexpression
conditionalexpression = equalityexpression { "?" equalityexpression ":" equalityexpression }
equalityexpression  = relationalexpression { "=="|"!=" relationalexpression }
relationalexpression = expression { "<"|">"|<="|>=" expression }
expression          = [ "+"|"- " ] term { "+"|"- " term }
term                = exponent { "*"|"/" exponent }
exponent            = factor { "^" factor }
factor              = parameter|number|"(" conditionalexpression ")"
integer             = "0"|...|"9" { "0"|...|"9" }
number              = integer [ "." { "0"|...|"9" } ]
parameter           = "A"|...|"Z" { "A"|...|"Z" } [ bitqualifier ]
bitqualifier        = integer
```

Beyond what we are all used to, formulas can come with conditional expressions like $=A>B?A:B$ which means the result is the bigger of the two parameters **A** and **B**.

Parameters available are **A** to **Z** followed by **AA** to **AZ**. **A** is the first byte of a response, **Z** is the 26th byte, **AA** is the 27th and **AZ** is the 52nd. Parameter can have a bit qualifier too, so **B2** e.g. means bit 2 from the **B** byte value. Bits are counted from right to left starting with 0 (i.e. a byte is **B7 B6 B5 B4 B3 B2 B1 B0**).

Parameters must not address bytes not available. In our example, we get only a one byte reply, so using **B** or any further parameter will result in an error. LapTimer uses parameters to understand how many bytes are expected in a response too. Let us assume a custom PID returns 4 bytes but we want to map the result simply to the second byte received (parameter **B**). To hint LapTimer there are actually 4 bytes to expect, you may set the formula to $=B+0 \cdot D$.

Using custom PIDs will degrade update rates: they will be used for parallel requests rarely as only requests using the same mode can be combined. Furthermore, switching the ECU address means two extra request needs to be sent to the ELM 327 adapter: if we have selected [ALL](#) or [ECM](#) as standard ECU to address, LapTimer will send a request to switch to [ABS](#) ahead of requesting the Steering Angle Channel, and send a request to switch back to [ALL](#) or [ECM](#) once the response has been received.

Configuration of Custom BTLE Sensors

Bluetooth Low Energy (BTLE) devices do not have a standard interface and LapTimer will need you to set four parameters in [Expert Settings](#). You can (and actually need to) investigate all of these parameters by using apps like **LightBlue** available on iOS and Android. Connect you device / sensor and browse through the data shown.

The items that need to be set for a Custom OBD BTLE sensor are as follows:

Peripheral Name: this parameter is actually a pattern that needs to match the device's name as advertised; a pattern like **Kiwi ..** will match devices named **Kiwi 09** or **Kiwi AH**, i.e. a device name starting with **Kiwi**, a space, and two arbitrary characters. You do not need to use wild cards (actually any regular expression can be used) like described here, and enter your device's name directly.

Service UUID: a Universally Unique Identifier (UUID) identifying the service to exchange data between app and device. LapTimer accepts three formats for UUIDs: a four digit version like **FFE0**, an eight digit version, and a fully qualified UUID like **00000002-0000-4B49-4E4F-525441474947**. You can identify the service UUID by looking for the [Read](#) and [Write Characteristics](#) described below. The service will group this characteristics in **LightBlue**.

Read Characteristic UUID: a UUID identifying the GATT / interface used to read data coming in from the sensor. You can identify this characteristic in **LightBlue**: it will come with properties named **Read** **Notify**. The UUID can be entered using the three version described above.

Write Characteristic UUID: a UUID identifying the GATT / interface used to write data to the sensor. Identify this characteristic by looking for property **Write**.

Some devices will use the same characteristic for read and write access. Properties will be **Read** **Notify** **Write** in this case. For configurations like this, enter the same UUID for both the read and write characteristic.

Once this items are set, disconnect **LightBlue** and goto LapTimer's [Sensor List](#). In case the device shows up as connected, the [Peripheral Name](#) has been set correctly. In case you get an update rate shown, [Service](#), [Read](#), and [Write](#) UUIDs are fine too.

Harry's Developer Program

Scripting with HarrysLua

Introduction

The section gives an introduction into **HarrysLua**, a scripting language embedded into Harry's apps and based on Lua 5.3 (see lua.org for reference and programming model).

We have planned to enable scripting for Harry's apps in several areas, but start with its sensor framework. Sensor integration has proven to be both complex and broad. "Broad" means there are hundreds of sensor interfaces out there we can't support using hard coded integration. By making this area scriptable, we address both complexity and broadness:

Reduced complexity: the complete path from connection making using the various connection technologies (BT, BT LE, WiFi, Mfi), configuration of sensors, and reading / writing proprietary protocols is supported by powerful extensions to standard Lua. In addition, standard dialects like NMEA for GNSS sensors, or ELM327 for OBD sensors are provided as predefined libraries allowing sensor integration scripts as small as 10 lines of code. Scripts are platform independent too. An integration scripted for iOS will work on Android and vice versa.

Allow broadness: scripting allows 3rd parties to provide their own integration - without our direct support and independent from app release cycles. Scripts can be written based on examples provided and tested "in real life" immediately. In case a 3rd party wants to have a sensor integration distributed to all LapTimer users, we can integrate it into release versions too. So far, we had to reject lots of requests by manufacturers and users to support some new device. Due to capacity and time limitations, we had to pick the main stream / most common devices to support as many users as possible. Scripting will allow much broader support.

In case you are interested in adding scripts, please read this document first. It should give you an understanding on how scripts are structured and how they can help to support your goals. Next, head to the Other Sources section and get your hands on the Programming in Lua paperback. It will give you a full understand of Lua, including embedded scenarios like **HarrysLua**. Finally, get you sensor and one of the sample scripts available and customize it to your requirements. In case you run into any problem, please use the **HarrysLua Developer Forum** to discuss solutions and request features. We are really excited to make this an universal and often used feature, so anyone starting the sensor integration adventure will get our full support!

Availability

HarrysLua will be supported in production versions of Harry's apps starting with major release v22. We will make this feature available for early adopters in v21.1 Please watch the forum and our Facebook page for announcements. To allow debugging, we recommend to use Harry's GPS/OBD Buddy for testing. Other than LapTimer et al, Buddy is logging enabled and will allow you to log both the framework and your script.

To join the developer program, contact us at Harry@gps-laptimer.de. Developer resources (and support through **HarrysLua** Developer Forum in particular) are made available for those joining the program only.

Is scripting required?

Probably not. Harry's apps supports several levels of sensor individualization:

Predefined sensors: we try to make genuine sensors used frequently available out of the box. Usually, connection making on operating system level is all a user needs to do. The accessory will be recognized automatically and it will just work.

Predefined sensors are either hard coded or scripted (without exposing this script). Predefined sensors are listed on

<http://www.gps-laptimer.de/compatibility/ios>

and

<http://www.gps-laptimer.de/compatibility/android>

Custom sensors: in case you have a plain vanilla GPS sensor feeding data using the NMEA, or an OBD sensor using the ELM327 protocol, you can connect it using the app's Expert Settings. The only thing to do is entering connection type and parameters. WiFi and BTLE connection types are supported for both iOS and Android. For Android, any Bluetooth (SPP) sensor speaking the named protocols can be added from Sensor List. See [Configuration of Custom BTLE Sensors](#) for instructions on how to add configure custom sensors in LapTimer.

Scripted sensors: for sensors requiring special connection configurations including custom initialization, or those featuring a proprietary protocol, scripting is required. Actually scripting is coming in two levels again. For NMEA and ELM327 sensors, a script will focus on connection and initialization only. For proprietary protocols, parsing and requesting data is scripted individually. See the sections below for the details.

General structure

We start with how a script is created and loaded into Harry's apps. A script files comes with an extension ".hscrl" and is an XML file on top level. To install the script into one of Harry's apps on Android, send it to yourself by mail and touch the attachment in e.g. Gmail. Select Harry's GPS/OBD Buddy from the list of apps shown and you are done. In case you had an older version of the script on your device already, it is replaced by the new version.

A minimal script file looks something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<scriptdefinition
  xmlns="http://www.gps-laptimer.de"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.gps-laptimer.de Scripts.xsd">

  <author>Harry</author>
  <version>1.0</version>
  <purpose>sensor.bt1e</purpose>

  <script><![CDATA[

    ...

  ]]></script>

</scriptdefinition>
```

The purpose defines the "plug-in" point into Harry's. If e.g. a purpose `sensor.bt1e` is chosen, the script will be used to implement a sensor using BT Low Energy technology.

Purposes available or planned so far are:

```
sensor.bluetooth
sensor.wifi
sensor.bluetooth [TODO]
sensor.mfi [TODO]
actioncam [TODO]
reporting [TODO]
export [TODO]
import [TODO]
```

Each purpose will define some mandatory and optional functions to be defined by the `<script>` tag. The other way around, each purpose will provide a set of functionality beyond pure Lua that can be used by the script. These sets of functionality are provided by modules.

Purposes and modules form a tight relation:

As a sample, the purpose `sensor.bluetooth` is a special case of a `sensor` - supported by functions predefined in the `sensor` module. BT LE specialities are supported by the `sensor.bluetooth` module. In addition to this module hierarchy, **HarrysLua** provides a number of global functions and constants.

Let us have a look into the **HarrysLua** code itself. For this initial discussion, we use a minimal script implementing a GPS sensor using a predefined NMEA parser back end. The code shown below replaces ... in the XML script structure introduced above.

```
-- sensor.oninit() is mandatory to define for any sensor purpose
-- it sets basic parameters and sets up communication

function sensor.oninit()

    tracecall(connect, "sensor.oninit()")

    -- set sensor parameters
    sensor.channelsets = { gnss } -- one or more channel sets, add from gnss, tires, engine
    sensor.nameprefix = "Custom BTLE" -- shown in Sensor List as "Custom BTLE GNSS"
    sensor.connectiontype = bluetooth -- one of bluetooth, bt, wifi, mfi

    -- set BTLE peripheral name pattern and service / characteristics we are interested in
    sensor.bluetooth.peripheralnamepattern = "Some name..." -- regex matching sensor name
    sensor.bluetooth.deviceinformation = true -- we assume this sensor provided device information

    -- set our main notifying read characteristic, we will use the tag returned later
    readcharacteristic = sensor.bluetooth.addcharacteristic(
        "some service uuid", "some characteristiuc uuid", true)

    -- GPS channel specific settings
    -- this is an important one: setting .nmea to true will use a predefined NMEA parser
    sensor.gnss.nmea = true

    tracereturn(connect, "sensor.oninit()")

end

-- sensor.bluetooth.onvaluechanged () is mandatory for bluetooth sensor
-- we redirect input from our read characteristic directly to the
-- predefined sensor.bytesread () function, which in turn will pass data to the
-- predefined NMEA parser (see sensor.gnss.nmea = true above)

function sensor.bluetooth.onvaluechanged (characteristic, value)

    tracecall(bluetooth, "sensor.bluetooth.onvaluechanged("
```

```
        .. characteristic .. ", " .. tracebytes(value) .. ")")

    if characteristic == readcharacteristic then
        sensor.bytesread(value)
    end

    tracereturn(bt1e, "sensor.bt1e.onvaluechanged()")

end
```

See comments in the code to understand what is actually done. Ignore the `trace*` statements for now, they are optional and covered in the section Debugging below.

When a script with sensor purpose is loaded, a function `sensor.oninit ()` is called. This function is mandatory to define. It describes primary properties of the sensor like which channelsets are actually supported, the wording in the app's front end (see `sensor.nameprefix`), and connection technology specific parameters.

In our BT LE sample, the sensor will go through a discovery phase after returning from `sensor.oninit ()`. The app is searching for BT LE devices around. To recognize devices, the device name is used. Valid names are defined by `sensor.bt1e.peripheralnamepattern` as regular expressions. If e.g. every actual sensor is coming with a name "FOO" followed by a 3 digit serial number made up from digits, it could be "FOO[0-9][0-9][0-9]". Once such a sensor (in BT LE a "peripheral") is discovered, the app will check if the services/characteristics declared using one or more call to `addcharacteristic ()` are available. Characteristics defined as notifying will automatically be subscribed to, the sensor is going to state "connected".

In our case, we will see bytes coming in through the read characteristic defined after a few seconds. Assuming the sensor sends data conforming to the NMEA standard, data is simply dispatched to `sensor.bytesread()` here. While this function will often be overridden when reading proprietary protocols, the default implementation will check if `sensor.gnss.nmea` (or, for engine channelsets, `sensor.engine.elm327`) is true and feed data into that predefined parser. The parser in turn will queue data into the apps engine once they are recognized.

So if your sensor is using a standard protocol, things are really easy and we are done here. :-)

Custom protocols

So let us have a look into how things work for a custom protocol instead of NMEA. The script implementing a custom protocol will look like the above one except is sets

```
sensor.gnss.nmea = false
```

```
in sensor.oninit ().
```

As the default implementation of `sensor.bytesread(value)` has no clue how to interpret "value" with a NMEA parser, we define our own version. Let us assume the sensor is sending (oversimplified) messages structured like:

```
seconds since 1/1/1970 (UTC):  4 bytes, unsigned integer
longitude:                    4 bytes, longitude
latitude:                     4 bytes, latitude
speed:                        1 byte, unsigned integer
accuracy:                     1 byte, unsigned integer
```

i.e. with 14 bytes overall length each.

```
function sensor.bytesread(message)
```



```
if #message == 14 then

    -- create a new and empty set of channels
    gpschannelset = {}

    -- extract data from message and fill gpschannelset table
    gpschannelset.seconds = string.unpack("I4", message, 1)+2082844800
    gpschannelset.longitude = string.unpack("f", message, 5)
    gpschannelset.latitude = string.unpack("f", message, 9)
    gpschannelset.speed = string.unpack("I1", message, 13)
    gpschannelset.accuracy = string.unpack("I1", message, 14)

    -- pass result to app
    sensor.queuechannelset(gpschannelset, gnss)

else

    -- signal app we have received an invalid fix
    sensor.queuechannelset(nil, gnss)

end

end
```

Other important call backs

You may want to initialize your sensor during connection. For situations like this, just define a function `sensor.onconnect()`. It will be called after `sensor.oninit()` and after the sensor has been connected on technical level. In case you send data to the sensor from here, it is a good idea to keep the current state to interpret messages correctly

```
local initialized = false

function sensor.onconnect ()

    initialized = false

    -- set up writecharacteristic using addcharacteristic(., ., false) in sensor.oninit()
    sensors.btle.writevalue(writecharacteristic, "some data")

end

function sensor.bytesread(message)

    if initialized then

        -- code as before

    else

        -- interpret reply from writecharacteristic

        initialized = true

    end

end

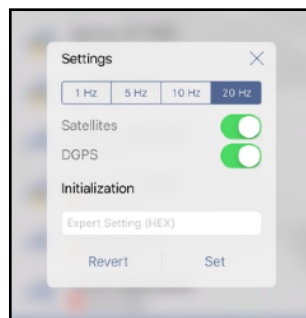
end
```

Other optional callbacks are listed in [HarrysLua Reference](#).

Sensor Configuration

Sensors will sometimes come with their own set of configuration options. Samples are configurable update rates, or features that can be turned on or off. **HarrysLua** provides a full features set of configuration tools to allow changing a sensor configuration from within our line of apps and without a requirement to handle a manufacturer app in parallel.

Sensor configurations are similar to preferences / settings used by apps to customize behavior. See the Settings app in iOS or Android. Sensor configurations are a set of items made up from a configuration name, a configuration label, a configuration type and a default value. Configuration sets can be read and written in scripts, and they can be changed by the user.



Sample configuration for a GPS sensor as shown in Sensor List

Configuration structures are defined in `sensor.oninit()`, the above sample's definition is something like this:

```
function sensor.oninit()
...
  sensor.configurationdefinitions =
  {
    rate = { type = "enumeration", values = { "1 Hz", "5 Hz", "10 Hz", "20 Hz" },
            default = "10 Hz" },
    satellites = { label = "Satellites", type = "boolean", default = true },
    differential = { label = "DGPS", type = "boolean", default = false },
    initialization = { label = "Initialization", type = "string",
                     hint = "Expert Setting (HEX)" }
  }
...
end
```

To access configuration from within a script, **HarrysLua** maintains a table of actual configuration values. With configuration values shown in the sample, `sensor.configurations.satellites` evaluates to `true` and `sensor.configurations.rate` evaluates to `"20 Hz"`.

Whenever the user changes configurations in the user interface, the sensor script will be notified by a call to `sensor.onconfigurationschanged()`. It is in the responsibility of the script to change the sensor's configuration accordingly.

This API is sufficient in case all actual configuration values are maintained in LapTimer: the script defines the meta structure and default values, the user is allowed to change this values, and the script is notified whenever changes occur and the sensor needs an update.

In case the actual state should be stored in the sensor itself, **HarrysLua** allows scripts to synchronize configuration stored in the device with those stored in the app: a function `sensor.changeconfiguration()` is provided and called with updated values. As an example, `sensor.changeconfigurations({ satellites = false})` would change

configuration `satellites` to `false`. A typical pattern is to read configuration from the sensor in `sensor.onconnect()` and write the results back into LapTimer using `sensor.changeconfigurations()`.

Platform Limitations

Scripts written in **HarrysLua** following instructions in this document are portable and will work for both Android and iOS. There are two limitations worth mentioning:

1. While serial Bluetooth connections (BT SPP) are "open" in Android, this type of connection is made available for iOS through Apple's Mfi (Made for iPhone) program. The only difference for the scripts is the purpose (`sensor.bt` vs. `sensor.mfi`) and the connection settings in `sensor.oninit()`. To allow Harry's apps to access a Mfi device, it needs to be white listed with Apple's Mfi team. This means for iOS adoptions, there are two approaches: a) White list our apps (starting with Buddy) for your accessory so the app is allowed to access the accessory. We will need to release a new app version for this but can do this as part of one or the frequent bug fix release. Once the app is white listed for the accessory protocol used, you can implement and test scripts with purpose `sensor.mfi`. b) Change the firmware of a testing accessory to one of the well known protocols. Use this protocol setting for implementation and testing of the script. Once you are ready, do the white listing for production accessories.
2. Scripting available to users is not allowed for iOS. Apps offering such a feature will be rejected during reviews. This means we do not support user level access for iOS. Instead, the script is developed by a 3rd party and made available by us and as part of an app release only. Going this path, the script is an implementation detail that is allowed and used commonly in games. Please contact us to get support for testing on iOS front up, or simply utilize an Android device for testing.

In case you have questions on this, please use the **HarrysLua Developer Forum**.

Debugging

For debugging, we use the same tracing capability LapTimer uses throughout the code. For selected apps (Harry's GPS/OBD Buddy in general, and all alpha / beta versions of LapTimer), this tracing capability is enabled and can be used. To allow traces of program flows, tracing needs to be turned on and the trace classes one is interested in need to be selected. Our apps and the scripts use the same set of trace classes although we provide only those that make sense as parameters to select from. Before going into the details, please have a look into the Tracing FAQ available from the Other Resources section below.

To add tracing output to your scripts, **HarrysLua** provides three global functions:

```
tracecall(), tracereturn(), and trace()
```

Each of these functions require two parameters: the trace class and the message to add.

```
trace(gnss, "sensor received data: " .. data)
```

will add the line "sensor received data: <content of data>" as a separate line to the trace log. `tracecall()` and `tracereturn()` always come as pairs and build an group in the tracing output. They are typically used as follows:

```
-- some function foogpsfctn defined in a script
function foogpsfctn (argument)

    tracecall(gnss, 'foogpsfctn(argument:' .. argument .. ')')

    -- some function implementation
    result = ...
```

```
-- some other trace output
trace(gnss, 'some other stuff')

tracereturn(gnss, 'foogpsfctn()', result)

return result

end
```

The output of the above code will be something like (provided tracing is enabled for the GPS Sensors trace class):

```
GPS      0176292887@main  D 01  --> call to foogpsfctn(argument: <content of argument>)
GPS      0176292887@main  D 02  --> some other stuff
GPS      0176292887@main  D 01  --> foogpsfctn() returns ...
```

In case the GPS Sensors trace class is not enabled in the app ([Expert Settings](#) ▶ [Trace Classes](#)), no output will be generated. Details on trace classes and the above functions can be found in [Harry's Lua Reference](#).

Result handling for `tracereturn()` is optional. In case a function does not return values, the function is called with just two arguments.

Our apps will check a proper pairing of `tracecall()` and `tracereturn()` calls. The function names need to match up to the opening brace or the argument list for both the call and the return. The "call to" prefix and "returns" postfix are added automatically, and anything between call and return will get an increased indent depth (see 02 above).

Our apps follow some conventions when augmenting code with trace statements. We appreciate your efforts to follow this conventions to receive consistent output. Once you will start with tracing, you will see that our apps are heavily instrumented with trace statements and those in our app and framework code will be mixed with trace statements you add.

When tracing arguments, add the argument name "argument:" and the argument value. This allows easier reading. Functions not issuing any other output than call and return should use a single trace statement like

```
function foognssfctn (argument)

    trace(gnss, 'foognssfctn(argument:' .. argument .. ')')

    -- some function implementation

end
```

Adding your own trace classes is not supported. For quick testing, use the `adhoc` trace class. It will always output a trace (i.e. there is no trace class that needs to be enabled, it is always active) and is convenient to use. Please do not forget to either remove the whole call once you do not need the output any more, or replace `adhoc` by one of the predefined trace classes before finalizing the script. The `adhoc` trace class must not be used in production versions of your scripts.

Performance considerations

A script based implementation is a lot slower than a hard coded integration using compiled code. Lua is considered a fast scripting language, but profiling shows it is still slower by a magnitude when compared e.g. to Objective-C like used in Harry's app on iOS. On the other hand, today's smartphones are quite fast and will compensate the impact of a scripted portion of code. For a modern device, any sensor processing up to 25 Hz should not create a relevant slow down. When integrating 50 or 100 Hz devices however, I'd recommend to monitor the load using either platform tools or by enabling the [Expert Display Mode](#) available in Harry's apps. It will show the current load in the bottom left of all Timer Views.

In case you run into limits, please contact the **Developer Forum** to discuss if either additional native **HarrysLua** APIs can be introduced, or if the script should be considered a prototype and go into a hard coded sensor implementation.

Other Resources

Programming in Lua, Fourth Edition: available on all Amazon stores, a great introduction into Lua

<https://www.amazon.com/Programming-Lua-Fourth-Roberto-Ierusalimschy/dp/8590379868>

[HarrysLua Reference](#): settings, functions and callback reference for HarrysLua. This reference is available for Developer Program members through the HarrysLua Developer Forum (see below).

Official Lua Site: provides documentation, sources etc: <https://www.lua.org>

Tracing / Logging Harry's Apps FAQ: <http://forum.gps-laptimer.de/viewtopic.php?f=39&t=1933>

HarrysLua Developer Forum: <http://forum.gps-laptimer.de/viewforum.php?f=47>

In case you can't access the **Developer Forum**, you have not yet contacted us to participate in our developer program.