Sensors and actuators

Contents

Sensors and actuators

Introduction

This documents possible approaches to designing an API for exposing vehicle sensor information and allowing interaction with actuators to application bundles on an Apertis system.

The major considerations with a sensors and actuators API are:

- Bandwidth and latency of sensor data such as that from parking cameras
- Enumeration of sensors and actuators
- Support for multiple vehicles or accessories
- Support for third-party and OEM accessories and customisations
- Multiplexing of access to sensors
- Privilege separation between application bundles using the API
- Policy to restrict access to sensors (privacy sensitive)
- Policy to restrict access to actuators (safety critical)

Terminology and concepts

Vehicle

For the purposes of this document, a *vehicle* may be a car, car trailer, motorbike, bus, truck tractor, truck trailer, agricultural tractor, or agricultural trailer, amongst other things.

Intra-vehicle network

The *intra-vehicle network* connects the various devices and processors throughout a vehicle. This is typically a CAN or LIN network, or a hierarchy of such networks. It may, however, be based on Ethernet or other protocols.

The vehicle network is defined by the OEM, and is statically defined — all devices which are supported by the network have messages or bandwidth allocated for them at the time of manufacture. No devices which are not known at the time of manufacture can be supported by the vehicle network.

Inter-vehicle network

An *inter-vehicle network* connects two or more *physically connected* vehicles together for the purposes of exchanging information. For example, a network between a truck tractor and trailer.

An inter-vehicle network (for the purposes of this document) does *not* cover transient communications between separate cars on a motorway, for example; or between a vehicle and static roadside infrastructure it passes. These are car-to-car (C2C) and car-to-infrastructure (C2X) communications, respectively, and are handled separately.

Sensor

A *sensor* is any input device which is connected to the vehicle's network but which is not a direct part of the dashboard user interface. For example: parking cameras, ultrasonic distance sensors, air conditioning thermometers, light level sensors, etc.

Actuator

An *actuator* is any output device which is connected to the vehicle's network but which is not a direct part of the dashboard user interface. For example: air conditioning heater, door locks, electric window motors, interior lights, seat height motors, etc.

Device

A sensor or actuator.

Use cases

A variety of use cases for application bundle usage of sensor data are given below. Particularly important discussion points are highlighted at the bottom of each use case.

Augmented reality parking

When parking, the feed from a rear-view camera should be displayed on the screen, with an overlay showing the distance between the back of the vehicle and the nearest object, taken from ultrasonic or radar distance sensors.

The information from the sensors has to be synchronised with the camera, so correct distance values are shown for each frame. The latency of the output image has to be low enough to not be noticed by the driver when parking at low speeds (for example, 5km·h).

Virtual mechanic

Provide vehicle status information such as tyre pressure, engine oil level, washer fluid level and battery status in an application bundle which could, for example, suggest routine maintenance tasks which need to be performed on the vehicle.

(Taken from *http://www.w3.org/2014/automotive/vehicle_spec.html#h2_abstract*.)

Trailer

The driver attaches a trailer to their vehicle and it contains tyre pressure sensors. These should be available to the virtual mechanic bundle.

Petrol station finder

Monitor the vehicle's fuel level. When it starts to get low, find nearby petrol stations and notify the driver if they are near one. Note that this requires programs to be notified of fuel level changes while not in the foreground.

Sightseeing application bundle

An application bundle could highlight sights of interest out of the windows by combining the current location (from GPS) with a direction from a compass sensor. Using a compass rather than the GPS' velocity angle allows the bundle to work even when the vehicle is stationary.

Privacy concern: Any application bundle which has access to compass data can potentially use dead reckoning to track the vehicle's location, even without access to GPS data.

Basic model vehicle

If a vehicle does not have a compass sensor, the sightseeing bundle cannot function at all, and the Apertis store should not allow the user to install it on their vehicle.

Changing bundle functionality when driving at speed

An application bundle may want to voluntarily change or disable some of its features when the vehicle is being driven (as opposed to parked), or when it is being driven fast (above a cut-off speed). It might want to do this to avoid distracting the driver, or because the features do not make sense when the vehicle is moving. This requires bundles to be able to access speedometer and driving mode information.

If the application bundle is using a cut-off speed for this decision, it should not have to continually monitor the vehicle's speed to determine whether the cut-off has been reached.

Changing audio volume with vehicle or cabin noise

Bundles may want to adjust their audio output volume, or disable audio output entirely, in response to changes in the vehicle's cabin or engine noise levels. For example, a game bundle could reduce its effects volume if a loud conversation can be heard in the cabin; but it might want to increase its effects volume if engine noise increases.

Privacy concern: This should be implemented by granting access to overall 'volume level' information for different zones in the vehicle; but *not* by granting access to the actual audio input data, which would allow the bundle to record conversations. The overall volume level information should be sufficiently smoothed or high-latency that a malicious application cannot infer audio information from it.

Night mode

Programs may wish to change their colour scheme according to the ambient lighting level in a particular zone in the cabin, for example by switching to a 'night mode' with a dark colour scheme if driving at night, but not if an interior light is on. This requires bundles to be able to read external light sensors and the state of internal lights.

Weather feedback or traffic jam feedback

A weather bundle may want to crowd-source information about local weather conditions to corroborate its weather reports. Information from external rain, temperature and atmospheric pressure sensors could be collected at regular intervals – even while the weather bundle is not active – and submitted to an online weather service as network connectivity permits.

Similarly, a traffic jam or navigation bundle may want to crowd-source information about traffic jams, taking input from the speedometer and vehicle separation distance sensors to report to an online service about the average speed and vehicle separation in a traffic jam.

Insurance bundle

A vehicle insurance company may want to offer lower insurance premiums to drivers who install its bundle, if the bundle can record information about their driving safety and submit it to the insurance company to give them more information about the driver's riskiness. This would need information such as driving duration, distances driven, weather conditions, acceleration, braking frequency, frequency of using indicator lights, pitch, yaw and roll when cornering, and potentially vehicle maintenance information. It would also require access to unique identifiers for the vehicle, such as its VIN. The data would need to be collected regardless of whether the vehicle is connected to the internet at the time — so it may need to be stored for upload later.

Privacy concern: Unique identification information like a VIN should not be given to untrusted bundles, as they may use it to track the user or vehicle.

Driving setup bundle

An application bundle may want to control the driving setup — the position of the steering wheel, its rake, the position of the wing mirrors, the seat position and shape, whether the vehicle is in sport mode, etc. If a guest driver starts using the vehicle, they could import their settings from the same bundle on their own vehicle, and the bundle would automatically adjust the physical driving setup in the vehicle to match the user's preferences. The bundle may want to restrict these changes to only happen while the vehicle is parked.

Odour detection

A vehicle manufacturer may have invented a new type of interior sensor which can detect foul odours in the cabin. They want to integrate this into an application bundle which will change the air conditioning settings temporarily to clear the odour when detected. The Sensors and Actuators API currently has no support for this new sensor. The manufacturer does not expect their bundle to be used in other vehicles.

Air conditioning control

An application bundle which connects to wrist watch body monitors on each of the passengers (through an out-of-band channel like Bluetooth, which is out of the scope of this document; see Bluetooth wrist watch and the Internet of Things may want to change the cabin temperature in response to thermometer readings from passengers' watches.

Automatic window feedback

In order to do this, the bundle may also need to close the automatic windows, but one of the passengers has their arm hanging out of the window and the hardware interlock prevents it closing. The bundle must handle being unable to close the window.

Agricultural vehicle

Apertis is used by an agricultural manufacturer to provide an IVI system for drivers to use in their latest tractor model. The manufacturer provides a pre-installed app for controlling their own brand of agricultural accessories for the tractor, so the driver can use it to (for example) control a tipping trailer and a baler which are hitched to each other behind the tractor, and also control a bale spear attached to the front of the tractor.

Roof box

A car driver adds a roof box to their car, provided by a third party, containing a safety sensor which detects when the box is open. The built-in application bundle for alerting the driver to doors which are open when the vehicle starts moving should be able to detect and use this sensor to additionally alert the driver if the roof box is open when they start moving.

Truck installations

Trucks are sold as a basis 'vanilla' truck with a special installation on top, which is customised for the truck's intended use. For example, a rubbish truck, tipping truck or police truck. The installation is provided by a third party who has a relationship with the basis truck manufacturer. Each installation has specific sensors and actuators, which are to be controlled by an application bundle provided by the third party or by the manufacturer.

Compromised application bundle

An application bundle on the system, A, is installed with permissions to adjust the driver's seat position, which is one of the features of the bundle. Another application bundle, B, is installed without such permissions (as they are not needed for its normal functionality).

Safety critical: An attacker manages to exploit bundle B and execute arbitrary code with its privileges. The attacker must not be able to escalate this exploit to give B permission to use actuators attached to the system, or extra sensors. Similarly, they must not be able to escalate the exploit to gain the privileges of bundle A, and hence bundle A's permissions to adjust the driver's seat position.

Ethernet intra-vehicle network

A vehicle manufacturer wants to support high-bandwidth devices on their intra-vehicle network, and decides to use Ethernet for all intra-vehicle communications, instead of a more traditional CAN or LIN network. Their use of a different network technology should not affect enumeration or functionality of devices as seen by the user.

Development against the SDK

An application developer wants to use a local gyroscope sensor attached to their development machine to feed input to their application while they are developing and testing it using the SDK.

Non-use-cases

Bluetooth wrist watch and the Internet of Things

A passenger gets into the vehicle with a Bluetooth wrist watch which monitors their heart rate and various other biological variables. They launch their health monitor bundle on the IVI display, and it connects to their watch to download their recent activity data.

This is not a use case for the Sensors and Actuators API; it should be handled by direct Bluetooth communication between the health monitor bundle and the watch. If the Sensors and Actuators API were to support third-party devices (as opposed to ones specified and installed by the vehicle manufacturer or suppliers), having full support for all available devices would become a lot harder. Additionally, devices would then appear and disappear while the vehicle was running (for example, if the user turned off their watch's Bluetooth connection while driving); this is not possible with fixed in- vehicle sensors, and would complicate the sensor enumeration API.

More generally, this use-case is a specific case of the internet of things (IoT), which is out of scope for this design for the reasons given above. Additionally, supporting IoT devices would mean supporting wireless communications as part of the sensors service, which would significantly increase its attack surface due to the complexity of wireless communications, and the fact they enable remote attacks.

Car-to-car and car-to-infrastructure communications

In C2C and C2X communications, vehicles share data with each other as they move into range of each other or static roadside infrastructure. This information may be anything from braking and acceleration information shared between convoys of vehicles to improve fuel efficiency, to payment details shared from a car to toll booth infrastructure.

While many of the use cases of C2C and C2X cover sharing of sensor data, the data being shared is typically a limited subset of what's available on one vehicle's network. Due to the dynamic nature of C2C and C2X networks, and the greater attack surface caused by the use of more complex technologies (radio communications rather than wired buses), a conservative approach to security suggests implementing C2C and C2X on a use-case-by-use-case basis, using separate system components to those handling intra-vehicle sensors and actuators. This ensures that control over actuators, which is safety critical, remains in a separate security domain from C2C and C2X, which must not have access to actuators on the local vehicle. See Security.

An initial suggestion for C2C and C2X communications would be to implement them as a separate service which consumes sensor data from the sensors and actuators service just like other applications.

Buddied and vehicle fleet communications

Similarly, long-range communications of sensor data between buddied vehicles or vehicles operating in a fleet (for example, a haulage or taxi fleet) should be handled separately from the sensors and actuators service, as such systems involve network communications. Typical use cases here would be reporting speed and fuel usage information from trucks or taxis back to headquarters; or letting two friends know each others' locations and traffic conditions when both doing the same journey.

Requirements

Enumeration of devices

An application bundle must be able to enumerate devices in the vehicle, including information about where they are located in the vehicle (for example, so that it can adjust the position and setup of the driver's seat but not others (see Driving setup bundle)).

It is expected that the set of devices in a vehicle may change dynamically while the vehicle is running, for example if a roof box were added while the engine was running (Roof box).

Enumeration is particularly important for bundles, as the set of sensors in a particular vehicle will not change, but the set of sensors seen by a bundle across all the vehicles it's installed in will vary significantly.

Enumeration of vehicles

An application bundle must be able to enumerate vehicles connected to the inter-vehicle network, for example to discover the existence of hitched trailers or agricultural vehicles (Trailer, Agricultural vehicle).

It is expected that the set of vehicles may change dynamically while the vehicles are running.

Retrieving data from sensors

An application bundle must be able to retrieve data from sensors. This data must be strongly typed in order to minimise the possibility of a bundle misinterpreting it, or sensors from different manufacturers using different units, for example. Sensor data could vary in type from booleans (see Night mode) through to streaming video data (see Augmented reality parking). Sensor data may be processed by the system to make it more useful for application bundles; they do not need direct access to raw sensor data.

Sending data to actuators

An application bundle must be able to send data to actuators (including invoking methods on them). This data must be strongly typed in order to minimise the possibility of a bundle misinterpreting it, or actuators from different manufacturers using different units, for example. Actuator data could vary in type from booleans through to enumerated types (see Driving setup bundle) and possibly larger data streams, though no concrete use cases exist for that.

Network independence

The API should be independent of the network used to connect to devices — whether it be Ethernet, LIN or CAN; or whether the device is connected directly to a host processor (Ethernet intra-vehicle network).

Bounded latency of processing sensor data

Certain sensor data has bounds on its latency. For example, pitch, yaw and roll information typically arrive as angular rate from sensors, and have to be integrated over time to be useful to application bundles — if sensor readings are missed, accuracy decreases. Sensor readings should be processed within the latency limits specified by the sensors. The limits on forwarding this processed data to bundles are less strict, though it is expected to be within the latency noticeable by humans (around 20ms) so that it can be displayed in real time (see Augmented reality parking, Sightseeing application bundle, Changing audio volume with vehicle or cabin noise).

Extensibility for OEMs

New types of device may be developed after the Sensors and Actuators API is released. As the set of sensors in a vehicle does not vary after release, already-deployed versions of the API do not need to handle unknown devices. However, there must be a mechanism for OEMs or third parties working with them to define new device types when developing a new vehicle or an installation or accessory to go with it. In order for new devices to be usable by non-OEM application bundle authors, the Sensors and Actuators API must be updatable or extensible to support them. (Odour detection, Truck installations.)

Third-party backends

If an OEM or third party produces a new device which can be connected to an existing vehicle, some code needs to exist to allow communication between the device and the Apertis sensors and actuators service. This code must be written by the device manufacturer, as they know the hardware, and must be installable on the Apertis system before or after vehicle production (so as a system or non-system application). (See Agricultural vehicle, Roof box, Truck installations.)

Third-party backend validation

If a third-party device is exposed to the sensors and actuators service, the third party might not be one who has contributed to or used Apertis before. There must be a process for validating backends for the sensors and actuators system, to ensure they have a certain level of code quality and security, in order to reduce the attack surface of the service as a whole. (See Roof box.)

Notifications of changes to sensor data

All sensor data changes over time, so the API must support notifying application bundles of changes to sensor data they are interested in, without requiring the bundle to poll for updates (see Petrol station finder, Sightseeing application bundle, Changing bundle functionality when driving at speed, Changing audio volume with vehicle or cabin noise, Night mode, Odour detection).

Application bundles should be able to request notifications only when a sensor value crosses a given threshold, to avoid unnecessary notifications (see Changing bundle functionality when driving at speed).

Uncertainty bounds

Sensors are not perfectly accurate, and additionally a sensor's accuracy may vary over time; each sensor measurement should be provided with uncertainty bounds. For example, the accuracy of geolocation by mobile phone tower varies with your location.

This is especially possible with data aggregated from multiple sensors, where the aggregate accuracy can be statistically modelled (for example, distance calculation from multiple sensors in Weather feedback or traffic jam feedback).

Failure feedback

As actuators are physical devices, they can fail. The API cannot assume automatic, immediate or successful application of its changes to properties, and needs to allow for feedback on all property changes.

For example, the air conditioning coolant on an older vehicle might have leaked, leaving the air conditioning system unable to cool the cabin effectively. Application bundles which wish to set the temperature need to have feedback from a thermometer to work out whether the temperature has reached the target value (see Air conditioning control).

Another example is failure to close windows: Automatic window feedback.

Timestamping

In-vehicle networks (especially Ethernet) may have variable latency. In order to correlate measurements from multiple sensors on the end of connections of varying latency, each measurement should have an associated timestamp, added at the time the measurement was recorded (see Augmented reality parking, Sightseeing application bundle).

Triggering bundle activation

Various use cases require a bundle to be able to trigger actions based on sensor data reaching a certain value, even if the program is not running at that time (see Petrol station finder, Changing audio volume with vehicle or cabin noise, Odour detection). This requires some operating system service to monitor a list of trigger conditions even while the programs which set those triggers are not running, and start the appropriate program so that it can respond to that trigger.

Bulk recording of sensor data

Some bundles require to be able to regularly record sensor measurements, with the intention of processing them (for example, uploading them to an online service) at a later time (see Weather feedback or traffic jam feedback, Insurance bundle). This is not latency sensitive. As an optimisation, a system service could record the sensor readings for them, to avoid waking up the programs regularly.

Data recorded in this way must only be accessible to the application bundle which requested it be recorded.

The requesting application bundle is responsible for processing the data periodically, and deleting it once processed. The system must be able to periodically overwrite recorded data if running low on space.

Sensor security

As highlighted by the privacy concerns in several of the use cases (Sightseeing application bundle, Changing audio volume with vehicle or cabin noise, Insurance bundle), there are security concerns with allowing bundles access to sensor data. The system must be able to restrict access to some or all types of sensor data unless the user has explicitly granted a bundle access to it. Bundles with access to sensor data must be in separate security domains to prevent privilege escalation (Compromised application bundle).

Actuator security

Control of actuators is safety critical but not privacy sensitive (unlike sensors). The system must be able to restrict write access to some or all types of actuator unless the user has explicitly granted a bundle access to it. Bundles with access to actuators must be in separate security domains to prevent privilege escalation (Compromised application bundle).

App store knowledge of device requirements

The Apertis store must know which devices (sensors *and* actuators) an application bundle requires to function, and should not allow the user to install a bundle which requires a device their vehicle does not have, or the bundle would be useless (Basic model vehicle).

Accessing devices on multiple vehicles

The API must support accessing properties for multiple vehicles, such as hitched agricultural trailers (Agricultural vehicle) or car trailers (Trailer). These vehicles may appear dynamically while the IVI system is running; for example, in the case where the driver hitches a trailer with the engine running.

Note: This requirement explicitly does not support C2C or C2X, which are out of scope of this document. (See Car-to-car and car-to-infrastructure communications).

Third-party accessories

The API must support accessing properties of third-party accessories — either dynamically attached to the vehicle (Roof box) or installed during manufacture (Truck installations).

SDK hardware support

The SDK must contain a backend for the system which allows appropriate hardware which is attached to the developer's machine to be used as sensors or actuators for development and testing of applications (see Development against the SDK).

This backend must not be available in target images.

Background on intra-vehicle networks

For the purposes of informing the interface design between the Sensors and Actuators API and the underlying intra-vehicle network, some background information is needed on typical characteristics of intra-vehicle networks.

CAN and LIN are common protocols in use, though future development may favour Ethernet or other protocols. In all cases, the OEM statically defines all protocols, data structures, and devices which can be on the network. Bandwidth is allocated for all devices at the time of manufacture; even for devices which are only optionally connected to the network, either because they're a premium vehicle feature, or because they are detachable, such as trailers. In these cases, data structures on the network relating to those devices are empty when the devices are not connected.

Sometimes flags are used in the protocol, such as 'is a trailer connected?'.

There are no common libraries for accessing vehicle networks: they differ between OEMs.

Existing sensor systems

This chapter describes the approaches taken by various existing systems for exposing sensor information to application bundles, because it might be useful input for Apertis' decision making. Where available, it also provides some details of the implementations of features that seem particularly interesting or relevant.

W3C Vehicle Information Service Specification (VISS)

The W3C Vehicle Information Service Specification[1] defines a WebSocket based API for a Vehicle Information Service (VIS) to enable client applications to get, set, subscribe and unsubscribe to vehicle signals and data attributes. This specification defines a number of methods for accessing vehicle data which are strictly agnostic to the data model Vehicle Signal Specification[2].

The Vehicle Signal Specification (VSS) focuses on vehicle signals, in the sense of classical sensors and actuators with the raw data communicated over vehicle buses and data which is more commonly associated with the infotainment system alike. This defines a 'tree-like' logical taxonomy of the vehicle, (formally a Directed Acyclic Graph), where major vehicle structures (e.g. body, engine) are near the top of the tree and the logical assemblies and components that comprise them, are defined as their child nodes.

The VSS supports both extensibility and the ability to define private branches.

GENIVI Web API Vehicle

The [GENIVI Web API Vehicle] (sic) is a proof of concept API for exposing and manipulating vehicle information to GENIVI apps via a JavaScript API. It is very similar to the W3C Vehicle Information Access API, and seems to expose a very similar set of properties.

The Web API Vehicle[3] is a proxy for exposing a separate Vehicle Interface API within a HTML5 engine. The Vehicle Interface API itself is apparently a D-Bus API for sharing vehicle information between the CAN bus and various clients, including this Web API Vehicle and any native apps. Unfortunately, the Vehicle Interface API seems to be unspecified as of August 2015, at least in publicly released GENIVI documents.

[1] https://www.w3.org/TR/vehicle-information-service/

[2] https://github.com/GENIVI/vehicle_signal_specification

[3] http://git.projects.genivi.org/?p=web-api-vehicle.git;a=blob_plain;f=doc/WebAPIforVehicleData.pdf;hb=HEAD

http://git.projects.genivi.org/?p=web-api-vehicle.git;a=blob_plain;f=doc/WebAPIforVehicleDataRI.pdf;hb=HEAD Section 2.2.3

The Web API Vehicle has the same features and scope as the W3C API, but its implementation is clumsier, relying a lot more on seemingly unstructured magic strings for accessing vehicle properties.

http://git.projects.genivi.org/?p=web-api-vehicle.git;a=blob_plain;f=doc/WebAPIforVehicleData.pdf;hb=HEAD

It was last publicly modified in May 2013, and might not be under development any more. Furthermore, a lot of the wiki links in the specification link to private and inaccessible data on collab.genivi.org.

Apple HomeKit

Apple HomeKit[4] is an API to allow apps on Apple devices to interact with sensors and actuators in a home environment, such as garage doors, thermostats, thermometers and light switches, amongst others. It is designed explicitly for the home environment, and does not consider vehicles. However, as it is effectively an API for allowing interactions between sandboxed apps and external sensors and actuators, it bears relevance to the design of such an API for vehicles.

At its core, HomeKit allows enumeration of devices ('accessories') in a home. A large part of its API is dedicated to grouping these into homes, rooms, service groups and zones so that collections of accessories can be interacted with simultaneously.

Each accessory implements one or more 'services' which are defined interfaces for specific functionality, such as a light switch interface, or a thermostat interface. Each service can expose one or more 'characteristics' which are readable or writeable properties of that interface, such as whether a light is on, the current temperature measured by a thermostat, or the target temperature for the thermostat.

It explicitly maintains separation between *current* and *target* states for certain characteristics, such as temperature controlled by a thermostat, acknowledging that changes to physical systems take time.

A second part of the API implements 'actions' based on sensor values, which are arbitrary pieces of code executed when a certain condition is met. Typically, this would be to set the value of a characteristic on some actuator when the input from another sensor meets a given condition. For example, switching on a group of lights when the garage door state changes to 'open' as someone arrives in the garage.

[4] https://developer.apple.com/homekit/

Critically, triggers and actions are handled by the iOS operating system, so are still checked and executed when the app which created them is not active.

HomeKit has a [simulator] for developing apps against.

Apple External Accessory API

As a precursor to HomeKit, Apple also supports an External Accessory API[5], which allows any iOS device to interact with accessories attached to the device (for example, through Bluetooth).

In order to use the External Accessory API, an app must list the accessory protocols it supports in its app manifest. Each accessory supports one or more protocols, defined by the manufacturer, which are interfaces for aspects of the device's functionality. They are equivalent to the 'services' in the HomeKit API. The code to implement these protocols is provided by the manufacturer, and the protocols may be proprietary or standard.

Each accessory exposes versioning information[6] which can be used to determine the protocol to use.

All communication with accessories is done via sessions[7], rather than one-shot reads or writes of properties. Each session is a bi-directional stream along which the accessory's protocol is transmitted.

iOS CarPlay

iOS CarPlay[8] is a system for connecting an iOS device to a car's IVI system, displaying apps from the phone on the car's display and allowing those apps to be controlled by the car's touchscreen or physical controls. It *does not give*[9] the iOS device access to car sensor data, and hence is not especially relevant to this design.

It does not[10] (as of August 2015) have an API for integrating apps with the IVI display.

Most vehicle manufacturers have pledged support for it in the coming years.

Android Auto

Android Auto[11] is very similar to iOS CarPlay: a system for connecting a phone

[5] https://developer.apple.com/library/ios/featuredarticles/ExternalAccessoryPT/Introduction/Introduction.html

[6] https://developer.apple.com/library/ios/documentation/ExternalAccessory/Reference/EAAccessory_class/index.html#//apple_ref/occ/instp/EAAccessory/modelNumber

[7] https://developer.apple.com/library/ios/documentation/ExternalAccessory/Reference/EASession_class/index.html#//apple_ref/occ/instp/EASession/accessory

[8] http://www.apple.com/uk/ios/carplay/

[9] http://www.tomsguide.com/us/apple-carplay-faq,news-18450.html

[10] https://developer.apple.com/carplay/

[11] https://www.android.com/auto/

to the vehicle's IVI system so it can use the display and touchscreen or physical controls. As with CarPlay, it does *not* give the Android device access to vehicle sensor data, although (as of August 2015) that is planned for the future.

As of August 2015, it has an API for apps[12], allowing audio and messaging apps to improve their integration with the IVI display.

Most vehicle manufacturers have pledged support for it in the coming years.

MirrorLink

MirrorLink[13] is a proprietary system for integrating phones with the IVI display — it is similar to iOS CarPlay and Android Auto, but produced by the Car Connectivity Consortium[14] rather than a device manufacturer like Apple or Google.

The specifications for MirrorLink are proprietary and only available to registered developers. In their brochure[15] (page 2), it is stated that support for allowing apps access to sensor data is planned for the future (as of 2014).

MirrorLink is apparently the technology behind Microsoft's Windows in the Car[16] system, which was announced in April 2014.

Android Sensor API

Android's Sensor API[17] is a mature system for accessing mobile phone sensors. There are a more constrained set of sensors available in phones than in vehicles, hence the API exposes individual sensors, each implementing an interface specific to its type of sensor (for example, accelerometer, orientation sensor or pressure sensor). The API places a lot of emphasis on the physical limitations of each sensor, such as its range, resolution, and uncertainty of its measurements.

The sensors required by an app are listed in its manifest file, which allows the Google Play store to filter apps by whether the user's phone has all the necessary sensors.

As Android runs on a multitude of devices from different manufacturers, each with different sensors, enumeration of the available sensors is also an emphasis of the API, using its SensorManager[18] class.

[12] https://developer.android.com/training/auto/index.html
[13] http://www.mirrorlink.com/apps
[14] http://carconnectivity.org/
[15] http://carconnectivity.org/public/files/files/MirrorLink_2pgBrochure_0.pdf
[16] http://www.techradar.com/news/car-tech/microsoft-sets-its-sights-on-apple-carplay-with-windows-in-the-car-concept-1240245
[17] http://developer.android.com/guide/topics/sensors/index.html
[18] http://developer.android.com/reference/android/hardware/SensorManager.html

Sensors[19] can be queried by apps, or apps can register for notifications when sensor values change, including when the app is not in the foreground or when the device is asleep (if supported by the sensor). Apps can also register[20] for notifications when sensor values satisfy some trigger, such as a 'significant' change.

Automotive Message Broker

Automotive Message Broker[21] is an Intel OTC project to broker information from the vehicle networks to applications, exposing a tweaked version[22] of the W3C Vehicle Information Access API (with a few types and naming conventions tweaked) over D-Bus to apps, and interfacing with whatever underlying networks are in use in the vehicle. In short, it has the same goals as the Apertis Sensors and Actuators API.

As of August 2015, it was last modified in June 2015, so is an active project (although Tizen is in decline, so this may change). Although it is written in C++, it uses GNOME technologies like GObject Introspection; but it also uses Qt. Its main daemon is the Automotive Message Broker daemon, ambd.

One area where it differs from the Apertis design is Security; it does not implement the polkit integration which is key to the vehicle device daemon security domain boundary. Modifying the security architecture of a large software project after its initial implementation is typically hard to get right.

Another area where ambd differs from the Apertis design is in the backend: ambd uses multiple plugins to aggregate vehicle properties from many places. Apertis plans to use a single OEM-provided, vehicle-specific plugin.

AllJoyn

The AllJoyn Framework[23] is an internet of things (IoT) framework produced under the Linux Foundation banner and the AllSeen Alliance[24]. (Note that IoT frameworks are explicitly out of scope for this design; this section is for background information only. See Bluetooth wrist watch and the Internet of Things) It allows devices to discover and communicate with each other. It is freely available (open source) and has components which run on various different operating systems.

As a framework, it abstracts the differences between physical transports, providing a session API for devices to use in one-to-one or one-to-many configurations

[19] http://developer.android.com/reference/android/hardware/SensorManager.html#registerListener%28android.hardware.SensorEventListener,%20android.hardware.Sensor,%20int%29

[20] http://developer.android.com/reference/android/hardware/SensorManager.html#requestTriggerSensor%28android.hardware.TriggerEventListener,%20android.hardware.Sensor%29

[21] https://github.com/otcshare/automotive-message-broker

[22] https://github.com/otcshare/automotive-message-broker/blob/master/docs/amb.in.fidl

[23] https://allseenalliance.org/framework

[24] https://allseenalliance.org/

for communication. A lot of its code is orientated towards implementing different physical transports.

It provides a security API for establishing different trust models between devices.

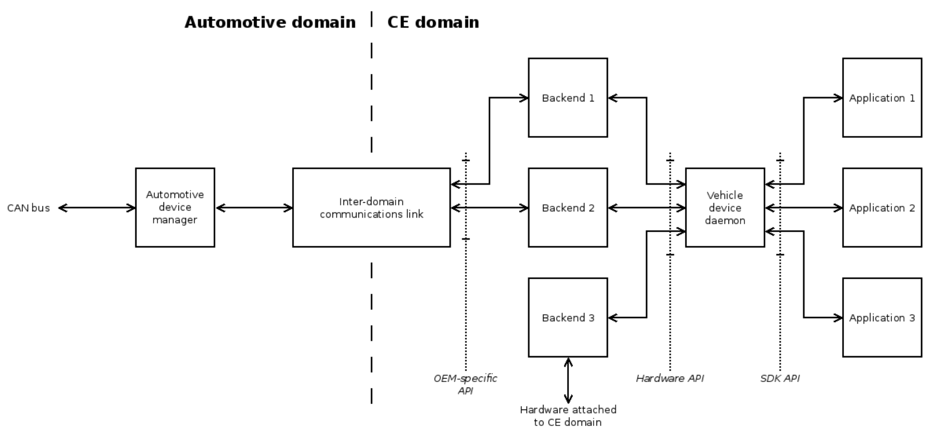
It provides various communication layer APIs for implementing RPC or raw I/O streams (or other things in-between) between devices. However, it does not specify the protocols which devices must use — they are specified by the device manufacturer.

AllJoyn provides common services for setting up new devices, sending notifications between devices, and controlling devices. It provides one example service for controlling lamps in a house, where each lamp manufacturer implements a well-defined OEM API for their lamp, and each application uses the lamp service API which abstracts over these.

Approach

Based on the above research (Existing sensor systems) and Requirements, we recommend the following approach as an initial sketch of a Sensors and Actuators API.

Overall architecture



Vehicle device daemon

Implement a vehicle device daemon which aggregates all sensor data in the vehicle, and multiplexes access to all actuators in the vehicle (apart from specialised high bandwidth devices; see High bandwidth or low latency sensors). It will connect to whichever underlying buses are used by the OEM to connect devices (for example, the CAN and LIN buses); see Hardware and app APIs. The implementation may be new, or may be a modified version of ambd, although it would need large amounts of rework to fit the Apertis design (see Automotive message broker).

The daemon needs to receive and process input within the latency bounds of the sensors.

The daemon should expose a D-Bus interface which follows the W3C Vehicle Information Access API[25]. The set of supported properties, out of those defined by the Vehicle Signal Specification[26], may vary between vehicles — this is as expected by the specification. It may vary over time as devices dynamically appear and disappear, which programs can monitor using the Availability interface[27].

The W3C specification was chosen rather than something like HomeKit due to its close match with the requirements, its automotive background, and the fact that it looks like an active and supported specification. Furthermore, HomeKit requires each device to define one or more protocols to use, allowing for arbitrary flexibility in how devices communicate with the controller. All the sensor and actuator use cases which are relevant to vehicles need only a property interface, however, which supports getting and setting properties, and being notified when they change.

If an OEM, third party or application developer wishes to add new sensor or actuator types, they should follow the extension process[28] and request that the extensions be standardised by Apertis — they will then be released in the next version of the Sensors and Actuators API, available for all applications to use. If a vehicle needs to be released with those sensors or actuators in the meantime, their properties must be added to the SDK API in an OEM-specific namespace. Applications from the OEM can use properties from this namespace until they are standardised in Apertis. See Property naming.

Multiple vehicles can be supported by exposing new top-level instances of the Vehicle interface[29]. For example, each vehicle could be exposed as a new object in D-Bus, each implementing the Vehicle interface, with changes to the set of vehicles notified using an interface like the standard D-Bus ObjectManager[30] interface.

This API can be exposed to application bundles in any binding language supported by GObject Introspection (including JavaScript), through the use of a client library, just as with other Apertis services. The client library may provide more specific interfaces than the D-Bus interface — the D-Bus API may be defined in terms of string keywords and variant values, whereas the client library API may be sensor-specific strongly typed interfaces.

[25] http://www.w3.org/2014/automotive/vehicle_spec.html
[26] https://github.com/GENIVI/vehicle_signal_specification
[27] http://www.w3.org/2014/automotive/vehicle_spec.html#data-availability
[28] https://genivi.github.io/vehicle_signal_specification/rule_set/private_branch/
[29] https://www.w3.org/Submission/vsso/#Vehicle
[30] http://dbus.freedesktop.org/doc/dbus-specification.html#standard-interfaces-objectmanager

Hardware and app APIs

The vehicle device daemon will have two APIs: the D-Bus SDK API exposed to applications, and the hardware API it consumes to provide access to the CAN and LIN buses (for example). The SDK API is specified by Apertis, and is standardised across all Apertis deployments in vehicles, so that a bundle written against it will work in all vehicles (subject to the availability of the devices whose properties it uses).

Open question: The exact definition of the SDK API is yet to be finalised. It should include support for accessing multiple properties in a single IPC round trip, to reduce IPC overheads.

The hardware API is also specified by Apertis, and implemented by one or more backend services which connect to the vehicle buses and devices and expose the information as properties understandable by the vehicle device daemon, using the hardware API.

At least one backend service must be provided by the vehicle OEM, and it must expose properties from the vehicle's standard devices from the vehicle buses. Other backend services may be provided by the vehicle OEM for other devices, such as optional devices for premium vehicle models; or truck installations. Similarly, backend services may be provided by third parties for other devices, such as after-market devices like roof boxes. Application bundles may provide backend services as well, to expose hardware via application-specific protocols. Consequently, backend services will likely be developed in isolation from each other.

Each backend service must expose zero or more properties — it is possible for a backend to expose zero properties if the device it targets is not currently connected, for example.

Each backend service must run as a separate process, communicating with the vehicle device daemon over D-Bus using the hardware API. The hardware API needs the following functionality:

- Bulk enumeration of vehicles
- Bulk notification of changes to vehicle availability
- Bulk enumeration of properties of a vehicle, including readability and writability
- Bulk notification of changes to property availability, readability or writability
- Subscription to and unsubscription from property change notifications
- Bulk property change notifications for subscribed properties

The hardware API will be roughly a similar shape to the SDK API, and hence a lot of complexity of the vehicle device daemon will be in the vehicle-specific backends (both operate on properties — Properties vs devices).

As vehicle networks differ, the backend used in a given vehicle has to be developed by the OEM developing that vehicle. Apertis may be able to provide some common utility functions to help in implementing backends, but cannot abstract all the differences between vehicles. (See Background on intra-vehicle networks).

It is expected that the main backend service for a vehicle, provided by that vehicle's OEM, will be access the vehicle-specific network implementation running in the automotive domain, and hence will use the inter-domain communications connection[31]. In order to avoid additional unnecessary inter-process communication (IPC) hops, it is suggested that the main backend service acts as *the* proxy for sensor data on the inter-domain connection, rather than communicating with a separate proxy in the CE domain — but only if this is possible within the security requirements on inter-domain connection proxies.

The path for a property to pass from a hardware sensor through to an application is long: from the hardware sensor, to the backend service, through the D-Bus daemon to the vehicle device daemon, then through the D-Bus daemon again to the application. This is at least 5 IPC hops, which could introduce non-negligible latency. See High bandwidth or low latency sensors for discussion about this.

Interactions between backend services

In order to keep the security model for the system simple, backend services must not be able to interact. Each device must be exposed by exactly one backend service — two backend services cannot expose the same device; and neither can they extend devices exposed by other backend services.

The vehicle device daemon must aggregate the properties exposed by its backends and choose how to merge them. For example, if one backend service provides a 'lights' property as an array with one element, and another backend service does similarly, the vehicle device daemon should append the two and expose a 'lights' array with both elements in the SDK API.

For other properties, the vehicle device daemon should combine scalar values. For example, if one backend service exposes a rain sensor measurement of 4/10, and another exposes a second measurement (from a separate sensor) of 6/10, the SDK API should expose an aggregated rain sensor measurement of (for example) 6/10 as the maximum of the two.

Open question: The exact means for aggregating each property in the Vehicle Signal Specification is yet to be determined.

Recommended hardware API design

[31] https://sjoerd.pages.apertis.org/apertis-website/concepts/inter-domain-communication/

Below is a pseudo-code recommendation for the hardware API. It is not final, but indicates the current best suggestion for the API. It has two parts — a management API which is implemented by the vehicle device daemon; and a property API which is implemented by each backend service and queried by the vehicle device daemon.

Types are given in the D-Bus type system notation[32].

Management API

Exposed on the well-known name `org.apertis.Rhosydd1` from the main daemon, the `/org/apertis/Rhosydd1` object implements the standard `org.freedesktop.DBus.ObjectManager`[33] interface to let client discover and get notified about the registered vehicles.

Vehicles are mapped under `/org/apertis/Rhosydd1/${vehicle_id}` and implement the `org.apertis.Rhosydd1.Vehicle` interface:

```
interface org.apertis.Rhosydd1.Vehicle {
  readonly property s VehicleId;
  method GetAttributes (
    in s node_path,
    out x current_time,
    out a(s(vdx)a{sv}(uu)) attributes)
  method GetAttributesMetadata (
    in s node_path,
    out x current_time,
    out a(sa{sv}(uu)) attributes_metadata)
  method SetAttributes (
    in a{sv} attributes_value)
  method UpdateSubscriptions (
    in a(sa{sv}) subscriptions,
    in a(sa{sv}) unsubscriptions)
  signal AttributesChanged (
    x current_time,
    a(s(vdx)a{sv}(uu)) changed_attributes,
    a(sa{sv}(uu)) invalidated_attributes))
  signal AttributesMetadataChanged (
    x current_time,
    a(sa{sv}(uu)) changed_attributes_metadata)
}
```

Backends register themselves on the bus with well-known names under the

[32] http://dbus.freedesktop.org/doc/dbus-specification.html#type-system
[33] http://dbus.freedesktop.org/doc/dbus-specification.html#standard-interfaces-objectmanager

org.apertis.Rhosydd1.Backends. prefix and implement the same interfaces and the main daemon, which will monitor the owned names on the bus and register to the object manager signals to multiplex access to the backends.

Each attribute managed via the vehicle attribute API is identified by a property name. Properties names come from the Vehicle Signal Specification, for example:

- Sunroof.Position[34]
- Horn.IsActive[35]
- Seat.FancySeatController.BackTemperature (oem specific property)

Each attribute has three values associated:

- its value (of type v)
- its accuracy (as a standard deviation of type d, set to `0.0` for non-numeric values)
- the timestamp when it was last updated (of type x)

In addition the current time is also returned for comparison to the time the value was last updated.

Values also have two set of metadata (of type u) associated:

- availability enum
  - AVAILABLE = 1
  - NOT_SUPPORTED = 0
  - NOT_SUPPORTED_YET = 2
  - NOT_SUPPORTED_SECURITY_POLICY = 3
  - NOT_SUPPORTED_BUSINESS_POLICY = 4
  - NOT_SUPPORTED_OTHER = 5
- access flags
  - NONE = 0
  - READABLE = (1 « 0)
  - WRITABLE = (1 « 1)

The GetAttributes method must return the value of all properties in the given branch indicated by the node path. If the node path represents a leaf node, then only the value corresponding to that property is returned. If no such branch or property exists on that vehicle, it must return an error. To get all properties of the vehicle an empty node path shall be passed.

To receive notification of attribute changes via the AttributesChanged and AttributesMetadataChanged signals, clients must first register their subscription with the UpdateSubscriptions method to specify the kind of properties for which they have some interest.

[34] https://www.w3.org/Submission/vsso/#SunroofPositionSensor
[35] https://www.w3.org/Submission/vsso/#HornIsActive

A backend service must emit an AttributesChanged signal when one of the properties it exposes changes, but it may wait to combine that signal with those from other changed properties — the trade-off between latency and notification frequency should be determined by backend service developers.

Hardware API compliance testing

As the vehicle-specific and third party backend services to the vehicle device daemon contain a large part of the implementation of this system, there should be a compliance test suite which all backend services must pass before being deployed in a vehicle.

If a backend service is provided by an application bundle, that application bundle must additionally undergo more stringent app store validation, potentially including a requirement for security review of its code. See Checks for backend services.

The compliance test suite must be automated, and should include a variety of tests to ensure that the hardware API is used correctly by the backend service. It should be implemented as a mock D-Bus service which mocks up the hardware management API (Recommended hardware API design), and which calls the hardware property API. The backend service must be run against this mock service, and call its methods as normal. The mock service should return each of the possible return values for each method, including:

- Success.
- Each failure code.
- Timeouts.
- Values which are out of range.

It must call property API methods with various valid and invalid input.

The backend service must not crash or obviously misbehave (such as consuming an unexpected amount of CPU time or memory).

As the backend service pushes data to the vehicle device daemon, the compliance test could be trivially passed by a backend service which pushes zero properties to it. This must not be allowed: backend services must be run under a test harness which triggers all of their behaviour, for all of the devices they support. Whether this harness simulates traffic on an underlying intra-vehicle network, or physically provides inputs to a hardware sensor, is implementation defined. The behaviour must be consistently reproducible for multiple compliance test runs.

SDK API compliance testing and simulation

Application bundle developers will not be able to test their bundles on real vehicles easily, so a simulator should be made available as part of the SDK, which exposes a developer-configurable set of properties to the bundle under test. The simulator must support all properties and configurations supported by the real vehicle device daemon, including multiple vehicles and third-party accessories; otherwise bundles will likely never be tested in such configurations. Similarly, it must support varying properties over time, simulating dynamic addition and removal of vehicles and devices, and simulating errors in controlling actuators (for example, Automatic window feedback).

The emulator should be implemented as a special backend service for the vehicle device daemon, which is provided by the emulator application. That way, it can directly feed simulated device properties into the daemon. This backend, and the emulator should only be available on the SDK, and must never be available on production systems.

Compliance testing of application bundles is harder, but as a general principle, any of the Apertis store validation checks which *can* be brought forward so they can be run by the bundle developers, *should* be brought forward.

SDK hardware

If a developer has appropriate sensors or actuators attached to their development machine, the development version of the sensors and actuators system should have a separate backend service which exposes that hardware to applications for development and testing, just as if it were real hardware in a vehicle.

This backend service must be separate from the emulator backend service (SDK API compliance testing and simulation), in order to allow them to be used independently.

Trip logging of sensor data

As well as an emulator for application developers to use when testing their applications, it would be useful to provide pre-recorded 'trip logs' of sensor data for typical driving trips which an application should be tested against. These trip logs should be replayable in order to test applications.

The design for this is covered in the 'Trip logging of SDK sensor data' section of the Debug and Logging design.

Properties vs devices

A major design decision was whether to expose individual sensors to bundles via the SDK API, or to expose properties of the vehicle, which may correspond to the reading from a single sensor or to the aggregate of readings from multiple sensors. For example, if exposing sensors, the API would expose a gyroscope plus several accelerometers, each returning individual one-dimensional measurements. Bundles would have to process and aggregate this data themselves — in the majority of cases, that would lead to duplication of code (and most likely to bugs in applications where they mis-process the data), but it would also allow more advanced bundles access to the raw data to do interesting things with. Conversely, if exposing properties, the vehicle device daemon would pre-aggregate the data so that the properties exposed to bundles are filtered and averaged acceleration values in three dimensions and three angular dimensions. This would simplify implementation within bundles, at the cost of preventing a small class of interesting bundles from accessing the raw data they need.

For the sake of keeping bundles simpler, and hence with potentially fewer bugs, this design exposes properties rather than sensors in the SDK API. This also means that the potentially latency sensitive aggregation code happens in the daemon, rather than in bundles which receive the data over D-Bus, which has variable latency.

Similarly, the hardware API must expose properties as well, rather than individual devices. It may aggregate data where appropriate (for example, if it has information which is useful to the aggregation process which it cannot pass on to the vehicle device daemon). This also means that a set of device semantics, separate from the W3C Vehicle Data property semantics, does not have to be defined; nor a mapping between it and the properties.

Property naming

Properties exposed in the SDK API must be named following the Vehicle Signal Specification (VSS) naming guidelines[36]. VSS defines a 'tree-like' logical taxonomy of the vehicle, (formally a Directed Acyclic Graph), where major vehicle structures (e.g. body, engine) are near the top of the tree and the logical assemblies and components that comprise them, are defined as their child nodes. Each of the child nodes in the tree is further decomposed into its logical constituents, and the process is repeated until leaf nodes are reached. A leaf node is a node at the end of a branch that cannot be decomposed because it represents a single signal or data attribute value. For example some of the properties of DriveTrain transmission and fuel system are exposed with these names:

- Drivetrain.Transmission.Speed[37]
- Drivetrain.Transmission.TravelledDistance[38]
- DriveTrain.FuelSystem.TankCapacity[39]

The element hops from the root to the leaf is called path. Properties are named according to their path from the root of the tree toward the node itself and each element in the path is delimited by using the dot notation.

[36] https://genivi.github.io/vehicle_signal_specification/rule_set/basics/#addressing-nodes
[37] https://www.w3.org/Submission/vsso/#VehicleSpeed
[38] https://www.w3.org/Submission/vsso/#TravelledDistance
[39] https://www.w3.org/Submission/vsso/#tankCapacity

Property names are formed of components in the data tree (which may contain the letters a-z, A-Z, and the digits 0-9; they must start with a letter a-z or A-Z, and must be in CamelCase) separated by dots. Property names must start and end with a component (not a dot) and contain one or more components.

If an OEM needs to expose a custom (non-standardised) property, they must define them underneath the private branch[40] which is provided by VSS to facilitate OEM specific properties.

High bandwidth or low latency sensors

Sensors which provide high bandwidth outputs, or whose outputs must reach the bundle within certain latency bounds (as opposed to simply being aggregated by the vehicle device daemon within certain latency bounds), will be handled out of band. Instead of exposing the sensor data via the vehicle device daemon, the address of some out of band communications channel will be exposed. For video devices, this might be a V4L device node; for audio devices it might be a PulseAudio device identifier. Multiplexing access to the device is then delegated to the out of band mechanism.

This considerably relaxes the performance requirements on the vehicle device daemon, and allows the more specialist high bandwidth use cases to be handled by more specialised code designed for the purpose.

Timestamps and uncertainty bounds

The W3C Vehicle Signal Specification does not define uncertainty fields for any of its data types (for example, VehicleSpeed[41] contains a single speed field measured in kilometres per hour). However, it allows the extensibility, so the data types exposed by the vehicle device daemon should all include an extension field specifying the uncertainty (accuracy) of the measurement, in appropriate units; and another specifying the timestamp when the measurement was taken, in monotonic time (in the CLOCK_MONOTONIC[42] sense).

For example, the Apertis VehicleSpeed update looks like this:

```
[('Drivetrain.Transmission.Speed',                -> property name
    (110,  0.3,  38003116),                                               -
> value field (speed, uncertainty, timestamp)
  {'description': 'Latereal vehicle accelaration', -> metadata
   'id': 54,
   'type': 'Int32',
   'unit': 'km/h'})
]
```

[40] https://genivi.github.io/vehicle_signal_specification/rule_set/private_branch/

[41] https://genivi.github.io/vehicle_signal_specification/rule_set/data_entry/sensor_actuator/

[42] http://linux.die.net/man/3/clock_gettime

which represents a measurement of *speed* ± *uncertainty* (110 ± 0.3) kilometres per hour.

Registering triggers and actions

When subscribing to notifications for changes to a particular property using the VehicleSignalInterface[43] interface, a program is also subscribing to be woken up when that property changes, even if the program is suspended or otherwise not in the foreground.

Once woken up, the program can process the updated property value, and potentially send a notification to the user. If the user interacts with this notification, the program may be brought to the foreground. The program must not be automatically brought to the foreground without user interaction or it will steal the user's focus, which is distracting.

> See the draft compositor security design

Alternatively, the program could process the updated property value in the background without notifying the user.

The VehicleSignalInterface interface may be extended to support notifications only when a property value is in a given range; a degenerate case of this, where the upper and lower bounds of the range are equal, would support notifications for property values crossing a threshold. This would most likely be implemented by adding optional min and max parameters to the VehicleSignalInterface.subscribe() method.

Bulk recording of sensor data

This is a slightly niche use case for the moment, and can be handled by an application bundle running an agent process which is subscribed to the relevant properties and records them itself. This is less efficient than having the vehicle device daemon do it, as it means more processes waking up for changes in sensor data, but avoids questions of data formats to use and how and when to send bulk data between the vehicle device daemon and the application bundle's agent.

If the implementation of this is moved into the vehicle device daemon, the lifecycle of recorded data must be considered: how space is allocated for the data's storage, when and how the application bundle is woken to process the data, and what happens when the allocated storage space is filled.

Security

The vehicle device daemon acts as a privilege boundary between all bundles accessing devices, between the bundles and the devices, and between each back-end service. Application bundles must request permissions to access sensor data

[43] http://www.w3.org/2014/automotive/vehicle_spec.html#widl-VehicleSignalInterface-subscribe-unsigned-short-VehicleInterfaceCallback-callback-Zone-zone

in their manifest (see the Applications Design document), and must separately request permissions to interact with actuators. The split is because being able to control devices in the vehicle is more invasive than passively reading from sensors — it is safety critical. A sensible security policy may be to further split out the permissions in the manifest to require specific permissions for certain types of sensors, such as cabin audio sensors or parking cameras, which have the potential to be used for tracking the user. As adding more permissions has a very low cost, the recommendation is to err on the side of finer-grained permissions.

The manifest should additionally separate lists of device properties which the bundle *requires* access to from device properties which it *may* access if they exist. This will allow the Apertis store to hide bundles which require devices not supported by the user's vehicle.

From the permissions in the manifest, AppArmor and polkit rules restricting the program's access to the vehicle device daemon's API can be generated on installation of the bundle. See Security domains for rationale.

When interacting with the vehicle device daemon, a program is securely identified by its D-Bus connection credentials, which can be linked back to its manifest — the vehicle device daemon can therefore check which permissions the program's bundle holds and accept or reject its access request as appropriate. Therefore, the vehicle device daemon acts as 'the underlying operating system' in controlling access, in the phrasing used by[44] the W3C specification. It enforces the security boundary between each bundle accessing devices, and between the intra- and inter-vehicle networks. The vehicle device daemon forms a separate security domain from any of the applications.

Each backend service is a separate security domain, meaning that the vehicle device daemon is in a separate security domain from the intra-vehicle networks.

The daemon may rate-limit API requests from each program in order to prevent one program monopolising the daemon's process time and effectively causing a denial of service to other bundles by making API calls at a high rate. This could result from badly implemented programs which poll sensors rather than subscribing to change notifications from them, for example; as well as malicious bundles.

Due to its complexity, low level in the operating system, and safety criticality, the vehicle device daemon requires careful implementation and auditing by an experienced developer with knowledge of secure software development at the operating system level and experience with relevant technologies (polkit, AppArmor, D-Bus).

The threat model under consideration is that of a malicious or compromised bundle which can execute any of the D-Bus SDK APIs exposed by the daemon, with full manifest privileges for sensor access. A second threat model is that of

[44] http://www.w3.org/2014/automotive/vehicle_spec.html#security

a compromised backend service, which can execute any of the D-Bus hardware APIs exposed by the daemon.

Security domains

There are various security technologies available in Apertis for use in restricting access to sensors and actuators. See the Security Design for background on them; especially §9, Protecting the driver assistance system from attacks. These technologies can only be used on the boundaries between security domains. In this design, each application bundle is a single security domain (encompassing all programs in the bundle, including agents and helper programs); the vehicle device daemon is another domain; and each of the backend services are in a separate domain (including the vehicle networks they each use).

Application bundle and another application bundle or the rest of the system

Separation of the security domains of different application bundles from each other and from the rest of the system is covered in the Applications and Security designs.

Application bundle and vehicle device daemon

The boundary between an application bundle and the vehicle device daemon is the Sensors and Actuators SDK API, implemented by the daemon and exposed over D-Bus. The bundle's AppArmor profile will grant access to call any method on this interface if and only if the bundle requests access to one or more devices in its manifest. Note that AppArmor is not used to separate access to different sensors or actuators — it is not fine-grained enough, and is limited to allowing or denying access to the API as a whole.

A separate set of polkit[45] rules for the bundle control which devices the bundle is allowed to access; these rules are generated from the bundle's manifest, looking at the specific devices listed. Given a set of polkit actions defined by the vehicle device daemon, these rules should permit those actions for the bundle.

For example, the daemon could define the polkit actions:

- org.apertis.vehicle_device_daemon.EnumerateVehicles: To list the available vehicles or subscribe to notifications of changes in the list.
- org.apertis.vehicle_device_daemon.EnumerateDevices: To list the available devices on a given vehicle (passed as the vehicle variable on the action) or subscribe to notifications of changes in the list.
- org.apertis.vehicle_device_daemon.ReadProperty: To read a property, i.e. access a sensor, or subscribe to notifications of changes to the property

[45] http://www.freedesktop.org/software/polkit/docs/master/polkit.8.html

value. The vehicle ID and property name are passed as the vehicle and property variables on the action.

- org.apertis.vehicle_device_daemon.WriteProperty: To write a property, i.e. operate an actuator. The vehicle ID, property name and new value are passed as the vehicle, property and value variables on the action.

The default rules for all of these actions must be polkit.Result.NO.

If a bundle has access to any device, it is safe and necessary to grant it access to enumerate *all* vehicles and devices (the Enumerate* actions above) — otherwise the bundle cannot check for the presence of the devices it requires. Knowledge of which devices are connected to the vehicle should not be especially sensitive — it is expected that there will not be a sufficient variety of devices connected to a single vehicle to allow fingerprinting of the vehicle from the device list, for example.

An application bundle, org.example.AccelerateMyMirror, which requests access to the vehicle.throttlePosition.value property (a sensor) and the vehicle.mirror.mirrorPan property (an actuator) would therefore have the following polkit rule generated in /etc/polkit-1/rules.d/20-org.example.AccelerateMyMirror.rules:

```
polkit.addRule (function (action, subject) {
  if (subject.credentials != 'org.example.AccelerateMyMirror') {
  /* This rule only applies to this bundle.
   * Defer to other rules to handle other bundles. */
  return polkit.Result.NOT_HANDLED;
}

if (action.id == 'org.apertis.vehicle_device_daemon.EnumerateVehicles' ||
   action.id == 'org.apertis.vehicle_device_daemon.EnumerateDevices') {
  /* Always allow these. */
  return polkit.Result.YES;
}

if (action.id == 'org.apertis.vehicle_device_daemon.ReadProperty' &&
    action.lookup ('property') == 'vehicle.throttlePosition.value') {
  /* Allow access to this specific property. */
  return polkit.Result.YES;
}

if (action.id == 'org.apertis.vehicle_device_daemon.WriteProperty' &&
    action.lookup ('property') == 'vehicle.mirror.mirrorPan') {
  /* Allow access to this specific property,
   * with user authentication. */
  return polkit.Result.AUTH\_USER;
}

/* Deny all other accesses. */
return polkit.Result.NO;
});
```

In the rules, the subject is always the program in the bundle which is requesting access to the device.

Open question: What is the exact security policy to implement regarding separation of sensors and actuators? For example, bundle access to sensors could always be permitted without prompting by returning polkit.Result.YES for all sensor accesses; but actuator accesses could always be prompted to the user by returning polkit.Result.AUTH_SELF. The choice here depends on the desired user experience.

Vehicle device daemon and a backend service

The boundary between the vehicle device daemon and one of its backend services is the Sensors and Actuators hardware API, implemented by the daemon and exposed over D-Bus. The backend service's AppArmor profile will grant access to call any method on this interface. Note that AppArmor is not used to grant or deny permissions to expose particular properties — it is not fine-grained enough, and is limited to allowing or denying access to the API as a whole.

In order to limit the potential for a compromised backend service to escalate its compromise into providing malicious sensor data for any sensor on the system, each backend service must install a file which lists the Vehicle Data properties it might possibly ever provide to the vehicle device daemon. The vehicle device daemon must reject properties from a backend service which are not in this list. The list must not be modifiable by the backend service after installation (i.e. it must be read-only, readable by the vehicle device daemon).

Furthermore, if a backend service is found to be exploitable after being deployed, it must be possible for the vehicle device daemon to disable it. This is expected to typically happen with backend services provided by application bundles, as opposed to those provided by OEMs or third parties (as these should go through stricter review, and disabling them would have a much larger impact). The vehicle device daemon must have a blacklist of backend services which it never loads. It must check the credentials of D-Bus messages from backend services against this blacklist.

> Using GetConnectionCredentials, which returns an unforgeable identifier for the peer: http://dbus.freedesktop.org/doc/dbus-specification.html#bus-messages-get-connection-credentials

In order to support one (vulnerable) version of a backend service being blacklisted, but not the next (fixed) version, the blacklist must contain version numbers, which should be compared against the installed version number of the backend service as listed in the system-wide application bundle manifest store.

Vehicle device daemon and the rest of the system

The vehicle device daemon itself must not be able to access any of the vehicle buses or any networks. It must be run as a unique user, which owns the daemon's binary, with its DAC permissions set such that other users (except root) cannot run it. It must not have access to any device files. See §9, Protecting the driver assistance system from attacks, of the Security design for more details.

Backend service and another backend service or the rest of the system

In order to guarantee it is the only program which can access a particular vehicle bus or network, each backend service should run as a unique user. The service's binary must be owned by that user, with its DAC permissions set such that other users (except root) cannot run it. Any device files which it uses for access to the underlying vehicle networks must be owned by that user, with their DAC permissions set such that other users cannot access them, and udev rules in place to prevent access by other users. If the backend needs access to a (local) network interface to communicate with the vehicle network buses, that interface must be put in a separate network namespace, and the CLONE_NEWNET flag used when spawning the backend service to put it in that namespace. This prevents the service from accessing other network interfaces; and prevents other processes from accessing the buses. See §9, Protecting the driver assistance system from attacks, of the Security design for more details.

SDK emulator

Typically, it should not be possible for one program to have access to both the vehicle device daemon's SDK API and its hardware API (this access is controlled by AppArmor). However, the SDK emulator is a special case which needs access to both — so either this must be possible as a special case, or the SDK emulator must be split into a backend service process and a UI process, which communicate via another D-Bus connection.

Apertis store validation

Application bundles which request permissions to access devices must undergo additional checks before being put on the Apertis store. This is especially important for bundles which request access to actuators, as those bundles are then potentially safety critical.

Checks for access to sensors

Suggested checks for bundles requesting read access to sensors:

- The bundle does not send privacy-sensitive data to services outside the user's control (for example, servers not operated by the user; see the User Data Manifesto[46]), either via network transmission, logging to local storage, or other means, without the user's consent. Any data sent *with* the user's consent must only be sent to services which follow the User Data Manifesto. For example (this list is not exhaustive):
  - Tracking the vehicle's movements.
  - Monitoring the user's conversations (audio recording).
- The bundle does not have access to uniquely identifiable information, such as a vehicle identification number (VIN). Any exceptions to this would need stricter review.
- The bundle clearly indicates when it is gathering privacy-sensitive data from sensors. For example, a 'recording' light displayed in the UI when listening using a microphone.

  1.

[46] https://userdatamanifesto.org/

Checks for access to actuators

Suggested checks for bundles requesting write access to actuators:

- The bundle does not additionally have network access.
- Actuators are only operated while the vehicle is not driving. Any exceptions to this would need even stricter review.
- Manual code review of the entire bundle's source code by a developer with security experience. The entire source code must be made available for review by the bundle developer, as it is all run in the same security domain. For example (this list is not exhaustive):
  - Looking for ways the bundle could potentially be exploited by an attacker.
  - Checking that the bundle cannot use the actuator inappropriately during normal operation if it encounters unexpected circumstances. (For example, checking that arithmetic bugs don't exist which could cause an actuator to be operated at a greater magnitude than intended by the bundle developer.)

Open question: The specific set of Apertis store validation checks for bundles which access devices is yet to be finalised.

Checks for backend services

Suggested checks for backend services for the vehicle device daemon, whether they are provided by an OEM, a third party or as part of an application bundle:

- The backend service does not additionally have network access.
- The backend service does not have write access to any of the file system except devices it needs, and the D-Bus socket.
- The backend service cannot access any more device nodes than it needs to support its devices.
- Manual code review of the entire bundle's source code by a developer with security experience. The entire source code must be made available for review by the bundle developer, as it is all run in the same security domain. For example (this list is not exhaustive):
  - Looking for ways the backend service could potentially be exploited by an attacker.
  - Checking that the backend service cannot use any of its actuator inappropriately during normal operation if it encounters unexpected circumstances. (For example, checking that arithmetic bugs don't exist which could cause an actuator to be operated at a greater magnitude than intended by the developer.)

- The backend service's D-Bus service is only accessible by the vehicle device daemon (as enforced by AppArmor).
- If other software is shipped in the same application bundle, it must be considered to be part of the same security domain as the backend service, and hence subject to the same validation checks.
- The backend service must pass the automated compliance test (Hardware API compliance testing).
- The backend service must not expose any properties which are not supported by the version of the vehicle device daemon which it targets as its minimum dependency (see Vehicle device daemon for information about the extension process).

Suggested roadmap

Due to the large amount of work required to write a system like this from scratch, it is worth exploring whether it can be developed in stages.

The most important parts to finalise early in development are the SDK and hardware APIs, as these need to be made available to bundle developers and OEMs to develop bundles and the backend services. There seems to be little scope for finalising these APIs in stages, either (for example by releasing property access APIs first, then adding vehicle and device enumeration), as that would result in early bundles which are incompatible with multi-vehicle configurations.

Similarly, it does not seem to be possible to implement one of the APIs before the other. Due to the fragmented nature of access to vehicle networks, the backend needs to be written by the OEM, rather than relying on one written by Apertis for early versions of the system.

Furthermore, the security implementation for the vehicle device daemon must be part of the initial release, as it is safety critical.

One area where phased development is possible is in the set of properties itself — initial versions of the daemon and backends could implement a small, core set of the properties defined in the VSS Ontology (VSSo)[47], and future versions could expand that set of properties as time is available to implement them. As each property is a public API, it must be supported as part of the SDK one it has appeared in a released version of the daemon, so it is important to design the APIs correctly the first time.

Similarly, the scope for backend services could be expanded over time. Initial releases of the system could allow only backend services written by vehicle OEMs to be used; with later releases allowing third-party backend services, then ones provided by installed application bundles.

[47] https://www.w3.org/Submission/vsso/

The emulator backend service (SDK API compliance testing and simulation) and any SDK hardware backend services (SDK hardware) should be implemented early on in development, as they should be relatively simple, and having them allows application developers to start writing applications against the service.

Requirements

- Enumeration of devices: The availability of known properties of the vehicle can be checked through the Availability interface[48]. The W3C approach considers properties, rather than devices, to be the enumerable items, but they are mostly equivalent (see Properties vs devices).
- Enumeration of vehicles: The availability of objects implementing the W3C Vehicle interface on D-Bus is exposed using an interface like the D-Bus ObjectManager API.
- Retrieving data from sensors: Properties can be retrieved through the VehicleInterface interface[49]. For high bandwidth sensors, or those with latency requirements for the end-to-end connection between sensor and bundle, data is transferred out of band (see High bandwidth or low latency sensors).
- Sending data to actuators: Properties can be set through the VehicleSignalInterface[50] interface. As with getting properties, data for high bandwidth or low latency sensors is transferred out of band.
- Network independence: The vehicle device daemon abstracts access to the underlying buses, so bundles are unaware of it.
- Bounded latency of processing sensor data: The vehicle device daemon should have its scheduling configuration set so that it can provide latency guarantees for the underlying buses.
- Extensibility for OEMs: Extensions are standardised through Apertis and released in the next version of the Sensors and Actuators API for use by the OEM.
- Third-party backends: Backend services for the vehicle device daemon can be installed as part of application bundles (either built-in or store bundles).
- Third-party backend validation: Backend services must be validated before being installed as bundles (see Checks for backend services).

[48] http://www.w3.org/2014/automotive/vehicle_spec.html#data-availability
[49] https://www.w3.org/Submission/vsso/#Vehicle
[50] http://www.w3.org/2014/automotive/vehicle_spec.html#widl-VehicleSignalInterface-subscribe-unsigned-short-VehicleInterfaceCallback-callback-Zone-zone

- Notifications of changes to sensor data: Property changes are notified via a publish–subscribe interface on VehicleSignalInterface[51]. Notification thresholds are supported by optional parameters on that interface.
- Uncertainty bounds: The W3C API is extended to include uncertainty bounds for measurements.
- Failure feedback: Through its use of Promises[52], the API allows for failure to set a property.
- Timestamping: The W3C API is extended to include timestamps for measurements.
- Triggering bundle activation: Programs are woken by subscriptions to property changes (see Registering triggers and actions).
- Bulk recording of sensor data: **Not currently implemented**, but may be implemented in future as a straightforward extension to the API. See Bulk recording of sensor data.
- Sensor security: Access to the Sensors and Actuators API is controlled by an AppArmor profile generated from permissions in the manifest. Access to individual sensors is controlled by a polkit rule generated from the same permissions. See Security.
- Actuator security: As with Sensor security; sensors and actuators are listed and controlled by the polkit profile separately.
- App-store knowledge of device requirements: As devices required by an application bundle are listed in the bundle's manifest (see Security), the Apertis store knows whether the bundle is supported by the user's vehicle.
- Accessing devices on multiple vehicles: Each vehicle is exposed as a separate D-Bus object, each implementing the W3C Vehicle interface.
- Third-party accessories: Properties for third-party accessories must be standardised through Apertis and exposed as separate interfaces on the vehicle object on D-Bus.
- SDK hardware support: SDK hardware should be supported through a separate development-only backend service written specifically for that hardware.

Open questions

1. Hardware and app APIs: The exact definition of the SDK API is yet to be finalised. It should include support for accessing multiple properties in a single IPC round trip, to reduce IPC overheads.

[51] http://www.w3.org/2014/automotive/vehicle_spec.html#widl-VehicleSignalInterface-subscribe-unsigned-short-VehicleInterfaceCallback-callback-Zone-zone

[52] http://www.w3.org/TR/2013/WD-dom-20131107/#promises

2. Interactions between backend services: The exact means for aggregating each property in the Vehicle Data specification is yet to be determined.
3. Security domains: What is the exact security policy to implement regarding separation of sensors and actuators? For example, bundle access to sensors could always be permitted without prompting by returning polkit.Result.YES for all sensor accesses; but actuator accesses could always be prompted to the user by returning polkit.Result.AUTH_SELF. The choice here depends on the desired user experience.
4. Apertis store validation: The specific set of Apertis store validation checks for bundles which access devices is yet to be finalised.

Summary of recommendations

As discussed in the above sections, we recommend:

- Implementing a vehicle device daemon which exposes the W3C Vehicle Information Access API; this will probably need to be developed from scratch.
- Documenting the hardware API and distributing it to OEMs, third parties and application developers along with a compliance test suite and a common utility library to allow them to build backend services for accessing vehicle networks.
- Documenting the SDK API and distributing it to application bundle developers along with a validation suite and simulator to allow them to build programs which use the API.
- Provide example trip logs for journeys to test against and a method for replaying them via the vehicle device daemon, so application developers can test their applications.
- Defining how to aggregate multiple values of each property in the W3C Vehicle Data API.
- Extending the W3C Vehicle Information Service Specification to expose uncertainty and timestamp data for each property.
- Extending the W3C Vehicle Information Service Specification to expose multiple vehicles and notify of changes using an interface like D-Bus ObjectManager.
- Extending the W3C Vehicle Information Service Specification to support a range of interest for property change notifications.
- Adding a property to the application bundle manifest listing which device properties programs in the bundle may access if they exist.
- Adding a property to the application bundle manifest listing which device properties programs in the bundle require access to.

- Extending the Apertis store validation process to include relevant checks when application bundles request permissions to access sensors (privacy sensitive) or actuators (safety critical). Or when application bundles request permissions to provide a vehicle device daemon backend service (safety critical).
- Modifying the Apertis software installer to generate AppArmor rules to allow D-Bus calls to the vehicle device daemon if device properties are listed in the application bundle manifest.
- Modifying the Apertis software installer to generate polkit rules to grant an application bundle access to specific devices listed in the application bundle manifest.
- Implementing and auditing strict DAC and MAC protection on the vehicle device daemon and each of its backend services, and identity checks on all calls between them.
- Defining a feedback and standardisation process for OEMs to request new properties or device types to be supported by the vehicle device daemon's API.

Sensors and Actuators API

This sections aims to compare the current status of the Vehicle device daemon for the sensors and actuators SDK API (Rhosydd[53]) with the latest W3C specifications: the Vehicle Information Service Specification[54] API and the Vehicle Signal Specification[55] data model.

It will also explain the required changes to align Rhosydd[56] to the new W3C specifications.

Rhosydd API Current State

The current Rhosydd API[57] is stable and usable implementing the Vehicle Information Service Specification[58] and using the data model specified by the Vehicle Signal Specification[59].

Considerations to align Rhosydd to the new VISS API

1. The original Vehicle API and the Rhosydd API don't exactly match 1:1 as the latter has been adapted to follow the inter-process D-Bus constraints

[53] https://docs.apertis.org/rhosydd/index.html
[54] https://www.w3.org/TR/vehicle-information-service/
[55] https://github.com/GENIVI/vehicle_signal_specification
[56] https://docs.apertis.org/rhosydd/index.html
[57] https://docs.apertis.org/rhosydd/index.html
[58] https://www.w3.org/TR/vehicle-information-service/
[59] https://github.com/GENIVI/vehicle_signal_specification

and best-practice, which are somewhat different than the ones for a in-process JavaScript API.

New vs Old Specification

1. The Vehicle Data Specification[60] data model uses attributes (data) and interface objects, where VISS uses the Vehicle Signal Specification[61] data model which is based on a signal tree structure containing different entities types (branches, rbranches, signals, attributes, and elements).

2. The Vehicle Information Service Specification[62] API objects are defined as JSON objects that will be passed between the client and the VIS Server, where Rhosydd is currently based on accessing attributes values using interface objects.

3. VISS defines a set of **Request Objects** and **Response Objects** (defined as JSON schemas), where the client must pass request messages to the server and they should be any of the defined request objects, in the same way, the message returned by the server must be one of the defined response objects.

4. The request and response parameters contain a number of attributes, among them the Action attribute which specify the type of action requested by the client or delivered by the server.

5. VISS lists well defined actions for client requests: authorize, getMetadata, get, set, subscribe, subscription, unsubscribe, unsubscribeAll.

6. The Vehicle Signal Specification[63] introduces the concept of **signals**. They are just named entities with a producer (or publisher) that can change its value over time and have a type and optionally a unit type defined.

7. The Vehicle Signal Specification[64] data model introduces a signal specification format. This specification is a YAML list in a single file called **vspec** file. This file can also be generated in other formats (JSON, FrancaIDL), and basically defines the signal and data structure tree.

8. The Vehicle Signal Specification introduces the concept of signal ID databases. These are generated from the vspec files, and they basically map signal names to ID's that can be used for easy indexing of signals without the need of providing the entire qualified signal name.

[60] http://www.w3.org/2014/automotive/data_spec.html
[61] https://github.com/GENIVI/vehicle_signal_specification
[62] https://www.w3.org/TR/vehicle-information-service/
[63] https://github.com/GENIVI/vehicle_signal_specification
[64] https://github.com/GENIVI/vehicle_signal_specification

Rhosydd New Changes

- The Vehicle Information Service Specification[65] API defines the Request and Response Objects using a JSON schema format. The Rhosydd API[66] (both the application-facing and backend-facing ones) has been updated to provide a similar API based on idiomatic DBus methods and types.
- Maps the different VISS Server actions to handle client requests to their respective DBus methods in Rhosydd.
- The internal Rhosydd data model has been updated to support all the element types defined in the Vehicle Signal Specification[67].
- It might also be required to add support to process signal ID databases in order for Rhosydd to recognize signals specified by the Vehicle Signal Specification.

Advantages

- The new VISS spec is based on a WebSocket API, and it resembles more closely the inter-process mechanism based on D-Bus in Rhosydd rather than the previous JavaScript in-process mechanism defined by the previous specification.

Conclusion

The main effort will be about updating the internal Rhosydd data model to reflect the changes introduced in the Vehicle Signal Specification[68] data model, with the extended types and metadata.

The DBus APIs, both on the application and backend sides, will need to be updated to map to the new data model. From a high-level point of view the old and new APIs are relatively similar, but a non-trivial amount of changes is expected to map the new concepts and to align to the new terminology.

The Rhosydd[69] client APIs for applications (librhosydd) and backends (libcroesor) will need to be updated to reflect the changes in the underlying DBus APIs.

Appendix: W3C API

For the purposes of completeness, the Vehicle Information Service Specification[70] is reproduced below. This is the version from the Final Business Group

[65] https://www.w3.org/TR/vehicle-information-service/
[66] https://docs.apertis.org/rhosydd/index.html
[67] https://github.com/GENIVI/vehicle_signal_specification
[68] https://github.com/GENIVI/vehicle_signal_specification
[69] https://docs.apertis.org/rhosydd/index.html
[70] https://www.w3.org/TR/vehicle-information-service/

Report 26 June 2018, and does not include the Vehicle Signal Specification[71] for brevity. The API is described as WebIDL[72], and partial interfaces have been merged.

[71] https://github.com/GENIVI/vehicle_signal_specification
[72] http://www.w3.org/TR/WebIDL/

```
[Constructor,
 Constructor(VISClientOptions options)]
interface VISClient {
  readonly attribute DOMString? host;
  readonly attribute DOMString? protocol;
  readonly attribute unsigned short? port;

  [NewObject] Promise< void> connect();
  [NewObject] Promise< unsigned long> authorize(object tokens);
  [NewObject] Promise< Metadata> getMetadata(DOMString path);
  [NewObject] Promise< VISValue> get(DOMString path);
  [NewObject] Promise< void> set(DOMString path, any value);
  VISSubscription subscribe(DOMString path, SubscriptionCallback subscriptionCallback, ErrorCallback err
  [NewObject] Promise< void> unsubscribe(VISSubscription subscription);
  [NewObject] Promise< void> unsubscribeAll();
  [NewObject] Promise< void> disconnect();
};

dictionary VISClientOptions {
  DOMString? host;
  DOMString? protocol;
  unsigned short? port;
};

dictionary VISValue {
  any value;
  DOMTimeStamp timestamp;
};

dictionary VISError {
  unsigned short number;
  DOMString? reason;
  DOMString? message;
  DOMTimeStamp timestamp;
};

enum Availability {
  "available",
  "not_supported",
  "not_supported_yet",
  "not_supported_security_policy",
  "not_supported_business_policy",
  "not_supported_other"
};
```