W3C Vehicle API Creation Guidelines and Rationale

The intent of the W3C Vehicle API specification is to provide a standard for developing HTML5/JavaScript applications to run on in-vehicle infotainment systems. The specification is intended to promote an API that allows for third party application development in a manner that is consistent across automakers. A secondary goal is to allow this API to be used by HTML5/JavaScript applications running on mobile phones to access the resources of a connected vehicle. This specification does not dictate or describe the nature of such a connection in either protocol or transport—Bluetooth, WiFi, or cloud connections are all possible.

It is recognized that mechanisms needed to access or control vehicle properties may not be identical across automotive trim levels, models, or makers. Even if technically achievable, not all automakers may be in consensus about which properties should be exposed. Some OEMs may wish for more access, while some may wish tighter control on what automotive features can be used. These factors guide this specification such that it will allow OEM extensions or restrictions. To prevent unnecessary fragmentation, extensions are only allowed in areas that are not already described by this API, and must follow the existing API format and the guidelines here. Restriction of functionality is handled by allowing an OEM to omit optional features in their implementation. Omitted features must be identifiable as "unsupported" as defined in the specification.

The target platform supported by this specification is exclusively passenger vehicles. Use of this specification for non-passenger applications (transportation vehicles, heavy machinery, farm equipment, airline infotainment, marine, military, etc.) is not prohibited, but is not covered in the design or content of the API and hence may be purposefully insufficient.

This document defines the consistent methodology applied to the set of attributes and attribute naming defined for the W3C Vehicle API standard. It provides a set of rules needed to grow/extend the APIs when necessary.

1) Data representation
    a. Units and types
        i. *Guideline*: All values of attributes are consistently represented as SI (metric) units, string, percentage, booleans, or enumerations.

           Rationale: Using percentages when possible instead of unit-based values allows a calling application to be easily adaptable when value ranges change between vehicle models. For numeric non-percentage values, SI is a consistent unit system that is globally understood and used for all scientific endeavours. Booleans or enumerations are used

when the values are from a small set of possible options. Enumerations should be created such that a caller can determine the range of possible responses without requiring this information a priori. This means that enumerations should use the standard JavaScript object.defineProperties method to set allowable values.

Attribute types should not change (for example from integer to enumeration, or from percentage to string), unless required for representation of special values (like "unknown", or "N/A").

b. Attribute value ranges and increments
    i. *Guideline*: Annotate the valid ranges for attributes through documentation.

    Rationale: Applications need to know the extent of the allowable values. However, the range of valid values should be constant for each implementation, and hence the range does not need to be dynamically accessible through the API. It is common practice in JavaScript to denote allowable values through documentation. This in turn simplifies the burden of implementing or adapting the API to each platform and vehicle.
    ii. *Guideline*: No standard mechanism is provided to query the valid subset of ranges or allowable increments for a particular attribute. Implementations can provide this feature optionally if they wish for selected attributes.

    Rationale: The implementation may not itself know the possible values that an underlying representation may take. Furthermore, a linear increment may not be possible. For example, a sensor on the fuel tank may return values from 0 to 15, which represents the position of a physical float in the tank or a liquid sensor against the side of the tank. The actual volume in the tank would be dependent on a lookup table, and computed by the service layer that receives the raw CAN data and provides the values into the JavaScript interface. Hence, the possible fuel level values received by JavaScript might be 0%, 1%, 3%, 5%, 15%, 25%, 40%, 50%, etc.. The resulting values are non-linear and may not be easily determined by the JavaScript API.

    It is complicated to generalize the mechanism to supply a valid set of values for any arbitrary attribute. Simplifying the implementation by allowing attributes to take any value within the range seems a reasonable solution, especially

since the value in providing that increment seems questionable and would be hard in practice to properly deal with within the application.

c. Update frequency
    i. *Guideline*: The API will be allowed to silently coalesce (or merge) multiple "set" or "get" calls to the same attribute if they occur faster than a reasonable rate. No API is provided to the application that allows querying the maximum update frequency.

    Terminology: "Reasonable rate" is defined uniquely against an execution profile, specific OS+web platform, bus characteristics, and vehicle model. Execution profile is used to mean that the maximum update rate may fluctuate depending on what simultaneous services the infotainment system is providing, what condition the vehicle is in (e.g. park vs drive), or potential state of the driver (e.g. estimate of cognitive workload).

    Rationale: The system needs to prevent excessive CAN/MOST/EAVB bus updates to ensure proper operation of the system under all conditions. This could be achieved through several mechanisms (adaptive CPU time partitioning, CAN message traffic monitoring, automatic message folding/caching, strict rate limit, etc). It may be extremely difficult in practice to compute a maximum rate that is usable by the application. This is especially true since that maximum rate may fluctuate and may be invalid by the time the application attempts to utilize it.

d. Value accuracy
    i. *Guideline*: All attribute values will use WebIDL types.

    Rationale. It is useful for the application to consistently apply data types that cover the attribute values to avoid introducing programming logic errors. Although an API for JavaScript only needs to support Strings, Numbers, and Booleans, WebIDL was chosen for two primary reasons. Firstly, it is compliant with W3C standards, and as such, all new W3C standards are enforcing its use. This by itself may be sufficient, but in addition, the open source cross-mobile HTML5 platform Cordova is on an eventual migration path to WebIDL (for W3C compatibility). One additional side benefit may be WebIDL promotes adaptation of this API for non-JavaScript languages.

2) List of attributes
    a. Naming rules

i. *Guideline*: Names should use standard terminology accepted within the automotive industry.

   Rationale: Adoption by all parties should use names well accepted by the industry.

ii. *Guideline:* Names should be all lower camel case (e.g. lowercaseFirstLetterFollowingWordsStartWithUpper).

   Rationale: Following relatively common and widespread JavaScript practice.

iii. *Guideline.* Names containing an acronym will contain the uppercase version of the acronym (eg. HVAC vs Hvac, VIN vs Vin), as an exception to lower camel case.

   Rationale: This is how people expect to encounter the names, also prevents issues with misunderstanding acronym as a "word".

iv. *Guideline:* Names should be consistently applied regardless of geographical region (e.g. use driver/passenger instead of left/right when applying to things that could switch sides depending on right-hand vs left-hand driving standards, use terminology that is not regional-only like British "boot" (for trunk), etc). Use American English spelling and terminology when words have multiple regional options (eg. use "Center" instead of "centre")

   Rationale: Almost all OEMs provide world-wide distributions, and creating an API that did not allow transparent operation of applications across multiple geographies is counter-productive.

v. *Guideline:* Do not use car maker specific terminology or brand names

   Rationale: Names should be consistent across automotive OEMs, and so we should avoid creating areas of conflict

vi. *Guideline:* Create attributes that are independent of vehicle capability assumptions

   Rationale: We do not want APIs to break when placed in a vehicle that does not have the capability, or that has additional capabilities. For example, a single wiper value setting does not accommodate a vehicle that has a front and rear wiper. Values that assume one fan setting for climate control do not handle driver+passenger climate, or front/mid/rear independent climate control zones.

vii. *Guideline:* Do not use attribute or enumeration names that hard-code assumptions on number. For example "ManualGear1".."ManualGear10", or "DoorsInRow1".."DoorsInRow3"

Rationale: Similar to the guideline for creating attributes independent of vehicle capability assumption, this pre-disposes knowledge of the maximum value that an attribute may attain. Artificially low limits constrain the API—what if your vehicle has more than 3 rows of seats (full-sized van)? Artificially high limits make the API look nonsensical. Both create an API that is not generic and non-orthogonal, and is not flexible for future extension. Data should be restructured such that the numerical information is independent of the attribute name (perhaps by using multiple independent attributes).

viii. *Guideline:* Create attributes that are independent of mechanism

Rationale: Vehicle buses or technologies change—we don't want the APIs to be describing obsolete technologies. For example, don't build APIs using CAN bus when the same service could be provided by MOST, FlexRay, or EAVB. Or don't assume gears are mandatory for vehicles with CVT.

ix. *Guideline:* Names should use abbreviations or acronyms when there exists a generally adopted shortened form. For example: HVAC, id. When words can provide alternative meanings if shortened, the abbreviated form should be avoided (eg. Temp for Temperature vs Temp for Temporary).

Rationale: As JavaScript is an interpreted language, longer names may have a performance effect. Shortening the names when they are unique and non-confusable can improve application execution speed.

x. *Guideline:* Attributes should be quantities that are available by querying a vehicle system, subsystem, or service, or are derived from such queried values. Fabricating attribute values by hardcoding within the API's implementation layer should not be required. A questionable attribute can be retained if evidence can be provided that any one manufacturer's vehicle supports that attribute.

Rationale: If the API implementation cannot access valid values for an attribute, it is forced with making the attribute optional (and hence dropping support for it), or programming the information somehow into the module.

Any changes that would require modifications of the manufacturing line are considered too intrusive. For example, an API that returns the color of the car would be disallowed because that information would not be contained anywhere within the vehicle. The only way that information could be made available to a JavaScript API would be if it were specially programmed into each vehicle coming off the assembly line with a color code. The provision to allow for questionable attributes to be left in the specification is because there are differences between all manufacturers on what data is supported. If the defender of an attribute under question can come up with a specific example where the attribute is implemented by some manufacturer, then it should be left in the specification. Coming up with a single counter-example is viewed as enough evidence of the value of retaining a property.

b. Create consistent organizational rules for attribute grouping
    i. *Guideline:* Base attribute organization on major underlying car systems (e.g. Power train, Body, Safety, Infotainment, Telematics, etc)

    Rationale. Data grouping should take advantage of an already existing logical organization.

3) Methods
    a. Attribute set/get
        i. *Guideline:* Additional attributes cannot be added in an implementation when the attribute in question is already present in the API specification, unless the new attribute is an alternative view and the W3C conformant attribute is already provided.

        Rationale: Applications should be able to rely on features being provided in a consistent way. Although optional attributes give the automaker flexibility to not supply attributes that are absent, unknown, or unallowed for a particular vehicle, an optional attribute mechanism should not be abused to avoid compliance with the specification. For example—if the specification calls for fuelLevel as a percentage, the implementer cannot remove fuelLevel while providing fuelInLitres, although the implementer could choose to supply both fuelLevel (in the spec) and fuelInLitres (additional extension). An application must be able to rely on attribute consistency for adherence to specification to have value.
    b. Decoding values

i. *Guideline:* Do not force applications to contain automotive specific logic (e.g. decoding VIN).

   Rationale: Developers may be mobile developers without an understanding of the proper way to decode automotive specific values. Decoding of values will need to be done by multiple applications, and it is better for the overall system if the computation can be done in a single function instead of having to be inserted into every application that requires a computed value.

c. Use of callbacks
   i. *Guideline:* All get/set methods will return their data through callbacks.

      Rationale: This is standard JavaScript practice, and allows the the greatest execution "concurrency" in a single threaded model. Blocking calls prevent other JavaScript code from running, and even though the block may be measured in milliseconds, this adds up over many calls. This is why most extensible JavaScript frameworks (like Cordova) use event-driven callbacks to return their data.
   ii. *Guideline:* Get/set functions will use error callbacks to return errors.

      Rationale: Very similar to above, this allows the caller to continue execution. It also places all error handling in centralized handlers as opposed to requiring error checking after each API function. This is the JavaScript equivalent of using an exception handler.

d. Overlap with W3C APIs
   i. *Guideline:* No functions will be provided that overlap with existing W3C APIs.

      Rationale: Reuse existing standards work, as well as ensure compatibility with the greatest number of existing programs.

e. Value histories
   i. *Guildline:* Although not prohibited, historical values will not be provided for any arbitrary attribute. If historical values are provided for an attribute, they should follow an API consistent with the existing get/set method signature; for example, getHistory(<attribute>, successCallback, errorCallback).

      Rationale: Some of the consulted contributions provide a mechanism to get the history of changes in an attribute. While this could be useful in certain use cases, this adds a

good deal of overhead to the implementation, and to the vehicle bus traffic. This also assumes that the attributes are continually queried and stored on a periodic basis, which may not be the case. Pushing support for this use case from the API to the application allows the application to control which attributes and how frequently they are queried, lessening the overall load on the system.

There may be a select subset of attributes where the value in maintaining historical values outweighs the overhead, or can potentially lessen the overhead for commonly requested attributes.

4) API meta-concerns
    a. RESTful APIs
        i. *Guideline:* All APIs should provide a REST equivalent whenever practical.

        Rationale: RESTful APIs allows calling functions from frameworks that do not otherwise allow JavaScript calls. This includes some commercial/open source JavaScript frameworks, but also makes it possible to call the functions from any language that can issue HTTP requests.
    b. Security
        i. *Guideline:* The API must include access control so that applications without sufficient privileges can be denied access to vehicle services. The granularity of this access should be fine grained (per attribute).

        Rationale: The need for security is paramount. Although security clearly affects “set” operations that could change the vehicle state, information about the driver and/or vehicle gained through unauthorized “get” requests should also be prevented.

        It is possible that this is already addressed through the W3C SysApps group <THIS NEEDS TO BE CONFIRMED>