GENIVI Web Vehicle API

11/14/2012

Justin (JongSeon) Park
LG Electronics Inc.

LG
Life's Good

Who am I?

Justin (JongSeon) Park

- Chief Research Engineer, SW Platform Lab. of LG Electronics
- 10 years experience in embedded system
- Working in automotive industry for 6 years
  - Developed IVI and Telematics system
- Participating in GENIVI Alliance regarding Web Vehicle APIs

Agenda

- Introduction of Web in Automotive
- Characteristics of Vehicle Data
- Considerations
  - Suggested Architecture
  - Principles to define Vehicle APIs
- Introduction of GENIVI Web Vehicle APIs
  - API descriptions
  - Issues
- Conclusion
- Q&A

Web Technologies for Automotive

The first target will be obviously IVI system

- ❑ Web Browsing in a vehicle
  - IVI Web Browser : Big Button, Driving Regulation, etc.
- ❑ GUI framework for HMI
  - Portability, MVC Pattern, Abundant Dev. Pool.
- ❑ Platform for App Store
  - Easily adding new features even if not for App Store

Requires Standardized Vehicle APIs

- ❑ Alternative Mirror Link
  - Exchange data via meta data instead of transferring the whole screen

How to Make Standard Vehicle APIs?

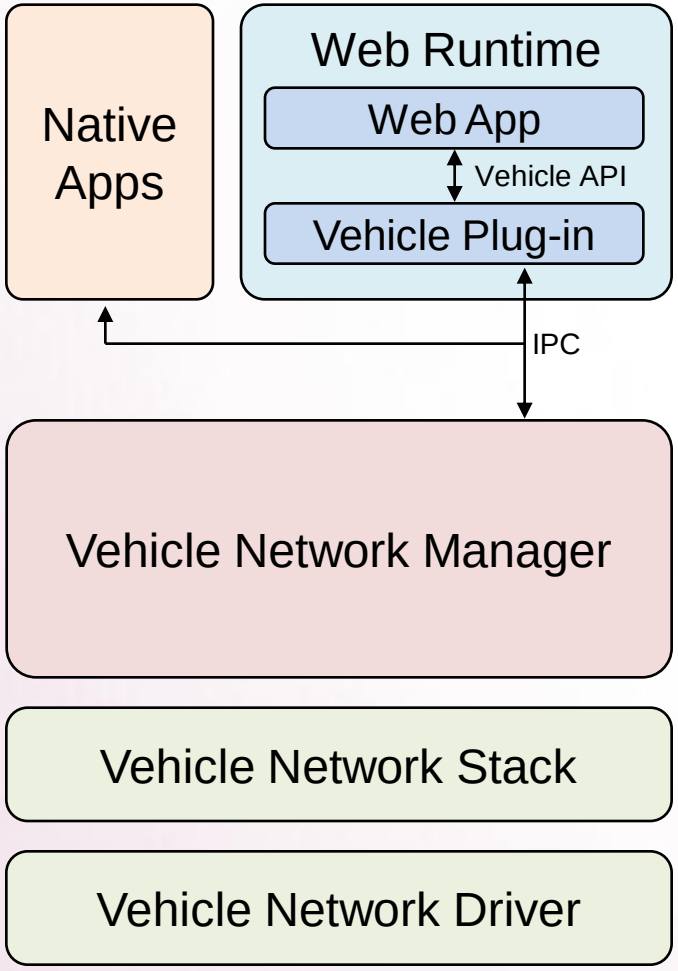
We have to understand and consider characteristics of vehicle data

- Data Characteristic
  - So many kinds of vehicle data and data types
  - A few Persistent Data - Car Type, VIN*, Model, WMI**, etc.
  - Most data are Transient; status at a moment
  - Only the latest value is meaningful (except GPS data)
- Vehicle Network Characteristic (usually CAN)
  - Real data exist somewhere else not in IVI
  - Data is broadcasted rather than query
- OEM Variations
  - Unit, Accuracy, Frequency, etc.
  - Policy - Which data are supported, Permissions

VIN* – Vehicle Identification Number (ISO 3779)
WMI** – World Manufacturer Identifier

Overall IVI Architecture for Vehicle APIs

Layered architecture according to characteristics of vehicle network



IVI Layered Architecture

- ❑ Various ways to implement it
- ❑ IPC should cover both web and native apps
- ❑ Gateway to vehicle network for Apps
  - Broadcast updates of values
  - Keep the latest values
  - Message encoding/decoding
- ❑ Commercial solution is usually used
  - Full tool chain – simulation, monitoring, automatic code-generation to apply the change of message database

How to Overcome OEM Variations

APIs must be very flexible to absorb variety

- Define as many data types as possible to prevent fragment
  - Need to gather OEM requirements as much as possible

- Allow OEMs much freedom to maintain their policy
  - A few mandatory data types
  - Most of data types need to be optional

- Consider flexibility of interface
  - Minimum number of common methods to support various data types
  - Less structured interfaces to absorb changes depending on OEMs

Web Vehicle API Project in GENIVI

GENIVI has full Web Vehicle API (draft version) and implementation

- Collected opinions to define the types of supported data
  - GENIVI has over 168 member companies including 11 OEMs
  - To reflect the realistic requirements, OEM survey was conducted

- Total 9 groups and 129 data types are defined
  - Vehicle Information (7)
  - Running Status (26)
  - Maintenance (8)
  - Personalization (20)
  - Driving Safety (16)
  - Vision System (11)
  - Parking (4)
  - Climate/Environment (29)
  - Electric Vehicle (8)

➔ 9 groups are defined as 9 Interfaces

➔ 2 methods(get/set) are defined to access all data as the unified way

➔ getSupportedTypes() method is defined

API Description – Common Interface

- All interfaces for data exchange are defined to inherit VehicleEvent interface.
- All vehicle data belong to a type of VehicleEvent and can be accessed as an attribute of that.

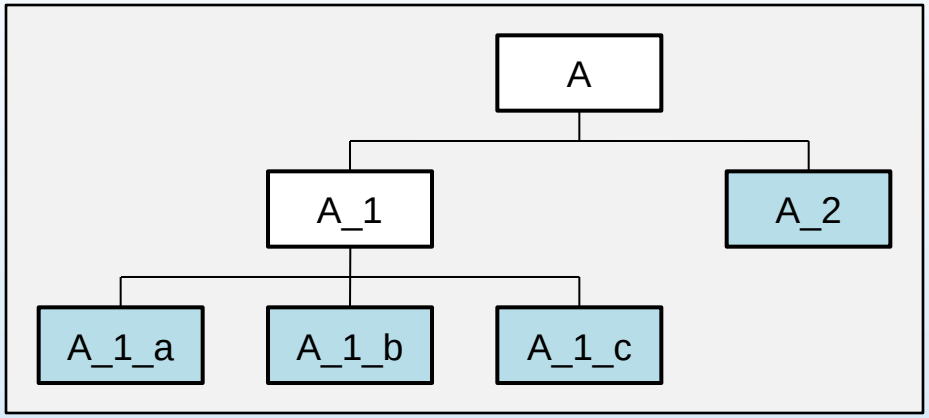
```
[NoInterfaceObject]
interface VehicleEvent : Event {};
interface RunningStatusEvent : VehicleEvent {
    ...
    readonly attribute unsigned short speedometer;
    readonly attribute unsigned short? engineSpeed;
    ...
};
```

- get/set/getSupportedEventTypes can be accessible via VehicleInterface

```
[NoInterfaceObject]
interface VehicleInterface : EventTarget {
    void get(VehicleEventType type, VehicleDataHandler handler, ErrorCallback errorCB);
    void set(VehicleEventType type, VehicleEvent data, SuccessCallback successCB, ErrorCallback errorCB);
    VehicleEventType[] getSupportedEventTypes(VehicleEventType type, boolean writable);
};
```

API Description – Multiple Data Access (1/3)

- Well-structured Interface
  - Some data have relations to others; these produce a type of data structure
  - Especially, a Setting method requires a set of attributes at a time
  - Usually, these are defined as a structured data types - Interfaces
  - Good for Clarity. But flexibility is inhibited

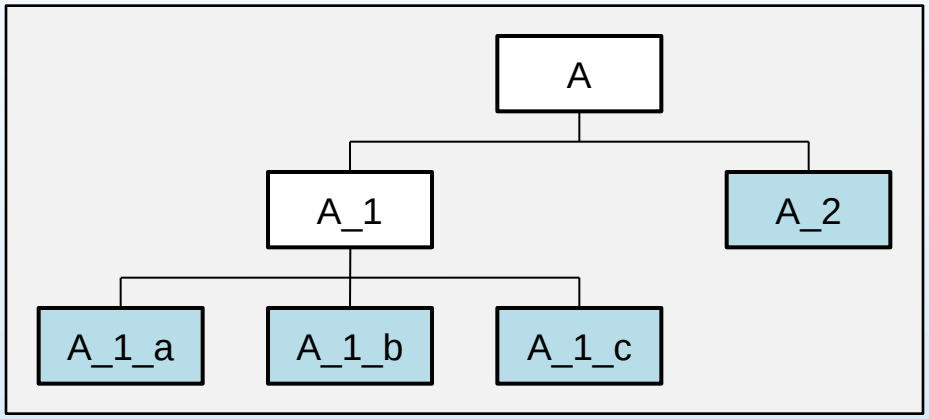



```
Interface A_1 : Event {
    attribute A_1_a;
    attribute A_1_b;
    attribute A_1_c;
}

Interface A : Event {
    attribute A_1;
    attribute A_2;
}
```

API Description – Multiple Data Access (2/3)

- Less structured interface for flexibility
  - Real data: A_1_a, A_1_b, A_1_c, A_2
  - Virtual type: A, A_1
  - Special attribute "Type" is used as an ID to identify the intended type and the range of validity of data.



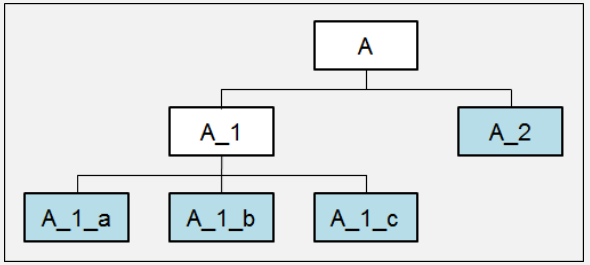

```
Interface A : Event {
    attribute Type;
    attribute A_1_a;
    attribute A_1_b;
    attribute A_1_c;
    attribute A_2;
}
```

```
const Type A = "A";
const Type A_1 = "A_1";
const Type A_1_a = "A_1_a";
const Type A_1_b = "A_1_b";
const Type A_1_c = "A_1_c";
const Type A_2 = "A_2";
```

API Description – Multiple Data Access (3/3)

- Handling multiple data at a time (cont'd)
  - Example code

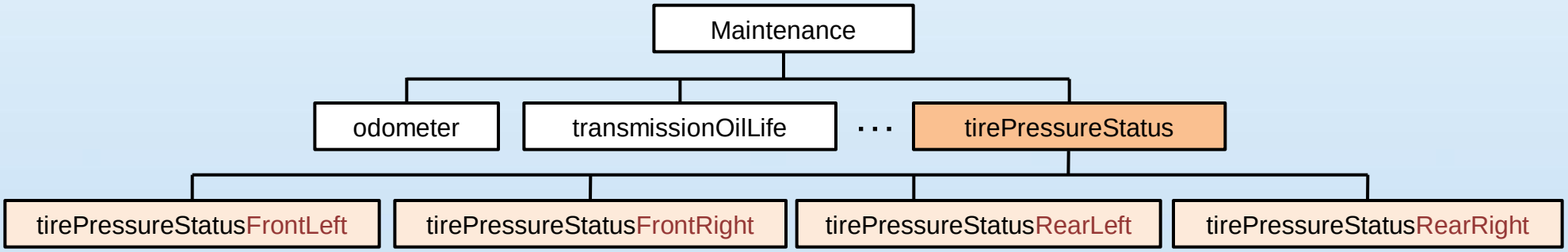
```
function handleInterfaceA(objA) {
    if (objA.type == "A_1") {
        console.log("value A_1_a = "+objA.A_1_a);      // It's valid.
        console.log("value A_1_b = "+objA.A_1_b);      // It's valid.
        console.log("value A_1_c = "+objA.A_1_c);      // It's valid.
        console.log("value A_2 = "+objA.A_2);          // It's possible but the value is invalid in our rules.
    }
    else if (objA.type == "A_2") {
        console.log("value A_2 = "+objA.A_2);          // It's valid.
    }
}
```



API Description – Example (1/6)

- Tire pressure status in MaintenanceEvent interface

```
interface MaintenanceEvent : VehicleEvent {
    const VehicleEventType MAINTENANCE = "maintenance";
…….
    const VehicleEventType MAINTENANCE_TIRE_PRESSURE_STATUS = "maintenance_tire_pressure_status";
    const VehicleEventType MAINTENANCE_TIRE_PRESSURE_STATUS_FRONT_LEFT = "maintenance_tire_pressure_status_front_left";
    const VehicleEventType MAINTENANCE_TIRE_PRESSURE_STATUS_FRONT_RIGHT = "maintenance_tire_pressure_status_front_right";
    const VehicleEventType MAINTENANCE_TIRE_PRESSURE_STATUS_REAR_LEFT = "maintenance_tire_pressure_status_rear_left";
    const VehicleEventType MAINTENANCE_TIRE_PRESSURE_STATUS_REAR_RIGHT = "maintenance_tire_pressure_status_rear_right";
…….
    const unsigned short TIRE_PRESSURE_STATUS_NORMAL = 0;
    const unsigned short TIRE_PRESSURE_STATUS_LOW = 1;
    const unsigned short TIRE_PRESSURE_STATUS_HIGH = 2;
…….
    readonly attribute unsigned short? tirePressureStatusFrontLeft;
    readonly attribute unsigned short? tirePressureStatusFrontRight;
    readonly attribute unsigned short? tirePressureStatusRearLeft;
    readonly attribute unsigned short? tirePressureStatusRearRight;
…….
};
```



API Description – Example (2/6)

- Getting a single vehicle data
  - Let's get the tire pressure status for the front left tire and notice the status to the driver
  - Call the get function with a callback function (handleVehicleData)

```
vehicle.get('maintenance_tire_pressure_status_front_left', handleVehicleData, handleError);
function handleVehicleData(data) {
        if (data.tirePressureStatusFrontLeft == 0) {
                alert('Tire pressure status (front-left) is normal.');
        } else if (data.tirePressureStatusFrontLeft == 1) {
                alert('Tire pressure status (front-left) is low.');
        } else if (data.tirePressureStatusFrontLeft == 2) {
                alert('Tire pressure status (front-left) is high.');
        }
}
```

API Description – Example (3/6)

- Getting multiple vehicle data
  - Let's get tire pressure status for all tires simultaneously
  - In the previous way, you have to get the status of each tire.

```
vehicle.get('maintenance_tire_pressure_status_front_left', handleVehicleData, handleError);
vehicle.get('maintenance_tire_pressure_status_front_right', handleVehicleData, handleError);
vehicle.get('maintenance_tire_pressure_status_rear_left', handleVehicleData, handleError);
vehicle.get('maintenance_tire_pressure_status_rear_right', handleVehicleData, handleError);
function handleVehicleData(data) {
        if ((data.tirePressureStatusFrontLeft != 0) || (data.tirePressureStatusFrontRight != 0) ||
           (data.tirePressureStatusRearLeft != 0) || (data.tirePressureStatusRearRight != 0)) {
                alert('Check tire pressure.');
        }
}
```

  - However, with the upper level type, the code becomes quite simple.

```
vehicle.get('maintenance_tire_pressure_status', handleVehicleData, handleError);
```

API Description – Example (4/6)

- Adding event listener(s)
  - Let's add an event listener to monitor the tire pressure status for the front left tire.

```
vehicle.addEventListener('maintenance_tire_pressure_status_front_left', handleVehicleData, false);
```

  - Also, you can use the upper level type to add multiple listeners.

```
vehicle.addEventListener('maintenance_tire_pressure_status', handleVehicleData, false);
```

  - A callback function (*handleVehicleData*) is called whenever any of tire pressure status is changed.

API Description – Example (5/6)

- Setting a single vehicle data
  - Assume that driver seat position can be set in this vehicle.
  - Let's set the driver seat position for recline seatback.

```
interface PersonalizationEvent : VehicleEvent {
…….
   const VehicleEventType PERSONALIZATION_DRIVER_SEAT_POSITION = "personalization_driver_seat_position";
   const VehicleEventType PERSONALIZATION_DRIVER_SEAT_POSITION_RECLINE_SEATBACK =
                                                                    "personalization_driver_seat_position_recline_seatback";
   const VehicleEventType PERSONALIZATION_DRIVER_SEAT_POSITION_SLIDE = "personalization_driver_seat_position_slide";
   const VehicleEventType PERSONALIZATION_DRIVER_SEAT_POSITION_CUSHION_HEIGHT = "personalization_driver_seat_position_cushion_height";
   const VehicleEventType PERSONALIZATION_DRIVER_SEAT_POSITION_HEADREST = "personalization_driver_seat_position_headrest";
   const VehicleEventType PERSONALIZATION_DRIVER_SEAT_POSITION_BACK_CUSHION = "personalization_driver_seat_position_back_cushion";
   const VehicleEventType PERSONALIZATION_DRIVER_SEAT_POSITION_SIDE_CUSHION = "personalization_driver_seat_position_side_cushion";
…….
   readonly attribute unsigned short? driverSeatPositionReclineSeatback;
   readonly attribute unsigned short? driverSeatPositionSlide;
   readonly attribute unsigned short? driverSeatPositionCushionHeight;
   readonly attribute unsigned short? driverSeatPositionHeadrest;
   readonly attribute unsigned short? driverSeatPositionBackCushion;
   readonly attribute unsigned short? driverSeatPositionSideCushion;
…….
};
```

API Description – Example (6/6)

- Setting a single vehicle data
  - Create an object (*obj*) and add an attribute in the *obj*.

```
var obj = new Object();
obj.driverSeatPositionReclineSeatback = 0;
vehicle.set('personalization_driver_seat_position_recline_seatback', obj, handleSuccess, handleError);
```

- Setting multiple vehicle data
  - Let's set all driver seat position.
  - Just add attributes to the *obj* and use the upper level type.

```
var obj = new Object();
obj.driverSeatPositionReclineSeatback = 0;
obj.driverSeatPositionSlide = 0;
obj.driverSeatPositionCushionHeight = 0;
obj.driverSeatPositionHeadrest = 0;
obj.driverSeatPositionBackCushion = 0;
obj.driverSeatPositionSideCushion = 0;
vehicle.set('personalization_driver_seat_position', obj, handleSuccess, handleError);
```

Pros and Cons

- Pros
  - Various data types are supported in accordance with GENIVI members
  - Seamless way of access for all data types via minimum APIs and interfaces
  - Flexibility for various supported types
  - Various granularity is possible
  - Easily modifiable to fit OEM's own purpose

- Cons
  - New way for multiple access might be unfamiliar
    - Especially, when an event handler is registered to listen a group ID, leaf node events are fired to it.
  - Data is exchanged as a unified structure - tens of bytes overhead

➔ GENIVI Web Vehicle API is still in progress
  - Hope to make it better to reflect many other opinions

Conclusion

How to standardize Web Vehicle API successfully?

- Flexibility
  - Vehicle API depends on rigid factors such as vehicle network protocol and OEM's policy
- Generality
  - Should be fit for many OEM's requirements
  - Limited coverage will cause additional work and fragmentation, which make it less meaningful
- Timing
  - Web Vehicle API needs to be standardized very soon
  - Many OEMs are now working on it in their own way
  - As time goes on, it will be harder to convince OEMs to adopt it

Thank you for your attention

Any Questions?