Webots for automobiles

release 1.0.0

www.cyberbotics.com

David Mansolino
david.mansolino@epfl.ch

October 24, 2014

Foreword

This document presents the tools developed to extend the Webots robots simulation software for automobile simulation. The reader is supposed to be already familiar with Webots software and documentation, including the Webots *User Guide* and *Reference Manual* (both available from Cyberbotics' web site).

The first section of this document focuses on various PROTOs that help the user to create realistic models of vehicles. The second section introduces programming libraries used to facilitate the control of theses vehicles.

Contents

1 PROTOs

This section presents the set of PROTOs developed specifically for automobile related simulations, from the PROTO of wheel to the PROTO of a complete car.

1.1 AutomobileWheel

The *AutomobileWheel* allows the user to easily create automobile wheels. It is designed to be generic and customizable to cover a wide range of wheel configurations. The base node of the *AutomobileWheel* PROTO is a *Slot* of type *automobile wheel* so that it can only be inserted as a wheel of an automobile PROTO featuring the corresponding *automobileWheel* slots.

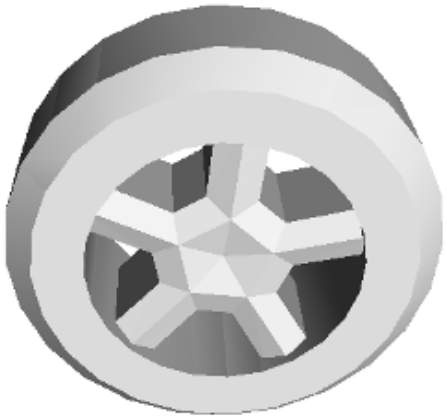

Figure 1: An AutomobileWheel PROTO with the default field values.

Here is the description of the fields of the *AutomobileWheel* PROTO:

```
1  SFFloat   thickness        0.3
2  SFFloat   tireRadius       0.4
3  SFInt32   subdivision      24
4  SFFloat   curvatureFactor  0.35
5  SFInt32   edgeSubdivision  1
6  SFFloat   rimRadius        0.28
7  SFInt32   rimBeamNumber    5
8  SFFloat   rimBeamWidth     0.1
9  SFFloat   centralInnerRadius 0.09
10 SFFloat   centralOuterRadius 0.13
11 SFFloat   rimBeamThickness 0.2
12 SFFloat   rimBeamOffset    0.03
13 SFString  contactMaterial  "default"
14 SFNode    tireAppearance   Appearance { material Material {} }
15 SFNode    rimAppearance    Appearance { material Material {} }
16 SFNode    physics          Physics {}
17 MFNode    logoSlot         []
```

thickness Defines the thickness of the wheel.

tireRadius Defines the outer radius of the wheel.

subdivision Defines the number of subdivisions for the cylinder approximation.

curvatureFactor Defines the curvature of the wheel, should be between 0 and 1.

edgeSubdivision Defines the number of subdivisions for the edge approximation.

rimRadius Defines the radius of the separation between the rim and the tire.

rimBeamNumber Defines the number of beams of the wheel.

rimBeamWidth Defines the width of the beams.

centralInnerRadius and centralOuterRadius Defines the geometry of the central part of the wheel.

rimBeamThickness Defines the lateral thickness of the beams.

rimBeamOffset Defines the lateral offset of the beams.

contactMaterial Defines the *contactMaterial* used for the wheel.

tireAppearance Specifies the appearance of the tire part of the wheel.

rimAppearance Specifies the appearance of the rim part of the wheel.

physics *Physics* node of the wheel defining all the physical characteristics.

logoSlot Extension slot allowing the user to add a *Shape* node for a brand logo.

Some sample instances of wheels are provided (see for example *BmwX5Wheel.proto*) which inherit from *AutomobileWheel*.

Figure 2: Example of modeling of a BMW X5 wheel using the AutomobileWheel PROTO.

1.2 AckermannVehicle

The *AckermannVehicle* PROTO allows the user to easily create any vehicle that complies with the ackermann model. Position and orientation of the wheels are automatically computed using the fields of the PROTO. These wheels are automatically connected to the appropriate joints in order to rotate along the correct axis. Actuators (*Motor* nodes called *right_steer* and *left_steer*) are connected to the front joints to be able to steer the vehicle.

The base node of the *AckermannVehicle* PROTO is a *Robot* from which it inherits its first seven fields. The rest of the vehicle (Shape, Sensors, other actuators) can be added using the *extensionSlot* field. Here is a description of the fields of the *AckermannVehicle* PROTO:

```
SFVec3f   translation                 0 0.4 0
SFRotation rotation                   0 1 0 0.0
SFString  name                        "vehicle"
SFString  model                       "AckermannVehicle"
SFString  controller                  "void"
SFString  controllerArgs              ""
SFBool    synchronization             TRUE
SFFloat   trackFront                  1.7
SFFloat   trackRear                   1.7
SFFloat   wheelbase                   4.0
SFFloat   minSteeringAngle           -1
SFFloat   maxSteeringAngle            1
SFFloat   suspensionFrontSpringConstant 100000
SFFloat   suspensionFrontDampingConstant 4000
SFFloat   suspensionRearSpringConstant 100000
SFFloat   suspensionRearDampingConstant 4000
SFFloat   wheelsDampingConstant       5
MFNode    extensionSlot               NULL
SFNode    boundingObject              NULL
SFNode    physics                     NULL
SFNode    wheelFrontRight             AutomobileWheel { }
SFNode    wheelFrontLeft              AutomobileWheel { }
SFNode    wheelRearRight              AutomobileWheel { }
SFNode    wheelRearLeft               AutomobileWheel { }
MFNode    axisDevicesFrontRight       [ ]
MFNode    axisDevicesFrontLeft        [ ]
MFNode    axisDevicesRearRight        [ ]
MFNode    axisDevicesRearLeft         [ ]
SFString  data                        ""
```

trackFront and trackRear Defines the front/rear distances between right and left wheels.

wheelbase Defines the distance between the front and the rear wheels axes.

minSteeringAngle and maxSteeringAngle Defines the minimum and maximum steering angle of the front wheels

suspension... Defines the characteristics of the suspension.

wheelsDampingConstant Defines the *dampingConstant* of each wheel joint used to simulate the frictions of the vehicle.

extensionSlot Extension slot allowing the user to add other nodes (e.g., sensors, shape of the vehicle, etc.).

boundingObject Physical geometry of the vehicle.

physics *Physics* node of the vehicle defining the physical parameters of the vehicle.

wheel*X* Slot to insert an *AutomobileWheel* (or any *AutomobileWheel* descendant PROTOs).

axisDevices*X* Slot to add devices in the wheels joints (such as *Brake*, *PositionSensor* and *Motor*).

data Defines an user *data* string of the *Robot* node.

The center of the vehicle (position 0 0 0) is in the center of the rear wheels axis. Any node added in the *extensionSlot* is added relatively to this position. A *Transform* node should be used to move an extension node away from this center.

You can easily create your own PROTO that inherits from the *AckermannVehicle* PROTO, see for example the *SimpleVehicle* PROTO which automatically computes the shapes of the wheel axes.

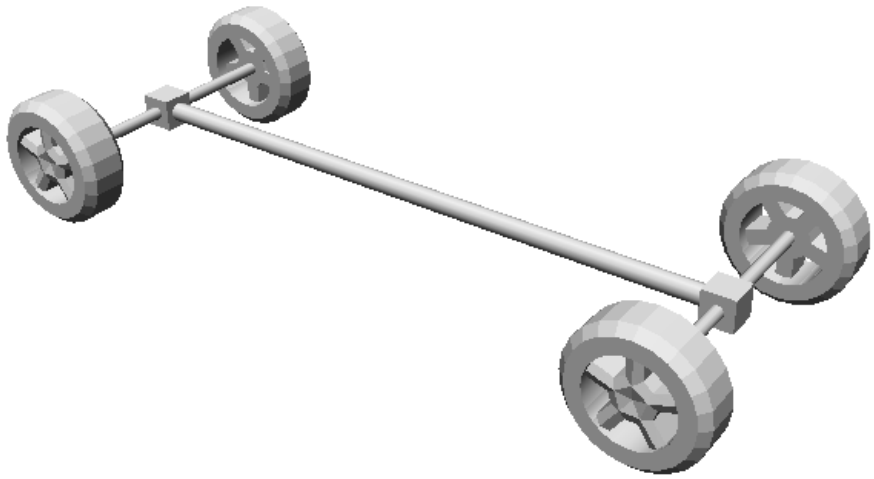

Figure 3: Result of the SimpleVehicle PROTO which inherits from the AckermannVehicle PROTO.

1.3 Car

The *Car* PROTO inherits from the *AckermannVehicle* PROTO and extends it. It should be used with the *driver* library (see section 2) in order to add a model of the engine, transmission, lights, gearbox and brake. The joint devices are automatically filled in with the appropriate devices in function of the transmission type set in the parameters.

Here is the description of the fields specific to the *Car* PROTO:

```
#fields specific to Car
  SFString  type                        "traction"
  SFString  engineType                  "combustion"
  SFFloat   brakeCoefficient            500
  SFFloat   time0To100                  10
  SFFloat   engineMaxTorque             250
  SFFloat   engineMaxPower              50000
  SFFloat   engineMinRPM                1000
  SFFloat   engineMaxRPM                4500
  SFVec3f   engineFunctionCoefficients  150 0.1 0
  MFFloat   gearRatio                   [-12 10 7 5 2.5 1]
  SFFloat   hybridPowerSplitRatio       0.25
  SFFloat   hybridPowerSplitRPM         3000
```

type Defines the transmission type of the car, supported types are: *traction* (front wheels), *propulsion* (rear wheels) or *4x4* (both).

engineType Defines the type of engine, supported types are: *combustion*, *electric*, *parallel hybrid* and *power-split hybrid* (for a serial hybrid please use electric instead). See section 2.1.1 for more information.

brakeCoefficient Defines the maximum *dampingConstant* applied by the brake on the wheels joint.

time0To100 Defines the time to accelerate from 0 to 100 km/h in seconds, this value is used to compute the wheels acceleration when controlling the car in cruising speed (see section 2.1).

engineMaxTorque Defines the maximum torque of the motor in *Nm* used to compute the electric engine torque.

engineMaxPower Defines the maximum power of the motor in *W* used to compute the electric engine torque.

engineMinRPM and engineMaxRPM Defines the working range of the engine (*engineMinRPM* not used in case of *electric engineType*).

engineFunctionCoefficients Define the coefficients of the second order function used to approximate the output torque in function of the rotational speed of the motor.

gearRatio Defines the total (not only the gearbox ratio) ratio between the rotational speed of the engine and the wheels, the number of elements defines the number of gears, the first element should be negative and is the reverse gear.

hybridPowerSplitRatio Defines the ratio of the output power of the combustion engine that is used for charging the battery in case of *power-split hybrid engineType*.

hybridPowerSplitRPM Defines the fixed rotational speed of the combustion engine in case of *power-split hybrid engineType*.

The *extensionSlot* field is filled in by default with the *AutomobileLights* PROTO. The *AutomobileLights* PROTO is used to add all the models of the regular lights present in a car (based on *LED* nodes). For each light you can specify its shape and the color emitted when the light is switched on. Of course if you don't need to have lights you can safely remove the *AutomobileLights* PROTO from *extensionSlot*.

```
1  MFNode   front          NULL
2  MFColor  frontColor     [ 0.8 0.8 0.8 ]
3  MFNode   rightIndicator NULL
4  MFNode   leftIndicator  NULL
5  MFColor  indicatorColor [ 1 0.7 0.1 ]
6  MFNode   antifog        NULL
7  MFColor  antifogColor   [ 0.8 0.8 0.8 ]
8  MFNode   braking        NULL
9  MFColor  brakingColor   [ 0.7 0.12 0.12 ]
10 MFNode   rear           NULL
11 MFColor  rearColor      [ 0.8 0.8 0.8 ]
12 MFNode   backwards      NULL
13 MFColor  backwardsColor [ 0.7 0.12 0.12 ]
```

Here again, you can easily create your own PROTO that inherits from the *Car* PROTO to define your own custom and complete model of car. Three PROTOs that inherit from the *Car* PROTO are provided. These PROTOs (see figure 4) represent three different models of car: the X5 from BMW, the C-Zero from Citroën and the Prius from Toyota.

Figure 4: Models of cars created using the AckermannVehicle PROTO.

An interesting aspect of these three PROTOs is that the *extensionSlot* is divided into four *sensorsSlot* in order to provide smart predefined positions where to put sensors (or actuators if needed), which are in the front, top, rear and center of the car. The position of the central sensors slot is always at 0 0 0 (which is the center of the rear wheels axis). For the three other sensor slots, the positions are different for each model (because the size of the cars differs), see table 1.3 for the exact positions.

Model	**Front slot translation**	**Top slot translation**	**Rear slot translation**
BmwX5	0.0 0.45 3.850	0.0 1.45 1.000	0.0 0.3 -1.000
CitroenCZero	0.0 0.05 3.075	0.0 1.35 1.075	0.0 0.3 -0.425
ToyotaPrius	0.0 0.40 3.635	0.0 1.30 1.100	0.0 0.3 -0.850

Table 1: Positions of the sensors slots.

2 Libraries

To ease the creation of controllers for any *Car* PROTOs (or any PROTOs inherited from *Car*), two libraries are provided. These two libraries are easy to use, provide high-level functionalities and save the user from knowing the name of the internal devices nodes (*Motors*, *Brakes*, etc.) of the car. This section presents and explains how to use these two libraries.

2.1 Driver library

The *driver* library provides all the usual functionality available to a human driving his own car. All the functions included in this library are explained below.

```
1 void wbu_driver_init();
2 void wbu_driver_cleanup();
3 int wbu_driver_step();
```

These functions are the equivalent of the init, cleanup and step function of any regular robot controller. As a reminder, the *init* function should be called at the very beginning of any controller program, the *cleanup* function at the end of the controller program just before exiting and the *step* function should be called in the main loop to run one simulation step. Unlike the robot step, the driver step does not have any argument, the default time step of the world being used.

```
1 void wbu_driver_set_steering_angle(double steering_angle);
2 double wbu_driver_get_steering_angle();
```

The first function is used to steer the car, it steers the front wheels according to the Ackermann geometry (left and right wheels are not steered with the exact same angle). The angle is set in radians, a positive angle steers right and a negative angle steers left. The formulas used in order to compute the right and left angles are the following (*trackFront* and *wheelbase* are the parameters of the *Car* PROTO):

$$angle_{right} = atan(\frac{1}{cot(steering\_angle) - \frac{trackFront}{2*wheelbase}})$$

$$angle_{left} = atan(\frac{1}{cot(steering\_angle) + \frac{trackFront}{2*wheelbase}})$$

The second function simply returns the steering angle (argument of the last call to the previous function).

```
1 void wbu_driver_set_cruising_speed(double speed);
2 double wbu_driver_get_target_cruising_speed();
```

Then first function activates the control in cruising speed of the car, the rotational speed of the wheels is forced (respecting the geometric differential constraint) in order

that the car moves at the speed given in argument of the function (in kilometers per hour). When the control in cruising speed is activated, the speed is directly applied to the wheel without any engine model simulation, therefore any call to functions like *wbu_driver_get_rpm()* will raise an error. The acceleration of the car is computed using the *time0To100* field of the *Car* PROTO (see section 1.3).

The second function simply returns the target cruising speed (argument of the last call to the previous function).

```
double wbu_driver_get_current_speed();
```

This function returns the current speed of the car (in kilometers per hour). The estimated speed is computed using the rotational speed of the actuated wheels and their respective radius.

```
void wbu_driver_set_throttle(double throttle);
double wbu_driver_get_throttle();
```

The first function is used in order to control the car in torque, it sets the state of the throttle. The argument should be between 0.0 and 1.0, 0 means that 0% of the output torque of the engine is sent to the wheels and 1.0 means that 100% of the output torque of the engine is sent to the wheels. For more information about how the output torque of the engine is computed see section 2.1.1.

The second function simply returns the state of the throttle (argument of the last call to the previous function).

```
void wbu_driver_set_brake(double brake);
double wbu_driver_get_brake();
```

This function brakes the car by increasing the *dampingConstant* coefficient of the rotational joints of each of the four wheels. The argument should be between 0.0 and 1.0, 0 means that no damping constant is added on the joints (no breaking), 1 means that the parameter *brakeCoefficient* of the *Car* PROTO (see section 1.3) is applied on the *dampingConstant* of each joint (the value will be linearly interpolated between 0 and *brakeCoefficient* for any arguments between 0 and 1).

The second function simply returns the state of the brake (argument of the last call to the previous function).

```
typedef enum {
  OFF=0,
  RIGHT,
  LEFT
} wbu_indicator_state;

void wbu_driver_set_indicator(int state);
void wbu_driver_set_indicator_warning(bool state);
```

The first function allows the user to set (using the *wbu_indicator_state* enum) if the indicator should be on only for the right side of the car, the left side of the car or should be off.

The second function allows the user to switch the warning on (indicator on on both side of the car) or off.

```
void wbu_driver_set_dipped_beams(bool state);
void wbu_driver_set_antifog_lights(bool state);
bool wbu_driver_get_dipped_beams();
bool wbu_driver_get_antifog_lights();
```

The first two functions are used to enable or disable the dipped beams and the anti-fog lights.

The second two functions return the state of the dipped beams or the anti-fog lights.

```
double wbu_driver_get_rpm();
```

This function returns the estimation of the engine rotation speed. Warning: if the control in cruising speed is enabled, this function returns an error because there is no engine model when control in cruising speed is enabled.

```
void  wbu_driver_set_gear(int gear);
int   wbu_driver_get_gear();
int   wbu_driver_get_gear_number();
```

The first function sets the engaged gear. An argument of -1 is used in order to engage the reverse gear, an argument of 0 is used in order to disengaged the gearbox. Any other arguments than 0 and -1 should be between 1 and the number of coefficients set in the *gearRatio* parameter of the *Car* PROTO (see section 1.3) -1.

The second function returns the currently engaged gear.

The last function simply returns the number of available gears (including the reverse gear).

2.1.1 Engine models

When the control in torque of the car is enabled, at each step the output torque of the engine is recomputed. First the rotational speed of the engine is estimated from the rotational speed of the wheels, then the output torque of the engine is computed (the formula depends on the engine type) using the rotational speed of the engine, then this output torque is multiplied by the state of the throttle and the gearbox coefficient, finally the torque is distributed (respecting the differential constraint) on the actuated wheels (depending on the car type).

Combustion engine If a, b and c are the values of the *engineFunctionCoefficients* parameter of the *Car* PROTO, the output torque is:

$$output\_torque = c * rpm^2 + b * rpm + a$$

Note: if the rpm is below the *engineMinRPM* parameter of the *Car* PROTO, *engineMinRPM* is used instead of the real rpm, but if the rpm is above the *engineMaxRPM* parameter, then the output torque is 0.

Electric engine If $maxP$ and $maxT$ are respectively the *engineMaxPower* and *engineMaxTorque* parameters of the *Car* PROTO, the ouput torque is:

$$output\_torque = min(maxT; \frac{maxP * 60}{2 * pi * rpm})$$

Parallel hybrid engine In that case, the output torque is simply the sum of the two previous models. But if the real rpm is below the *engineMinRPM* parameter of the *Car* PROTO, the combustion engine is switched off.

Serial hybrid engine Since this case is very similar to the electric engine model (from a simulation point of view), you should use the electric model instead.

Power-split hybrid engine If $ratio$ and $splitRpm$ are respectively the *hybridPowerSplitRatio* and *hybridPowerSplitRPM* parameters of the *Car* PROTO, the output torque is computed as follow:

$$output\_torque_c = c * splitRpm^2 + b * splitRpm + a$$

$$output\_torque_e = min(maxT; \frac{maxP * 60}{2 * pi * rpm})$$

$$output\_torque_{total} = output\_torque_e + (1 - ratio) * output\_torque_c$$

Here again, if the real rpm is below the *engineMinRPM* parameter of the *Car* PROTO the combustion engine is switched off.

2.2 Car library

The *car* library is supposed to be used together with the *driver* library. It provides additional information and functions to which a normal human driver of a car does not have access (e.g., changing the blinking period of the indicator or getting the value of the wheels encoders). All the functions included in this library are explained below.

```
1 void wbu_car_init();
2 void wbu_car_cleanup();
```

These two functions are respectively used to initialize and properly close the *car* library (the first one should be called at the very beginning of the controller program and the second one at the very end). If you use the *driver* library (see section 2.1) it is not needed to call these functions since they are already called from the corresponding functions of the *driver* library.

```
1  typedef enum {
2    WBU_CAR_TRACTION=0,
3    WBU_CAR_PROPULSION=1,
4    WBU_CAR_FOUR_BY_FOUR=2
5  } wbu_car_type;
6
7  typedef enum {
8    WBU_CAR_COMBUTSION_ENGINE=0,
9    WBU_CAR_ELECTRIC_ENGINE=1,
10   WBU_CAR_PARALLEL_HYBRID_ENGINE=2,
11   WBU_CAR_POWER_SPLIT_HYBRID_ENGINE=3
12 } wbu_car_engine_type;
13
14 wbu_car_type wbu_car_get_type();
15 wbu_car_engine_type wbu_car_get_engine_type();
```

These two functions return respectively the type of transmission and of engine of the car.

```
1 void wbu_car_set_indicator_period(double period);
2 double wbu_car_get_indicator_period();
```

The first function is used to change the blinking period of the indicators. The argument should be specified in seconds.

The second function returns the current blinking period of the indicators.

```
1 bool  wbu_car_get_backwards_lights();
2 bool  wbu_car_get_brake_lights();
```

This two functions return respectively the state of the backwards and brake lights (these two lights are switched on automatically by the library when appropriated).

```
double wbu_car_get_track_front();
double wbu_car_get_track_rear();
double wbu_car_get_wheelbase();
double wbu_car_get_front_wheel_radius();
double wbu_car_get_rear_wheel_radius();
```

All these functions provide important physical characteristics of the car.

```
typedef enum {
  WBU_CAR_WHEEL_FRONT_RIGHT=0,
  WBU_CAR_WHEEL_FRONT_LEFT=1,
  WBU_CAR_WHEEL_REAR_RIGHT=2,
  WBU_CAR_WHEEL_REAR_LEFT=3,
  WBU_CAR_WHEEL_NB
} wbu_car_wheel_index;

double wbu_car_get_wheel_encoder(int wheel_index);
double wbu_car_get_wheel_speed(int wheel_index);
```

These two functions return respectively the state of the wheel encoder (in radians) and the instantaneous wheel rotational speed (in radians per second). The *whee_index* argument should match a value of the *wbu_car_wheel_index* enum.

```
double wbu_car_get_right_steering_angle();
double wbu_car_get_left_steering_angle();
```

These two functions return respectively the right and left steering angle (because of the Ackermann steering geometry, the two angles are slightly different).

```
void wbu_car_enable_limited_slip_differential(bool enable);
```

These functions allow the user to enable or disable the limited differential slip (it is enabled by default). When the limited differential slip is enabled, at each time step, the torque (when control in torque is enabled) is redistributed among all the actuated wheels so that they rotate at the same speed (except the difference due to the geometric differential constraint). If the limited differential slip is disabled, when a wheel starts to slip, it will rotate faster than the others.

2.3 C++ version of the libraries

If needed, it is possible to write the controller in C++ too. Indeed a C++ version of the *driver* and *car* libraries are provided, they work the same way as the C versions of the libraries (except that the *init* and *cleaup* functions are called automatically from the constructor/destructor of the *Driver* and *Car*) classes. The methods names are very similar to the names of the C functions, see appendix A for more information.

3 Acknowledgements

This work was sponsored by the project RO2IVSim of the Swiss Commission for Technology and Innovation CTI:
(`http://transport.epfl.ch/simulator-for-mobile-robots-and-intelligent-vehicles`).

A Appendix: C++ libraries

The two libraries (*libCppDriver* and *libCppCar*) provide a C++ wrapper of the C libraries *libdriver* and *libcar*.

A.1 CppDriver

The *Driver* class provides the following methods:

```
enum { INDICATOR_OFF=0, INDICATOR_RIGHT, INDICATOR_LEFT };

Driver();
virtual ~Driver();

virtual int step();

void  setSteeringAngle(double steeringAngle);
double getSteeringAngle();

void  setCruisingSpeed(double speed);
double getTargetCruisingSpeed();

double getCurrentSpeed();

void  setThrottle(double throttle);
double getThrottle();

void  setBrake(double brake);
double getBrake();

void setIndicator(int state);
void setIndicatorWarning(bool state);

void setDippedBeams(bool state);
void setAntifogLights(bool state);

bool getDippedBeams();
bool getAntifogLights();

double getRpm();
int   getGear();
void  setGear(int gear);
int   getGearNumber();
```

A.2 CppCar

The *Car* class inherits from the *Driver* class and provides the following methods:

```
enum { TRACTION=0, PROPULSION=1, FOUR_BY_FOUR=2 };
enum {
  COMBUTSION_ENGINE=0,
  ELECTRIC_ENGINE=1,
  PARALLEL_HYBRID_ENGINE=2,
  POWER_SPLIT_HYBRID_ENGINE=3
};
enum {
  WHEEL_FRONT_RIGHT=0,
  WHEEL_FRONT_LEFT=1,
  WHEEL_REAR_RIGHT=2,
  WHEEL_REAR_LEFT=3,
  WHEEL_NB
};

Car() : Driver() {}
virtual ~Car() {}

int getType();
int getEngineType();

void  setIndicatorPeriod(double period);
double getIndicatorPeriod();

bool getBackwardsLights();
bool getBrakeLights();

double getTrackFront();
double getTrackRear();
double getWheelbase();
double getFrontWheelRadius();
double getRearWheelRadius();

double getWheelEncoder(int wheel);
double getWheelSpeed(int wheel);
double getRightSteeringAngle();
double getLeftSteeringAngle();

void enableLimitedSlipDifferential(bool enable);
```