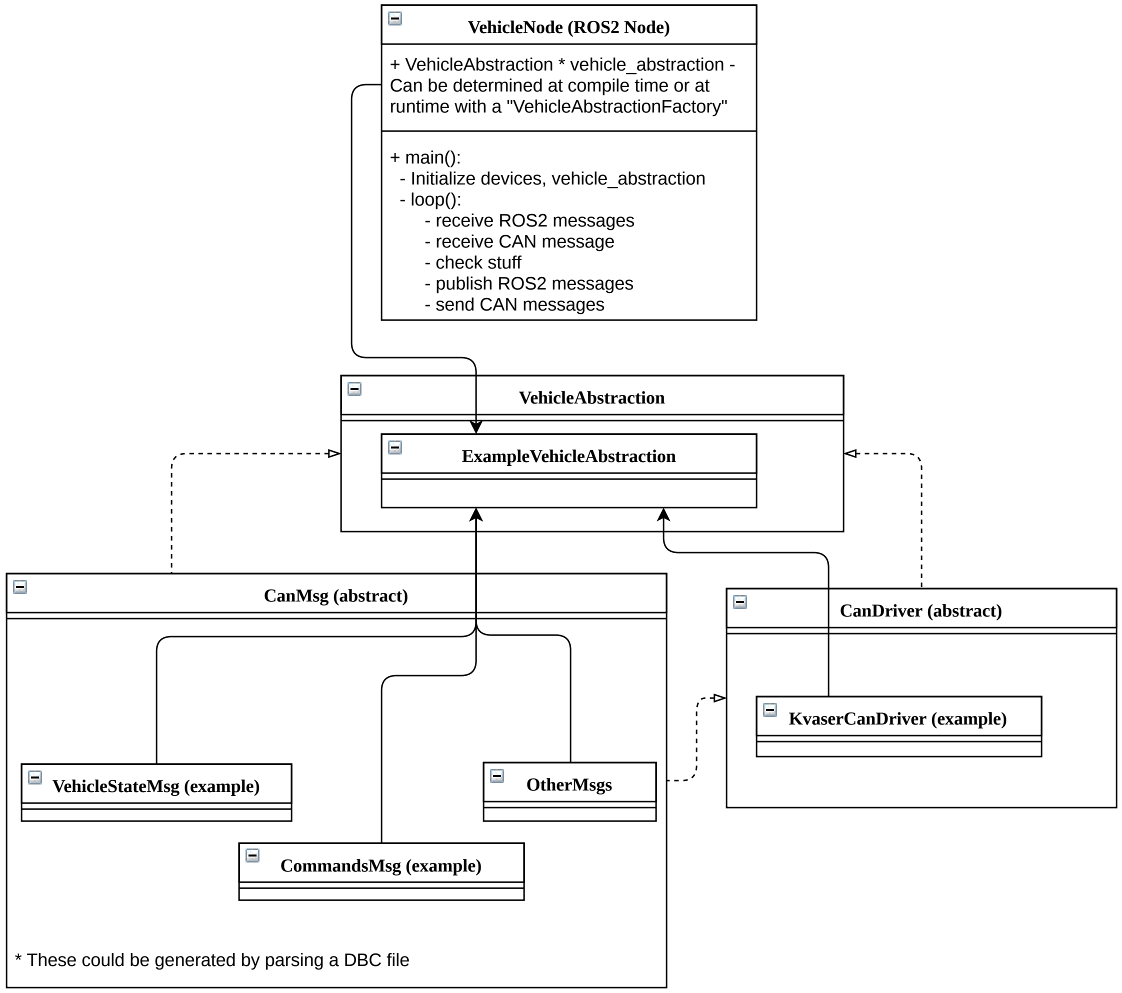
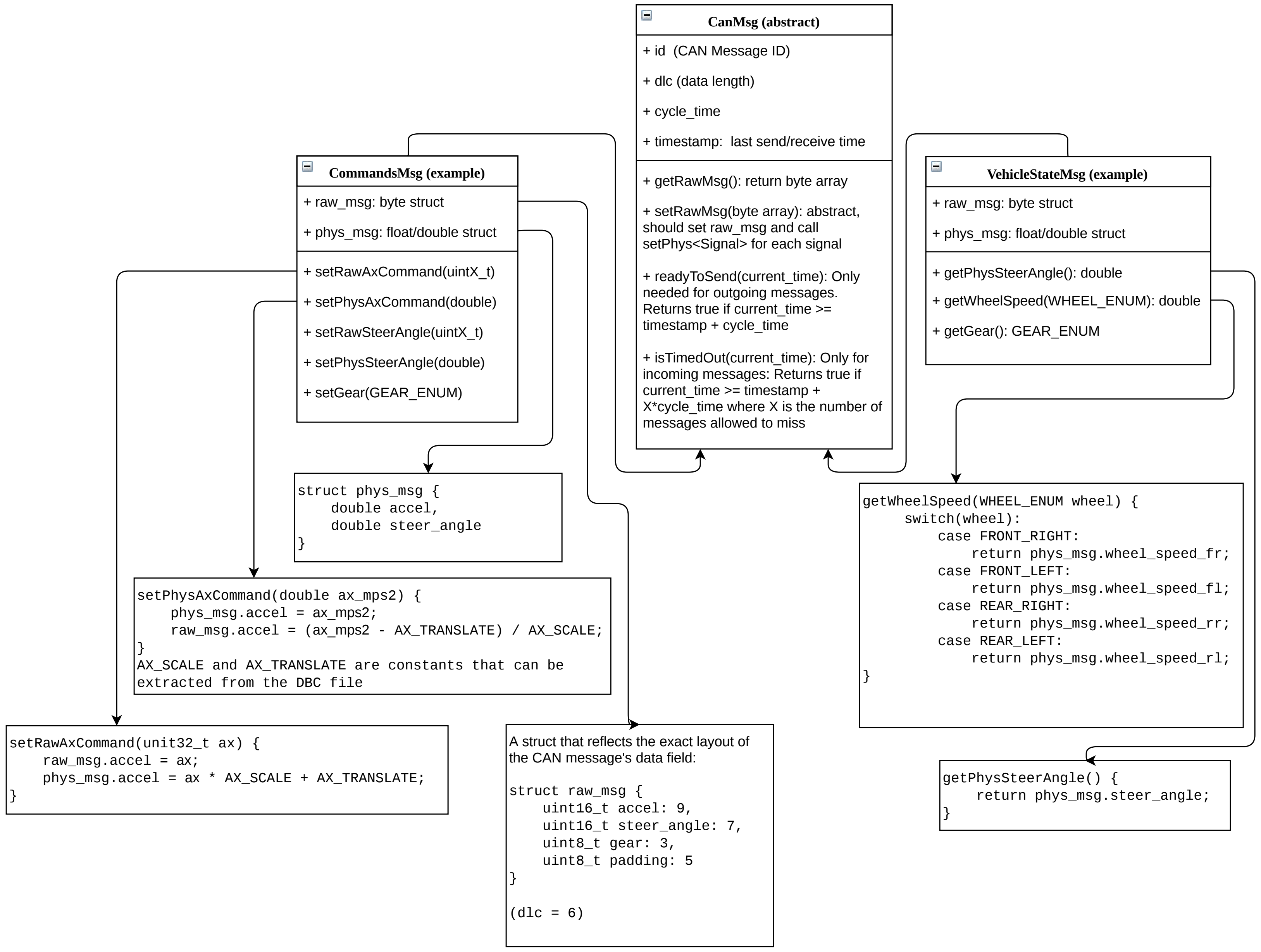

VehicleNode (ROS2 Node)
+ VehicleAbstraction * vehicle_abstraction - Can be determined at compile time or at runtime with a "VehicleAbstractionFactory"
+ main():
- Initialize devices, vehicle_abstraction
- loop():
- receive ROS2 messages
- receive CAN message
- check stuff
- publish ROS2 messages
- send CAN messages
VehicleAbstraction
ExampleVehicleAbstraction
CanMsg (abstract)
VehicleStateMsg (example)
OtherMsgs
CommandsMsg (example)
* These could be generated by parsing a DBC file
CanDriver (abstract)
KvaserCanDriver (example)



**CanMsg (abstract)**

+ id (CAN Message ID)
+ dlc (data length)
+ cycle_time
+ timestamp: last send/receive time

+ getRawMsg(): return byte array
+ setRawMsg(byte array): abstract, should set raw_msg and call setPhys<Signal> for each signal
+ readyToSend(current_time): Only needed for outgoing messages. Returns true if current_time >= timestamp + cycle_time
+ isTimedOut(current_time): Only for incoming messages: Returns true if current_time >= timestamp + X*cycle_time where X is the number of messages allowed to miss

**CommandsMsg (example)**

+ raw_msg: byte struct
+ phys_msg: float/double struct

+ setRawAxCommand(uintX_t)
+ setPhysAxCommand(double)
+ setRawSteerAngle(uintX_t)
+ setPhysSteerAngle(double)
+ setGear(GEAR_ENUM)

**VehicleStateMsg (example)**

+ raw_msg: byte struct
+ phys_msg: float/double struct

+ getPhysSteerAngle(): double
+ getWheelSpeed(WHEEL_ENUM): double
+ getGear(): GEAR_ENUM

```
struct phys_msg {
    double accel,
    double steer_angle
}
```

```
setPhysAxCommand(double ax_mps2) {
    phys_msg.accel = ax_mps2;
    raw_msg.accel = (ax_mps2 - AX_TRANSLATE) / AX_SCALE;
}
AX_SCALE and AX_TRANSLATE are constants that can be
extracted from the DBC file
```

```
setRawAxCommand(unit32_t ax) {
    raw_msg.accel = ax;
    phys_msg.accel = ax * AX_SCALE + AX_TRANSLATE;
}
```

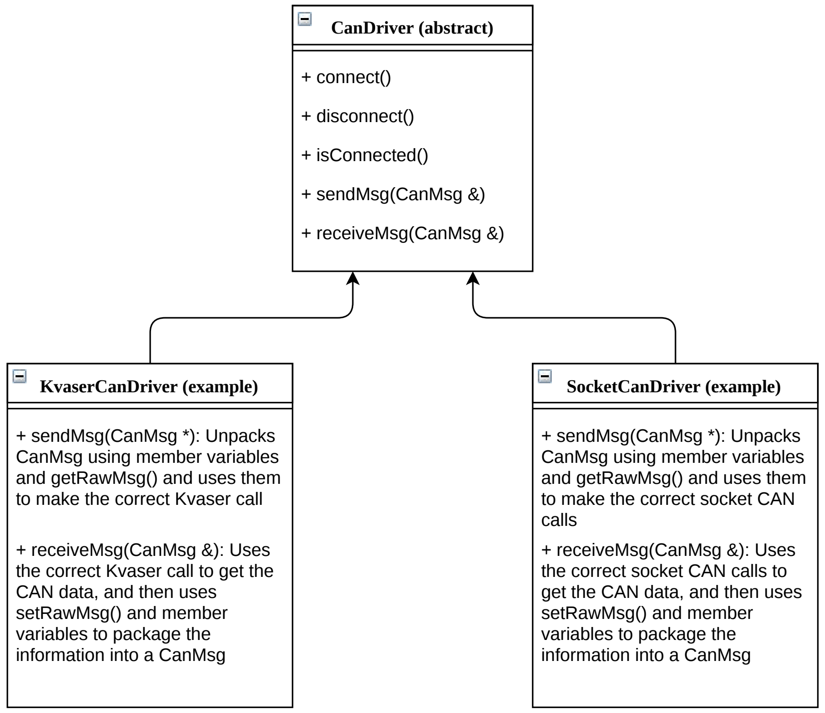
A struct that reflects the exact layout of the CAN message's data field:

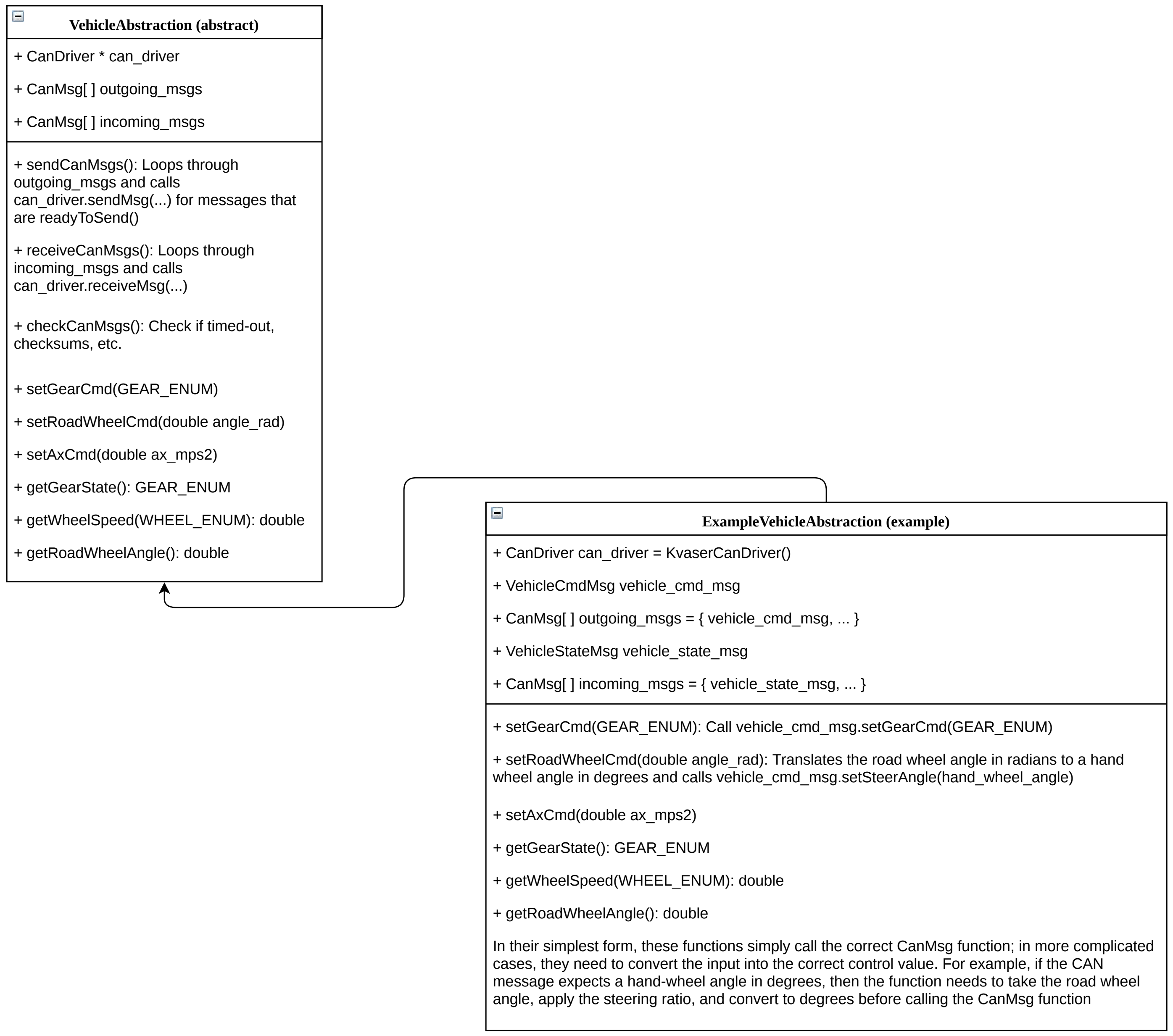
```
struct raw_msg {
    uint16_t accel: 9,
    uint16_t steer_angle: 7,
    uint8_t gear: 3,
    uint8_t padding: 5
}

(dlc = 6)
```

```
getWheelSpeed(WHEEL_ENUM wheel) {
     switch(wheel):
         case FRONT_RIGHT:
             return phys_msg.wheel_speed_fr;
         case FRONT_LEFT:
             return phys_msg.wheel_speed_fl;
         case REAR_RIGHT:
             return phys_msg.wheel_speed_rr;
         case REAR_LEFT:
             return phys_msg.wheel_speed_rl;
}
```

```
getPhysSteerAngle() {
    return phys_msg.steer_angle;
}
```


CanDriver (abstract)
+ connect()
+ disconnect()
+ isConnected()
+ sendMsg(CanMsg &)
+ receiveMsg(CanMsg &)
KvaserCanDriver (example)
+ sendMsg(CanMsg *): Unpacks CanMsg using member variables and getRawMsg() and uses them to make the correct Kvaser call
+ receiveMsg(CanMsg &): Uses the correct Kvaser call to get the CAN data, and then uses setRawMsg() and member variables to package the information into a CanMsg
SocketCanDriver (example)
+ sendMsg(CanMsg *): Unpacks CanMsg using member variables and getRawMsg() and uses them to make the correct socket CAN calls
+ receiveMsg(CanMsg &): Uses the correct socket CAN calls to get the CAN data, and then uses setRawMsg() and member variables to package the information into a CanMsg



**VehicleAbstraction (abstract)**

+ CanDriver * can_driver
+ CanMsg[ ] outgoing_msgs
+ CanMsg[ ] incoming_msgs

+ sendCanMsgs(): Loops through outgoing_msgs and calls can_driver.sendMsg(...) for messages that are readyToSend()
+ receiveCanMsgs(): Loops through incoming_msgs and calls can_driver.receiveMsg(...)
+ checkCanMsgs(): Check if timed-out, checksums, etc.
+ setGearCmd(GEAR_ENUM)
+ setRoadWheelCmd(double angle_rad)
+ setAxCmd(double ax_mps2)
+ getGearState(): GEAR_ENUM
+ getWheelSpeed(WHEEL_ENUM): double
+ getRoadWheelAngle(): double

**ExampleVehicleAbstraction (example)**

+ CanDriver can_driver = KvaserCanDriver()
+ VehicleCmdMsg vehicle_cmd_msg
+ CanMsg[ ] outgoing_msgs = { vehicle_cmd_msg, ... }
+ VehicleStateMsg vehicle_state_msg
+ CanMsg[ ] incoming_msgs = { vehicle_state_msg, ... }

+ setGearCmd(GEAR_ENUM): Call vehicle_cmd_msg.setGearCmd(GEAR_ENUM)
+ setRoadWheelCmd(double angle_rad): Translates the road wheel angle in radians to a hand wheel angle in degrees and calls vehicle_cmd_msg.setSteerAngle(hand_wheel_angle)
+ setAxCmd(double ax_mps2)
+ getGearState(): GEAR_ENUM
+ getWheelSpeed(WHEEL_ENUM): double
+ getRoadWheelAngle(): double

In their simplest form, these functions simply call the correct CanMsg function; in more complicated cases, they need to convert the input into the correct control value. For example, if the CAN message expects a hand-wheel angle in degrees, then the function needs to take the road wheel angle, apply the steering ratio, and convert to degrees before calling the CanMsg function