LOOP IN PRACTICE

An Example of Autonomous Vehicle Hazard Detection

CONTEXT

As the automotive industry accelerates towards autonomy, software is quickly becoming the vital driver of the 21st century vehicle. Features like lane change assistance, blind spot detection, parking guidance, collision avoidance, and other now-commonplace components are direct results of innovative thinking and software craftsmanship. The autonomous vehicle of tomorrow will be a product of even further innovation and craftsmanship—a congregation of all of these data sources which will allow for instantaneous, accurate decisioning.

Building the Autonomous Drive Controller of the vehicle, while vital, will not be the great hurdle in the race to autonomy. The group who is able to successfully integrate a vehicle with the myriad of sensors and hardware components will be the one who leads the world into the future of smart mobility.

Pillar Technology has significant experience in this testing space, and now we're introducing a revolutionary product to shift automotive organizations into the next gear. This solution, called LOOP, is a proprietary testing framework which allows embedded software to be tested before the physical components are available—or before they even exist. LOOP flips the typical development scheme on its head, allowing software to precede hardware, reducing time to market, catching defects early, and accelerating innovation.

In this article, we'll review a challenge faced by automotive manufacturers today, discuss how it's traditionally been solved, and then introduce a new paradigm to solve problems in the autonomous climate of tomorrow.

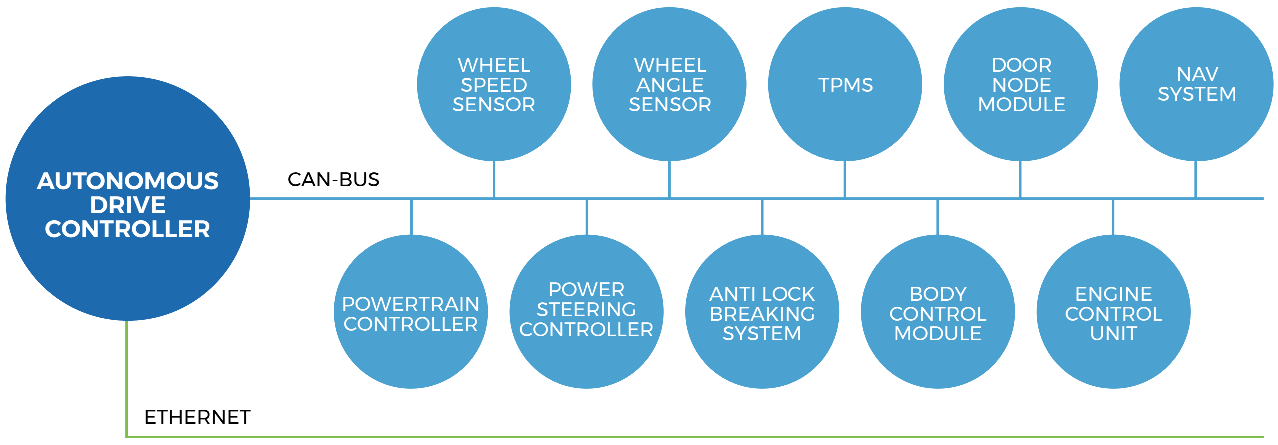
A REAL WORLD EXAMPLE

When talking about a concept as complex as the testing of self-driving automobiles, it's very easy to get lost in the intricacies. Let's take a step back and put this into real context.

For our purposes, imagine you are part of an engineering group that is developing an autonomous vehicle, and in this hypothetical scenario, the code base of the Autonomous Drive Controller subsystem already exists. You have been tasked with creating the Hazard Detector: the software component that detects hazardous conditions and suggests evasive actions. To do so, you've been tasked with the following initial use case:

When the vehicle is cruising on the freeway, and the Tire Pressure Monitoring System (TPMS) detects a loss of tire pressure, the vehicle must pull off to the closest side of the road and come to a stop within 30 seconds of the detection of the low tire pressure hazard.

Fairly straightforward. Now, the Autonomous Drive Controller subsystem drives the vehicle by communicating to automotive devices and sensors via the automotive CAN-Bus. It also communicates to other subsystems over an automotive Ethernet backbone, such as surveillance sensors (radar, lidar, video, etc.). The organization of the CAN-Bus is shown in the diagram below.



Note that, in our example above, only the pertinent CAN-Bus devices and sensors are shown.

Let's go back to our use case for further context. To satisfy the essential conditions of the case, the vehicle must be "cruising on the freeway." This means that all systems must satisfy the conditions of cruising. If, for example, the braking system is activated, then the vehicle is surely not cruising. The necessary essential conditions for the vehicle to be considered "cruising" are outlined in Chart A.

Similarly, in our case, a tire is considered inflated if it is equal to or above 30 psi. Anything below 30 psi is considered a tire pressure hazard, and warrants that the vehicle begin to pull over.

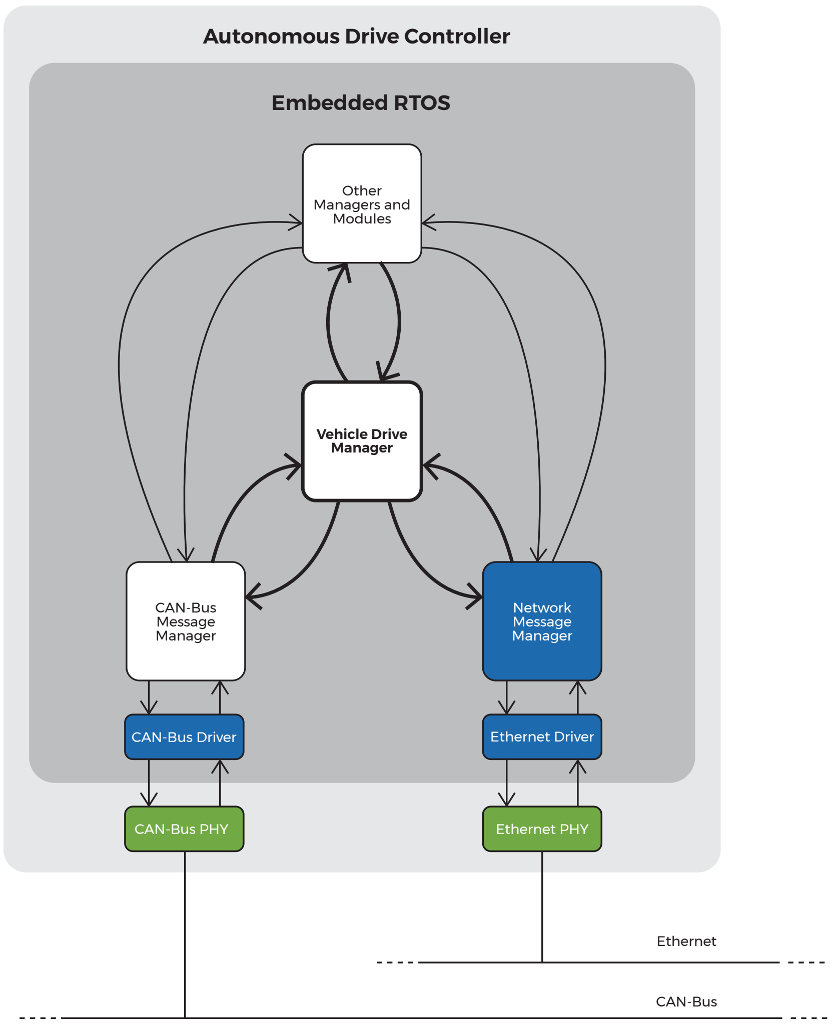
To fully satisfy the use case for a hazard detection, the vehicle must "pull off to the closest side of the road and come to a stop within 30 seconds of the detection of the low tire pressure hazard." The essential conditions for the vehicle to be considered "stopped" are also listed in Chart A.

Chart A	Cruising Vehicle	Stopped Vehicle
Wheel speed sensor =	65 mph	0 mph
ABS state =	Not stopped	Stopped
Nav system location =	Right lane	>5 ft. to right of lane
Nav system speed =	65 mph	0 mph
Powertrain controller speed =	3000 rpm	Between 600 and 1000 rpm

Architecture

Earlier in our example, we said that the Autonomous Drive Controller for our hypothetical vehicle had already been built. To understand the role that the Autonomous Drive Controller plays in this scenario, we first need to define what exactly it is.

The Autonomous Drive Controller is a high-performance Single Board Computer (SBC) that runs an embedded Real-Time Operating System (RTOS). While the RTOS vendor supplies a CAN-Bus Driver, Ethernet Driver, and a Network Message Manager, our hypothetical autonomous vehicle engineering group has crafted its own CAN-Bus Message Manager to coordinate and abstract the CAN-Bus message traffic, alleviating the need for the other software components to depend on the details of CAN message communication operations.

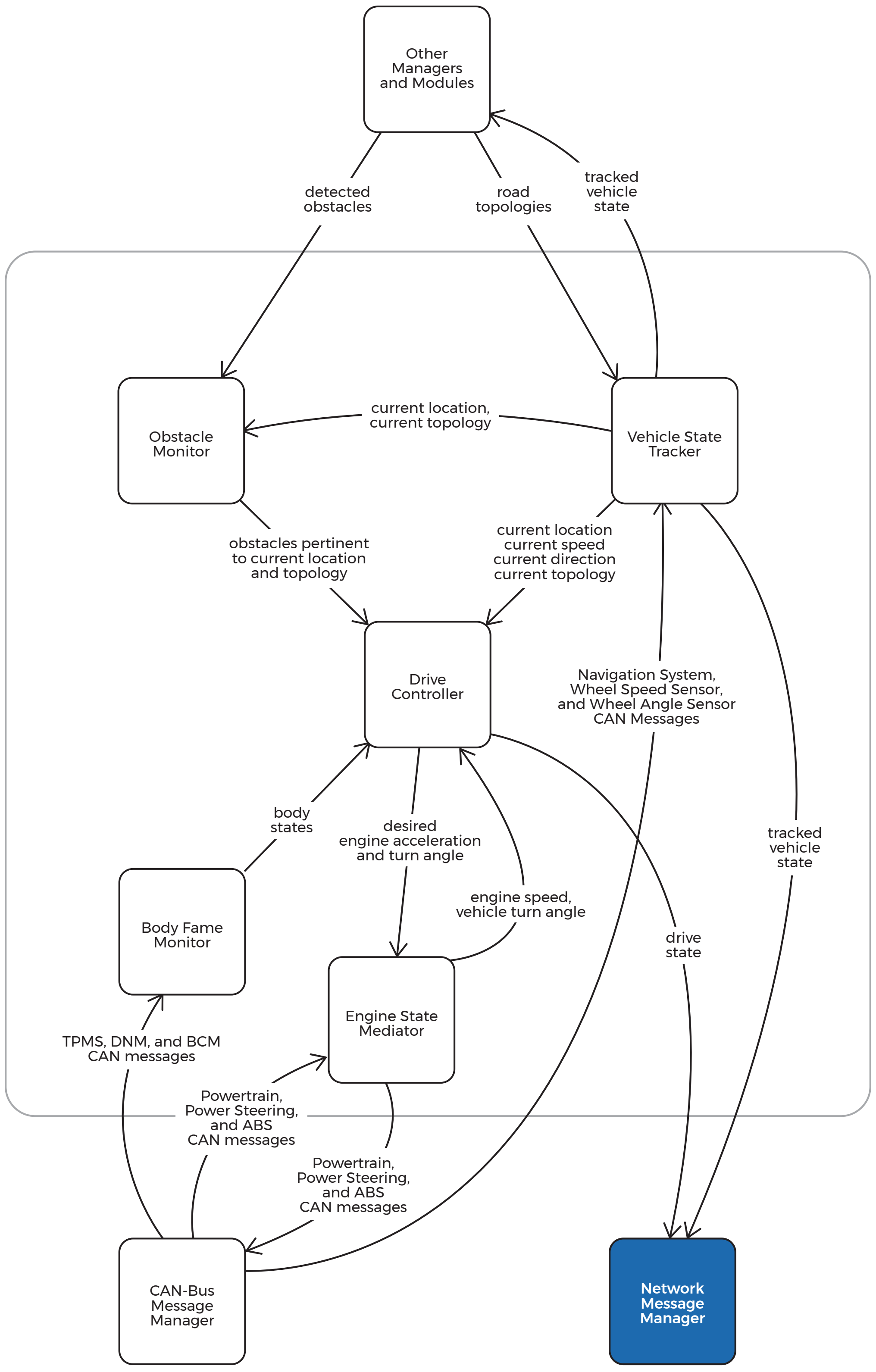


For our example, the software subsystem of interest is the Vehicle Drive Manager, because that is where the new Hazard Detector will reside.

The Vehicle Drive Manager is the software subsystem of the Autonomous Drive Controller that manages the vehicle driving operations. It monitors data collected from the other devices and sensors on the vehicle and produces acceleration and turn commands to the vehicle powertrain and power steering controllers on the CAN-Bus.

Within the Vehicle Drive Manager, the Drive Controller is essentially the brain of the subsystem. Its fundamental purpose is to drive the car by sending desired acceleration and turn angle commands to the Powertrain Controller, Power Steering Controller, and ABS devices residing on the automotive CAN-Bus.

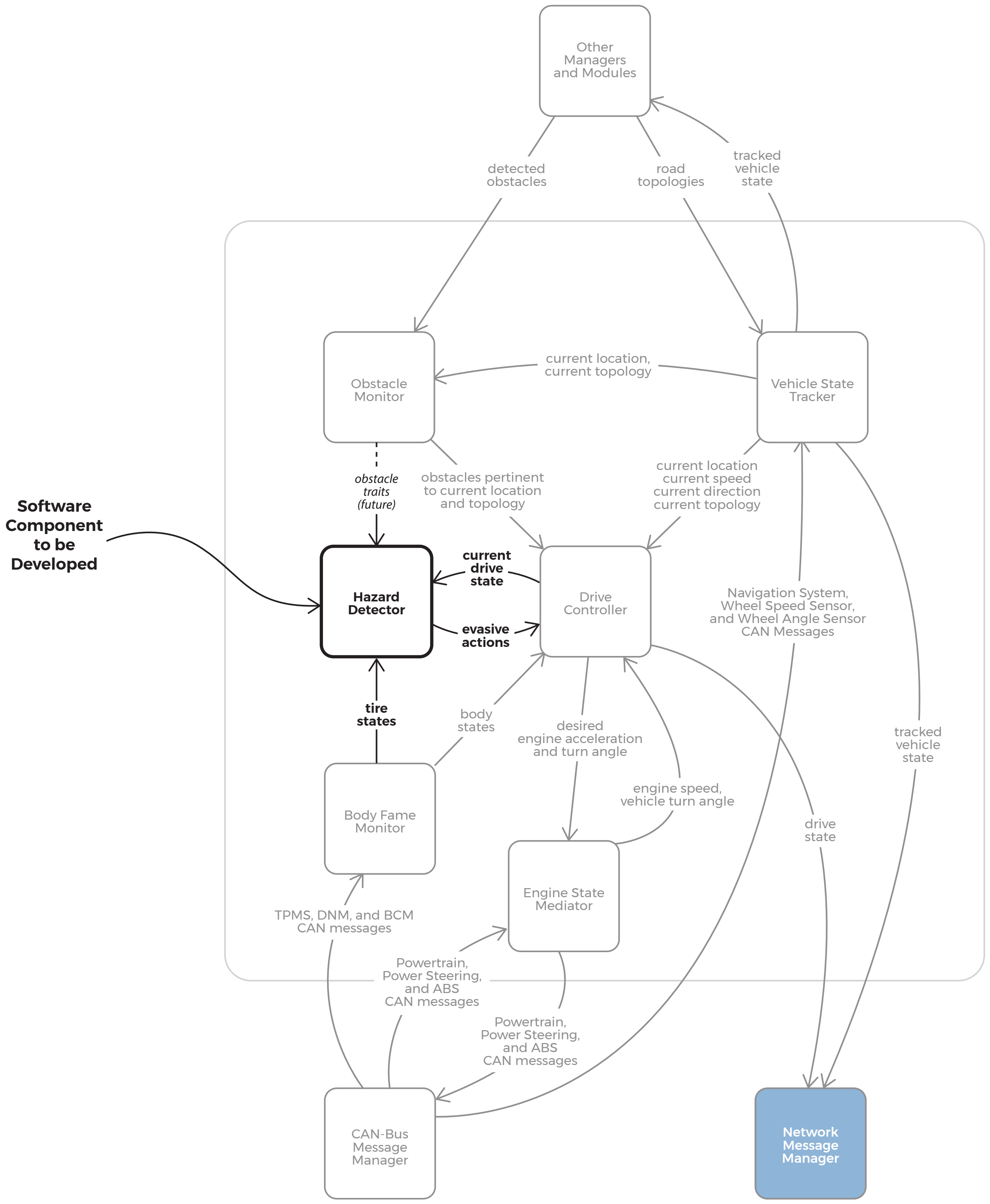
The Drive Controller makes decisions based on a number of informative sources. These sources include the Vehicle State Tracker, the Obstacle Monitor, the Body Frame Monitor, and the Engine State Mediator. Any of these sources can inform the Drive Controller on actions like acceleration and turn angle.


Other Managers and Modules
detected obstacles
road topologies
tracked vehicle state
Obstacle Monitor
current location, current topology
Vehicle State Tracker
obstacles pertinent to current location and topology
current location current speed current direction current topology
Drive Controller
Navigation System, Wheel Speed Sensor, and Wheel Angle Sensor CAN Messages
body states
desired engine acceleration and turn angle
engine speed, vehicle turn angle
tracked vehicle state
drive state
Body Fame Monitor
Engine State Mediator
TPMS, DNM, and BCM CAN messages
Powertrain, Power Steering, and ABS CAN messages
Powertrain, Power Steering, and ABS CAN messages
CAN-Bus Message Manager
Network Message Manager

Our task, if you recall, is to develop the Hazard Detector, which will notify the Drive Controller of evasive actions required to avoid hazardous conditions. Our newly developed solution will need to interact with the Drive Controller, the Obstacle Monitor, and the Body Frame Monitor.

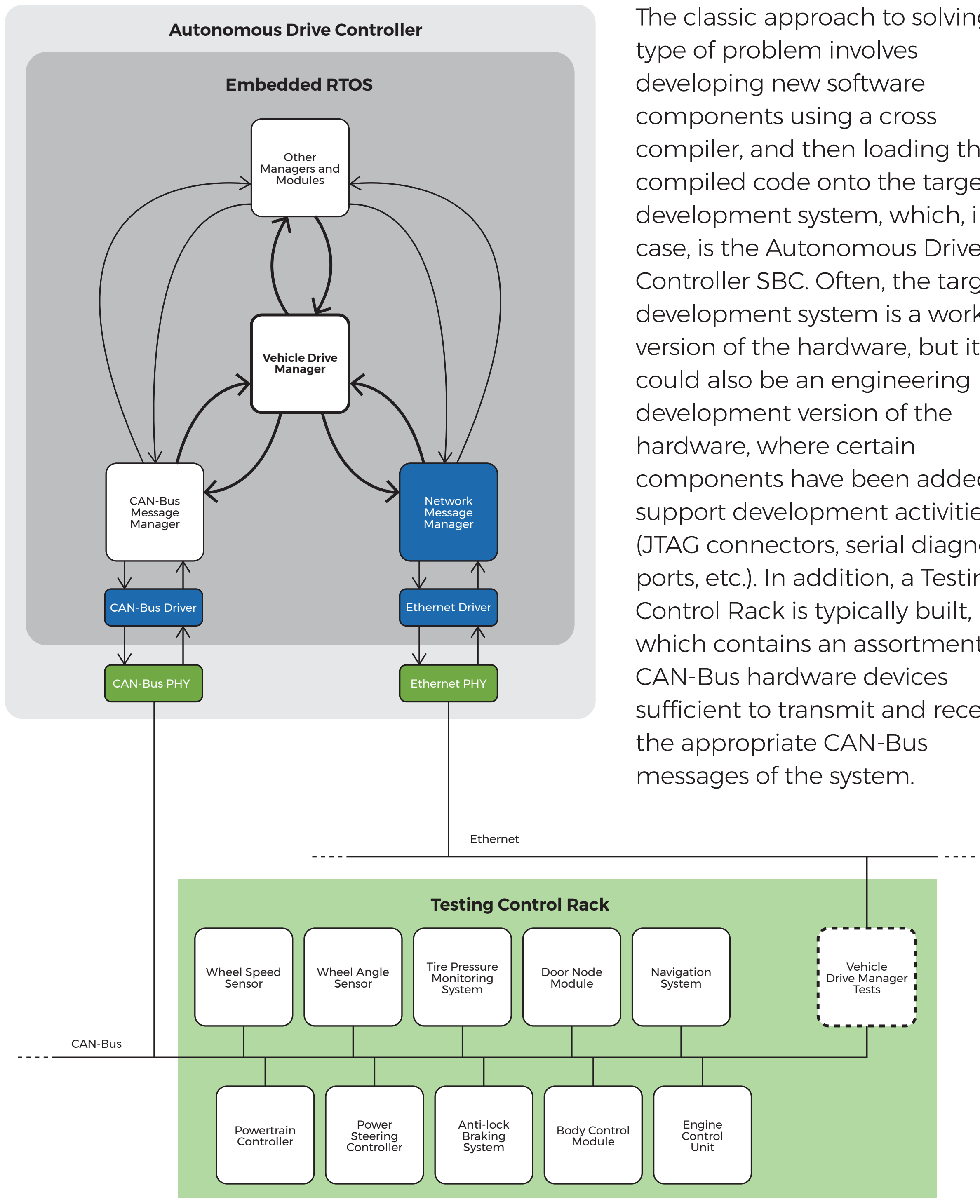
In our case, the Hazard Detector monitors the current drive state (e.g., cruising conditions, road topology, etc.) of the Drive Controller and the tire states obtained from the Body Frame Monitor. Upon detection of a flat tire condition, the Hazard Detector suggests evasive actions to the Drive Controller (the evasive actions are suggested because they may potentially conflict with higher-priority reactions and decisions of the Drive Controller). The Drive Controller will then alter its acceleration and turn angle commands accordingly so that the vehicle can safely pull off to the side of the road.

A depiction of these system interactions is outlined below.


Other Managers and Modules
detected obstacles
road topologies
tracked vehicle state
Obstacle Monitor
current location, current topology
Vehicle State Tracker
obstacle traits (future)
obstacles pertinent to current location and topology
current location current speed current direction current topology
Software Component to be Developed
Hazard Detector
current drive state
evasive actions
Drive Controller
Navigation System, Wheel Speed Sensor, and Wheel Angle Sensor CAN Messages
tire states
body states
desired engine acceleration and turn angle
engine speed, vehicle turn angle
tracked vehicle state
drive state
Body Fame Monitor
Engine State Mediator
TPMS, DNM, and BCM CAN messages
Powertrain, Power Steering, and ABS CAN messages
Powertrain, Power Steering, and ABS CAN messages
CAN-Bus Message Manager
Network Message Manager

The Classic Approach to Solving the Problem

The scenario we've described above is not a new one for automakers. So, how have they solved this problem in the past?



The classic approach to solving this type of problem involves developing new software components using a cross compiler, and then loading the compiled code onto the target development system, which, in this case, is the Autonomous Drive Controller SBC. Often, the target development system is a working version of the hardware, but it could also be an engineering development version of the hardware, where certain components have been added to support development activities (JTAG connectors, serial diagnostic ports, etc.). In addition, a Testing Control Rack is typically built, which contains an assortment of CAN-Bus hardware devices sufficient to transmit and receive the appropriate CAN-Bus messages of the system.

The developer, or often a testing engineer, then creates Vehicle Drive Manager tests which are usually controlled and run by the Testing Control Rack itself. This means that these tests are external to the Vehicle Drive Manager. Because of this, the tests are constructed after a baseline version of the software has already been developed. More importantly, these tests do not lend visibility into the software code being tested. In other words, they are “black box” tests and their focus is at a relatively high level of integration.

In our example, the classic approach would verify the conditions of a “hazardous” or “nonhazardous” scenario through the readings on actual hardware. If the TPMS reported tire pressure of 20 psi, the system would suggest that the car begin slowing down, pulling over safely, and stopping within 30 seconds of the initial detection.

These scenarios are tested, verified, and applied in the physical world. In this classic approach, sensors interact with relevant systems to incite action (or inaction). In this testing environment, the systems and the hardware are equally important, and one cannot be validated without the other. This means that development can only occur with the availability of the Drive Manager target and Testing Control Rack hardware.

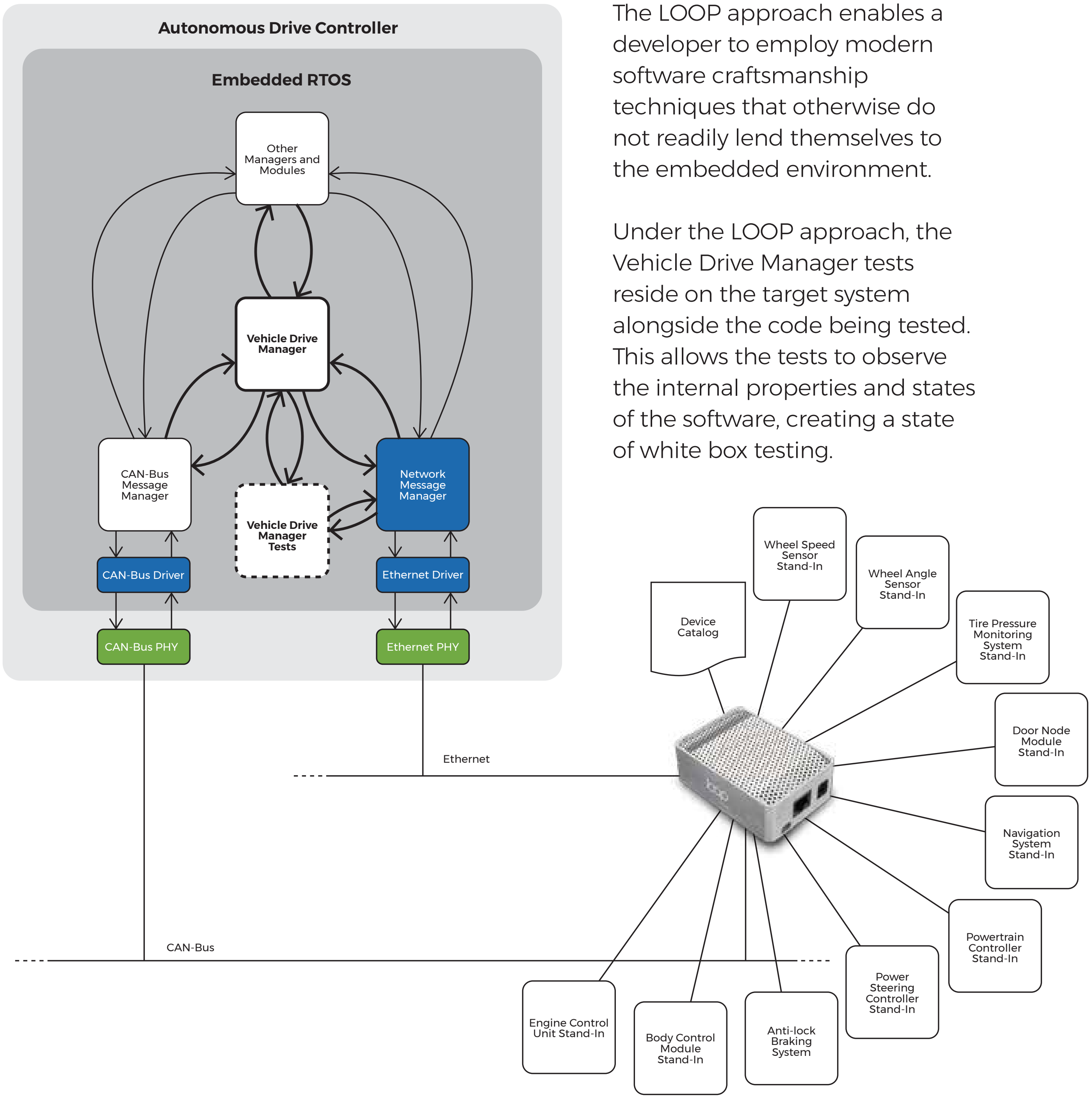
In this approach, the test runner is externally attempting to test a closed system. This requires testing after the component has been developed, leading to lengthy, costly debug times and significant code change barriers. Additionally, because these testing models are based on physics and validated through physical hardware, the testing time is quite long. And with limited visibility into software properties and statuses, a “black box” testing state is inevitable.

This testing model, the current status quo, is overly lengthy, costly, and inefficient.

The LOOP Approach

The LOOP approach enables a developer to employ modern software craftsmanship techniques that otherwise do not readily lend themselves to the embedded environment.

Under the LOOP approach, the Vehicle Drive Manager tests reside on the target system alongside the code being tested. This allows the tests to observe the internal properties and states of the software, creating a state of white box testing.


Autonomous Drive Controller
Embedded RTOS
Other Managers and Modules
Vehicle Drive Manager
CAN-Bus Message Manager
Vehicle Drive Manager Tests
Network Message Manager
CAN-Bus Driver
Ethernet Driver
CAN-Bus PHY
Ethernet PHY
Ethernet
CAN-Bus
Device Catalog
Wheel Speed Sensor Stand-In
Wheel Angle Sensor Stand-In
Tire Pressure Monitoring System Stand-In
Door Node Module Stand-In
Navigation System Stand-In
Powertrain Controller Stand-In
Power Steering Controller Stand-In
Anti-lock Braking System
Body Control Module Stand-In
Engine Control Unit Stand-In

In our usage example, under the LOOP approach, the Hazard Monitor software component would determine conditions based on the state of the Drive Controller software component and make recommendations accordingly.

Under this approach, the tests use LOOP to set up and tear down stand-in CAN-Bus devices as needed by each test. This provides a high degree of control over the specification of test scenarios because stand-in devices can be instantiated and configured very efficiently as compared to real devices.

The LOOP approach addresses several significant issues which continue to stall development organizations. First, because the LOOP box is highly available compared to Testing Control Rack hardware, it greatly increases the ability for team members to continually test and develop software. Additionally, because LOOP creates a state of white box testing, it greatly increases the visibility of internal properties and states.

LOOP's unique combination of features and benefits leads to decreased development time, increased efficiency, and faster time to market.

Example

As an example of how LOOP can be used at the "white-box" testing level, consider the following test scenario:

When

Wheel Speed Sensor = **65 mph**
Wheel Angle Sensor = **0 degrees**
Navigation System location = **right lane of freeway**
Navigation System direction = **0 degrees along track of freeway lane**
Navigation System speed = **65 mph**
Powertrain Controller speed = **3000 rpm**
Power Steering Controller state = **straight**
ABS state = **not stopped**

Then

the Hazard Monitor determines a "cruising on the freeway" condition from from the "current drive state" data obtained from the Drive Controller

Because the Hazard Monitor and Drive Controller software components are accessible to the Vehicle Drive Manager Tests (because they coexist with the UUT software on the SBC), and because the testing code has the ability to talk to LOOP, this test can be easily constructed and executed.

The LOOP stand-in devices are defined via a hierarchical Device Catalog that exists on the LOOP unit. Each device is specified via a set of properties (i.e., values) and CAN-Bus messages (i.e., actions).

Without getting into the specifics of the internals of the Device Catalog, each of the stand-in device types needed for our test can be described using the following pseudo-definitions:

WheelSpeedSensor
Properties:
- LeftFront: MPH
- RightFront: MPH
- LeftRear: MPH
- RightRear: MPH

Messages:
- LF-RPM: { 0x0101, LeftFront }
- RF-RPM: { 0x0102, RightFront }
- LR-RPM: { 0x0103, LeftRear }
- RR-RPM: { 0x0104, RightRear }

TPMS
Properties:
- LeftFront: PSI
- RightFront: PSI
- LeftRear: PSI
- RightRear: PSI
- Inflated: Boolean

Messages:
- LF-PSI: { 0x0301, LeftFront }
- RF-PSI: { 0x0302, RightFront }
- LR-PSI: { 0x0303, LeftRear }
- RR-PSI: { 0x0304, RightRear }
- Status: { 0x0305: Inflated }

WheelAngleSensor
Properties:
- Angle: Degrees CW

Messages:
- Angle: { 0x0201, Angle }

ABS
Properties:
- Engaged: Boolean

Messages:
- Status: { 0x0501, Engaged }

DNM
Properties:
- LeftFrontClosed: Boolean
- RightFrontClosed: Boolean
- LeftRearClosed: Boolean
- RightRearClosed: Boolean

Messages:
- LF-State: { 0x0401, LeftFrontClosed }
- RF-State: { 0x0402, RightFrontClosed }
- LR-State: { 0x0403, LeftRearClosed }
- RR-State: { 0x0404, RightRearClosed }

BCM
Properties:
- LeftFrontWindow: Level
- RightFrontWindow: Level
- LeftRearWindow: Level
- RightRearWindow: Level

Messages:
- LF-State: { 0x0601, LeftFrontWindow }
- RF-State: { 0x0602, RightFrontWindow }
- LR-State: { 0x0603, LeftRearWindow }
- RR-State: { 0x0604, RightRearWindow }

Nav
Properties:
- Latitude: Degrees N
- Longitude: Degrees E
- Height: Feet MSL
- Heading: Degrees CWN
- HSpeed: MPH
- VSpeed: FPM
- Region: RoadTopology

Messages:
- Lat: { 0x0701, Latitude }
- Lon: { 0x0702, Longitude }
- Hgt: { 0x0703, Height }
- Hdg: { 0x0704, Heading }
- HS: { 0x0704, HSpeed }
- VS: { 0x0704, VSpeed }
- Rgn: { 0x705: Region }

ECU
Properties:
- Temperature: Degrees C

Messages:
- Temp: { 0x0801, Temperature }

PowertrainController
Properties:
- EngineRPM: RPM

Messages:
- RPM: { 0x0901, EngineRPM }

PowerSteeringController
Properties:
- RequestedTurnAngle: Degrees CW

Messages:
- Angle: { 0x0A01, RequestedTurnAngle }

The device entries describe the value properties of each device, and the messages define the CAN-Bus ID tags and property associations for the message payloads.

The LOOP unit is controlled via a REST interface via its Ethernet port. The basic command breakdown of the REST API is as follows:

```
http://<LOOP-server>/api/v2/CreateDevice/{device-name} [GET]
http://<LOOP-server>/api/v2/SetDeviceProperty/{device-code}/{property-name}/{property-value} [POST]
http://<LOOP-server>/api/v2/GetDeviceProperty/{device-code}/{property-name} [GET]
http://<LOOP-server>/api/v2/SendDeviceMessage/{device-code}/{message-name} [POST]
http://<LOOP-server>/api/v2/DeleteDevice/{device-code} [DELETE]
```

The “CreateDevice” endpoint instantiates a new device from the catalog and returns a device ID code in the HTTP response body. The “SetDeviceProperty” endpoint sets the property value of an instantiated device to a new value. The “GetDeviceProperty” endpoint retrieves the property value of an instantiated device, the value being returned in the body of the HTTP response. The “SendDeviceMessage” endpoint triggers a sending of a CAN-Bus message for the specified device, where the message payload is populated from the associated property value as defined in the Device Catalog. There is no need for a “ReceiveDeviceMessage” endpoint because CAN-Bus messages received by LOOP that match existing message ID and payload signatures will cause their associated device property values to get updated from the received message payloads. The “DeleteDevice” endpoint deallocates an existing device.

For our example, we will assume the existence of a set of “serverGet()”, “serverPost()”, and “serverDelete()” pseudo-code functions that allow the test code to abstract the LOOP server address. Each function has the ability to return the HTTP response, with the body containing the LOOP return values (e.g., device codes, property values, etc.).

Using our pseudo-code functions, at the beginning of the test scenario, the devices are created as follows:

```
response = serverGet(“/api/v2/CreateDevice/WheelSpeedSensor”)
wheelSpeedSensorID = int(response.body)
response = serverGet(“/api/v2/CreateDevice/WheelAngleSensor”)
wheelAngleSensorID = int(response.body)
response = serverGet(“/api/v2/CreateDevice/TPMS”)
tpmsID = int(response.body)
response = serverGet(“/api/v2/CreateDevice/DNM”)
dnmID = int(response.body)
response = serverGet(“/api/v2/CreateDevice/ABS”)
absID = int(response.body)
```

```
response = serverGet("/api/v2/CreateDevice/BCM")
bcmID = int(response.body)
response = serverGet("/api/v2/CreateDevice/Nav")
navID = int(response.body)
response = serverGet("/api/v2/CreateDevice/ECM")
ecmID = int(response.body)
response = serverGet("/api/v2/CreateDevice/PowertrainController")
powertrainID = int(response.body)
response = serverGet("/api/v2/CreateDevice/PowerSteeringController")
powerSteeringID = int(response.body)
```

The test scenario is then executed:

```
//
// Set LOOP device property values
//

// typical wheel rotation for a 205/60R16 tire = 785.32 rev/mile
// 785.32 rev/mile * 65 mile/hr * 1 hr / 60 min = 850.76 rev/min
// => scale = 785.32 / 60 = 13.089
serverPost("/api/v2/SetDeviceProperty/{wheelSpeedSensorID}/LeftFront/65/13.089/0")
serverPost("/api/v2/SetDeviceProperty/{wheelSpeedSensorID}RightFront/65/13.089/0")
serverPost("/api/v2/SetDeviceProperty/{wheelSpeedSensorID}/LeftRear/65/13.089/0")
serverPost("/api/v2/SetDeviceProperty/{wheelSpeedSensorID}/RightRear/65/13.089/0")

serverPost("/api/v2/SetDeviceProperty/{wheelAngleSensorID}/Angle/0")

// heading east on I-270 passing Port Columbus radar
serverPost("/api/v2/SetDeviceProperty/{navID}/Latitude/40.008092")
serverPost("/api/v2/SetDeviceProperty/{navID}/Longitude/-82.888978")
serverPost("/api/v2/SetDeviceProperty/{navID}/Height/902")
serverPost("/api/v2/SetDeviceProperty/{navID}/Heading/90")
serverPost("/api/v2/SetDeviceProperty/{navID}/HSpeed/65")
serverPost("/api/v2/SetDeviceProperty/{navID}/VSpeed/0")
serverPost("/api/v2/SetDeviceProperty/{navID}/Region/12345")

serverPost("/api/v2/SetDeviceProperty/{powertrainID}/EngineRPM/3000")

serverPost("/api/v2/SetDeviceProperty/{absID}/Engaged/false")

//
// Ensure we start from a known DriveController state
//
```

```
DriveController.Reset()

// ensure that DriveController does not think it’s cruising on the freeway
Assert(DriverController.CruiseState != “At Speed On Freeway”)

//
// Tell DriveController to begin running
//

DriveController.Run()

//
// Tell LOOP devices to send their messages
//

serverPost(“/api/v2/SendDeviceMessage/{wheelSpeedSensorID}/LF-RPM”)
serverPost(“/api/v2/SendDeviceMessage/{wheelSpeedSensorID}/RF-RPM”)
serverPost(“/api/v2/SendDeviceMessage/{wheelSpeedSensorID}/LR-RPM”)
serverPost(“/api/v2/SendDeviceMessage/{wheelSpeedSensorID}/RR-RPM”)

serverPost(“/api/v2/SendDeviceMessage/{wheelAngleSensorID}/Angle”)

serverPost(“/api/v2/SendDeviceMessage/{navID}/Lat”)
serverPost(“/api/v2/SendDeviceMessage/{navID}/Lon”)
serverPost(“/api/v2/SendDeviceMessage/{navID}/Hgt”)
serverPost(“/api/v2/SendDeviceMessage/{navID}/Hdg”)
serverPost(“/api/v2/SendDeviceMessage/{navID}/HS”)
serverPost(“/api/v2/SendDeviceMessage/{navID}/VS”)
serverPost(“/api/v2/SendDeviceMessage/{navID}/Rgn”)

serverPost(“/api/v2/SendDeviceMessage/{powertrainID}/RPM”)

serverPost(“/api/v2/SendDeviceMessage/{absID}/Status”)

//
// Wait a few seconds for all received messages to be delivered through the RTOS
//

wait(5)

//
// Verify that DriveController thinks it’s cruising on the freeway
//

Assert(DriverController.CruiseState == “At Speed On Freeway”)

//
// Stop the DriveController
//

DriveController.Stop()
```

The devices can then be deallocated using:

```
serverDelete("/api/v2/DeleteDevice/WheelSpeedSensor/{wheelSpeedSensorID}")
serverDelete("/api/v2/DeleteDevice/WheelSpeedSensor/{wheelAngleSensorID}")
serverDelete("/api/v2/DeleteDevice/WheelSpeedSensor/{tpmsID}")
serverDelete("/api/v2/DeleteDevice/WheelSpeedSensor/{dnmID}")
serverDelete("/api/v2/DeleteDevice/WheelSpeedSensor/{absID}")
serverDelete("/api/v2/DeleteDevice/WheelSpeedSensor/{bcmID}")
serverDelete("/api/v2/DeleteDevice/WheelSpeedSensor/{navID}")
serverDelete("/api/v2/DeleteDevice/WheelSpeedSensor/{ecmID}")
serverDelete("/api/v2/DeleteDevice/WheelSpeedSensor/{powertrainID}")
serverDelete("/api/v2/DeleteDevice/WheelSpeedSensor/{powerSteeringID}")
```

A fundamental observation to be made from this example is that a white-box test using LOOP is able to completely control both the UUT software component state and the external CAN-Bus message state with absolute predictability. In addition, this level of control is simple and efficient, and, tests of this type are easily automated and can be controlled by any one of the many available continuous integration solutions (e.g., Jenkins). Lastly, because the LOOP interface is language agnostic (i.e., it relies a REST interface), any language chosen by the developer is sufficient to use LOOP; it does not force the developer to use a specific testing language.

Conclusion

It's plainly evident that software will play a vital role in shaping the automotive climate of the 21st century and beyond. And although the industry is speeding towards ideas like autonomy and shared mobility—both of which will require a keen ability to quickly develop and deliver software—we don't know exactly how people will choose to get around five, ten, or a hundred years from now.

In this uncertainty lies the real value in LOOP. Regardless of macro, micro, or industry trends, LOOP enables your organization to deliver reliable, valuable software at lightning speed. By removing barriers and increasing visibility, LOOP enables groups to decrease time to market, foster collaborative practices, and further cement their organization's place in the future of 21st century mobility.