iOS Framework API Guide

The iOS Framework provides a simple mechanism to connect to an OpenXC VI via an iOS app. The iOS Framework is written in Swift2.

The Framework must be included in any iOS app that needs to connect to a VI by copying the openXCiOSFramework.Framework file into the app project. In the app project settings this framework must be added to the "Embedded Binaries" section, and import openXCiOSFramework must be added to the top of any file that makes use of the framework. Additionally, the CoreBluetooth framework must be added to any project that uses this framework. The framework also makes use of a third party framework for ProtocolBuffer support. The ProtocolBuffer framework must be included in the framework, and any app using the framework.

The VehicleManager

The VehicleManager (VM) handles all communication between the iOS app and the OpenXC VI. It can be configured to handle many different usage situations.

Firstly, the VM must be instantiated. Typically this is done in the viewDidLoad method of the app's principal View Controller.

```
override func viewDidLoad() {
super.viewDidLoad()
        . . .
        var vm: VehicleManager!
        vm = VehicleManager.sharedInstance
        . . .
}
```

The VM is only instantiated once across the entire app. The sharedInstance class variable gives any Controller access to the VM. The first Controller to access the sharedInstance will result in the sharedInstance being created.

Configuring the VehicleManager

The VM can be configured at any time, before or after a connection to the VI is made, with the exception of choice of data format for communicating with the VI.

The VM will optionally send status updates to a ViewController that registers a callback function. Only one ViewController can be registered to receive the status updates. The status updates arrive as an NSDictionary object containing at least a *"status"* key-value pair. Certain values will indicate that other keys are present.

```
override func viewDidLoad() {
        . . .
        vm.setManagerCallbackTarget(self,
        action: ViewController.manager_status_updates)
        . . .
}

func manager_status_updates(rsp:NSDictionary) {
let status = rsp.objectForKey("status") as! Int
let msg = VehicleManagerStatusMessage(rawValue: status)
print("VM status : ",msg!)
if msg==VehicleManagerStatusMessage.C5CONNECTED {
                // .. do something
        }
}
```

Once the status message has been decoded (as in *"msg"* above), the status type can be checked as per the following enums

```
.C5DETECTED                 - C5 VI detected
.C5CONNECTED                - C5 VI connected
.C5DISCONNECT               - C5 VI disconnected
.C5SERVICEFOUND             - openXC service discovered
.C5NOTIFYON                 - openXC notify enabled
.TRACE_SOURCE_END           - input trace file reached EOF
.TRACE_SINK_WRITE_ERROR   - error when writing to trace file
.BLE_RX_DATA_PARSE_ERROR  - error when decoding rx data
```

The VM can be configured to either autoconnect to the first discovered VI, or the VI can be selected from a discovered list. Autoconnect defaults to true.

```
override func viewDidLoad() {
        . . .
vm.setAutoconnect(true)
        . . .
}
```

The VM can optionally output debug logging

```
override func viewDidLoad() {
        . . .
vm.setManagerDebug(true)
        . . .
}
```

The VM must be told what data format the BLE packets will arrive in. By default the VM is configured to JSON mode, but can be changed to ProtoBuf mode if necessary.

```
override func viewDidLoad() {
        . . .
 vm.setProtobufMode(true)
        . . .
}
```

When ready, the app must first tell the VM to perform a scan for nearby VIs.

```
override func viewDidLoad() {
        . . .
vm.scan()
        . . .
}
```

If autoconnect is enabled, the connection initiates when the first VI is discovered.

Otherwise, whenever a VI is discovered, the VM will return the C5DETECTED status update to the client. The app has access to a list of VI that were discovered.

```
func someFunction() {
        . . .
        let list = vm.discoveredVI()
    // list is an array of VI names
        . . .
}
```

The VM can connect to a specific VI in the list of discovered VI.

```
func someFunction() {
        . . .
let name = vm.discoveredVI()[selected]
        vm.connect(name)
        . . .
}
```

There are a few status variables exposed from the VM that a user of the framework can use to find out details about the connection to the VI.

messageCount shows the total number of messages that have been received since the VM has been in Operational state.

```
func someRecurringFunction() {
        . . .
if vm.connectionState==
          VehicleManagerConnectionState.Operational {
        print("VM is receiving data from VI!")
        print("So far we've had ",vm.messageCount,
                    " messages")
  }
        . . .
}
```

connectionState shows the connection state of the VM compared to the following enums

```
.NotConnected                    - not connected
.Scanning                        - scan is underway
.ConnectionInProgress            - connection is underway
.Connected                       - VM is connected to VI
.Operational                - VM is receiving messages
. . .
```

Measurement Messages

Once connected, the VM will begin to receive all measurement messages sent from the VI. These messages can be read on demand by requesting a specific measurement from the VM. The last recorded message of that type is returned.

```
func doSomething() {
        . . .
        let rsp = vm.getLatest("fuel_level")
 print("FUEL_LEVEL value:",rsp.value)
 print("FUEL_LEVEL timestamp:",rsp.timestamp)
        . . .
}
```

The VM can also be configured to trigger a callback if a specific type of measurement message arrives.

```
override func viewDidLoad() {
        . . .
vm.addMeasurementTarget("ignition_status",
        target: self,
                action:ViewController.display_ignition_status_change)
        . . .
}

func display_ignition_status_change(rsp:NSDictionary) {
        let vr = rsp.objectForKey("vehiclemessage") as!
                VehicleMeasurementResponse
print("specified meas:",vr.name," -- ",vr.value!)
}
```

The callback returns an NSDictionary object containing a *"vehiclemessage"* key-value pair. The value in the case of this measurement message callback is a VehicleMeasurementResponse object.

The VM can additionally be configured to trigger a default callback for any types of measurement messages that have not already been configured with a callback as above.

```
    override func viewDidLoad() {
            . . .
    vm.setMeasurementDefaultTarget(self,
                        action: ViewController.default_measurement_change)
            . . .
    }

func default_measurement_change(rsp:NSDictionary) {
    let vr = rsp.objectForKey("vehiclemessage") as!
                        VehicleMeasurementResponse
    if vr.isEvented {
            print("default meas(evented):",vr.name," -- "
                                ,vr.event," -- ",vr.value)
    } else {
            print("default meas:",vr.name," -- ",vr.value)
            }
}
```

Callbacks can be removed for specific and default measurements.

```
    func doSomething() {
            . . .
     vm.clearMeasurementTarget("fuel_level")
     vm.clearMeasurementDefaultTarget()
            . . .
    }
```

Callbacks are also able to be overwritten simply by calling the set functions again.

Command Messages

The VM supports all currently available openXC commands that can be sent to the VI. The commands can be sent with a callback where the command response will be returned.

```
    override func viewDidLoad() {
            . . .
     let cm = VehicleCommandRequest()
    cm.command = .version
    let cmdcode = vm.sendCommand(cm, target: self, action:
                        ViewController.display_version_rsp)
            print("version cmd sent:",cmdcode)
            . . .
    }

  func display_version_rsp(rsp:NSDictionary) {
    let vr = rsp.objectForKey("vehiclemessage") as!
                        VehicleCommandResponse
    let code = rsp.objectForKey("key") as! String
    print("cmd_rsp \(code) : \(vr.command_response)")
  }
```

The callback returns an NSDictionary object containing a *"vehiclemessage"* key-value pair. The value in the case of this command message callback is a VehicleCommandResponse object. It also returns a *"key"* key-value pair where the value matches the return code returned when the command is sent. It can be used to match a response to a sent command if necessary.

A command message can also be sent where no specific callback is requested.

```
    func doSomething() {
            . . .
     let cm = VehicleCommandRequest()
    cm.command = .device_id
            vm.sendCommand(cm)
            . . .
    }
```

A default command response callback is also available, which in many cases simplifies the client app.

```
override func viewDidLoad() {
        . . .
vm.setCommandDefaultTarget(self, action:
                StatusViewController.handle_cmd_response)
        . . .
}

func handle_cmd_response(rsp:NSDictionary) {
 // extract the command response message
 let cr = rsp.objectForKey("vehiclemessage") as! VehicleCommandResponse
 print("cmd response is \(cr.command_response)")
 }
```

If this default callback is declared, then all command responses will be sent to the callback function, even if a command is sent specifying its own callback function. It is effectively a global override for any command response.

The default callback can be removed.

```
func doSomething() {
        . . .
 vm.clearCommandDefaultTarget()
        . . .
}
```

Diagnostic Messages

The VM can initiate diagnostic messages to the VI in a variety of ways.

Since a diagnostic command is effectively a command message followed by one or more diagnostic response messages, there are several combinations of how to interact with them.

The simplest way to have the VM interact with diagnostic messages is to setup a default handler for diagnostic responses.

```
  override func viewDidLoad() {
        . . .
  vm.setDiagnosticDefaultTarget(self,
                action: ViewController.default_diag_change)
        . . .
  }


func default_diag_change(rsp:NSDictionary) {
  let vr = rsp.objectForKey("vehiclemessage") as!
                VehicleDiagnosticResponse
  print("default diag msg: bus=",vr.bus,
                " success=",vr.success," value=",vr.value)
}
```

The callback returns an NSDictionary object containing a *"vehiclemessage"* key-value pair. The value in the case of this measurement message callback is a VehicleDiagnosticResponse object.

Diagnostic requests can then be sent in a similar way to generic command requests, and any diagnostic response message will be forwarded to the default diagnostic handler defined above

```
func doSomething() {
        . . .
let dr = VehicleDiagnosticRequest()
dr.bus = 1
dr.message_id = 0x7e0
dr.mode = 1
dr.pid = 12
 let cmdcode = vm.sendDiagReq (dr,
                target: self,
                cmdaction: ViewController.handle_diag_cmd_rsp)
        print("diag cmd sent:",cmdcode)
        . . .
}

func handle_diag_cmd_rsp(rsp:NSDictionary) {
        let cr = rsp.objectForKey("vehiclemessage") as!
                VehicleCommandResponse
let code = rsp.objectForKey("key") as! String
print("cmd response : \(code) : \(cr.command_response)")
}
```

It is also possible to ignore the diagnostic command response when sending a diagnostic request

```
func doSomething() {
        . . .
let dr = VehicleDiagnosticRequest()
dr.bus = 1
dr.message_id = 0x7e0
dr.mode = 1
dr.pid = 12
        vm.sendDiagReq (dr)
        . . .
}
```

If different handling of certain diagnostic responses is required, then a handler can be registered for a specific diagnostic response, based on a key created from the bus, id,mode,pid (or lack of pid).

```
  override func viewDidLoad() {
        . . .
  vm.addDiagnosticTarget([1,0x7e9,7], target: self,
                action: ViewController.diag_handler_a)
  vm.addDiagnosticTarget([1,0x7e8,1,12], target: self,
                action: ViewController.diag_handler_b)
        . . .
  }


func diag_handler_a(rsp:NSDictionary) {
  let vr = rsp.objectForKey("vehiclemessage") as!
                VehicleDiagnosticResponse
  print("only type a response: bus=",vr.bus,
                " success=",vr.success," value=",vr.value)
}
func diag_handler_b(rsp:NSDictionary) {
  let vr = rsp.objectForKey("vehiclemessage") as!
                VehicleDiagnosticResponse
  print("only type a response: bus=",vr.bus,
                " success=",vr.success," value=",vr.value)
}
```

If a bus,id,mode,pid key is registered for a specific callback, then all diagnostic responses matching that key will be sent to that callback and not to the default callback if it has been defined.

Diagnostic callbacks can be removed as well if necessary

```
  func doSomething() {
        . . .
   vm.clearDiagnosticTarget([1,0x7e9,7])
   vm.clearDiagnosticTarget([1,0x7e8,1,12])
   vm.clearDiagnosticDefaultTarget()
        . . .
  }
```

CAN Messages

The VM can send a CAN message with a specified bus, message_id and data.

```
func doSomething() {
        . . .
let cr = VehicleCanRequest()
cr.bus = 1
cr.message_id = 0x7e0
cr.data = “0102030405060708”
 vm.sendCanReq (cr)
        print("can rqst sent:")
        . . .
}
```

The VM can be configured to trigger on any CAN messages received from the VI. A default CAN message handler can be added to capture all CAN messages.

```
override func viewDidLoad() {
        . . .
vm.setCanDefaultTarget(self,
                action: ViewController.default_can_change)
        . . .
}


func default_can_change(rsp:NSDictionary) {
  let vr = rsp.objectForKey("vehiclemessage") as!
                VehicleCanResponse
  print("default can msg: bus=",vr.bus,
                " data=",vr.data)
}
```

The callback returns an NSDictionary object containing a *“vehiclemessage”* key-value pair. The value in the case of this measurement message callback is a VehicleCanResponse object.

The VM can also trigger on a specific bus,id key and direct these CAN messages to a different handler.

```
  override func viewDidLoad() {
        . . .
  vm.addCanTarget([1,0x7e9], target: self,
                  action: ViewController.can_handler_a)
  vm.addCanTarget([1,0x7e8], target: self,
                  action: ViewController.can_handler_b)
        . . .
  }


func can_handler_a(rsp:NSDictionary) {
  let vr = rsp.objectForKey("vehiclemessage") as!
                  VehicleCanResponse
  print("only type a CAN: data=",vr.data)
}
func can_handler_b(rsp:NSDictionary) {
  let vr = rsp.objectForKey("vehiclemessage") as!
                  VehicleCanResponse
  print("only type b CAN: data=",vr.data)
}
```

If a bus,id key is registered for a specific callback, then all CAN responses matching that key will be sent to that callback and not to the default callback if it has been defined.

Similar to other handlers, CAN handlers can be removed.

```
  func doSomething() {
        . . .
   vm.clearCanTarget([1,0x7e9])
   vm.clearCanTarget([1,0x7e8])
   vm.clearCanDefaultTarget()
        . . .
  }
```

Trace File Processing

The VM can be configured to capture all incoming messages into a trace output file, or to read data from a trace log file instead of via the VI. In both cases, the trace file is accessible via iTunes File Sharing. Apps using trace file capabilities must include the UIFileSharingEnabled=true key in the Info.plist file.

Configuring a trace output logfile should generally be set up before calling the VM connect function. If the file already exists in the app's documents folder, it will be overwritten by this command. The output file will contain all of the messages received by the VM in JSON format, each line terminated with a linefeed.

```
override func viewDidLoad() {
    . . .
vm.enableTraceFileSink("tracefile.txt")
    . . .
}
```

The VM can be told to stop capturing a trace output file.

```
func doSomething() {
    . . .
vm.disableTraceFileSink()
    . . .
}
```

When replaying from a trace logfile, the VM will process the file line by line at a configurable rate in ms until reaching the end of the file. At this time, the VM will stop automatically.

```
override func viewDidLoad() {
    . . .
vm.enableTraceFileSource("tracefile.txt",speed:50)
    . . .
}
```

If speed is omitted from the function call, the timestamps in the input trace file will be used to process the input messages at the same rate as when they were originally captured.

A trace input replay can be stopped before the end of file is reached, if desired.

```
func doSomething() {
        . . .
vm.disableTraceFileSource()
        . . .
}
```