OpenXC VI Firmware Documentation

Release 6.0.1

Ford Motor Company

July 04, 2014

Contents

Version 6.0.1

Web http://openxcplatform.com

Documentation http://vi-firmware.openxcplatform.com

Source http://github.com/openxc/vi-firmware

The OpenXC vehicle interface (VI) firmware runs on a microcontroller connected to one or more CAN buses. It receives either all CAN messages or a filtered subset, performs any unit conversion or factoring required and outputs a generic version to a USB, Bluetooth or network interface.

Warning: This portion of the site covers advanced topics such as writing and compiling your own firmware from source. These steps are NOT required for flashing a pre-compiled binary firmware.
If you've downloaded a pre-built binary firmware for your car, locate your VI in the list of supported interfaces to find instructions for programming it. You don't need anything from the VI firmware documentation itself - most users don't need anything in this documentation. Here be dragons!

CHAPTER 1

Getting Started

If you've downloaded a pre-built binary firmware for your car, locate your VI in the list of supported interfaces to find instructions for programming it. You don't need anything from the VI firmware documentation itself - most users don't need anything in this documentation. Really, you can stop here!

The first steps for building custom firmware are:

1. Make sure you have one of the *supported hardware platforms*.
2. *Set up your development environment*.
3. *Compile with emulated data output* to make sure your development environment is set up, including whatever tool or library you plan to use to interact with the VI.
4. *Compile the default configuration* so you can start reading standard On-Board Diagnostics (OBD-II) data from your car.
5. To go beyond OBD-II, create a *create a configuration file* for your car.
6. With a custom configuration in hand, *compile with your new config*.

1.1 Development Environment Setup

Before we can compile, we need to set up our development environment. Bear with it...there's a few steps but you only have to do it once!

When you see `$` it means this is a shell command - run the command after the `$` but don't include the `$`. The example shell commands may also be prefixed with a folder name, if it needs to be run from a particular location, e.g. `foo/ $ ls` means to run `ls` from the `foo` folder.

1.1.1 Windows

If you already use Cygwin for development in Windows and are comfortable with its command line its quirks that pop up occasionally, the 32-bit version of Cygwin is a fully supported build environment for the VI firmware.

If you do not use Cygwin for anything else, you might consider running an Ubuntu Linux virtual machine. Ubuntu is also a 100% supported build environment for the firmware, and it tends to be more straightforward and easy to set up. If debugging strange Cygwin errors isn't something you feel like doing, we strongly recommend the Linux virtual machine method.

If you wish to use an Ubuntu Linux virtual machine, follow one of the many quickstart guides available online (a good example) to install VirtualBox, download an Ubuntu disc image, and install Ubuntu into a virtual machine. Once installed, jump down to the *Linux* section of this guide to wrap up.

If you still wish to use Cygwin, download 32-bit version of Cygwin (even if you're on 64-bit Windows) and run the installer - during the installation process, select these packages:

```
make, gcc-core, patchutils, git, unzip, python, check, curl, libsasl2, python-setuptools
```

After it's installed, open a new Cygwin terminal and configure it to ignore Windows-style line endings in scripts by running this command:

```
$ set -o igncr && export SHELLOPTS
```

1.1.2 Linux

Install Git with your Linux distribution's package manager:

Ubuntu:

```
$ sudo apt-get install git
```

Arch Linux:

```
$ [sudo] pacman -S git
```

1.1.3 OS X

Open the Terminal app and install Homebrew by running this command:

```
$ ruby -e "$(curl -fsSkL raw.github.com/mxcl/homebrew/go)"
```

Once Homebrew is installed, use it to install Git:

```
$ brew install git
```

1.1.4 All Platforms

If we're on a network that requires an Internet proxy (e.g. at work on a corporate network) set these environment variables.

```
$ export http_proxy=<your proxy>
$ export https_proxy=$http_proxy
$ export all_proxy=$http_proxy
```

Clone the vi-firmware repository:

```
$ git clone https://github.com/openxc/vi-firmware
```

Run the `bootstrap.sh` script:

```
$ cd vi-firmware
vi-firmware/ $ script/bootstrap.sh
```

If there were no errors, we are ready to compile. If there are errors, try to follow the recommendations in the error messages. You may need to *manually install the dependencies* if your environment is not in a predictable state. The `bootstrap.sh` script is tested in 32-bit Cygwin, OS X Mountain Lion and Mavericks, Ubuntu 13.04 and Arch Linux.

1.2 Testing with Emulated Data

At this point, we will assume you've *set up your development environment* and continue on to test it.

You can confirm your development environment is set up correctly by compiling the firmware with the default configuration. This build is not configured to read any particular CAN signals or messages, but it allow you to send On-Board Diagnostic (OBD-II) requests and raw CAN messages for experimentation.

Assuming you have a reference VI from Ford, move to the `vi-firmware/src` directory and compile for the `FORDBOARD` platform:

```
vi-firmware/ $ cd src
vi-firmware/src $ PLATFORM=FORDBOARD DEFAULT_EMULATED_DATA_STATUS=1 make -j4
Compiling for FORDBOARD...
...lots of output...
Compiled successfully for FORDBOARD running under a bootloader.
```

There will be a lot more output when you run this but it should end with `Compiled successfully...`. If you got an error, try and follow what it recommends, then look at the troubleshooting section, and finally ask for help on the Google Group.

The compiled firmware is located at `src/build/FORDBOARD/vi-firmware-FORDBOARD.bin`. Find the instructions for re-flashing your VI on the *supported hardware platforms* page and flash with this new firmware. For other platforms, the location will be slightly different in the `build` directory - e.g. for the `CHIPKIT` platform it will be at `src/build/CHIPKIT/vi-firmware-CHIPKIT.hex`.

Finally, test that you can receive the emulated data output stream using the OpenXC Python library:

1. You should already have the OpenXC Python library installed after running the `bootstrap.sh` script, but if not, install the library with `pip`. Don't forget a USB backend.
2. Attach the programmed VI to your computer with a USB cable. In Windows, install the VI windows driver.
3. Run `openxc-control version` from the command line - it should print out the current firmware version of the attached vehicle interface. If you instead get an error about not being able to find the USB device, make sure the VI has power (look for an LED).
4. If the version check was successful, run `openxc-dump` to view the raw data stream of emulated vehicle data coming from the VI.

1.3 Compiling the Default Configuration

Going beyond emulated data, you can start using your VI with a real vehicle by using the default configuration. default configuration. This build is not configured to read any particular CAN signals or messages, but it allow you to send On-Board Diagnostic (OBD-II) requests and raw CAN messages for experimentation.

Again, assuming you've *set up your development environment* and you have a reference VI from Ford, move to the `vi-firmware/src` directory and compile for the `FORDBOARD` platform:

```
vi-firmware/ $ cd src
vi-firmware/src $ export PLATFORM=FORDBOARD
vi-firmware/src $ make clean
vi-firmware/src $ make -jj4
Compiling for FORDBOARD...
...lots of output...
Compiled successfully for FORDBOARD running under a bootloader.
```

Make sure you run `make clean` first whenever changing the environment variable flags (e.g. we omitted `DEFAULT_EMULATED_DATA_STATUS` this time, so the emulated data isn't generated).

Just as with the emulated data build, there will be a lot more output when you run this but it should end with `Compiled successfully...`. If you got an error, try and follow what it suggests, then look at the troubleshooting section, and finally ask for help on the Google Group.

Re-flash your VI (go back to the section on *Testing with Emulated Data* if you forgot how to do that), and try the `openxc-version` command again to make sure it's running your new version.

You can use the `openxc-diag` tool (also from the OpenXC Python library) to send a simple OBD-II request for the engine speed (RPM) to your car. Plug the VI into your car, then attach via USB and run:

```
$ openxc-diag --id 0x7df --mode 1 --pid 0xc
{"success": true, "bus": 1, "id": 2016, "mode": 1, "pid": 12, "payload": "0x0"}
```

1.4 Customizing the VI Configuration

The open source VI firmware doesn't include any CAN message definitions. If you know the details of your CAN signals, you can add your own implementation of the functions defined in `signals.h` to a file named `signals.cpp`.

In this example, we'll pick a simple use case and walk through how to configure and compile the firmware. You'll need to be comfortable getting around at the command line, but you don't need to know any C++. The VI firmware can be configured and built in Windows, Linux (Ubuntu and Arch are tested) or Mac OS X.

Let's say we have a vehicle with a high speed CAN bus on the standard HS pins, connecting to the "CAN1" controller on our vehicle interface. There's a CAN message on this bus sent by an ECU, and we want to read one numeric signal from the message - for this example, let it be the accelerator pedal position as a percentage.

1.4.1 CAN Message and Signal Details

The message contains driver control signals, so we'll give it the name `Driver_Controls` so we can keep track of it. The message ID is `0x102`.

In the message, there is a signal we'll call `Accelerator_Pedal_Pos` that starts at bit `5` and is `7` bits wide - enough to represent pedal positions from 0 to 100.

The value on the bus is exactly how we want it to appear in the translated version over USB or Bluetooth. We want the name to be `accelerator_pedal_position` and we want to hide the rest of the details. We want the output format, sent via USB and UART (i.e. Bluetooth) from the VI to be the standard translated OpenXC message format (see the message format specification):

```
{"name": "accelerator_pedal_position", "value": 42}
```

1.4.2 JSON Configuration

The configuration files for the VI firmware are stored as a single JSON object in a file. JSON is a human-readable data format that's a alternative to XML - we use it because it's easy to parse, (sorta) easy to write by hand and the syntax is fairly obvious.

CAN Bus Definition

We'll start by defining the CAN buses that we want to connect - open a file called `accelerator-config.json` and add this:

```
{   "name": "accelerator",
    "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000
        }
    }
}
```

- We gave this configuration the name `accelerator` - that will show up when we query for the query from the VI.
- We defined 1 CAN bus and called it `hs` for "high speed" - the name is arbitrary but we'll use it later on, so make it short and sweet. `hs`, `ms`, `info` - these are good names.
- We configured this bus to be connected to the `#1` controller on the VI - that's typically what's connected to the high speed bus in most vehicles.
- We set the speed of this CAN bus at 500Kbps - the `speed` attribute is in bytes per second, so we set it to `500000`.

CAN Message Definition

Next up, let's define the CAN message we want to translate from the bus. Modify the file so it looks like this:

```
{   "name": "accelerator",
    "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000
        }
    },
    "messages": {
        "0x102": {
            "name": "Driver_Controls",
            "bus": "hs"
        }
    }
}
```

- We added a `messages` field to the JSON object.
- We added a `0x102` field to `messages` - that's our CAN message's ID and we use it here as a "key" for the object.
- Within the `0x102` message object:
- We set the name of the message. This is just used in comments so we can keep track of which message is which, rather than memorizing the ID.
- We set the `bus` field to `hs`, so this message will be pulled from the bus we defined (and named `hs`).

CAN Signal Definition

Don't stop yet...we have to define our CAN signal before anything will be translated. Modify the file again:

```
{   "name": "accelerator",
    "buses": {
        "hs": {
```

```
            "controller": 1,
            "speed": 500000
        }
    },
    "messages": {
        "0x102": {
            "name": "Driver_Controls",
            "bus": "hs",
            "signals": {
                "Accelerator_Pedal_Pos": {
                    "generic_name": "accelerator_pedal_position",
                    "bit_position": 5,
                    "bit_size": 7
                }
            }
        }
    }
}
```

- We added a `signals` field to the `0x102` message object, after the `name`. The order doesn't matter, just watch out for the commas required after each field and value pair. There's no comma after the last field in an object.
- We added an `Accelerator_Pedal_Pos` field in the `signals` object - that's the name of the signal, and like the message name, this is just for human readability.
- The `generic_name` is what the `name` field will be in the translated format over USB and Bluetooth - we set it to `accelerator_pedal_position`.
- We set the `bit_position` and `bit_size` for the signal.

That's it - the configuration is finished. When we compile the VI firmware with this configuration, it will read our CAN message from the bus, parse and translate it into a JSON output message with a `name` and `value`, and send it out over USB and Bluetooth. Next, we'll *walk through how to do the compilation with your config*.

1.5 Compiling with a Custom Configuration

The VI firmware doesn't understand the JSON configuration file format natively, so next you have to convert it to a C++ implementation. The OpenXC Python library includes a tool that will *generate the C++* for `signals.cpp` from your configuration file, so you still don't have to write any code.

You should already have the Python library installed from the environment setup you did earlier (and if not, run *pip install openxc*). Assuming the `accelerator-config.json` file we created is in our home directory, run this to generate a valid `signals.cpp` for our CAN signal:

```
vi-firmware/src $ openxc-generate-firmware-code --message-set ~/accelerator-config.json > signals.cpp
```

and then re-compile the firmware:

```
vi-firmware/src $ export PLATFORM=FORDBOARD
vi-firmware/src $ make clean
vi-firmware/src $ make -j4
Compiling for FORDBOARD...
...
15 Compiling build/FORDBOARD/signals.o
Producing build/FORDBOARD/vi-firmware-FORDBOARD.elf
Producing build/FORDBOARD/vi-firmware-FORDBOARD.bin
Compiled successfully for FORDBOARD running under a bootloader.
```

Success! The compiled firmware is located at `build/FORDBOARD/vi-firmware-FORDBOARD.bin`. We can use the same VI re-flashing procedure that we used for a binary firmware from an automaker with our custom firmware - the process is going to depend on the VI you have, so see the list of supported VIs to find the right instructions.

There's a *lot* more you can do with the firmware - many more CAN signals simultaneously, raw CAN messages, diagnostic reqwuests, advanced data transformation, etc. For complete details, see the VI Firmware docs. You can find the right `PLATFORM` value for your VI in the VI firmware supported platforms page.

CHAPTER 2

Firmware Configuration

If you cannot use a pre-built binary firmware from an automaker you can create a **VI configuration file** and use the code generation tool in the OpenXC Python library. Many *examples of configuration files* are included in the docs, as well as a *complete reference* for all configuration options

Knowledge of the car's CAN messages is required to build a custom configuration file. If you're just looking to get some data out of your car you most likely want a binary firmware from your car's maker. If they don't offer one, get together with the community to reverse engineer it!

2.1 Read-only Configuration Examples

If you haven't created a custom firmware for the OpenXC VI yet, we recommend the getting started with custom data guide.

For all examples, the `name` field for a message is optional but strongly encouraged to help keep track of the mapping between a message ID and something human readable.

When an example refers to "sending" a translated or raw message, it means sending to the app developer via one of the output interfaces (e.g. USB, Bluetooth) and not sending to the CAN bus. For examples of configuring writable messages and signals that *do* write back to the CAN bus, see the *write configuration examples*.

2.1.1 One Bus, One Numeric Signal

We want to read a single, numeric signal from a high speed bus on controller 1. The signal is 7 bits wide, starting from bit 5 in message ID `0x102`. We want the name of the signal for OpenXC app developers to be

my_openxc_measurement.

```
{   "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000
        }
    },
    "messages": {
        "0x102": {
            "bus": "hs",
            "signals": {
                "My_Signal": {
                    "generic_name": "my_openxc_measurement",
                    "bit_position": 5,
                    "bit_size": 7
                }
            }
        }
    }
}
```

With this configuration, the VI will publish the received CAN signal using the OpenXC single-valued, translated message format, e.g. when using the JSON output format:

```
{"name": "my_openxc_measurement", "value": 42}
```

2.1.2 Transformed Numeric Signal

We want to read the same signal as in the *One Bus, One Numeric Signal* example, but we want to transform the value with a factor and offset before publishing it. The value on CAN must be multiplied by -1.0 and offset by 1400.

```
{   "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000
        }
    },
    "messages": {
        "0x102": {
            "bus": "hs",
            "signals": {
                "My_Signal": {
                    "generic_name": "my_openxc_measurement",
                    "bit_position": 5,
                    "bit_size": 7,
                    "factor": -1.0,
                    "offset": 1400
                }
            }
        }
    }
}
```

We added the `factor` and `offset` attributes to the signal.

2.1.3 One Bus, One Boolean Signal

We want to read a boolean signal from a high speed bus on controller 1. The signal is 1 bits wide, starting from bit 32 in message ID `0x103`. We want the name of the signal for OpenXC app developers to be `my_boolean_measurement`. Because it is a boolean type, the value will appear as `true` or `false` in the JSON for app developers.

```
{   "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000
        }
    },
    "messages": {
        "0x103": {
            "bus": "hs",
            "signals": {
                "My_Boolean_Signal": {
                    "generic_name": "my_boolean_measurement",
                    "bit_position": 32,
                    "bit_size": 1,
                    "decoder": "booleanDecoder"
                }
            }
        }
    }
}
```

We set the `decoder` for the signal to the `booleanDecoder`, one of the *built-in signal decoders* - this will transform the numeric value from the bus (a `0` or `1`) into first-class boolean values (`true` or `false`).

With this configuration, the VI will publish the received CAN signal using the OpenXC single-valued, translated message format, e.g. when using the JSON output format:

```
{"name": "my_boolean_measurement", "value": true}
```

2.1.4 One Bus, One State-based Signal

We want to read a signal from a high speed bus on controller 1 that has numeric values corresponding to a set of states - what we call a state-based signal

The signal is 3 bits wide, starting from bit 28 in message ID `0x104`. We want the name of the signal for OpenXC app developers to be `active_state`. There are 6 valid states from 0-5, and we want those to appears as the state strings `a` through `f` in the JSON for app developers.

```
{   "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000
        }
    },
    "messages": {
        "0x104": {
            "bus": "hs",
            "signals": {
                "My_State_Signal": {
                    "generic_name": "active_state",
                    "bit_position": 28,
```

```
                    "bit_size": 3,
                    "states": {
                        "a": [0],
                        "b": [1],
                        "c": [2],
                        "d": [3],
                        "e": [4],
                        "f": [5]
                    }
                }
            }
        }
    }
}
```

We set the `states` field for the signal to a JSON object, mapping the string value for each state to the numerical values to which it corresponds. This automatically will set the `decoder` to the `stateDecoder`, one of the *built-in signal decoder functions*.

With this configuration, the VI will publish the received CAN signal using the OpenXC single-valued, translated message format, e.g. when using the JSON output format:

```
{"name": "active_state", "value": "a"}
```

2.1.5 Combined State-based Signal

We want to read the same state-based signal from *One Bus, One State-based Signal* but we want the values 0-3 on the bus to all correspond with state `a` and values `4-5` to the string state `b`.

```
{   "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000
        }
    },
    "messages": {
        "0x104": {
            "bus": "hs",
            "signals": {
                "My_State_Signal": {
                    "generic_name": "active_state",
                    "bit_position": 28,
                    "bit_size": 3,
                    "states": {
                        "a": [0, 1, 2, 3],
                        "b": [4, 5]
                    }
                }
            }
        }
    }
}
```

Each state string maps to an array - this can seem unnecessary when you only have 1 numeric value for each state, but it allows combined mappings as in this example.

2.1.6 Two Buses, Two Signals

We want to read two numeric signals - one from a message on a high speed bus on controller 1, and the other from a message on a medium speed bus on controller 2.

The signal on the high speed bus is 12 bits wide, starting from bit 11 in message ID `0x108`. We want the name of the signal for OpenXC app developers to be `my_first_measurement`.

The signal on the medium speed bus 14 bits wide, starting from bit 0 in message ID `0x90`. We want the name of the signal for OpenXC app developers to be `my_second_measurement`.

```
{   "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000
        },
        "ms": {
            "controller": 2,
            "speed": 125000
        }
    },
    "messages": {
        "0x108": {
            "bus": "hs",
            "signals": {
                "My_Signal": {
                    "generic_name": "my_first_measurement",
                    "bit_position": 11,
                    "bit_size": 12
                }
            }
        },
        "0x90": {
            "bus": "ms",
            "signals": {
                "My_Other_Signal": {
                    "generic_name": "my_second_measurement",
                    "bit_position": 0,
                    "bit_size": 14
                }
            }
        }
    }
}
```

We added the second bus to the `buses` field and assigned it to controller 2. We added the second message object and made sure to set its `bus` field to `ms`.

With this configuration, the VI will publish the received CAN signals using the OpenXC single-valued, translated message format, e.g. when using the JSON output format:

```
{"name": "my_first_measurement", "value": 42}
{"name": "my_second", "value": 942}
```

2.1.7 Limited Translated Signal Rate

We want to read the same signal as in the *One Bus, One Numeric Signal* example, but we want it to be sent at a maximum of 5Hz. We want the firmware to pick out messages at a regular period, but we don't care which data is

dropped in order to stay under the maximum.

```
{   "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000
        }
    },
    "messages": {
        "0x102": {
            "bus": "hs",
            "signals": {
                "My_Signal": {
                    "generic_name": "my_openxc_measurement",
                    "bit_position": 5,
                    "bit_size": 7,
                    "max_frequency": 5
                }
            }
        }
    }
}
```

We set the `max_frequency` field of the signal to 5 (meaning 5Hz) - the firmware will automatically handle skipping messages to stay below this limit.

2.1.8 Limited Translated Signal Rate if Unchanged

We want the same signal from *Limited Translated Signal Rate* at a limited rate, but we don't want to lose any information - if the value of the signal changes, we want it to be sent regardless of the max frequency. Repeated, duplicate signal values are fairly common in vehicles, where a signal is sent at a steady frequency even if the value hasn't changed. For this example, we want to preserve all information - if a signal changes, we want to make sure the data is sent.

```
{   "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000
        }
    },
    "messages": {
        "0x102": {
            "bus": "hs",
            "signals": {
                "My_Signal": {
                    "generic_name": "my_openxc_measurement",
                    "bit_position": 5,
                    "bit_size": 7,
                    "max_frequency": 5,
                    "force_send_changed": true
                }
            }
        }
    }
}
```

We added the `force_send_changed` field to the signal, which will make sure the signal is sent immediately when the value changes. This rate limiting is lossless.

2.1.9 Send Signal on Change Only

We want to limit the rate of a signal as in *Limited Translated Signal Rate if Unchanged*, but we want to be more strict - the signal should only be published if it actually changes.

```
{    "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000
        }
    },
    "messages": {
        "0x102": {
            "bus": "hs",
            "signals": {
                "My_Signal": {
                    "generic_name": "my_openxc_measurement",
                    "bit_position": 5,
                    "bit_size": 7,
                    "send_same": false
                }
            }
        }
    }
}
```

We accomplish this by setting the `send_same` field to false. This is most appropriate for boolean and state-based signals where the transition is most important. Considering that a host device may connect to the VI *after* the message has been sent, using this field has the potential of making it difficult to tell the current state of the vehicle on startup - you have to wait for a state change before knowing any values. For that reason, we've moved away from using this for most firmware (using a combination of a `max_frequency` of 1Hz and `force_send_changed == true`) but the option is still available.

2.1.10 Separate Files for Message Sets

Starting from the *Two Buses, Two Signals* example, we want to split up the configuration into mutiple files because it's getting too big and hard to follow. This will especially be true as we add more message and signals.

Starting from this complete configuration:

```
{    "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000
        },
        "ms": {
            "controller": 2,
            "speed": 125000
        }
    },
    "messages": {
        "0x108": {
            "bus": "hs",
            "signals": {
                "My_Signal": {
                    "generic_name": "my_first_measurement",
                    "bit_position": 11,
```

```
                    "bit_size": 12
                }
            }
        },
        "0x90": {
            "bus": "ms",
            "signals": {
                "My_Other_Signal": {
                    "generic_name": "my_second_measurement",
                    "bit_position": 0,
                    "bit_size": 14
                }
            }
        }
    }
}
```

we move the messages that we want to read from the `hs` bus to the file `hs.json`:

```
{
    "messages": {
        "0x108": {
            "signals": {
                "My_Signal": {
                    "generic_name": "my_first_measurement",
                    "bit_position": 11,
                    "bit_size": 12
                }
            }
        }
    }
}
```

and we move the messages that we want to read from the `ms` bus to the file `ms.json`:

```
{
    "messages": {
        "0x90": {
            "signals": {
                "My_Other_Signal": {
                    "generic_name": "my_second_measurement",
                    "bit_position": 0,
                    "bit_size": 14
                }
            }
        }
    }
}
```

Notice in both of these files, the messages no longer have the `bus` attribute - we're instead going to specify that in the top level configuration:

```
{   "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000
        },
        "ms": {
            "controller": 2,
```

```
            "speed": 125000
        }
    },
    "mappings": [
        {"mapping": "hs.json", "bus": "hs"},
        {"mapping": "ms.json", "bus": "ms"}
    ]
}
```

The primary advantage of using separate files is readability, but it also makes the message definitions more re-usable between vehicle platforms and buses. For example, we could quickly parse all of the messages from the `ms.json` mapping file from the `hs` bus instead of `ms` by flipping the `bus` attribute in the top-level config file.

2.1.11 Mapped from a DBC File

If you use Vector DBC files to store your "gold standard" CAN signal definitions, you can save some effort by exporting the DBC to an XML file and merging it with your VI configuration file. You won't need to manually copy the `bit_position`, `bit_size`, `factor` and `offset` attributes.

If we are to implement *One Bus, One Numeric Signal* manually, we would use this configuration file:

```
{   "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000
        }
    },
    "messages": {
        "0x102": {
            "bus": "hs",
            "signals": {
                "My_Signal": {
                    "generic_name": "my_openxc_measurement",
                    "bit_position": 5,
                    "bit_size": 7,
                    "factor": -1.0,
                    "offset": 1400
                }
            }
        }
    }
}
```

If the message and signal is defined in a DBC file, we can save some effort. Using a program like Vector CANdb++, export the DBC file to XML. Place the XML file in the same directory as your JSON configuration file. We need to first split up the configuration into a mapped messages file and a top-level config, as in *Separate Files for Message Sets* example.

In our `config.json`:

```
{   "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000
        }
    },
    "mappings": [
         {"mapping": "hs.json", "bus": "hs"}
```

```
    ]
}
```

and in `hs.json`:

```
{
    "messages": {
        "0x102": {
            "signals": {
                "My_Signal": {
                    "generic_name": "my_openxc_measurement",
                    "bit_position": 5,
                    "bit_size": 7,
                    "factor": -1.0,
                    "offset": 1400
                }
            }
        }
    }
}
```

Now that we have the DBC exported to an XML file (we'll assume it's named `exported-hs.xml`), we can remove the `bit_position`, `bit_size`, `factor` and `offset` fields and let them be imported from the XML - the only thing required is a `generic_name`:

In our `config.json`:

```
{   "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000
        }
    },
    "mappings": [
         {"mapping": "hs.json", "bus": "hs",
            "database": "exported-hs.xml"}
    ]
}
```

and in `hs.json`:

```
{
    "messages": {
        "0x102": {
            "signals": {
                "My_Signal": {
                    "generic_name": "my_openxc_measurement"
                }
            }
        }
    }
}
```

It's not huge savings for 1 signal, but once you get a dozen it can save a lot of effort and opportunities for bugs.

2.1.12 Same Message ID, Two Buses

One shortcoming of a single configuration file is that you can't define a CAN message with the same ID to exist on two different, buses. For example, this isn't value JSON because the `0x100` key is repeated:

```
{
    "messages": {
        "0x100": {
            "bus": "hs"
         },
         "0x100": {
            "bus": "ms"
         }
    }
}
```

Instead, you can use mappings files as in *Separate Files for Message Sets* and put the messages for each bus in separate files. Here's the main configuration file:

```
{   "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000
        },
        "ms": {
            "controller": 2,
            "speed": 125000
        }
    },
    "mappings": [
        {"mapping": "hs.json", "bus": "hs"},
        {"mapping": "ms.json", "bus": "ms"}
    ]
}
```

and here's `hs.json`:

```
{
    "messages": {
        "0x100": {
             "My_Signal": {
                 "generic_name": "my_first_measurement",
                 "bit_position": 3,
                 "bit_size": 7
             }
        }
    }
}
```

and finally, `ms.json`:

```
{
    "messages": {
        "0x100": {
            "signals": {
                "My_Other_Signal": {
                    "generic_name": "my_second_measurement",
                    "bit_position": 0,
                    "bit_size": 14
```

```
                    }
                }
            }
        }
}
```

The two different CAN messages with the same ID can co-exist in these separate files, linked as mappings through the main config.

2.2 Read-only Raw CAN Configuration Examples

If you don't care about abstracting the details of CAN messages from developers (or perhaps you're the only developer and you're working directly with CAN data), you can configure the VI to output full, low-level CAN messages. You can read every message or a filtered subset, but be aware that not every VI has enough horsepower to send every CAN messages through as JSON.

2.2.1 Unfiltered Raw CAN

We want to read all raw CAN messages from a bus at full speed. Be aware that the VI hasn't been optimized for this level of throughput, and it's not guaranteed at this time that messages will not be dropped. We recommend using rate limiting, which can dramatically decrease the bandwidth required without losing any information.

```
{   "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000,
            "raw_can_mode": "unfiltered"
        }
    }
}
```

With this configuration, the VI will publish all CAN messages it receives using the OpenXC raw CAN message format, e.g. when using the JSON output format:

```
{"bus": 1, "id": 1234, "value": "0x12345678"}
```

2.2.2 Filtered Raw CAN

We want to read only the message with ID `0x21` from a high speed bus on controller 1.

```
{   "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000,
            "raw_can_mode": "filtered"
```

```
        }
    },
    "messages": {
      "0x21": {
        "bus": "hs"
      }
    }
}
```

We added the `0x21` message and assigned it to bus `hs`, but didn't define any signals (it's not necessary when using the raw CAN mode).

This configuration will cause the VI to publish using the OpenXC raw CAN message format, but you will only received message `0x21`.

2.2.3 Unfiltered Raw CAN with Limited, Variable Data Rate

We want to read all raw CAN messages from a bus, but we don't want the output interface to be overwhelmed by repeated duplicate messages. This is fairly common in vehicles, where a message is sent at a steady frequency even if the value hasn't changed. For this example, we want to preserve all information - if a message changes, we want to make sure the data is sent.

```
{   "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000,
            "raw_can_mode": "unfiltered",
            "max_message_frequency": 1,
            "force_send_changed": true
        }
    }
}
```

We combine two attributes to both limit the data rate from raw CAN messages, and also make sure the transfer is lossless. The `max_message_frequency` field sets the maximum send frequency for CAN messages that have not changed to 1Hz. We also set the `force_send_changed` field to `true`, which will cause a CAN message with a new value to be sent to the output interface immediately, even if it would go above the 1Hz frequency. The default is `true`, so we could also leave this parameter out for the same effect. The result is that each CAN message is sent at a minimum of 1Hz and a maximum of the true rate of change for the message.

2.2.4 Unfiltered Raw CAN with Strict, Limited Data Rate

We want to read all raw CAN messages as in *Unfiltered Raw CAN with Limited, Variable Data Rate* but we want to set a strict limit on the read frequency of each CAN message. We don't care if we skip some CAN messages, even if they have new data - the maximum frequency is the most important thing.

```
{   "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000,
            "raw_can_mode": "unfiltered",
            "max_message_frequency": 1,
            "force_send_changed": false.
        }
    }
}
```

We set the `force_send_changed` field to false so the firmware will strictly enforce the max message frequency.

2.2.5 Translated and Raw CAN Together

We want to read the same signal as in the *One Bus, One Numeric Signal* example, but we also want to receive all unfiltered raw CAN messages simultaneously.

```
{    "buses": {
         "hs": {
             "controller": 1,
             "raw_can_mode": "unfiltered",
             "speed": 500000
         }
     },
     "messages": {
         "0x102": {
             "bus": "hs",
             "signals": {
                 "My_Signal": {
                     "generic_name": "my_openxc_measurement",
                     "bit_position": 5,
                     "bit_size": 7
                 }
             }
         }
     }
}
```

We added set the `raw_can_mode` for the bus to `unfiltered`, as in *Unfiltered Raw CAN*. No other changes are required - the raw and translated message co-exist peacefully. If we set `raw_can_mode` to `filtered`, it would only send the raw message for `0x102`, where we're getting the numeric signal.

With this configuration, the VI will publish a mixed stream of OpenXC messages, both the raw CAN message format, and the translated message format, e.g. when using the JSON output format:

```
{"bus": 1, "id": 258, "value": "0x12345678"}
{"name": "my_openxc_measurement", "value": 42}
```

2.3 Custom Handler Configuration

Sometimes a bit of C++ is required beyond the JSON configuration to implement advanced features. You can optionally inject your own code into the generated C++ file, so you can still take advantage of the configuration file for the simple signals.

2.3.1 Custom Transformed Numeric Signal

Similar to the *Transformed Numeric Signal* example, we want to modify a numeric value read from a CAN signal before sending it to the app developer, but the the desired transformation isn't as simple as an offset. We want to read the same signal as before, but if it's below 100 it should be rounded down to 0. We want our custom transformation to happen *after* using the existing factor and offset.

To accomplish this, we need to know a little C - we will write a custom signal decoder to make the transformation. Here's the JSON configuration:

```
{   "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000
        }
    },
    "messages": {
        "0x102": {
            "bus": "hs",
            "signals": {
                "My_Signal": {
                    "generic_name": "my_openxc_measurement",
                    "bit_position": 5,
                    "bit_size": 7,
                    "factor": -1.0,
                    "offset": 1400,
                    "decoder": "ourRoundingDecoder"
                }
            }
        }
    },
    "extra_sources": [
      "my_handlers.cpp"
    ]
}
```

We set the `decoder` for the signal to `ourRoundingDecoder`, and we'll define that in a separate file named `my_handlers.cpp`. We also added the `extra_sources` field, which is a list of the names of C++ source files on our path to be included with the generated firmware code.

In `my_handlers.cpp`:

```
/* Round the value down to 0 if it's less than 100. */
float ourRoundingDecoder(CanSignal* signal, CanSignal* signals,
      int signalCount, float value, bool* send) {
   if(value < 100) {
      value = 0;
   }
   return value;
}
```

After being transformed with the factor and offset for the signal from the configuration file, the value is passed to our decoder. We make whatever custom transformation required and return the new value.

There are a few other valid type signatures for these *custom value decoders* - for converting numeric values to boolean or state-based signals.

2.3.2 Transformed with Signal Reference

We need to combine the values of two signals from a CAN message to create a single value - one signal is the absolute value, the other is the sign.

Both signals are on the high speed bus in the message with ID `0x110`. The absolute value signal is 5 bits wide, starting from bit 2. The sign signal is 1 bit wide, starting from bit 12 - when the value of the sign signal is 0, the final value should be negative. We want to the final value to be sent to app developers with the name `my_signed_measurement`.

We will use a custom decoder for the signal to reference the sign signal's last value when transforming the absolute value signal.

```
{   "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000
        }
    },
    "messages": {
        "0x110": {
            "bus": "hs",
            "signals": {
                "My_Value_Signal": {
                    "generic_name": "my_signed_measurement",
                    "bit_position": 2,
                    "bit_size": 5,
                    "decoder": "ourSigningDecoder"
                },
                "My_Sign_Signal": {
                    "generic_name": "sign_of_signal",
                    "bit_position": 12,
                    "bit_size": 1,
                    "decoder": "ignoreDecoder"
                }
            }
        }
    },
    "extra_sources": [
      "my_handlers.cpp"
    ]
}
```

We don't want to the sign signal to be sent separately on the output interfaces, but we need the firmware to read and store its value so we can refer to it from our custom decoder. We set the sign signal's `decoder` to `ignoreDecoder` which will still process and store the value, but withold it from the output data stream.

For the absolute value signal, we set the `decoder` to a custom function where we look up the sign signal and use its value to transform the absolute value. In `my_handlers.cpp`:

```
/* Load the last value for the sign signal and multiply the absolute value
by it. */
float ourRoundingDecoder(CanSignal* signal, CanSignal* signals,
      int signalCount, float value, bool* send) {
    CanSignal* signSignal = lookupSignal("sign_of_signal",
            signals, signalCount);

    if(signSignal == NULL) {
        debug("Unable to find sign signal");
```

```
        *send = false;
    } else {
        if(signSignal->lastValue == 0) {
            // left turn
            value *= -1;
        }
    }
    return value;
}
```

We use the *lookupSignal'* function to load a struct representing the `sign_of_signal` CAN signal we defined in the configuration, and check the `lastValue` attribute of the struct. If for some reason we aren't able to find the configured sign signal, `lookupSignal` will return NULL and we can stop hold the output of the final value by flipping `*send` to false. The firmware will check the value of `*send` after each call to a decoder to confirm if the translation pipeline should continue.

One slight problem with this approach: there is currently no guaranteed ordering for the signals. It's possible the `lastValue` for the sign signal isn't from the same message as the absolute value signal you are current handling in the function. With a continuous value, there's only a small window where this could happen, but if you must be sure the values came from the same message, you may need to write a *Composite Signal*.

2.3.3 Composite Signal

We want complete control over the output of a measurement from the car. We have a CAN message that includes 3 different signals that represent a GPS latitude value, and want to combine them into a single value in degrees.

The three signals are in the message `0x87` on a high speed bus connected to controller 1. The three signals:

- The whole latitude degrees signal starts at bit 10 and is 8 bits wide. The value on CAN requires an offset of -89.0.
- The latitude minutes signal starts at bit 18 and is 6 bits wide.
- The latitude minute fraction signal starts at bit 24 and is 14 bits wide. The value on CAN requires a factor of .0001.

```
{   "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000
        }
    },
    "messages": {
        "0x87": {
            "bus": "hs",
            "handlers": ["latitudeMessageHandler"],
            "signals": {
                "Latitude_Degrees": {
                    "generic_name": "latitude_degrees",
                    "bit_position": 10,
                    "bit_size": 8,
                    "offset": -89,
                    "ignore": true
                },
                "Latitude_Minutes": {
                    "generic_name": "latitude_minutes",
                    "bit_position": 18,
                    "bit_size": 6,
```

```
                    "ignore": true
                },
                "Latitude_Minute_Fraction": {
                    "generic_name": "latitude_minute_fraction",
                    "bit_position": 24,
                    "bit_size": 14,
                    "factor": 0.0001,
                    "ignore": true
                },
            }
        }
    },
    "extra_sources": [
      "my_handlers.cpp"
    ]
}
```

We made two changes to the configuration from a simple translation config:

- We set the `ignore` field to `true` for each of the component signals in the message. The signal definitions (i.e. the position, offset, etc) will be included in the firmware build so we can access it from a custom message handler, but the signals will not be processed by the normal translation stack.
- We set the `handlers` for the `0x87` message to an array containing our custom message handler, `latitudeMessageHandler`. This field should be an array as you can provide multiple message handlers. You could just call multiple handlers from a single handler function, but having them all defined in the configuration file makes the behavior more clear at a glance.

In `my_handlers.cpp`:

```
/* Combine latitude signals split into their components (degrees,
 * minutes and fractional minutes) into 1 output message: latitude in
 * degrees with with decimal precision.
 *
 * The following signals must be defined in the signal array, and they must
 * all be contained in the same CAN message:
 *
 *      * latitude_degrees
 *      * latitude_minutes
 *      * latitude_minutes_fraction
 *
 * This is a message handler, and takes care of sending the output message.
 *
 * messageId - The ID of the received GPS latitude CAN message.
 * data - The CAN message data containing all GPS latitude information.
 * signals - The list of all signals.
 * signalCount - The length of the signals array.
 * send - (output) Flip this to false if the message should not be sent.
 * pipeline - The pipeline that wraps the output devices.
 *
 * This type signature is required for all custom message handlers.
 */
void latitudeMessageHandler(CanMessage* message, CanSignal* signals,
        int signalCount, Pipeline* pipeline) {
    // Retrieve the CanSignal struct representations of the 3 latitude
    // component signals. These are still included in the firmware build
    // when the 'ignore' flag was true for the signals.
    CanSignal* latitudeDegreesSignal =
        lookupSignal("latitude_degrees", signals, signalCount);
```

```
    CanSignal* latitudeMinutesSignal =
        lookupSignal("latitude_minutes", signals, signalCount);
    CanSignal* latitudeMinuteFractionSignal =
        lookupSignal("latitude_minute_fraction", signals, signalCount);

    // Confirm that we have all required signal components
    if(latitudeDegreesSignal == NULL ||
            latitudeMinutesSignal == NULL ||
            latitudeMinuteFractionSignal == NULL) {
        debug("One or more GPS latitude signals are missing");
        return;
    }

    // begin by assuming we will send the message, no errors yet
    bool send = true;

    // Decode and transform (using any factor and offset defined in the
    // CanSignal struct) each of the component signals from the message data
    // preTranslate is intended to be used in conjunction with postTranslate
    // - together they keep metadata about the receive signals in memory.
    float latitudeDegrees = preTranslate(latitudeDegreesSignal, data, &send);
    float latitudeMinutes = preTranslate(latitudeMinutesSignal, data, &send);
    float latitudeMinuteFraction = preTranslate(
            latitudeMinuteFractionSignal, data, &send);

    // if we were able to decode all 3 component signals (i.e. none of the
    // calls to preTranslate flipped 'send' to false
    if(send) {
        float latitude = (latitudeMinutes + latitudeMinuteFraction) / 60.0;
        if(latitudeDegrees < 0) {
            latitude *= -1;
        }
        latitude += latitudeDegrees;

        // Send the final latitude value to the output interfaces (via the
        // pipeline)
        sendNumericalMessage("latitude", latitude, pipeline);
    }

    // Conclude by updating the metadata for each of the component signals
    // with postTranslate
    postTranslate(latitudeDegreesSignal, latitudeDegrees);
    postTranslate(latitudeMinutesSignal, latitudeMinutes);
    postTranslate(latitudeMinuteFractionSignal, latitudeMinuteFraction);
}
```

A more complete, functional example of a message handler is included in the VI firmware repository - one that handles both latitude and longitude in a CAN message. There is also additional documentation on the *message handler type signature*.

2.3.4 Initializer Function

We want to initialize a counter when the VI powers up that we will use from some custom signal decoder.

```
{   "buses": {
        "hs": {
            "controller": 1,
```

```
            "raw_can_mode": "unfiltered",
            "speed": 500000
        }
    },
    "messages": {
        "0x102": {
            "bus": "hs",
            "signals": {
                "My_Signal": {
                    "generic_name": "my_openxc_measurement",
                    "bit_position": 5,
                    "bit_size": 7
                }
            }
        }
    },
    "initializers": [
       "initializeMyCounter"
    ],
    "extra_sources": [
      "my_initializers.cpp"
    ]
}
```

We added an `initializers` field, which is an array containing the names of C functions matching the *initializer type signature*.

In `my_initializers.cpp`:

```
int MY_COUNTER;
void initializeMyCounter() {
   MY_COUNTER = 42;
}
```

This isn't a very useful initializer, but there much more you could do - you'll want to look into the lowest level APIs in the firmware source. Look through the `.h` files, where most functions are documented.

2.3.5 Looper Function

We want to increment a counter every time through the main loop of the firmware, regardless of whatever CAN messages we may have received.

```
{   "buses": {
        "hs": {
            "controller": 1,
            "raw_can_mode": "unfiltered",
            "speed": 500000
        }
    },
    "messages": {
        "0x102": {
            "bus": "hs",
            "signals": {
                "My_Signal": {
                    "generic_name": "my_openxc_measurement",
                    "bit_position": 5,
                    "bit_size": 7
                }
```

```
            }
        }
    },
    "loopers": [
       "incrementMyCounter"
    ],
    "extra_sources": [
      "my_loopers.cpp"
    ]
}
```

We added a `loopers` field, which is an array containing the names of C functions matching the *looper type signature*.

In `my_loopers.cpp`:

```
void incrementMyCounter() {
   static int myCounter = 0;
   ++myCounter;
}
```

As with the *initializer*, this isn't a very functional example, but there much more you could do - you'll want to look into the lowest level APIs in the firmware source. Look through the `.h` files, where most functions are documented.

2.3.6 Ignore Depending on Value

We want to ignore a signal (and not translate it and send over USB/Bluetooth) if the value matches certain critera. We'll use a custom decoder as in *Custom Transformed Numeric Signal* but instead of modifying the value of the signal, we'll use the `send` flag to tell the VI if it should process the value or not.

```
{   "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000
        }
    },
    "messages": {
        "0x102": {
            "bus": "hs",
            "signals": {
                "My_Signal": {
                    "generic_name": "my_openxc_measurement",
                    "bit_position": 5,
                    "bit_size": 7,
                    "factor": -1.0,
                    "offset": 1400,
                    "decoder": "ourFilteringDecoder"
                }
            }
        }
    },
    "extra_sources": [
      "my_handlers.cpp"
    ]
}
```

In `my_handlers.cpp`:

```
/* Ignore the signal if the value is less than 100 */
float ourFilteringDecoder(CanSignal* signal, CanSignal* signals,
      int signalCount, float value, bool* send) {
   if(value < 100) {
      *send = false;
   }
   return value;
}
```

2.4 Writable Configuration Examples

For applications that need to send data back to the vehicle, you can configure a variety of CAN writes: raw CAN messages, performing reverse translation of an individual CAN signal, or send diagnostic requests.

Most of these examples build on configurations started for reading data from the bus, so you are strongly encouraged to read, understand and try the *read-only configurations* before continuing.

2.4.1 Translated Numeric Signal Write Request

We want to send a single numeric value to the VI, and have it translated back into a CAN signal in a message on a high speed bus attached to controller 1. The signal is 7 bits wide, starting from bit 5 in message ID `0x102`. We want the name of the signal that will be sent to the VI to be `my_openxc_measurement`.

```
{   "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000
        }
    },
    "messages": {
        "0x102": {
            "bus": "hs",
            "signals": {
                "My_Signal": {
                    "generic_name": "my_openxc_measurement",
                    "bit_position": 5,
                    "bit_size": 7,
                    "writable": true
                }
            }
        }
    }
}
```

This configuration is the same as the read-only example *One Bus, One Numeric Signal* with the addition of the `writable` flag to the signal. When this flag is true, an OpenXC message sent back to the VI from an app with the name `my_openxc_measurement` will be translated to a CAN signal in a new message and written to the bus.

If the VI is configured to use the JSON output format, sending this OpenXC single-valued message to the VI via USB or UART (Bluetooth) would trigger a CAN write:

```
{"name": "my_openxc_measurement", "value": 42}
```

With the tools from the OpenXC Python library you can send that from a terminal with the command:

```
openxc-control write --name my_openxc_measurement --value 42
```

and if you compiled the firmware with `DEBUG=1`, you can view the view the logs to make sure the write went through with this command:

```
openxc-control write --name my_openxc_measurement --value 42 --log-mode stderr
```

2.4.2 Translated Boolean Signal Write Request

We want to send a single boolean value to the VI, and have it translated back into a CAN signal in a message on a high speed bus attached to controller 1. The signal is 1 bits wide, starting from bit 3 in message ID `0x201`. We want the name of the signal that will be sent to the VI to be `my_boolean_request`.

```
{   "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000
        }
    },
    "messages": {
        "0x102": {
            "bus": "hs",
            "signals": {
                "My_Signal": {
                    "generic_name": "my_boolean_request",
                    "bit_position": 3,
                    "bit_size": 1,
                    "writable": true,
                    "encoder": "booleanWriter"
                }
            }
        }
    }
}
```

In addition to setting `writable` to true, We set the `encoder` for the signal to the built-in `booleanWriter`. This will handle converting a `true` or `false` value from the user back to a 1 or 0 in the outgoing CAN message.

If the VI is configured to use the JSON output format, sending this OpenXC single-valued message to the VI via USB or UART (Bluetooth) would trigger a CAN write:

```
{"name": "my_boolean_request", "value": true}
```

With the tools from the OpenXC Python library you can send that from a terminal with the command:

```
openxc-control write --name my_boolean_request --value true
```

2.4.3 Translated State-based Signal Write Request

We want to send a state as a string to the VI, and have it translated back into a numeric CAN signal in a message on a high speed bus attached to controller 1. As in *One Bus, One State-based Signal*, the signal is 3 bits wide, starting from

bit 28 in message ID `0x104`. We want the name of the signal for OpenXC app developers to be `active_state`. There are 6 valid states from 0-5 in the CAN signal, but we want the app developer to send the strings `a` through `f` to the VI.

```
{   "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000
        }
    },
    "messages": {
        "0x102": {
            "bus": "hs",
            "signals": {
                "My_Signal": {
                    "generic_name": "my_state_request",
                    "bit_position": 28,
                    "bit_size": 3,
                    "states": {
                        "a": [0],
                        "b": [1],
                        "c": [2],
                        "d": [3],
                        "e": [4],
                        "f": [5]
                    },
                    "writable": true
                }
            }
        }
    }
}
```

The `writable` field is all that is required - the signal will be automatically configured to use the built-in `stateWriter` as its `encoder` because the signal has a `states` array. If a user sends the VI the value `c` in a write request with the name `my_state_request`, it will be encoded as `2` in the CAN signal in the outgoing message.

If the VI is configured to use the JSON output format, sending this OpenXC single-valued message to the VI via USB or UART (Bluetooth) would trigger a CAN write:

```
{"name": "my_state_request", "value": "a"}
```

With the tools from the OpenXC Python library you can send that from a terminal with the command:

```
openxc-control write --name my_state_request --value "\"a\""
```

Becuase of the way string escaping works from the command prompt, you have to add escaped `\"` characters so the tool knows you want to send a string.

2.4.4 Translated, Transformed Written Signal

We want to write the same signal as *Translated Numeric Signal Write Request* but round any values below 100 down to 0 before sending (similar to the read-only example *Custom Transformed Numeric Signal*).

To accomplish this, we need to know a little C - we will write a custom signal encoder to make the transformation. Here's the JSON configuration:

```
{   "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000
        }
    },
    "messages": {
        "0x102": {
            "bus": "hs",
            "signals": {
                "My_Signal": {
                    "generic_name": "my_openxc_measurement",
                    "bit_position": 5,
                    "bit_size": 7,
                    "factor": -1.0,
                    "offset": 1400,
                    "encoder": "ourRoundingWriteEncoder"
                }
            }
        }
    },
    "extra_sources": [
      "my_handlers.cpp"
    ]
}
```

We set the `encoder` for the signal to `ourRoundingWriteEncoder`, and we'll define that in a separate file named `my_handlers.cpp`. The `extra_sources` field is also set, meaning that our custom C/C++ code will be included with the firmware build.

In `my_handlers.cpp`:

```
/* Round the value down to 0 if it's less than 100 before writing to CAN. */
uint64_t ourRoundingWriteEncoder(CanSignal* signal, CanSignal* signals,
     int signalCount, double value, bool* send) {
   if(value < 100) {
      value = 0;
   }
   // encodeSignal pulls the CAN signal definition from the CanSignal struct
   // and encodes the value into the right bits of a 64-bit return value.
   return encodeSignal(signal, value);
}
```

Signal encoders are responsible for transforming a float, string or boolean value into a 64-bit integer, to be used in the outgoing message.

2.4.5 Composite Write Request

When the app developer sends a numeric measurement to the VI, we want to send:

- 1 arbitrary CAN message with the ID `0x34` on a high speed bus connected to controller 1, with the value `0x1234`.
- The value sent by the developer encoded into the message ID `0x35` in a signal starting at bit 0, 4 bits wide on the same high speed bus. We don't want this value to be writable by the app developer unless a part of these 3 writes combined.

- A boolean signal in the message `0x101` on a medium speed bus connected to controller 2, starting at bit 12 and 1 bit wide. If the numeric value from the user is greater than 100, the boolean value should be `true`.

```
{    "name": "passthrough",
     "buses": {
         "hs": {
             "controller": 1,
             "raw_writable": true,
             "speed": 500000
          },
          "ms": {
             "controller": 2,
             "speed": 125000
          }
     },
     "messages": {
         "0x35": {
             "bus": "hs",
             "signals": {
                 "My_Numeric_Signal": {
                     "generic_name": "my_number_signal",
                     "bit_position": 0,
                     "bit_size": 4
                 }
             }
         }
         "0x101": {
             "bus": "ms",
             "signals": {
                 "My_Other_Signal": {
                     "generic_name": "my_value_is_over_100_signal",
                     "bit_position": 12,
                     "bit_size": 1
                 }
             }
         }
     },
     "commands": [
        {"name": "my_command",
           "handler": "handleMyCommand"}
     ],
     "extra_sources": [
       "my_handlers.cpp"
     ]
}
```

We added a `commands` field, which contains an array of JSON objects with `name` and `handler` fields. The name of the command, `my_command` is what app developers will send to the VI. The `handler` is the name of a C++ function will define in one of the files listed in `extra_sources`.

In the configuration, also note that:

- The raw CAN message that we want to send isn't included. Since `raw_writable` is true for the `hs` bus, there's no need to define it in the configuration.
- The `my_number_signal` signal doesn't have the `writable` flag set to true (it's omitted, and the default is `false`). This means an app developer will not be able to send write requests for `my_number_signal` directly.

In `my_handlers.cpp`:

```
void handleMyCommand(const char* name, openxc_DynamicField* value,
      openxc_DynamicField* event, CanSignal* signals, int signalCount) {

   // Look up the numeric and boolean signals we need to send and abort if
   // either is missing
   CanSignal* numericSignal = lookupSignal("my_number_signal", signals,
         signalCount);
   CanSignal* booleanSignal = lookupSignal("my_value_is_over_100_signal",
         signals, signalCount);
   if(numericSignal == NULL || booleanSignal == NULL) {
      debug("Unable to find signals, can't send trio");
      return;
   }

   // Build and enqueue the arbitrary CAN message to be sent - note that none
   // of the CAN messages we enqueue in the handler will be sent until after
   // it returns - interaction with the car via CAN must be asynchronous.
   CanMessage message = {0x34, 0x12345};
   CanBus* bus = lookupBus(0, getCanBuses(), getCanBusCount());
   if(bus != NULL) {
     can::write::enqueueMessage(bus, &message);
   }

   // Send the numeric signal
   can::write::encodeAndSendSignal(numericSignal, value,
          // the last parameter is true, meaning we want to force sending
          // this signal even though it's not marked writable in the
          // config
          true);

   // For the boolean signal, send 'true' if the value sent by the user is
   // greater than 100
   can::write::encodeAndSendBooleanSignal(booleanSignal, value > 100, true);
}
```

2.5 Writable Raw CAN Configuration Examples

Writing raw CAN messages to the vehicle's bus is the lowest level you can get, so please make sure you know what you're doing before trying any of these examples. In addition to *reading raw CAN messages* you can write arbitrary CAN messages back to the bus.

2.5.1 Write a CAN messages

There's not much to configure for raw CAN writes, since your application is decided what message to build instead of the VI. The only requirement is that the bus you wish to write on is configured as writable with the `raw_writable` flag, otherwise the VI will reject the request to send.

```
{   "buses": {
        "hs": {
```

```
            "controller": 1,
            "speed": 500000,
             "raw_writable": true
        }
    }
}
```

With this configuration, you can send a CAN message write request to the VI using the OpenXC raw CAN message format (via any of the I/O interfaces) and it will send it out to the bus. For example, when using the JSON output format, sending this to the VI:

```
{"bus": 1, "id": 1234, "value": "0x12345678"}
```

would cause it to write a CAN message with the arbitration ID 1234 and the payload 0x12345678 to the bus attached to the first CAN controller.

With the tools from the OpenXC Python library you can send that from a terminal with the command:

```
openxc-control write --bus 1 --id 1234 ---data 0x12345678
```

and if you compiled the firmware with `DEBUG=1`, you can view the view the logs to make sure the write went through with this command:

```
openxc-control write --bus 1 --id 1234 ---data 0x12345678 --log-mode stderr
```

2.5.2 Writing with Filtered CAN

It's also possible to write arbitrary CAN messages with you are using *filtered raw CAN reads*. Your write requests aren't restricted to the filtered set, simply add the `raw_writable` flag and you can send any message.

```
{   "buses": {
        "hs": {
            "controller": 1,
            "speed": 500000,
            "raw_can_mode": "filtered",
             "raw_writable": true
        }
    },
    "messages": {
      "0x21": {
        "bus": "hs"
      }
    }
}
```

The rest is the same as in the unfiltered write example.

2.6 Diagnostic Configuration Examples

The firmware configuration examples shown so far are for so-called "normal mode" CAN messages, that are sent and received without any formal request by a node on the bus. Diagnostic type messages use a request / response style protocol, and are also supported by the VI firmware. You can add pre-defined, diagnostic recurring requests to a VI config file (e.g. request the engine RPM once per second).

To perform a one-time request, you don't need anything special in the configuration - just use a *diagnostic request command*.

2.6.1 Recurring Simple Diagnostic PID Request

We want to send a request for a mode 0x22 PID once per second and publish the full details of the response out as an OpenXC message.

```
{   "buses": {
        "hs": {
            "controller": 1,
            "raw_writable": true,
            "speed": 500000
        }
    },
    "diagnostic_messages": [
        {
            "bus": "hs",
            "id": 2015,
            "mode": 34,
            "pid": 42,
            "frequency": 1
        }
    ]
}
```

We added a `diagnostic_messages` array and one diagnostic request object in that array. The `id`, `mode`, and `frequency` are the only required fields, and we added a `pid` to this as well.

It's also important that the CAN controller is configured as writable with the `raw_writable` flag, otherwise the VI will not be able to send the diagnostic request.

With this configuration, the VI will publish the diagnostic response received from the bus using the OpenXC diagnostic response message format, e.g. when using the JSON output format:

```
{"bus": 1,
  "id": 2016,
  "mode": 34,
  "pid": 42,
  "success": true,
  "payload": "0x1234"}
```

Note that the `id` of the response is different from the request, because we sent the request to the functional broadcast address `0x7df`. If you use a physical address with your request, the ID of the response will match. The `bus` is also different from that specified in the mapping - it's the controller ID, not the name of the bus.

2.6.2 Recurring Parsed OBD-II PID Request

We want to send a request for the OBD-II PID for engine RPM once per second, and publish the full details of the response out as an OpenXC message. We want to use a decoder to parse the response's payload to get the actual engine RPM value.

```
{   "buses": {
        "hs": {
            "controller": 1,
            "raw_writable": true,
            "speed": 500000
        }
    },
    "diagnostic_messages": [
        {
            "bus": "hs",
            "id": 2015,
            "mode": 1,
            "pid": 12,
            "frequency": 1,
            "decoder": "handleObd2Pid"
        }
    ]
}
```

Besides changing the `mode` and `pid`, we added a `decoder`. The `handleObd2Pid` decoder is included by default in the vi-firmware repository, and knows how to decode a number of the most interesting and widely implemented OBD-II PIDs.

With this configuration, the VI will publish the diagnostic response received from the bus using the OpenXC diagnostic response message format, e.g. when using the JSON output format:

```
{"bus": 1,
  "id": 2016,
  "mode": 34,
  "pid": 42,
  "success": true,
  "payload": "0x1234"}
```

Unlike the configuration example without a `decoder`, this response has a `value` instead of the raw `payload`. The value is whatever your `decoder` function returns.

2.6.3 Recurring Named Diagnostic PID Request

Just like before, we want to request the OBD-II PID for engine RPM once per second, but this time we don't care about returning the full details in the response message. We just want a named message like the OpenXC "translated" message type.

```
{   "buses": {
        "hs": {
            "controller": 1,
            "raw_writable": true,
            "speed": 500000
        }
    },
    "diagnostic_messages": [
        {
            "bus": "hs",
            "id": 2015,
            "mode": 1,
            "pid": 12,
            "frequency": 1,
            "decoder": "handleObd2Pid",
```

```
            "name": "engine_speed"
        }
    ]
}
```

We simply added a `name` field to the diagnostic message configuration. This will change the output format to the OpenXC single-valued, translated message format, e.g. when using the JSON output format:

```
{"name": "engine_speed", "value": 45}
```

where `value` is the return value from the `decoder`.

2.6.4 On-Demand Request from Command Handler

You can generate a new recurring or one-off diagnostic request from any custom command handler signal decoder, or CAN message handler. Take a look at the `diagnostics.h` module for functions that may be useful.

For this example, we want to generate a mode 3 diagnostic request to get trouble codes when a "collect_trouble_codes" command is sent. We will register a callback function to handle the payload of the response to parse out the trouble code we are looking for. Here's our VI config:

```
{   "buses": {
        "hs": {
            "controller": 1,
            "raw_writable": true,
            "speed": 500000
        }
    },
    "commands": [
        {"name": "collect_trouble_codes",
         "handler": "collectTroubleCodes"}
    ],
    "extra_sources": [
      "my_handlers.cpp"
    ]
}
```

Just as in the *Composite Write Request*, we added a `commands` field with one custom command, mapping `collect_trouble_codes` to the command handler function `collectTroubleCodes` (to be defined in `my_handlers.cpp`).

In `my_handlers.cpp`:

```
void handleTroubleCodeResponse(
        DiagnosticsManager* manager,
        const ActiveDiagnosticRequest* request,
        const DiagnosticResponse* response,
        float parsed_payload) {
    // Received a response to our mode 3 request

    // response->payload is an array (with length response->payload_length)
    // that contains the trouble code data - do whatever you need to do to parse
    // out your trouble codes.

    // If you need to send anything out on the I/O interfaces (e.g. to let
    // a client know about a particular trouble code), you can use the
    // openxc::pipeline::publish(...) function.
}
```

```
void handleMyCommand(const char* name, openxc_DynamicField* value,
      openxc_DynamicField* event, CanSignal* signals, int signalCount) {

   // Build and broadcast a non-recurring mode 3 diagnostic request
   DiagnosticRequest request = {
       arbitration_id: 0x7df,
       mode: 3
   };

   addRequest(&getConfiguration()->diagnosticsManager,
      // use the CAN bus on controller 0 (this is a little bit dangerous,
      // you'll want to do some error checking to amke sure this bus exists.
      getCanBuses()[0],
      &request,
      NULL, // no human readable name
      false, // don't wait for multiple responses
      NULL, // no response decoder
      handleTroubleCodeResponse); // when a response is received, call our handler
}
```

This combination of a command handler and diagnostic response callback requests trouble codes from the vehicle whenever the command is received, and can take any action on the response (in the callback.

2.7 Bit Numbering

Because of different software tools and conventions in the industry, there are multiple ways to refer to bits within a CAN message. This doesn't change the actual data representation (like a different *byte* order would) but it changes how you refer to different bit positions for CAN signals.

The vehicle interface C++ source assumes the number of the highest order bit of a 64-bit CAN message is 0, and the numbering continuous left to right:

```
Hex:          0x83                    46
Binary:       10000011              01000110
Bit pos:    0 1 2 3 4 5 6 7   8 9 10 11 12 13 14 15 ...etc.
```

The tool used at Ford to document CAN messages (Vector DBC files) uses an "inverted" numbering by default. In each byte of a CAN message, they start counting bits from the *rightmost bit*, e.g.:

```
Hex:          0x83                    46
Binary:       10000011              01000110
Bit pos:    7 6 5 4 3 2 1 0   15 14 13 12 11 10 9 8 ...etc.
```

You can control the bit numbering with the `bit_numbering_inverted` flag on a message set or message (where the property will cascade down to all signals) or an individual signal in a VI configuration file. By default the VI assumes normal bit ordering for each signal **unless** you are using a database-backed mapping - the DBC files we've seen so far have all stored signal information in the inverted format.

2.8 All Configuration Options Reference

There are many configuration options - we recommend looking for your use case in the list of *read* or *write* to find a starting point, and refer to this section as a reference on particular configuration options.

The root level JSON object maps CAN bus addresses to CAN bus objects, CAN message IDs to CAN message objects in each bus, and CAN signal name to signal object within each message. Other top-level sections are available to list one-time initialization functions and to list arbitrary functions that should be added to the main loop.

In all cases when we refer to a "path," either an absolute or relative path will work. If you use relative paths, however, they must be relative to the root of wherever you run the build scripts.

Once you've defined your message set in a JSON file, run the `openxc-generate-firmware-code` tool from the OpenXC Python library to create an implementation of `signals.cpp`:

```
vi-firmware/ $ openxc-generate-firmware-code --message-set mycar.json > src/signals.cpp
```


2.8.1 Message Set

Each JSON mapping file defines a "message set," and it should have a name. Typically this identifies a particular model year vehicle, or possibly a broader vehicle platform. The `name` field is required.

`bit_numbering_inverted` (optional) This flag controls the default *bit numbering* for all messages included in this message set. You can override the bit numbering for any particular message or mapping, too.

`false` by default, `true` by default for database-backed mappings.

`max_message_frequency` Set a default value for all buses for this attribute - see the *Can Buses* section for a description.

`raw_can_mode` Set a default value for all buses for this attribute - see the *Can Buses* section for a description.

2.8.2 Parent Message Sets

Message sets are composable - you can extend a set by adding a path to the parent(s) to the `parents` key.

2.8.3 Initializers

The key `initializers` should have as its value an array of strings. Each string should be the name of a function with the type signature:

```
void function();
```

These functions will be called once at the beginning of execution, before reading any CAN messages.

2.8.4 Loopers

The key `loopers` should have as its value an array of strings. Each string should be the name of a function with the type signature:

```
void function();
```

These functions will be called once each time through the main loop function, after reading and processing any CAN messages.

2.8.5 CAN Buses

The key `buses` must be an object, where each field is a CAN bus uses by this message set, and which CAN controllers are attached on the microcontroller. The

`controller` The integer ID of the CAN controller to which this bus is attached. The platforms we are using now only have 2 CAN controllers, identified here by `1` and `2` - these are the only acceptable bus addresses. If this field is not defined, the bus and any messages associated with it will be ignored (but it won't cause an error, so you can swap between buses very quickly).

`speed` The CAN bus speed in Kbps, most often 125000 or 500000.

`raw_can_mode` Controls sending raw CAN messages (encoded as JSON objects) from the bus over the output channel. Valid modes are `off` (the default if you don't specify this attribute), `filtered` (if messages are defined for the bus, will enable CAN filters and only transmit those messages), or `unfiltered` (disable acceptance filters and send all received CAN messages). If this attribute is set on a CAN bus object, it will override any default set at the message set level (e.g. you can have all buses configured to send `filtered` raw CAN messages, but override one to send `unfiltered`).

`raw_writable` Controls whether or not raw CAN messages from the user can be written back to this bus, without any sort of translation. This is false by default. Even when this is false, messages may still be written to the bus if a signal is configured as `writable`, but they will translated from the user's input first.

`max_message_frequency` The default maximum frequency for all CAN messages when using the raw passthrough mode. To put no limit on the frequency, set this to 0 or leave it out. If this attribute is set on a CAN bus object, it will override any default set at the message set level. This value cascades to all CAN message objects for their `max_frequency` attribute, which can also be overridden at the message level.

`force_send_changed` (optional) Meant to be used in conjunction with `max_message_frequency`, if this is true a raw CAN message will be sent regardless of the given frequency if the value has changed (when using raw CAN passthrough). Setting the value here, on the CAN bus object, will cascade down to all CAN messages unless overridden. Defaults to `true`.

2.8.6 CAN Messages

The `messages` key is a object with fields mapping from CAN message IDs to signal definitions. The fields must be hex IDs of CAN messages as strings (e.g. `0x90`).

Message

The attributes of each message object are:

`bus` The name of one of the previously defined CAN buses where this message can be found.

`bit_numbering_inverted` (optional) This flag controls the default *bit numbering* for the signals in this message. Defaults to the value of the mapping, then default of the message set.

signals A list of CAN signal objects (described in the *Signal* section) that are in this message, with the name of the signal as the key. If this is a database-backed mappping, this value must match the signal name in the database exactly - otherwise, it's an arbitrary name.

name (optional) The name of the CAN message - this is not required and has no meaning in code, it can just be handy to be able to refer back to an original CAN message definition in another document.

handlers (optional) An array of names of functions that will be compiled with the firmware and should be applied to the entire raw message value (see *Message Handlers*).

enabled (optional) Enable or disable all processing of a CAN message. By default, a message is enabled. If this flag is false, the CAN message and all its signals will be left out of the generated source code. Defaults to `true`.

max_frequency (optional) If sending raw CAN messages to the output interfaces, this controls the maximum frequency (in Hz) that the message will be process and let through. The default value (`0`) means that all messages will be processed, and there is no limit imposed by the firmware. If you want to make sure you don't miss a change in value even when rate limiting, see the `force_send_changed` attribute. Defaults to 0 (no limit).

max_signal_frequency (optional) Setting the max signal frequency at the message level will cascade down to all of the signals within the message (unless overridden). The default value (`0`) means that all signals will be processed, and there is no limit imposed by the firmware. See the `max_frequency` flag documentation for the signal mapping for more information. If you want to make sure you don't miss a change in value even when rate limiting, see the `force_send_changed_signals` attribute. Defaults to 0 (no limit).

force_send_changed (optional) Meant to be used in conjunction with `max_frequency`, if this is true a raw CAN message will be sent regardless of the given frequency if the value has changed (when using raw CAN passthrough). Defaults to `true`.

force_send_changed_signals Setting this value on a message will cascade down to all of the signals within the message (unless overridden). See the `force_send_changed` flag documentation for the signal mapping for more information. Defaults to `false`.

Message Handlers

If you need additional control, you can provide custom handlers for the entire message to combine multiple signals into a single value (or any other arbitrary processing). You can generate 0, 1 or many translated messages from each call to a custom handler function.

```
void handleSteeringWheelMessage(int messageId, uint64_t data,
        CanSignal* signals, int signalCount, Pipeline* pipeline);
    float steeringWheelAngle = decodeCanSignal(&signals[1], data);
    float steeringWheelSign = decodeCanSignal(&signals[2], data);

    float finalValue = steeringWheelAngle;
    if(steeringWheelSign == 0) {
        // left turn
        finalValue *= -1;
    }

    char* message = generateJson(signals[1], finalValue);
    sendMessage(usbDevice, (uint64_t*) message, strlen(message));
}
```

Using a custom message handler will not automatically stop the normal translation workflow for individual signals. To mute them (but still store their values in `signal->lastvalue`), specify `ignoreHandler` as the `handler`. This is not done by default because not every signal in a message is always handled by a message handler.

Signal

The attributes of a `signal` object within a `message` are:

`generic_name` The name of the associated generic signal name (from the OpenXC specification) that this should be translated to. Optional - if not specified, the signal is read and stored in memory, but not sent to the output bus. This is handy for combining the value of multiple signals into a composite measurement such as steering wheel angle with its sign.

`bit_position` The starting bit position of this signal within the message. Required unless this is a database-backed mapping.

`bit_size` The width in bits of the signal. Required unless this is a database-backed mapping.

`factor` The signal value is multiplied by this if set. Required unless this is a database-backed mapping.

`offset` This is added to the signal value if set. Required unless this is a database-backed mapping.

`decoder` (optional) The name of a function that will be compiled with the firmware and should be applied to the signal's value after the normal translation. See the *Signal Decoder* section for details.

`ignore` (optional) Setting this to `true` on a signal will silence output of the signal. The VI will not monitor the signal nor store any of its values. This is useful if you are using a custom decoder for an entire message, want to silence the normal output of the signals it handles, *and* you don't need the VI to keep track of the values of any of the signals separately (in the `lastValue` field). If you need to use the previously stored values of any of the signals, you can use the `ignoreDecoder` as the decoder for the signal. Defaults to `false`.

`enabled` (optional) Enable or disable all processing of a CAN signal. By default, a signal is enabled; if this flag is false, the signal will be left out of the generated source code. Defaults to `true`.

The difference between `ignore`, `enabled` and using an `ignoreDecoder` can be confusing. To summarize the difference:

- The `enabled` flag is the master control switch for a signal - when this is false, the signal (or message, or mapping) will not be included in the firmware at all. A common time to use this is if you want to have one configuration file with many options, only a few of which are enabled in any particular build.
- The `ignore` flag will not exclude a signal from the firmware, but it will not include it in the normal message processing pipeline. The most common use case is when you need to reference the bit field information for the signal from a custom decoder.
- Finally, use the `ignoreDecoder` for your signal's `decoder` to both include it in the firmware and handle it during the normal message processing pipeline, but just silence its output. This is useful if you need to track the last known value for this signal for a calculation in a custom decoder.

`states` This is a mapping between the desired descriptive states (e.g. `off`) and the corresponding numerical values from the CAN message (usually an integer). The raw values are specified as a list to accommodate multiple raw states being coalesced into a single final state (e.g. key off and key removed both mapping to just "off"). Required unless this is a database-backed mapping.

`max_frequency` (optional) Some CAN signals are sent at a very high frequency, likely more often than will ever be useful to an application. This attribute sets the maximum frequency (Hz) that the signal will be processed and let through. The default value (`0`) means that all values will be processed, and there is no limit imposed by the firmware. If you want to make sure you don't miss a change in value even when dropping messages, see the `force_send_changed` attribute. You probably don't want to combine this attribute with `send_same` or else you risk missing a status change message if wasn't one of the messages the VI decided to let through. Defauls to 0 (no limit).

`send_same` (optional) By default, all signals are translated every time they are received from the CAN bus. By setting this to `false`, you can force a signal to be sent only if the value has actually changed. This works best with boolean and state based signals. Defaults to `true`.

force_send_changed (optional) Meant to be used in conjunction with `max_frequency`, if this is true a signal will be sent regardless of the given frequency if the value has changed. This is useful for state-based and boolean states, where the state change is the most important thing and you don't want that message to be dropped. Defaults to `false`.

writable (optional) Set this attribute to `true` to allow this signal to be written back to the CAN bus by an application. OpenXC JSON-formatted messages sent back to the VI that are writable are translated back into raw CAN messages and written to the bus. By default, the value will be interpreted as a floating point number. Defaults to `false`.

encoder (optional) If the signal is writable and is not a plain floating point number (i.e. it is a boolean or state value), you can specify a custom function here to encode the value for a CAN messages. This is only necessary for boolean types at the moment - if your signal has states defined, we assume you need to encode a string state value back to its original numerical value. Defaults to a built-in numerical encoder.

Signal Decoder

The default decoder for each signal is a simple passthrough, translating the signal's value from engineering units to something more usable (using the defined factor and offset). Some signals require additional processing that you may wish to do within the VI and not on the host device. Other signals may need to be combined to make a composite signal that's more meaningful to developers.

An good example is steering wheel angle. For an app developer to get a value that ranges from e.g. -350 to +350, we need to combine two different signals - the angle and the sign. If you want to make this combination happen inside the VI, you can use a custom decoder.

You may also need a custom decoder to return a value of a type other than float. A decoder is provided for dealing with boolean values, the `booleanDecoder` - if you specify that as your signal's `decoder` the resulting JSON will contain `true` for 1.0 and `false` for 0.0. There is also a `stateDecoder` for translating integer state values to string names.

For this example, we want to modify the value of `steering_wheel_angle` by setting the sign positive or negative based on the value of the other signal (`steering_angle_sign`). Every time a CAN signal is received, the new value is stored in memory. Our custom decoder `decodeSteeringWheelAngle` will use that to adjust the raw steering wheel angle value. Modify the input JSON file to set the `decoder` attribute for the steering wheel angle signal to `decodeSteeringWheelAngle`.

Add this to the top of `signals.cpp` (or if using the mapping file, add it to a separate `.cpp` file and then add that filename to the `extra_sources` field):

```
openxc_DynamicField decodeSteeringWheelAngle(CanSignal* signal,
        CanSignal* signals, int signalCount,
        openxc::pipeline::Pipeline* pipeline,
        float value, bool* send) {
    if(signal->lastValue == 0) {
        // left turn
        value *= -1;
    }
    return openxc::payload::wrapNumber(value);
}
```

The function declaration of a custom decoder must match:

```
openxc_DynamicField customDecoder(CanSignal* signal, CanSignal* signals,
    int signalCount, openxc::pipeline::Pipeline* pipeline,
        float value, bool* send);
```

where `signal` is a pointer to the `CanSignal` this is handling, `signals` is an array of all signals, `value` is the raw value from CAN and `send` is a flag to indicate if this should be sent over USB.

The `bool* send` parameter is a pointer to a `bool` you can flip to `false` if this signal value need not be sent over USB. This can be useful if you don't want to keep notifying the same status over and over again, but only in the event of a change in value (you can use the `lastValue` field on the CanSignal object to determine if this is true). It's also good practice to inspect the value of `send` when your custom decoder is called - the normal translation workflow may have decided the data shouldn't be sent (e.g. the value hasn't changed and `sendSame == false`). Decoders are called every time a signal is received, even if `send == false`, so that you have the flexibility to implement custom processing that depends on receiving every data point.

A known issue with this method is that there is no guarantee that the last value of another signal arrived in the message or before/after the value you're current modifying. For steering wheel angle, that's probably OK - for other signals, not so much.

2.8.7 Diagnostic Messages

The `diagnostic_messages` key is an array of objects describing a recurring diagnostic message request.

Diagnostic Message

The attributes of each diagnostic message object are:

`bus` The name of one of the previously defined CAN buses where this message should be requested.

`id` the arbitration ID for the request.

`mode` The diagnostic request mode, e.g. Mode 1 for powertrain diagnostic requests.

`frequency` The frequency in Hz to request this diagnostic message. The maximum allowed frequency is 10Hz.

`pid` (optional) If the mode uses PIDs, the pid to request.

`name` (optional) A human readable, string name for this request. If provided, the response will have a `name` field (much like a normal translated message) with this value in place of `bus`, `id`, `mode` and `pid`.

`decoder` (optional) When using a `name`, you can also specify a custom decoder function to parse the payload. This field is the name of a function (that matches the `DiagnosticResponseDecoder` function prototype). When a decoder is specified, the decoded value will be returned in the `value` field in place of `payload`.

`callback` (optional) This field is the name of a function (that matches the `DiagnosticResponseCallback` function prototype) that should be called every time a response is received to this request.

2.8.8 Mappings

The `mappings` field is an optional field allows you to move the definitions from the `messages` list to separate files for improved composability and readability.

For an detailed explanation of mapped message sets, see the example of a message set using mappings, see the *Separate Files for Message Sets* configuration example.

The `mappings` field must be a list of JSON objects with:

`mapping` - A path to a JSON file containing a single object with the key `messages`, containing objects formatted as the *CAN Messages* section describes. In short, you can pull out the `messages` key from the main file and throw it into a separate file and link it in here. You can also do the same with a `diagnostic_messages` field containing *Diagnostic Messages*.

`bus` (optional) The name of one of the defined CAN buses where these messages can be found - this value will be set for all of the messages contained the mapping file, but can be overridden by setting `bus` again in an individual message.

database (optional) A path to a CAN message database associated with these mappings. Right now, XML exported from Vector CANdb++ is supported. If this is defined, you can leave the bit position, bit size, factor, offset, max and min values out of the `mapping` file - they will be picked up automatically from the database.

bit_numbering_inverted (optional) This flag controls the default *bit numbering* for the messages contained in this mapping. Messages in the mapping can override the bit numbering by explicitly specifying their own value for this flag. Defaults to the value of the message set, or `true` if this mapping is database-backed.

enabled (optional) Enable or disable all processing of the CAN messages in a mapping. By default, a mapping is enabled; if this flag is false, all CAN message and signals from the mapping will be excluded from the generated source code. Defaults to `true`.

2.8.9 Extra Sources

The `extra_sources` key is an optional list of C++ source files that should be injected into the generated `signals.cpp` file. These may include signal decoders, message handlers, initializers or custom loopers.

2.8.10 Commands

The `commands` field is a mapping of arbitrary command names to functions that should be called to run arbitrary code in the VI on-demand (e.g. sending multiple CAN signals at once). The value of this attribute is a list of objects with these attributes:

name The name of the command to be recognized on the OpenXC translated interface.

enabled (optional) Enable or disable all processing of a command. By default, a command is enabled. If this flag is false, the command will be excluded from the generated source code. Defaults to `true`.

handler The name of a custom command handler function (that matches the `CommandHandler` function prototype from `canutil.h`) that should be called when the named command arrives over the translated VI interface (e.g. USB or Bluetooth).

```
void (*CommandHandler)(const char* name, openxc_DynamicField* value,
        openxc_DynamicField* event, CanSignal* signals, int signalCount);
```

Any message received from the USB host with that given command name will be passed to your handler. This is useful for situations where there isn't a 1 to 1 mapping between OpenXC command and CAN signal, e.g. if the left and right turn signal are split into two signals instead of the 1 state-based signal used by OpenXC. You can use the `sendCanSignal` function in `canwrite.h` to do the actual data sending on the CAN bus.

2.9 FAQ

While building custom firmware, if you find yourself thinking:

- "I need to run some code in the main loop, regardless of any CAN signals." - use a *Looper Function*.
- "I need to run initialization code at startup." User an *Initializer Function*.
- "I need to do some extra processing on a CAN signal before sending it out." Use a *custom signal decoder*.
- "I need to combine the value of multiple signals in a message to generate a value." Use a *message handler*.
- "I need to send less data." Control the *send frequency*.
- "I don't want this signal to send unless it changes." Configure it to *Send Signal on Change Only*.

CHAPTER 3

Compiling

For a detailed walkthrough, see *Getting Started*.

The build process uses GNU Make and works with Linux (tested in Arch Linux and Ubuntu), OS X and Cygwin 32-bit in Windows. For documentation on how to build for each platform, see the *supported platform details*.

When compiling you need to specify which *VI platform* you are compiling for with the `PLATFORM` flag. All other flags are optional.

3.1 Example Build Configurations

There are many *Makefile Options*, so it may be difficult to tell which you need to configure to get a working build. This page collects a few examples of popular configurations.

3.1.1 Default Build

By default, if you just run `make` with no environment variables set, the firmware is built with these options:

```
$ make
<...snip lots of output...>
Compiled with options:
-       FORDBOARD = PLATFORM
-       1         = BOOTLOADER
-       0         = DEBUG
-       0         = DEFAULT_METRICS_STATUS
-       1         = DEFAULT_ALLOW_RAW_WRITE_USB
-       0         = DEFAULT_ALLOW_RAW_WRITE_UART
-       0         = DEFAULT_ALLOW_RAW_WRITE_NETWORK
-       0         = DEFAULT_UART_LOGGING_STATUS
-       JSON      = DEFAULT_OUTPUT_FORMAT
-       0         = DEFAULT_EMULATED_DATA_STATUS
-       SILENT_CAN= DEFAULT_POWER_MANAGEMENT
-       0x1       = DEFAULT_USB_PRODUCT_ID
-       0         = DEFAULT_CAN_ACK_STATUS
```

```
-      1          = DEFAULT_OBD2_BUS
-      0          = DEFAULT_RECURRING_OBD2_REQUESTS_STATUS
```

The Makefile will always print the configuration used so you can double check.

- This default configuration will run on a Ford reference VI <http://vi.openxcplatform.com/> (`PLATFORM` is `FORDBOARD`) running the pre-loaded bootloader (`BOOTLOADER` is `1`).
- Debug mode is off (`DEBUG` is `0`) so no log messages will be output via USB for maximum performance.
- If the VI configuration *allows raw CAN writes*, they will only be permitted if set via USB (`DEFAULT_ALLOW_RAW_WRITE_USB` is `1` but the `*_UART` and `*_NETWORK` versions are `0`.
- The data sent from the VI will be serialized to JSON in the format defined by the OpenXC message format <https://github.com/openxc/openxc-message-format>.
- The VI will go into sleep mode only when no CAN bus activity is detected for a few seconds (the `DEFAULT_POWER_MANAGEMENT` mode is `SILENT_CAN`).
- The CAN controllers will be initialized as listen only unless the VI configuration explicitly states they are writable (`DEFAULT_CAN_ACK_STATUS` is `1`). This means that the VI may not work in a bench testing setup where nothing else on the bus is ACKing.

3.1.2 Automatic Recurring OBD-II Requests Build

Another common build is one that automatically queries the vehicle to check if it supports a pre-defined set (see the file `obd2.cpp`) of interesting OBD-II parameters, and if so, sets up recurring requests for them. Compile with these options:

```
$ export DEFAULT_RECURRING_OBD2_REQUESTS_STATUS=1
$ export DEFAULT_POWER_MANAGEMENT=OBD2_IGNITION_CHECK
$ make
<...snip lots of output...>

Compiled with options:
-      FORDBOARD  = PLATFORM
-      1          = BOOTLOADER
-      0          = DEBUG
-      0          = DEFAULT_METRICS_STATUS
-      1          = DEFAULT_ALLOW_RAW_WRITE_USB
-      0          = DEFAULT_ALLOW_RAW_WRITE_UART
-      0          = DEFAULT_ALLOW_RAW_WRITE_NETWORK
-      0          = DEFAULT_UART_LOGGING_STATUS
-      JSON       = DEFAULT_OUTPUT_FORMAT
-      0          = DEFAULT_EMULATED_DATA_STATUS
-      OBD2_IGNIT= DEFAULT_POWER_MANAGEMENT
-      0x1        = DEFAULT_USB_PRODUCT_ID
-      0          = DEFAULT_CAN_ACK_STATUS
-      1          = DEFAULT_OBD2_BUS
-      1          = DEFAULT_RECURRING_OBD2_REQUESTS_STATUS
```

Notice we changed:

- `DEFAULT_RECURRING_OBD2_REQUESTS_STATUS` to `1`. This enables the automatic OBD-II queries.
- `DEFAULT_POWER_MANAGEMENT` to `OBD2_IGNITION_CHECK` (the Makefile summary display truncates this value). This changes the power management mode to actively probe the vehicle for the engine and vehicle speed. Some vehicles will keep modules alive if anyone is making diagnostic requests (e.g. the VI), and we want to avoid that because it could drain the car's battery. This mode actively infers if the ignition is on and

stops sending diagnostic queries if we think the car is off. The combination of an engine and vehicle speed check should be compatible with hybrid vehicles.

3.1.3 Emulated Data Build

If you want to test connectivity to a VI from your client device without going to a vehicle, but you don't care about the actual vehicle data being generated, you can compile a build that generates random vehicle data and sends it via the normal I/O interfaces.

If you are building an app, you'll want to use a trace file <http://openxcplatform.com/resources/traces.html> or the vehicle simulator <https://github.com/openxc/openxc-vehicle-simulator>.

The config a VI to emulate a vehicle:

```
$ export DEFAULT_EMULATED_DATA_STATUS=1
$ export DEFAULT_POWER_MANAGEMENT=ALWAYS_ON
$ make
<...snip lots of output...>

Compiled with options:
-       FORDBOARD = PLATFORM
-       1         = BOOTLOADER
-       0         = DEBUG
-       0         = DEFAULT_METRICS_STATUS
-       1         = DEFAULT_ALLOW_RAW_WRITE_USB
-       0         = DEFAULT_ALLOW_RAW_WRITE_UART
-       0         = DEFAULT_ALLOW_RAW_WRITE_NETWORK
-       0         = DEFAULT_UART_LOGGING_STATUS
-       JSON      = DEFAULT_OUTPUT_FORMAT
-       0         = DEFAULT_EMULATED_DATA_STATUS
-       OBD2_IGNIT= DEFAULT_POWER_MANAGEMENT
-       0x1       = DEFAULT_USB_PRODUCT_ID
-       0         = DEFAULT_CAN_ACK_STATUS
-       1         = DEFAULT_OBD2_BUS
-       1         = DEFAULT_RECURRING_OBD2_REQUESTS_STATUS
```

There are 2 changes from the default build:

- `DEFAULT_EMULATED_DATA_STATUS` is `1`, which will cause fake data to be generated and published from the VI.
- `DEFAULT_POWER_MANAGEMENT` is `ALWAYS_ON`, so the VI will not go to sleep while plugged in. Make sure to clear this configuration option before making a build to run in a vehicle, or you could drain the battery!

3.2 Makefile Options

These options are passed as shell environment variables to the Makefile, e.g.

```
$ PLATFORM=FORDBOARD make
```

Note: Try adding the `-j4` flag to your calls to `make` to build 4 jobs in parallel - the speedup can be quite dramatic.

`PLATFORM` Select the target *microcontroller platform*.

Values: `FORDBOARD, CHIPKIT, CROSSCHASM_C5, BLUEBOARD`

Default: `CHIPKIT`

DEBUG Set to `1` to compile with debugging symbols and to enable debug logging. By default the logging will be available via the logging USB endpoint - for UART output, see the `UART_LOGGING` flag. This also implies `DEFAULT_POWER_MANAGEMNET=ALWAYS_ON` and `DEFAULT_CAN_ACK_STATUS=1`.

Values: `0` or `1`

Default: `0`

BOOTLOADER By default, the firmware is built to run on a microcontroller with a bootloader (if one is available for the selected platform), allowing you to update the firmware without specialized hardware. If you want to build to run on bare-metal hardware (i.e. start at the top of flash memory) set this to `0`.

Values: `0` or `1`

Default: `1`

TRANSMITTER Set this to `1` to imply `DEFAULT_POWER_MANAGEMENT=ALWAYS_ON` and `DEFAULT_USB_PRODUCT_ID=0x2`. This is useful if you are using the VI as a transmitter in a local CAN bus for bench testing. You can address it separately from a receiving VI because of the different USB product ID.

Values: `0` or `1`

Default: `0`

DEFAULT_UART_LOGGING_STATUS When combined with `DEBUG`, set to `1` to enable debug logging via UART. See the *platform docs* for details on how to read this output.

Values: `0` or `1`

Default: `0`

DEFAULT_METRICS_STATUS Set to `1` to enable logging CAN message and output message statistics over the normal DEBUG output.

Values: `0` or `1`

Default: `0`

DEFAULT_CAN_ACK_STATUS If 1, the VI will be an active CAN bus participant and send low-level ACKs. If the bus speed is incorrect, can interfere with normal bus operation. This is useful if you are bench testing with 2 VIs and you need the CAN messages to be propagated up the stack.

If 0, the VI will be a listen only node and will not ACK messages. An incorrect bus speed will not have a negative impact on the bus, but you still won't be able to read anything.

See the *testing section* for more details.

Values: `0` or `1`

Default: `0`

DEFAULT_ALLOW_RAW_WRITE_NETWORK By default, raw CAN message write requests are not allowed from the network interface even if the CAN bus is configured to allow raw writes - set this to `1` to accept them.

Values: `0` or `1`

Default: `0`

DEFAULT_ALLOW_RAW_WRITE_UART By default, raw CAN message write requests are not allowed from the Bluetooth interface even if the CAN bus is configured to allow raw writes - set this to `1` to accept them.

Values: `0` or `1`

Default: `0`

DEFAULT_ALLOW_RAW_WRITE_USB By default, raw CAN message write requests *are* allowed from the wired USB interface (if the CAN bus is also configured to allow raw writes) - set this to `0` to block them.

Values: `0` or `1`

Default: `1`

DEFAULT_OUTPUT_FORMAT By default, the output format is `JSON`. Set this to `PROTOBUF` to use a binary output format, described more in *Binary Output Format*.

Values: `JSON`, `PROTOBUF`

Default: `JSON`

DEFAULT_RECURRING_OBD2_REQUESTS_STATUS Set this to `1` to include a set of recurring OBD-II requests in the build, to be requests immediately on startup.

Values: `0` or `1`

Default: `0`

DEFAULT_POWER_MANAGEMENT Valid options are `ALWAYS_ON`, `SILENT_CAN` and `OBD2_IGNITION_CHECK`.

Values: `ALWAYS_ON`, `SILENT_CAN`, `OBD2_IGNITION_CHECK` (will cause the VI to write messages to the bus)

Default: `SILENT_CAN`

DEFAULT_USB_PRODUCT_ID Change the default USB product ID for the device. This is useful if you want to address 2 VIs connected to the same computer.

Values: `0x0` to `0xffff`

Default: `0x1`

DEFAULT_EMULATED_DATA_STATUS Set this to `1` to have the VI generate random data and publish it as OpenXC vehicle messages.

Values: `0` or `1`

Default: `0`

DEFAULT_OBD2_BUS Sets the default CAN controller to use for sending OBD-II requests. Valid options are `0` (don't send any OBD-II requests), `1` or `2`. The default value is `1`.

Values: `0` (off), `1` or `2`

Default: `1`

NETWORK By default, TCP output of OpenXC vehicle data is disabled. Set this to `1` to enable TCP output on boards that have an Network interface. Note that the NETWORK option is broken on the chipKIT Max32 build for the moment, see https://github.com/openxc/vi-firmware/issues/189.

Values: `0` or `1`

Default: `0`

3.3 Troubleshooting

If the compilation didn't work:

- Make sure the submodules are up to date - run `git submodule update --init` and then `git status` and make sure there are no modified files in the working directory.

- Did you download the .zip file of the `vi-firmware` project from GitHub? Use git to clone the repository instead - the library dependencies are stored as git submodules and do not work when using the zip file.
- If you get a lot of errors about `undefined reference to getSignals()'` and other functions, you need to make sure you defined your CAN messages - read through *Firmware Configuration* before trying to compile.

3.4 Dependencies

If the `bootstrap.sh` script didn't work, see below for more information on the dependencies.

To compile the VI firmware, you need:

- Git
- `vi-firmware` *source code* cloned with Git - not from a .zip file
- OpenXC Python library
- *MPIDE*
- Digilent's USB and CAN *libraries for the chipKIT*
- *FTDI driver*
- Mini-USB cable

If instead of the chipKIT, you are compiling for the Blueboard (based on the NXP LPC1768/69), instead of MPIDE you will need:

- *GCC for ARM* toolchain
- *OpenOCD*
- JTAG programmer compatible with `openocd` - we've tested the Olimex ARM-OCD-USB programmer.

The easiest way to install these dependencies is to use the script/bootstrap.sh script in the `vi-firmware` repository. Run the script in Linux, 32-bit Cygwin in Windows or OS X and if there are no errors you should be ready to go:

```
$ script/bootstrap.sh
```

If there are errors, continue reading in this section to install whatever piece failed manually.

3.4.1 Source Code

Clone the repository from GitHub:

```
$ git clone https://github.com/openxc/vi-firmware
```

Some of the library dependencies are included in this repository as git submodules, so before you go further run:

```
$ git submodule update --init
```

If this doesn't print out anything or gives you an error, make sure you cloned this repository from GitHub with git and that you didn't download a zip file. The zip file is missing all of the git metadata, so submodules will not work.

3.4.2 PIC32 Dependencies

If you are planning to build for the PIC32 platforms (e.g. the chipKIT Max32 or C5), you need these dependencies.

3.4.3 MPIDE

Building the source for the VI for the chipKIT microcontroller requires MPIDE (the development environment and compiler toolchain for chipKIT provided by Digilent). Installing MPIDE can be a bit quirky on some platforms, so if you're having trouble take a look at the installation guide for MPIDE.

Although we just installed MPIDE, building via the GUI is **not supported**. We use GNU Make to compile and upload code to the device. You still need to download and install MPIDE, as it contains the PIC32 compiler.

You need to set an environment variable (e.g. in `$HOME/.bashrc`) to let the project know where you installed MPIDE (make sure to change these defaults if your system is different!):

```
# Path to the extracted MPIDE folder (this is correct for OS X)
export MPIDE_DIR=/Applications/Mpide.app/Contents/Resources/Java
```

Remember that if you use `export`, the environment variables are only set in the terminal that you run the commands. If you want them active in all terminals (and you probably do), you need to add these `export ...` lines to the file `~/.bashrc` (in Linux) or `~/.bash_profile` (in OS X) and start a new terminal.

3.4.4 Digilent / Microchip Libraries

It also requires some libraries from Microchip that we are unfortunately unable to include or link to as a submodule from the source because of licensing issues:

- Microchip USB device library (download DSD-0000318 from the bottom of the Network Shield page)
- Microchip CAN library (included in the same DSD-0000318 package as the USB device library)

You can read and accept Microchip's license and download both libraries on the Digilent download page.

Once you've downloaded the .zip file, extract it into the `libs` directory in this project. It should look like this:

```
- /Users/me/projects/vi-firmware/
---- libs/
-------- chipKITUSBDevice/
         chipKitCAN/
        ... other libraries
```

3.4.5 FTDI Driver

If you're using Mac OS X or Windows, make sure to install the FTDI driver that comes with the MPIDE download. The chipKIT uses a different FTDI chip than the Arduino, so even if you've used the Arduino before, you still need to install this driver.

3.4.6 LPC176x Dependencies

If you are planning to build for the LPC176x platforms (e.g. the Ford Reference VI), you need these dependencies.

3.4.7 GCC for ARM Toolchain

Download the binary version of the toolchain for your platform (Linux, OS X or Windows) from this Launchpad site.

Arch Linux

In Arch Linux you can install the `arm-none-eabi-gcc` from the `[community]` repository.

3.4.8 OpenOCD

If you plan to flash an LPC17xx based board with JTAG and not use the USB bootloader included with the Ford Reference VI, you need OpenOCD.

Arch Linux

```
$ pacman -S openocd
```

OS X

Install Homebrew. Then:

```
$ brew install libftdi libusb
$ brew install --enable-ft2232_libftdi openocd
```

Remove the Olimex sections from the FTDI kernel module, and then reload it:

```
$ sudo sed -i "" -e "/Olimex OpenOCD JTAG A/{N;N;N;N;N;N;N;N;N;N;N;N;N;N;N;d;}" /System/Library/Ext
$ sudo kextunload /System/Library/Extensions/FTDIUSBSerialDriver.kext/
$ sudo kextload /System/Library/Extensions/FTDIUSBSerialDriver.kext/
```

CHAPTER 4

Supported Platforms

For information on how to add support for another platform, see the *board support* section.

4.1 Ford Reference Vehicle interface

The Ford Reference VI is an open source hardware implementation of a VI, and the complete documentation is available at vi.openxcplatform.com.

To build for the Ford reference VI, compile with the flag `PLATFORM=FORDBOARD`.

4.1.1 Flashing a Pre-compiled Firmware

Pre-compiled binaries (built with the `BOOTLOADER` flag enabled, see all *compiler flags*) are compatible with the OpenLPC USB bootloader - follow the instructions for Flashing User Code to update the vehicle interface.

4.1.2 Compiling

USB Bootloader

If you are running a supported bootloader, you don't need any special programming hardware. Compile the firmware to run under the bootloader:

```
$ export PLATFORM=FORDBOARD
$ make clean
$ make -j4
```

The compiled firmware will be located at `build/lpc17xx/vi-firmware-lpc17xx.bin`. See reference VI programming instructions to find out how to re-flash the VI.

Bare Metal

Attach a JTAG adapter to your computer and the VI, then compile and flash:

```
$ export PLATFORM=FORDBOARD
$ export BOOTLOADER=0
$ make clean
$ make -j4
$ make flash
```

The config files in this repository assume your JTAG adapter is the Olimex ARM-USB-OCD unit. If you have a different unit, modify the `src/lpc17xx/lpc17xx.mk` Makefile to load your programmer's OpenOCD configuration.

4.1.3 UART

The software configuration is identical to the *Blueboard*. The reference VI includes an RN-41 on the PCB attached to the RX, TX, CTS and RTS pins, in addition to the UART status pin.

When a Bluetooth host pairs with the RN-42 and opens an RFCOMM connection, pin 0.18 will be pulled high and the VI will being streaming vehicle data over UART.

4.1.4 Debug Logging

In most cases the logging provided via USB is sufficient, but if you are doing low-level development and need the simpler UART interface, you can enable it with the `UART_LOGGING` Makefile flag.

Logging will be on UART0, which is exposed on the bottom of the board at J3, a 5-pin ISP connector.

4.1.5 LED Lights

The reference VI has 2 RGB LEDs. If the LEDs are a dim green and red, then the firmware was not flashed properly and the board is not running.

LED A

- CAN activity detected - Blue
- No CAN activity on either bus - Orange

LED B

- USB connected, Bluetooth not connected - Green
- Bluetooth connected, USB in either state - Blue
- Neither USB or Bluetooth connected - Off

4.1.6 Bootloader

The OpenLPC USB bootloader is tested and working, and enables the LPC17xx to appear as a USB drive. See the documentation in that repository for instructions on how to flash the bootloader (a JTAG programmer is required). The reference VI from Ford is pre-programmed with this bootloader.

4.2 NGX Blueboard LPC1768-H

To build for the Blueboard, compile with the flag `PLATFORM=BLUEBOARD`.

4.2.1 UART

On the LPC17xx, `UART1` is used for OpenXC output at the 230000 baud rate. Like on the chipKIT, hardware flow control (RTS/CTS) is enabled, so CTS must be pulled low by the receiving device before data will be sent.

- Pin 2.0 - `UART1 TX`, connect this to the RX line of the receiver.

- Pin 2.1 - `UART1 RX`, connect this to the TX line of the receiver.
- Pin 2.2 - `UART1 CTS`, connect this to the RTS line of the receiver.
- Pin 2.7 - `UART1 RTS`, connect this to the CTS line of the receiver.

UART data is sent only if pin 0.18 is pulled high. If you are using a Bluetooth module like the BlueSMiRF from SparkFun, you need to hard-wire 5v into pin 0.18 to actually enabling UART. Other hardware implementations (like the *Ford Reference VI*) may be able to hook the Bluetooth connection status to this pin instead, to make the status of UART more dynamic.

4.2.2 UART Debug Logging

In most cases the logging provided via USB is sufficient, but if you are doing low-level development and need the simpler UART interface, you can enable it with the `UART_LOGGING` Makefile flag.

On the Blueboard LPC1768H, logging will be on `UART0` at `115200` baud:

- Pin 0.2 - `UART0 TX`, connect this to the RX line of the receiver
- Pin 0.3 - `UART0 RX`, connect this to the TX line of the receiver

4.2.3 LED Lights

LEDs are not currently supported on the Blueboard.

4.3 Digilent chipKIT Max32

To build for the chipKIT-based Vehicle Interface, compile with the flag `PLATFORM=CHIPKIT`. The chipKIT is also the default platform, so the flag is optional.

The chipKIT VI supports up to 2 of the CAN1, CAN2-1 or CAN2-2 buses simultaneously.

For more details, see the chipKIT's documentation.

For instructions on flashing a new firmware version to the chipKIT, see the chipKIT firmware programming documentation.

4.3.1 USB

The micro-USB port on the Digilent Network Shield is used to send and receive OpenXC messages. The mini-USB cable on the Max32 itself is only used for re-programming.

4.3.2 UART

On the chipKIT, `UART1A` is used for OpenXC output at the 230000 baud rate. Hardware flow control (RTS/CTS) is enabled, so CTS must be pulled low by the receiving device before data will be sent. There are a few tricky things to watch out for with UART (i.e. Bluetooth) output on the chipKIT, so make sure to read this entire section.

`UART1A` is also used by the USB-Serial connection, so in order to flash the PIC32, the Tx/Rx lines must be disconnected. Ideally we could leave that UART interface for debugging, but there are conflicts with all other exposed UART interfaces when using flow control.

- Pin 0 - `U1ARX`, connect this to the TX line of the receiver.

- Pin 1 - `U1ATX`, connect this to the RX line of the receiver.
- Pin 18 - `U1ARTS`, connect this to the CTS line of the receiver.
- Pin 19 - `U1ACTS`, connect this to the RTS line of the receiver.

UART data is sent only if pin A1 is pulled low (to ground). If you are using a Bluetooth module like the BlueSMiRF from SparkFun, you need to hard-wire GND into this pin to actually enabling UART. To disable UART, pull A1 high (hard-wire to 5v) or leave it floating.

No data received over UART (i.e. Bluetooth)? If you are powering the device via USB but not also reading data via USB, it may be blocked waiting to send data. Try unplugging the USB connection and powering the device via the OBD connector.

4.3.3 Debug Logging

In most cases the logging provided via USB is sufficient, but if you are doing low-level development and need the simpler UART interface, you can enable it with the `UART_LOGGING` Makefile flag, but be aware that UART logging will dramatically decrease the performance of the VI.

On the chipKIT Max32, UART logging will be on UART2 (Pin 16 - Tx, Pin 17 - Rx) at 115200 baud.

4.3.4 LED Lights

The chipKIT has 1 user controllable LED. When CAN activity is detected, the LED will be enabled (it's green).

4.3.5 Compiling and Flashing

Attach the chipKIT to your computer with a mini-USB cable, `cd` into the `src` subdirectory, build and upload to the device.

```
$ export PLATFORM=CHIPKIT
$ make clean
$ make
$ make flash
```

If the flash command can't find your chipKIT, you may need to set the `SERIAL_PORT` variable if the serial emulator doesn't show up as `/dev/ttyUSB*` in Linux, `/dev/tty.usbserial*` in Mac OS X or `com3` in Windows. For example, if the chipKIT shows up as `/dev/ttyUSB4`:

```
$ SERIAL_PORT=/dev/ttyUSB4 make flash
```

and if in Windows it appeared as COM4:

```
$ SERIAL_PORT=com4 make flash
```

4.3.6 IDE Support

It is possible to use an IDE like Eclipse to edit and compile the project.

- Follow the directions in the above "Installation" section.
- Install Eclipse with the CDT project
- In Eclipse, go to `File -> Import -> C/C++ -> Existing Code as Makefile Project` and then select the `vi-firmware/src` folder.

- In the project's properties, under `C/C++ General -> Paths and Symbols`, add these to the include paths for C and C++:
  - `${MPIDE_DIR}/hardware/pic32/compiler/pic32-tools/pic32mx/include`
  - `${MPIDE_DIR}/hardware/pic32/cores/pic32`
  - `/src/libs/CDL/LPC17xxLib/inc` (add as a "workspace path")
  - `/src/libs/chipKITCAN` (add as a "workspace path")
  - `/src/libs/chipKITUSBDevice` (add as a "workspace path")
  - `/src/libs/chipKITUSBDevice/utility` (add as a "workspace path")
  - `/src/libs/chipKITEthernet` (add as a "workspace path")
  - `/usr/include` (only if you want to use the test suite, which requires the `check` C library)
- In the same section under Symbols, if you are building for the chipKIT define a symbol with the name `__PIC32__`
- In the project folder listing, select `Resource Configurations -> Exclude from Build` for these folders:
  - `src/libs`
  - `build`

If you didn't set up the environment variables from the `Installation` section (e.g. `MPIDE_HOME`), you can also do that from within Eclipse in `C/C++` project settings.

There will still be some errors in the Eclipse problem detection, e.g. it doesn't seem to pick up on the GCC `__builtin_*` functions, and some of the chipKIT libraries are finicky. This won't have an effect on the actual build process, just the error reporting.

4.3.7 Bootloader

All stock chipKITs are programmed with a compatible bootloader at the factory. The PIC32 avrdude bootloader is also tested and working and allows flashing over USB with `avrdude`.

4.4 CrossChasm C5 Interface

CrossChasm's C5 OBD interface is compatible with the OpenXC VI firmware. To build for the C5, compile with the flag `PLATFORM=CROSSCHASM_C5`.

CrossChasm has made the C5 available for purchase from their website, and it comes pre-loaded with the correct bootloader, so you don't need any additional hardware to load the OpenXC firmware.

The C5 connects to the CAN1 bus pins on the OBD-II connector.

4.4.1 Flashing a Pre-compiled Firmware

Assuming your C5 has the *bootloader* already flashed, once you have the USB cable attached to your computer and to the C5, follow the same steps to upload as for the *chipKIT Max32*.

The C5 units offered directly from the CrossChasm website are pre-programmed with the bootloader.

4.4.2 Bootloader

The C5 can be flashed with the same PIC32 avrdude bootloader, as the chipKIT.

The OpenXC fork of the bootloader (the previous link) defines a *CROSSCHASM_C5* configuration that exposes a CDC/ACM serial port function over USB. Once the bootloader is flashed, there is a 5 second window when the unit powers on when it will accept bootloader commands.

In Linux and OS X it will show up as something like */dev/ACM0*, and you can treat this just as if it were a serial device.

In Windows, you will need to install the *stk500v2.inf <https://raw.github.com/openxc/PIC32-avrdude-bootloader/master/Stk500v2.inf>* driver before the CDC/ACM modem will show up - download that file, right click and choose Install. The C5 should now show up as a COM port for for 5 seconds on bootup.

The C5 units offered directly from the CrossChasm website are pre-programmed with the bootloader.

If you need to reflash the bootloader yourself, a ready-to-go .hex file is available in the GitHub repository and you can flash it with MPLAB IDE/IPE and an ICSP programmer like the Microchip PICkit 3. You can also build it from source in MPLAB by using the *CrossChasm C5* configuration.

4.4.3 Compiling

The instructions for compiling from source are identical to the *chipKIT Max32* except that `PLATFORM=CROSSCHASM_C5` instead of `CHIPKIT`.

If you will not be using the avrdude bootloader and will be flashing directly via ICSP, make sure to also compile with `BOOTLOADER=0` to enable the program to run on bare metal.

4.4.4 USB

The micro-USB port on the board is used to send and receive OpenXC messages.

4.4.5 UART

On the C5, `UART1A` is used for OpenXC output at the 230000 baud rate. Hardware flow control (RTS/CTS) is enabled, so CTS must be pulled low by the receiving device before data will be sent.

TODO add pinout of expansion header, probably a picture

UART data is sent only if pin 0.58 (or PORTB BIT 4, RB4) is pulled high (to 5vv). If you are using a Bluetooth module like the BlueSMiRF from SparkFun, you need to hard-wire 5v into this pin to actually enabling UART. To disable UART, pull this pin low or leave it floating.

4.4.6 Debug Logging

In most cases the logging provided via USB is sufficient, but if you are doing low-level development and need the simpler UART interface, you can enable it with the `UART_LOGGING` Makefile flag.

On the C5, logging is on UART3A at 115200 baud (if the firmware was compiled with `DEBUG=1`).

4.4.7 LED Lights

The C5 has 2 user controllable LEDs. When CAN activity is detected, the green LED will be enabled. When USB or Bluetooth is connected, the blue LED will be enabled.

CHAPTER 5

Advanced Firmware Features

5.1 Low-level CAN Features

The OpenXC message format specification defines a "raw" CAN message type. You can configure the VI firmware to output raw CAN messages using this format.

For example, this JSON configuration will output all CAN messages received on a high speed bus connecetd to the CAN1 controller:

```
{    "name": "passthrough",
     "buses": {
         "hs": {
             "controller": 1,
             "raw_can_mode": "unfiltered",
             "speed": 500000
         }
     }
}
```

The only change from a typical configuration is the addition of the `raw_can_mode` attribute to the bus, set to `unfiltered`. When using the raw CAN configuration, there's no need to configure any messages or signals.

You may use both translated and raw output simultaneously - the 2 message types will be interleaved on the output interfaces, so you'll need to check for the right fields before reading the output.

If you're only interested in a few CAN messages, you can send a filtered set of raw messages. Change the `raw_can_mode` to `filtered` and add the messages ID's you want:

```
{    "name": "passthrough",
     "buses": {
         "hs": {
             "controller": 1,
             "raw_can_mode": "filtered",
             "speed": 500000
         }
     },
     "messages": {
       "0x21": {
         "bus": "hs"
       }
     }
}
```

This will read and send the message with ID `0x21` only.

The `raw_can_mode` flag can be applied to all active CAN buses at once by defining it at the top level of the configuration. For example, this configuration will enable unfiltered raw CAN output from 2 buses simultaneously:

```
{    "name": "passthrough",
     "raw_can_mode": "filtered",
     "buses": {
         "hs": {
             "controller": 1,
             "speed": 500000
         },
         "ms": {
             "controller": 2,
             "speed": 125000
         }
     }
}
```

When defined at the top level, the `raw_can_mode` can be overridden by any of the individual buses (e.g. `hs` could inherit the `unfiltered` setting but `ms` could override and set it to `filtered`).

There are yet more ways to configure and control the low-level output (e.g. limiting the data rate as to not overwhelm the VI's output channels) - see the *raw configuration examples* for more information.

5.1.1 Writing to CAN

By default, the CAN controllers on the VI are configured to be in a read-only mode - they won't even send ACK frames. If you configure one of the buses to be `raw_writable` in the firmware configuration, the controller will be write-enabled for raw CAN messages, e.g.:

```
{    "name": "passthrough",
     "buses": {
         "hs": {
             "controller": 1,
             "raw_can_mode": "unfiltered",
             "raw_writable": true,
             "speed": 500000
         }
     }
}
```

With a writable bus, you can send CAN messages (in the OpenXC "raw" message JSON format) to the VI's input interfaces (e.g. USB, Bluetooth) and they'll be written out to the bus verbatim.

Obviously this is an **advanced** feature with many security and safety implications. The CAN controllers are configured as read-only by default for good reason - make sure you understand the risks before enabling raw CAN writes.

For additional security, by default the firmware will not accept raw CAN write requests from remote interfaces even if `raw_writable` is true. Write requests from Bluetooth and network connections will be ignored - only USB is allowed by default. If you wish to write raw CAN messages wirelessly (and understand that those words make security engineers queasy), compile with the `NETWORK_ALLOW_RAW_WRITE` or `BLUETOOTH_ALLOW_RAW_WRITE` flags (see *all compile-time flags*).

The raw CAN write support is intended soley for protoyping and advanced development work - for any sort of consumer-level app, it's much better to use writable translated messages.

5.2 Binary Output Format

For applications that need to maximize the amount of data transferred from the vehicle, the firmware includes an *experimental* binary output format. Instead of JSON, it uses Google Protocol Buffers to more efficiently pack the data. Translated-style OpenXC messages are on average 30% smaller when using protobufs instead of JSON, and raw messages are around 60% smaller. This space savings comes at the cost of decreased flexibility and increased complexity in receiving and parsing the data.

The firmware does not currently support *receiving* binary-encoded messages - CAN write requests must still be sent in JSON.

This output format is supported by the official OpenXC Android library and OpenXC Python library, both of which will auto-detect the output format being used by an attached VI. The protobuf objects are defined in an experimental branch in the openxc-message-format repository if you wish to add support to another language or environment.

The output stream uses the common delimiting technique of writing the length of each protobuf message before the message itself in the stream.

5.2.1 Compiling with Binary Output

To use the binary output format, compile with the `BINARY_OUTPUT` flag (see *all compile-time flags*).

5.2.2 Motivation

The default output format encodes data from the vehicle as JSON, using the OpenXC message format. There are JSON formats for both translated and raw messages. The format is human-readable and very simple to parse, but it's not very compact.

For one, it representes everything in human-readable text, so for example, the number `999` (which could be stored in only 10 bits) takes 3 bytes - more than twice the space.

This isn't an issue for most users, as the output interfaces have plenty of bandwidth (USB is around 125-160 KB/s and the popular RN-42 Bluetooth tops out at 23KB/s) and the data rate of most release VI firwmares is on the order of only 3-6KB/s using JSON.

However, some applications need to pull a lot more data out of the car, perhaps by reading *raw CAN messages* or reading many signals at high frequencies. These can quickly overwhelm the output pipe using JSON.

5.3 CAN Bench Testing

Normally, the CAN controllers are initialized in a "listen only" mode. They are configured as writable only if using raw CAN passthrough with a CanBus that is marked `writable` or if a CanSignal is `writable`.

This works fine in a vehicle, but when testing on a bench with a simulated CAN network (e.g. another VI sending CAN messages directly to your VI under test), there's a problem - every CAN message must be acknowledged by the controller, and in "listen only" mode it does not send these ACKs. With nobody ACKing on the bus, the messages never propagate up from the network layer to the VI firmware.

When compiling the VI firmware to use to receive data on a bench test CAN bus, use the `DEFAULT_CAN_ACK_STATUS` flag to make sure CAN messages are ACked:

```
$ make clean
$ DEFAULT_CAN_ACK_STATUS=1 make
```

The CAN controllers will also be write-enabled if either CAN bus is configured to accept raw CAN or diagnostic message requests, or a writable translated signal is included.

5.3.1 Simulating a CAN Network

Speaking of CAN bus bench testing, it's possible to create a 2-node CAN network on your desk with 2 VIs to test your firmware. This is especially handy for prototyping new translated signals, since you can record a raw CAN bus trace from a vehicle and do all of your work inside where it's warm and comfortable.

Requirements

- 2 VIs
- A female-female OBD-II cable, or whatever combination of cable ends you need to connect one VI directly to another
- 120 ohm resistors across the H/L CAN wires in that cable or...
  - If one VI is a chipKIT, there are jumpers you can flip to enable 120 ohm resistors on each CAN controller (only one of the VIs needs the resistors)
  - If one VI is a Ford prototype, that has the termination resistors enabled by default on version 1.0. In an updated version that may change, as this is actually a mistake - they should have been only available if you short a solder jumper.

We recommend building your own cable with female OBD-II connectors from Molex (with the matching crimp connectors and a crimper wired together with 120ohm resistors crimped directly into one end of the cable (some pictures of a wired cable). We also recommend including the 12v and a ground wire, and shaving off a little ring of insulation on each at one end of the connector so you can inject bus power if needed. Make sure to offset the sections of shaved off insulation by a half inch so they don't short.

Alternatively, there are OBD-II splitter cables available online that may also work, but these have not been tested.

5.3.2 Preparing the Transmitter

1. Decide which of the VIs is the transmitter and which is the receiver. The receiver is most like your VI under test.
2. Follow the normal firmware build process (the CAN signals defined don't matter, just the bus speeds) but set the `TRANSMITTER` environment variable: `TRANSMITTER=1 make`. This changes the USB product ID from 1 to 2, so you can address the transmitter VI independently from the receiver. TODO create a transmitter config file with raw writable bus and no power management.

5.3.3 Preparing the Receiver

Compile the VI firmware for the receiver as usual, but instead of just `make`, run `DEFAULT_CAN_ACK_STATUS=1 make` This configures the CAN controllers as write-enabled, so that your VI under test can ACK the CAN messages. If nobody on the bus ACKs, you will receive nothing. In a car there are usually many other things ACKing, so we can be "listen only".

5.3.4 Sending Data on the Bus

You need a pre-recorded raw CAN trace file from a vehicle. To create one, compile the firmware in the passthrough mode (see the *low-level CAN docs*) and flash a VI, then use `openxc-dump` to receive the raw messages - point that at a file on disk to create a trace.

With a raw CAN trace file named `raw-can-trace.json`:

```
$ openxc-control writeraw --usb-product 2 -f raw-can-trace.json
```

That will send all of the message in the file, correctly spaced according to the timestamps. If you want to send continuously, you can loop the file like this:

```
$ watch -n 0 openxc-control writeraw --usb-product 2 -f raw-can-trace.json
```

5.3.5 Receiving Data

Receive data as usual from the VI under test, as if it was in real car.

5.4 Board Support

The code base is expanding very organically from supporting only one board to supporting multiple architectures and board variants. The strategy we have now:

- Switch between "platforms" with the PLATFORM flag - a platform encapsulates a micro architecture and a board variant.
- Implement different architecture-specific code in a subfolder for the micro
- Switch pins for board variants in in those same architecture-specific files (like in lights.cpp)

CHAPTER 6

I/O, Data Format and Commands

The OpenXC message format is specified and versioned separately from any of the individual OpenXC interfaces or libraries, in the OpenXC Message Format repository.

6.1 Commands

The firmware supports all commands defined in the OpenXC Message Format specification. Both UART and USB interfaces accept commands (serialized as JSON) sent in the normal data stream back to the VI. USB also supports these commands via USB control requests - see the USB section for the specifics.

The following is a summary of each command type - for the full specification, see the OpenXC Message Format.

6.1.1 Translated Writes

Translated write commands require that the firmware is pre-configured to understand the named signal, and also allows it to be written.

```
{"name": "seat_position", "value": 20}
```

With the tools from the OpenXC Python library you can send that from a terminal with the command:

```
openxc-control write --name seat_position --value 20
```

6.1.2 RAW CAN Message Writes

The RAW CAN message write requires that the VI is configured to allow raw writes to the given CAN bus. If the `bus` attribute is omitted, it will write the message to the first CAN bus found that permits writes.

```
{"bus": 1, "id": 1234, "value": "0x12345678"}
```

With the tools from the OpenXC Python library you can send that from a terminal with the command:

```
openxc-control write --bus 1 --id 1234 --value 0x12345678
```

6.1.3 Vehicle Diagnostic Requests

Diagnostic requests can either be one-time or recurring (if the `frequency` option is specified). The command requires that the VI is configured to allow raw writes to the given CAN bus (with the `raw_writable` flag in the config file). If the `bus` attribute is omitted, it will write the message to the first CAN bus found that permits writes.

```
{ "command": "diagnostic_request",
  "request": {
      "bus": 1,
      "id": 1234,
      "mode": 1,
      "pid": 5
    }
  }
}
```

With the tools from the OpenXC Python library you can send that from a terminal with the command:

```
openxc-diag --bus 1 --id 1234 --mode 1 --pid 5
```

6.1.4 Version Query

This asynchronous command will query for the version of firmware that the VI is running:

```
{ "command": "version"}
```

The response is injected into the normal output data stream:

```
{ "command_response": "version", "message": "v6.0 (default)"}
```

You can request the version with the tools from the OpenXC Python library:

```
openxc-control version
```

6.1.5 Device ID Query

This asynchronous command will query for a unique device ID for the VI:

```
{ "command": "device_id"}
```

If no device ID is available, the response message will be "Unknown". The response is injected into the normal output data stream:

```
{ "command_response": "device_id", "message": "0012345678"}
```

You can request the device ID with the tools from the OpenXC Python library:

```
openxc-control id
```

6.2 UART (Serial, Bluetooth)

The UART (or serial) connection for a VI is often connected to a Bluetooth module, e.g. the Roving Networks RN-41 on the Ford Reference VI. This allows wireless I/O with the VI.

The VI will send all messages it is configured to received out over the UART interface using the OpenXC message format. The data may be serialized as either JSON or protocol buffers, depending on the selected output format. Each message is followed by a `\r\n` delimiter.

The UART interface also accepts all valid OpenXC commands. JSON is the only support format for commands in this version. Commands must be delimited with a `\0` (NULL) character.

For details on your particular platform (i.e. the baud rate and pins for UART on the board) see the *supported platforms*.

6.3 USB Device

The VI is configured as a USB device, so you can connect it to a computer or mobile device that supports USB OTG. USB is best if you need to stream a lot of to or from the VI - the UART connection caps out at around 23KB/s, but USB can go about 100KB/s.

The VI will publish all messages it is configured to received to USB bulk `IN` endpoint 2 using the OpenXC message format. The data may be serialized as either JSON or protocol buffers, depending on the selected output format. Each message is followed by a `\r\n` delimiter. A larger read request from the host request will allow more messages to be batched together into one USB request and give high overall throughput (with the downside of introducing delay depending on the size of the request).

Bulk `OUT` endpoint 5 will accept valid OpenXC commands from the host, serialized as JSON (the Protocol Buffer format is not supported for commands). Commands must be delimited with a `\0` (NULL) character. Commands must be no more than 256 bytes (4 USB packets).

Finally, the VI publishes log messages to bulk `IN` endpoint 11 when compiled with the `DEBUG` flag. The log messages are delimited with `\r\n`.

If you are using one of the support libraries (e.g. openxc-python or openxc-android, you don't need to worry about the details of the USB device driver, but for creating new libraries the endpoints are documented here.

6.3.1 Control Transfers

The VI accepts a few control transfer requests on the standard endpoint 0.

Version

Transfer request type: `0x80`

The host can retrieve the version of the VI using the `0x80` control request. The data returned is the same format as the *version query*.

Device ID

Transfer request type: `0x82`

The host can retrieve a unique device identifier for the VI (if one is available) using the `0x82` control request. The data returned is the same format as *device ID query*.

CHAPTER 7

Testing

7.1 Windows USB Device Driver

If you want to send and receive vehicle data in Windows via USB, you must install the VI Windows Driver.

7.2 Python Library

The OpenXC Python library, in particular the `openxc-dashboard` tool, is useful for testing a VI. A quick "smoke test" using the Python tools is described in the Getting Started Guide for Python developers at the OpenXC website.

Keep in mind when bench testing that the VI will suspend if no CAN bus activity is detected. Compiled with `DEFAULT_POWER_MANAGEMENT=ALWAYS_ON` to stop this behavior, but don't leave it plugged into your car with power management off.

7.3 Debugging

To view debugging information, first compile the firmware with the debugging flag:

```
$ make clean
$ DEBUG=1 make
$ make flash
```

When compiled with `DEBUG=1`, two things happen:

- Debug symbols are available in the .elf file generated in the `build` directory.
- Log messages will be output over a separate USB endpoint required) - see *I/O, Data Format and Commands* for details. You can optionally enable logging via UART with the `UART_LOGGING` flag, but there may be a performance hit - see the *Makefile Options*.

To view the logs via USB, you can use the `--log-mode` flag with the Python CLI tools. See the `--help` text for any of those tools for more information.

To view UART logs, you can use an FTDI cable and any of the many available serial terminal monitoring programs, e.g. `screen`, `minicom`, etc. The pins for this UART output are different for each board, so see the *platform specific docs*.

7.4 Test Suite

The non-embedded platform specific code in this repository includes a unit test suite. It's a good idea to run the test suite before committing any changes to the git repository.

The test suite uses the check library. It should already be installed if you used `bootstrap.sh` to set up your development environment.

7.4.1 Running the Suite

```
vi-firmware/src $ make clean && make test
```

CHAPTER 8

Contributing

Please see our Contributing Guide.

8.1 Mailing list

For discussions about the usage, development, and future of OpenXC, please join the OpenXC mailing list.

8.2 Bug tracker

If you have any suggestions, bug reports or annoyances please report them to our issue tracker at http://github.com/openxc/vi-firmware/issues/

8.3 Authors and Contributors

A complete list of all authors is stored in the repository - thanks to everyone for the great contributions.

8.3.1 Related Projects

8.4 Python Library

The OpenXC Python library, in particular the *openxc-dashboard* tool, is useful for testing the VI with a regular computer, to verify the data received from a vehicle before introducing an Android device. Documentation for this tool (and the list of required dependencies) is available on the OpenXC vehicle interface testing page.

8.5 Android Library

The OpenXC Android library is the primary entry point for new OpenXC developers. More information on this library is available at in the applications section of the OpenXC website.

CHAPTER 9

License

This software depends on other open source projects, and a binary distribution may contain code covered by `other licenses`.