OpenXC VI Firmware Documentation

Release 8.2.1

Ford Motor Company

Dec 13, 2021

Contents

1 Getting Started **3**
- 1.1 Development Environment Setup . . . 3
- 1.2 Testing with Emulated Data . . . 5
- 1.3 Compiling the Default Configuration . . . 6
- 1.4 Customizing the VI Configuration . . . 6
- 1.5 Compiling with a Custom Configuration . . . 9

2 Firmware Configuration **11**
- 2.1 Read-only Configuration Examples . . . 11
- 2.2 Read-only Raw CAN Configuration Examples . . . 22
- 2.3 Custom Handler Configuration . . . 25
- 2.4 Writable Configuration Examples . . . 33
- 2.5 Writable Raw CAN Configuration Examples . . . 39
- 2.6 Diagnostic Configuration Examples . . . 40
- 2.7 Bit Numbering and Byte Order . . . 44
- 2.8 All Configuration Options Reference . . . 44
- 2.9 FAQ . . . 51

3 Compiling **53**
- 3.1 Example Build Configurations . . . 53
- 3.2 Makefile Options . . . 57
- 3.3 Troubleshooting . . . 60
- 3.4 Dependencies . . . 60
- 3.5 Developing in a Native Host OS . . . 63

4 Supported Embedded Platforms **65**
- 4.1 Ford Reference Vehicle interface . . . 65
- 4.2 NGX Blueboard LPC1768-H . . . 67
- 4.3 Digilent chipKIT Max32 . . . 67
- 4.4 CrossChasm C5 Interfaces . . . 69
- 4.5 CrossChasm C5 Bluetooth LE . . . 71
- 4.6 CrossChasm C5 Cellular . . . 72

5 Advanced Firmware Features **73**
- 5.1 Low-level CAN Features . . . 73
- 5.2 Binary Output Format . . . 75
- 5.3 CAN Bench Testing . . . 75

5.4 Firmware Developer Guide . . . 77
5.5 C5 Cellular Config . . . 78
5.6 Mass Storage Device . . . 80
5.7 Real Time Clock . . . 81

6 I/O, Data Format and Commands **83**
6.1 Commands . . . 83
6.2 UART (Serial, Bluetooth) . . . 84
6.3 USB Device . . . 85

7 Testing **87**
7.1 Windows USB Device Driver . . . 87
7.2 Python Library . . . 87
7.3 Debugging . . . 87
7.4 Test Suite . . . 88
7.5 Functional Test Suite . . . 88

8 Contributing **91**
8.1 Mailing list . . . 91
8.2 Bug tracker . . . 91
8.3 Authors and Contributors . . . 91
8.4 Related Projects . . . 91

9 License Disclosure **93**

10 License **105**

Version 8.2.1

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 vehicle interfaces*.
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 you can compile, you need to set up our development environment. You only need to do this once.

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

1.1.1 Install Git

You need to install Git to retrieve the source code repository for the `vi-firmware`.

Windows

Install GitHub for Windows, which includes a GUI for cloning Git repositories hosted at GitHub as well as a Bash terminal for using Git from the command line.

Mac OS X

Install GitHub for Mac, which includes a GUI for cloning Git repositories hosted at GitHub and also installs the `git` command line tools.

Linux

If you're using Linux, you hopefully know what you're doing and can install Git for your distro of choice. A couple of examples:

Ubuntu:

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

Arch Linux:

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

1.1.2 Clone the vi-firmware Repository

On Windows or Mac and you installed the GitHub app, open the vi-firmware repository in you browser and click the "Clone in Desktop" button.

If you are using `git` from the command line, clone it like so:

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

1.1.3 Install Vagrant (Recommended for All Platforms)

On all platforms, we recommend using the included Vagrant configuration to get compiling as quickly as possible. Vagrant is a tool that helps developers create and use identical development environments. We include a `Vagrantfile` in the repository that Vagrant uses to create a small Linux virtual machine with all of the vi-firmware dependencies installed. Once installed, you can open a shell in this VM just by running `vagrant ssh`. The `vi-firmware` on the host OS (e.g. your Linux, Windows or OS X machine) is shared to the Vagrant VM at the path `/vagrant`. You can use whatever text editor or IDE you prefer in your native OS, and simply do the compilation from within the VM.

1. Install VirtualBox.
2. If using Windows 7 (not required in Windows 8), add the VirtualBox tools to your PATH: `PATH=%PATH%;c:\Program Files\Oracle\VirtualBox`. If you aren't sure how to edit your `PATH`, see this guide for all versions of Windows. Log out and back in for the change to take effect.
3. Install Vagrant.
4. Navigate to the `vi-firmware` repository in the GitHub app, click the gear icon in the top right corner and select "Open in Git Shell".
5. In the shell, run `vagrant up` to initialize the Vagrant VM - this may take up to 10 minutes as it downloads and installs a number of dependencies in the VM.

6. If the initialization completes with no errors, run `vagrant ssh` to open a shell in the VM.
7. The `vi-firmware` directly is shared with the VM in the default home directory. Move into that directory and compile away:

```
$ cd vi-firmware
~/vi-firmware $ fab reference build
```

You can edit the source code from your native OS and re-run `fab reference build` from the Vagrant shell to compile (to compile for the reference VI, for example). Run `vagrant suspend` to suspend the VM and `vagrant up` again to restore it - it will be much faster after the first run.

1.1.4 Native Development

Don't want to use Vagrant? There are varying levels of support for compiling in your native OS - see the *native development environment docs*.

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.

If you are using Vagrant, `cd` into the `/vagrant` directory (which is actually a pointer to the `vi-firmware` directory on your host computer) and run `fab reference build`:

```
$ cd /vagrant
/vagrant $ fab reference emulator build
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. If you don't already have the OpenXC Python library installed, 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, compile the default- the default build for the reference VI.

If you are using Vagrant, `cd` into the `/vagrant` directory (which is actually a pointer to the `vi-firmware` directory on your host computer) and run `fab reference build`:

```
$ cd /vagrant
/vagrant $ fab reference build
Compiling for FORDBOARD...
...lots of output...
Compiled successfully for FORDBOARD running under a bootloader.
```

If you are doing native development, it's the same process but `cd` into wherever you put the `vi-firmware` directory instead of `/vagrant`.

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": 2015, "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 developed in Windows, Linux 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"
```

(continues on next page)

(continued from previous page)

```
        }
    }
}
```

- 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 $ fab reference build
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 simple vehicle message or CAN 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*.

- *One Bus, One Numeric Signal*
- *Transformed Numeric Signal*
- *One Bus, One Boolean Signal*
- *One Bus, One State-based Signal*
- *Combined State-based Signal*
- *Two Buses, Two Signals*
- *Limited Simple Vehicle Message Signal Rate*

- *Limited Simple Message Rate if Unchanged*
- *Send Signal on Change Only*
- *Separate Files for Message Sets*
- *Mapped from a DBC File*
- *Same Message ID, Two Buses*

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 simple vehicle 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",
```

(continues on next page)

(continued from previous page)

```
            "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 simple vehicle 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 simple vehicle 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
        }
```

(continues on next page)

(continued from previous page)

```
    },
    "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",
```

(continues on next page)

(continued from previous page)

```
            "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 signal using the OpenXC simple vehicle 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 Simple Vehicle Message 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 Simple Message Rate if Unchanged

We want the same signal from *Limited Simple Vehicle Message 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 Simple Message 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
                }
```

(continues on next page)

(continued from previous page)

```
            }
        }
    }
}
```

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,
```

(continues on next page)

(continued from previous page)

```
            "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
        },
```

(continues on next page)

(continued from previous page)

```
        "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.

- *Unfiltered Raw CAN*

- *Filtered Raw CAN*
- *Unfiltered Raw CAN with Limited, Variable Data Rate*
- *Unfiltered Raw CAN with Strict, Limited Data Rate*
- *Simple Messages and CAN Messages Together*

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 Simple Messages and CAN Messages 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 CAN messages simultaneously.

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

(continues on next page)

(continued from previous page)

```
        }
    },
    "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 CAN and simple vehicle messages 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 CAN message format, and the simple vehicle 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.

- *Custom Transformed Numeric Signal*
- *Transformed with Signal Reference*
- *Composite Signal*
- *Initializer Function*
- *Looper Function*
- *Ignore Depending on Value*

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(const CanSignal* signal, const CanSignal* signals,
      SignalManager* signalManager, SignalManager* signalManagers,
              int signalCount, Pipeline* pipeline, 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(const CanSignal* signal, const CanSignal* signals,
      SignalManager* signalManager, SignalManager* signalManagers,
              int signalCount, Pipeline* pipeline, float value, bool* send) {
    SignalManager* signSignal = lookupSignalManagerDetails("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;
```

(continues on next page)

(continued from previous page)

```
}
```

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,
```

(continues on next page)

(continued from previous page)

```
                    "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(const CanSignal* signal, const CanSignal* signals,
    SignalManager* signalManager, SignalManager* signalManagers,
int signalCount, Pipeline* pipeline, float value, bool* send) {
    // 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 =
```

(continues on next page)

(continued from previous page)

```
        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",
```

(continues on next page)

(continued from previous page)

```
            "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,
```

(continues on next page)

(continued from previous page)

```
                    "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(const CanSignal* signal, const CanSignal* signals,
      SignalManager* signalManager, SignalManager* signalManagers,
              int signalCount, Pipeline* pipeline, 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.

- *Named Numeric Signal Write Request*
- *Named Boolean Signal Write Request*
- *Named State-based Signal Write Request*
- *Named, Transformed Written Signal*
- *Composite Write Request*

2.4.1 Named 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
                }
            }
```

(continues on next page)

(continued from previous page)

```
        }
    }
}
```

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 Named 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 Named 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 Named, Transformed Written Signal

We want to write the same signal as *Named 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"
```

(continues on next page)

(continued from previous page)

```
    ]
}
```

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.

- *Write a CAN messages*
- *Writing with Filtered CAN*

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,
```

(continues on next page)

(continued from previous page)

```
            "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*.

- *Recurring Simple Diagnostic PID Request*
- *Recurring Parsed OBD-II PID Request*
- *Recurring Named Diagnostic PID Request*
- *On-Demand Request from Command Handler*

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
        }
```

(continues on next page)

(continued from previous page)

```
    ]
}
```

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 an OpenXC simple vehicle message.

```
{   "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 simple vehicle 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": {
```

(continues on next page)

(continued from previous page)

```
            "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 and Byte Order

When dealing with binary data like CAN messages, there are two important details - byte order and bit numbering.

Byte order, or endianness, determines the convention used to interpret a sequence of bytes as a number. Given 4 bytes of data, e.g. `0x01 02 03 04`, the endianness determines which byte is the "zero-th" byte and which is the last. There are only two options: big endian (a.k.a. Motorola order) and little endian (Intel order). Big endian is most common in automotive networks.

Bit numbering is an automotive specific term and must less standardized. The bit numbering can be what we will call normal or inverted. 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 most commonly used format in the industry, DBC files from Vector's software, use an inverted numbering by default when viewed in CANdb++, for example. 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
```

- *Message Set*
- *Parent Message Sets*
- *Initializers*
- *Loopers*
- *CAN Buses*
- *CAN Messages*
- *Diagnostic Messages*
- *Mappings*
- *Extra Sources*
- *Commands*

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 VI firmware can be compiled natively in Linux and Mac OS X. In Windows, you can use the included virtual machine configuration (which uses [Vagrant](http://www.vagrantup.com/). Cygwin is unnoficially supported, and is no longer recommended.

When compiling you need to specify which *embedded 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, many of which have shortcuts that can be run with `fab` at the command line, without remembering all of the configuration options.

- *Default Build*
- *Automatic Recurring OBD-II Requests Build*
- *Emulated Data Build*
- *Test Mode Build*
- *Fabric Shortcuts*

3.1.1 Default Build

You must always specify your target platform with the `PLATFORM` environment variable. For this example, we'll set `PLATFORM=FORDBOARD`.

By default, if you just run `make` with no other environment variables set, the firmware is built with these options. This is a good build to use if you want to send diagnostic requests through a VI:

```
$ make
<...snip lots of output...>
Compiled with options:
-       FORDBOARD = PLATFORM
-       1         = BOOTLOADER
-       0         = TEST_MODE_ONLY
-       0         = DEBUG
-       0         = MSD_ENABLE
-       180       = DEFAULT_FILE_GENERATE_SECS
-       0         = DEFAULT_METRICS_STATUS
-       1         = DEFAULT_ALLOW_RAW_WRITE_USB
-       0         = DEFAULT_ALLOW_RAW_WRITE_UART
-       0         = DEFAULT_ALLOW_RAW_WRITE_NETWORK
-       USB       = DEFAULT_LOGGING_OUTPUT
-       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 (`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.
- 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.
- *Mass storage device* (`MSD_ENABLE` is `0`) is disabled by default and is available on certain C5 devices which have a provision to connect a SD card. The time intervals at which the data is logged is (`DEFAULT_FILE_GENERATE_SECS` is `180`) set to 180 seconds by default.

Note: There's a shortcut for this default build, using the Fabric tool and an included script. This will build the default build for the reference VI platform:

```
fab reference build
```

and for the CrossChasm C5 Devices, choose one of:

```
fab c5bt build
fab c5ble build
fab c5cell build
```

and finally, for the chipKIT Max32:

```
fab chipkit build
```

Get the idea? These shortcuts will make sure the flags are set to their defaults, regardless of what you may have in your current shell environment. See the fabfile.py or run % fab -l for a complete list of options.

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         = TEST_MODE_ONLY
-       0         = DEBUG
-       0         = MSD_ENABLE
-       180       = DEFAULT_FILE_GENERATE_SECS
-       0         = DEFAULT_METRICS_STATUS
-       1         = DEFAULT_ALLOW_RAW_WRITE_USB
-       0         = DEFAULT_ALLOW_RAW_WRITE_UART
-       0         = DEFAULT_ALLOW_RAW_WRITE_NETWORK
-       USB       = DEFAULT_LOGGING_OUTPUT
-       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.

Note: This build also has a shortcut using the Fabric script. Just add the keyword `translated_obd2` before `build` in your call to `fab` at the command line. For example, this compiles for the reference VI with the automatic recurring, translated OBD2 requests:

```
fab reference translated_obd2 build
```

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 or the 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         = TEST_MODE_ONLY
-       0         = DEBUG
-       0         = MSD_ENABLE
-       180       = DEFAULT_FILE_GENERATE_SECS
-       0         = DEFAULT_METRICS_STATUS
-       1         = DEFAULT_ALLOW_RAW_WRITE_USB
-       0         = DEFAULT_ALLOW_RAW_WRITE_UART
-       0         = DEFAULT_ALLOW_RAW_WRITE_NETWORK
-       USB       = DEFAULT_LOGGING_OUTPUT
-       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!

Note: This build also has a shortcut using the Fabric script. Just add the keyword `emulator` before `build` in your call to `fab` at the command line. For example, this compiles for the reference VI with emulatded data:

```
fab reference emulator build
```

3.1.4 Test Mode Build

It is often desired to know if a hardware is functional before performing other operations on the device. Currently supported on CrossChasm C5 hardware this option creates a firmware build that will check the hardware and glow an onboard LED depending on the result. More description about this test can be found in *Other Hardware Tests*.

- `TEST_MODE_ONLY` is `1`,will cause a hardware testing firmware to be generated on build.

In fabric use the following command along with the correct C5 platform, in this case CrossChasm C5 BLE is used as an example:

```
fab c5ble build test_mode_only
```

3.1.5 Fabric Shortcuts

The repository includes a `fabfile.py` script, which works with the `Fabric` commmand line utility to simplify some of these build configurations. The `fab` commands are composable, following this simple formula:

- Start your command with `fab`
- Specify the target platform with `chipkit`, `c5bt`, `c5ble` , `c5cell`, or `reference`.
- Optionally include `emulator` or `translated_obd2` to enable one of the example builds described above.
- End with `build` to start the compilation.

For example, this builds the firmware for a chipKIT and includes emulated data:

```
fab chipkit emulator build
```

while this builds the default firmware, ready for OBD2 requests for the chipKIT:

```
fab chipkit obd2 build
```

You can specify the message format with `json`, or `protobuf`:

```
fab reference protobuf build
```

Other useful options are `debug`, `msd_enable`, `test`, and `clean`. See the fabfile.py or run % fab -l for a complete list of options. The `fab` commands can be run from any folder in the vi-firmware repository.

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.

Note: Don't miss the `Fabric` helper scripts for the common build configurations in *Fabric Shortcuts*.

`PLATFORM` Select the target *microcontroller platform*.

Values: **`FORDBOARD`**, **`CHIPKIT`**, **`CROSSCHASM_C5_BT`**, **`CROSSCHASM_C5_BLE`**, `CROSSCHASM_C5_CELLULAR`, `BLUEBOARD`

Default: `CHIPKIT`

Note: The old Platform variable `CROSSCHASM_C5` has been renamed to `CROSSCHASM_C5_BT`

TEST_MODE_ONLY Set to `1`, will create a firmware build which will simply test onboard hardware and glow a LED based on the result of the test.

Values: `0` or `1`

Default: `0`

Note: Currently supported on CROSSCHASM C5 Platforms

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 `DEFAULT_LOGGING_OUTPUT` flag. This also forces `DEFAULT_POWER_MANAGEMENT=ALWAYS_ON` and `DEFAULT_CAN_ACK_STATUS=1`.

Values: `0` or `1`

Default: `0`

MSD_ENABLE Set to `1` to enable logging to SD card and mass storage device(MSD) over USB. In this mode the device will startup as *MSD* only when powered up directly from a USB port. In other cases when the device is connected to a vehicle the firmware will perform logging operations on the SD card.

Values: `0` or `1`

Default: `0`

DEFAULT_FILE_GENERATE_SECS Enabled only when `MSD_ENABLE=1` sets the time interval for generating files on the SD card. All files are generated within the VI_LOG folder on the FAT file system. Read the *mass storage document* for more details.

Values: `15` to `86400`

Default: `180`

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 force `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_LOGGING_OUTPUT When combined with `DEBUG`, controls the output interface used for debug logging. See the *platform docs* for details on how to read this output.

Values: `OFF`, `USB`, `UART` or `BOTH`

Default: `USB`

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

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.

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.

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

We also require some libraries from Microchip that are available from the bottom of the Network Shield page. You can read and accept Microchip's license and download the libraries on the Digilent download page. The bootstrap script does the above and then the `libs` directory should look like this:

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

(continues on next page)

(continued from previous page)

```
  chipKITUSBDevice/
   ... 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 `arm-none-eabi-gcc`, `arm-none-eabi-binutils` and `arm-none-eabi-gdb` 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;N;d;}" /
↪System/Library/Extensions/FTDIUSBSerialDriver.kext/Contents/Info.plist
$ sudo kextunload /System/Library/Extensions/FTDIUSBSerialDriver.kext/
$ sudo kextload /System/Library/Extensions/FTDIUSBSerialDriver.kext/
```

3.5 Developing in a Native Host OS

On all platforms, we recommend using the included Vagrant configuration to get compiling as quickly as possible with a Linux virtual machine (VM).

If you are an advanced developer and you want to compile in your host OS, we may have support for it. Beware tha there are number of dependencies and we've found it finicky to replicate on the huge variety of possible configurations. Unless you have a compelling reason not to, we strongly recommend using *Vagrant*.

The dependencies can be installed using the included `bootstrap.sh` shell script. The script officially supports and is tested regularly in:

- Ubuntu Linux 13.04 - 14.04
- Arch Linux

and it has unofficial support (i.e. it's been tested, but not regularly) for:

- Mac OS X Mavericks
- Cygwin in Windows. We've found Cygwin can be a huge pain to get consistently set up, so we strongly discourage you from using it unless you are an expert in debugging Cygwin-specific problems.

Even if you already use Linux, we still recommend using Vagrant to set up a VM. The huge variety of Linux installations means that it's often much more pleasant to use the Vagrant VM which has a stable, repeatable setup.

3.5.1 Pre-requisites

Follow the steps in *Development Environment Setup* to install Git and clone the `vi-firmware` repository.

3.5.2 Cygwin Pre-Setup

Compiling natively in Windows (e.g. at a Windows Command Prompt) is not supported. Cygwin (32-bit) is unofficially supported, but is no longer recommended. You'll be much happier using Vagrant, promise!

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, git, gcc-core, patchutils, 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
```

3.5.3 Bootstrap

Run the `bootstrap.sh` script:

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

If there were no errors, you 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. If you're stuck, you might seriously consider using *Vagrant*!

CHAPTER 4

Supported Embedded Platforms

For information on how to add support for another embedded system, 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:

```
$ fab reference build
```

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 `DEFAULT_LOGGING_OUTPUT="UART"` build option.

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 `DEFAULT_LOGGING_OUTPUT="UART"` build option.

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 `DEFAULT_LOGGING_OUTPUT="UART"` build option, 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.

```
vi-firmware/src/ $ fab chipkit build
vi-firmware/src/ $ 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 Interfaces

CrossChasm C5 family of OBD interfaces include versions with different communication radios such as the Bluetooth Classic, *Bluetooth Low Energy* and *Cellular(GPRS)*. Most things are common between the different devices and are described on this page. This page covers the Bluetooth Classic version or `PLATFORM=CROSSCHASM_C5_BT`. See the particular BLE and Cellular pages for specific differences.

CrossChasm's C5 OBD interface is compatible with the OpenXC VI firmware. To build for one of the C5s, compile with one of the flags: `PLATFORM=CROSSCHASM_C5_BT`, `PLATFORM=CROSSCHASM_C5_BLE` or `PLATFORM=CROSSCHASM_C5_CELLULAR`.

Note: The old `CROSSCHASM_C5` Platform has been renamed to `CROSSCHASM_C5_BT` and the fabric shortcut `c5` has been updated to `c5bt`. Both old variables are aliased to the new BT forms.

All C5s come pre-loaded with the correct bootloader, so you don't need any additional hardware to load the OpenXC firmware. You do have flash an appropriate firmware for your application.

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

Note: CAN2 pins `CROSSCHASM_C5_BLE` are mapped to pin 3 and pin 11 of the OBD-II connector.

4.4.1 Flashing a Pre-compiled Firmware

Assuming your C5 has the *bootloader* already flashed, follow these `instructions`.

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 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.

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_BT`, `PLATFORM=CROSSCHASM_C5_BLE` or `PLATFORM=CROSSCHASM_C5_CELLULAR` 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 CROSSCHASM_C5_BT, `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.

UART data is sent only if pin 0.58 (or PORTB BIT 4, RB4) is pulled high (to 5v). 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 Mass Storage Device

The `CROSSCHASM_C5_BT` and `CROSSCHASM_C5_CELLULAR` are equipped with an SD card reader. If enabled, device messages (except debug messages) are logged to the SD card. See *MSD* for more info.

4.4.7 Real Time Clock

The C5 family of devices have a low power RTC chip that is connected to the PIC32 over the I2C bus. The RTC enables timestamping of vehicle messages at the time of generation. Timestamps are generated with millisecond resolution. See *RTC* for more info.

4.4.8 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 `DEFAULT_LOGGING_OUTPUT="UART"` build option.

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

Note: If `MSD_ENABLE=1` debug logging is not available as these pins are shared with the RTC for time stamping.

4.4.9 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.

4.5 CrossChasm C5 Bluetooth LE

4.5.1 Bluetooth Low Energy Specifications

The Bluetooth Low Energy or BLE protocol supports communication through a GATT server - client architecture. The device hosts a single service with following read and write characteristic UUIDs -

- Service 6800-d38b-423d-4bdb-ba05-c9276d8453e1
- Write Characteristic 6800-d38b-5262-11e5-885d-feff819cdce2
- Notify Characteristic 6800-d38b-5262-11e5-885d-feff819cdce3

The phone application or the GATT client should enable notifications on the notify characteristic in order to recieve OpenXC messages from the device. The write characteristic can be used to send commands and configuration messages to the device.

4.5.2 Connection Details

To connect to C5_BLE using a host such as a Smart Phone, the application must perform a scan of available bluetooth low devices in the network. The C5_BLE device includes the 128 bit service UUID in its scan response payload. The phone application should then attempt to connect to this device once a match is found. After establishing connection sucessfully over BLE the host application must enable notifications, failing to do so will result in automatic drop in connection after a timeout.

4.5.3 Sleep Mode

In the power saving mode the power to the BLE radio is shut-off

4.5.4 LED Lights

The BLE_C5 has 2 user controllable LEDs. When CAN activity is detected, the green LED will be enabled. When USB or bluetooth low energy is connected, the blue LED will be enabled. If CAN is silent the red LED will be enabled. All LEDs will be turned off when sleep mode is entered

---

Note: UART for Debug logging or commands is not available on the C5_BLE device.

4.6 CrossChasm C5 Cellular

The C5 Cellular device requires an OBD-II to DB9 Serial cable (which is included) in the shipment. This requires the device to be attached elsewhere in the vehicle since it does not plug directly into the port.

Users are required to supply their own SIM card. The SIM card is the Mini-SIM (2FF) size that supports 3G GSM.

The C5 Cellular device also has it's own GPS signal which can be configured to be inserted into the OpenXC JSON data.

4.6.1 TeLit Power Source

It is important to note that the TeLit radio (the 3G modem) is powered from the OBD-II 12V power supply. Therefore, in USB powered emulator mode, the radio will not send data or respond to commands unless there is also a benchtop 12V supply to the device.

4.6.2 Sample Web Server

The below project is a sample Azure server to receive data from a C5 Cellular device.

https://github.com/openxc/openxc-azure-webserver

4.6.3 Configuration Options

For full description of all the compile options for the Cellular connection, see *C5 Cellular Configuration Options*

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 simple vehicle messages and CAN message 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
         }
```

(continues on next page)

(continued from previous page)

```
    },
    "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 `DEFAULT_ALLOW_RAW_WRITE_NETWORK` or `DEFAULT_ALLOW_RAW_WRITE_UART` 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 simple vehicle 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 `DEFAULT_OUTPUT_FORMAT=PROTOBUF` environment variable set (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 actual amount of data sent from the pre-compiled binary firmware (from Ford, for example) is only about 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 Firmware Developer Guide

As much as possible, the firmware is written as vanilla C that can compile and run on any platform. The advantages of this are that we can write unit tests and run them on a development computer, which really speeds up the code-compile-debug cycle.

Each embedded platform that the firmware supports needs a set of board support implementations in the `platform` folder. Right now we support the PIC32 (2 different variants) and LPC1768. There are a number of functions declared in header files that are indicated as "platform-specific" that must be implemented for each board you want to support. There isn't a good central place to find this list right now, but if you start from one of the existing platforms it should be clear what needs to be implemented. The code only uses platform-specific C extensions and library includes in these `platform` folders.

The unit tests are all in `src/tests`, and the tests implement their own platform specific code in `src/tests/platform` - many of these include test spy functionality so the unit tests can verify that an LED turned on, a CAN message was sent to the controller, etc. You can run the test suite with `make test`.

The firmware runs on bare metal with a main event loop. To understand the firmware, start walking through the code from there, as that is the primary driver of the application. At a high level it runs this each time through the loop:

1. Process any received CAN messages that have been queued up during a CAN rx interrupt - call any registered handler functions, translate messages and queue output to send back to the client via USB/BT/network.
2. Send any pending diagnostic requests to the bus.
3. Handle any data received from a USB/BT/network connection, including incoming commands and CAN write requests.
4. Flush all queued outgoing CAN messages to the bus.

5. Update the LEDs to indicate if the CAN bus is connected, and which output interface is active.
6. Finally, flush any queued data to the output interfaces (USB/BT/network).

You'll notice when receiving and sending data, we make use of a buffer - this is to avoid doing very much work in the interrupt handlers for CAN/USB/UART.

5.5 C5 Cellular Config

When building for the C5 Cellular, there are some modem-specific build options that you can choose. These options are set directly in the firmware, in the configuration struct from config.cpp. Some common scenarios are described here.

5.5.1 Building for a particular mobile network (SIM card):

The following five options govern how the modem finds and connects to mobile network(s). These options should be considered whenever you compile the firmware for use with a particular SIM.

- **networkOperatorSettings.allowDataRoaming**
  - set to "true" to let the SIM use any network it can find, whether or not the SIM sees it as a "home" network. This is appropriate for many "global" SIM cards, which see all mobile networks as roaming. If you are using a SIM that is tied to a specific carrier, you can set this to "false" to stop the modem from connecting to a foreign network on start-up.
- **networkOperatorSettings.operatorSelectMode**
  - **AUTOMATIC**: Find and use any available mobile network which accepts the SIM. The options "PLMN" and "networkType" are ignored in this mode.
  - **MANUAL**: Find and connect to only the mobile network with a PLMN matching the "PLMN" option, and of the type specified by networkType.
  - **DEREGISTER**: This mode forces the modem to deregister from the current network. The modem will stay disconnected from any mobile network until the mode changes. Not particularly useful as a connection strategy, but can be used by custom code looking for implement a custom connection strategy.
  - **SET_ONLY**: Only useful for custom code implementations. Forces the modem to ignore the specified "PLMN" and "networkType", and to only interpret the format specifier for the network name (which is fixed in this firmware).
  - **MANUAL_AUTOMATIC**: Modem follows the MANUAL connection strategy. If a suitable network cannot be found, modem falls back to the "AUTOMATIC" connection strategy.
- **networkOperatorSettings.networkDescriptor.PLMN**
  - a 5-6 digit number that combines the "mobile country code" (MCC) and "mobile network code" (MNC) to uniquely identify a particular mobile network. Using the PLMN avoids any issues with variations in network text names. For example, 302220 is the PLMN for TELUS. A list of PLMNs is easily found online, e.g. "http://en.wikipedia.org/wiki/Mobile_country_code".
- **networkOperatorSettings.networkDescriptor.networkType**
  - an enumeration that specifies the access technology of the desired mobile network. Generally, "GSM" specifies a 2G network and "UTRAN" specifies a 3G network. A particular mobile network (PLMN) can have both types of networks available for the modem to use.
- **networkDataSettings.APN**

  - is the familiar "access point name" the a mobile operator will generally require in order for the device to access data services on the network. It is typically a string like "internet.com" or "apn".

5.5.2 Setting up GPS signals:

The GPS receiver in the C5 Cellular publishes a GPS message that contains a variety of signals. You can select which signals in this message will be published through the VI, and how often. Note that all of the selected GPS signals share a single publish interval.

The following are available in **telit.config.globalPositioningSettings**:

1. To enable/disable GPS functionality, set the "gpsEnable" flag to "true" or "false".
2. Set the interval (in milliseconds) to publish the GPS message via "gpsInterval".
3. Select the GPS signals you want to include in the published GPS message by setting the appropriate "gpsEnableSignal_gps_" flag. The following signal "enable" flags are available:
   - gpsEnableSignal_gps_time
   - gpsEnableSignal_gps_latitude
   - gpsEnableSignal_gps_longitude
   - gpsEnableSignal_gps_hdop
   - gpsEnableSignal_gps_altitude
   - gpsEnableSignal_gps_fix
   - gpsEnableSignal_gps_course
   - gpsEnableSignal_gps_speed
   - gpsEnableSignal_gps_speed_knots
   - gpsEnableSignal_gps_date
   - gpsEnableSignal_gps_nsat

5.5.3 Server connection settings:

The following configuration options tell the server how to connect to your OpenXC web server:

- **serverConnectSettings.host**
  - the web server address, for example "openxcserverdemo.azurewebsites.net". Only supply the base address of the web server here, do not include any particular resources (e.g. /api/{IMEI}/data). Specific, standard OpenXC resources are requested by the firmware's server API calls.
- **serverConnectSettings.port**
  - simply the TCP/IP port on which the web service operates. Typically this is port 80 (HTTP).

5.5.4 Socket connection settings:

More advanced settings that control the specific behaviour of the TCP/IP sockets used to communicate with thr OpenXC web server. You typically do not have to modify these unless you have some specific network issues or advanced requirements (or you find the default values to be less than optimal). TCP/IP socket settings are:

- **socketConnectSettings.packetSize**

  - the TCP/IP packet size. Set to 0 for default (300 bytes).

- **socketConnectSettings.idleTimeout**
  - the number of seconds that the modem will tolerate an idle socket before closing it. An idle socket is a socket without any data exchange. Set to 0 for no timeout, and the modem will keep an idle socket open indefinitely. Note that the server on the other end can still close the socket due to it's own idle timeout (or other) criteria.
- **socketConnectSettings.connectTimeout**
  - the duration of time that the modem will attempt to open a socket before timing out. The duration is specified in "hundreds of milliseconds", so a value of 600 would be 60000 milliseconds, or 60 seconds. A longer timeout period allow the modem a higher chance of gaining access to the server in a single connect request, at the expense of hanging the command interface for the duration of the timer (other parallel operations cannot be performed). Shortening the timer will increase the odds of a failed connection attempt, but the command interface will become more responsive.
- **socketConnectSettings.txFlushTimer**
  - the duration of time that the modem will hold a partial TCP/IP packet before flushing it to the TCP/IP socket. Specified in "hundreds of milliseconds".

5.6 Mass Storage Device

5.6.1 Operation

MSD mode is supported on `CROSSCHASM_C5_BT`, and `CROSSCHASM_C5_CELLULAR` devices. With this mode enabled the devices store messages(except debug log messages)on to the SD card storage that is formatted with FAT file system. The maximum storage capacity supported is upto 2GB. To view the contents of the SD card the user can unplug the SD card and connect to a card reader or simply connect the device to a USB port. In the latter case the device will be mounted as a mass storage device. The user can thereby view and edit the contents of the SD card. To enable MSD mode set `MSD_ENABLE` as `1` in the build configuration options.

5.6.2 USB connected modes

With the MSD mode enabled the devices still continue to support the standard USB logging and command interface support. However at a time either MSD or standard USB is supported, to decide which(correct mode), the firmware checks how the device was powered. To enable MSD mode the device must be unplugged from the vehicle and connected to the USB. In the other scenario where the device was connected to the vehicle USB will function as standard mode.

Note: To view emulated signals with `DEFAULT_EMULATED_DATA_STATUS` set to `1` the device must be connected to a vehicle or powered by a 12V supply on the OBD connectors

---

5.6.3 File generation and naming convention

To maintain unique and sequential files a 8.3 filenaming convention is used in which the first eight ascii characters represent the unix time in hex of the time the file was created.

5.6.4 File splitting technique

It is desired to have multiple log entries based on time rather than one large file. The build configuration option `DEFAULT_FILE_GENERATE_SECS` creates a new file each time this interval is elapsed. A larger interval will yeild to lesser files however the size of the files will depend upon the build configuration and the speed at which data is generated. By default the option `DEFAULT_FILE_GENERATE_SECS` is set to `180`

5.6.5 SD card status message

It may happen that the SD card which connected has become full or is unformatted. In such a scenario a message is sent to the user over the USB, Bluetooth and Cellular. User may also send a command to check the status synchronously. Based on this the user can perform a format by connecting the device over USB or insert a new empty card.

Example JSON command

{ “command”: “sd_mount_status”}

Response 1 (SD card correctly mounted)

{ “command_response”: “sd_mount_status”, “status”: true}

Response 2 (SD card mounting failed)

{ “command_response”: “sd_mount_status”, “status”: false}

Note: `SD_MOUNT_STATUS` command is not available when device is connected as a mass storage device. Issuing a command in this mode returns a `false` status

---

5.6.6 UART Debug Support

With `MSD_ENABLE` as `1` the UART3 on the PIC32 is configured as a I2C peripheral, hence debug logging is not supported at this time.

5.7 Real Time Clock

5.7.1 Real time Clock Configuration

For correct time stamping it is necessary that the RTC is configured with the correct time. A new rtc configuration command is added to the OpenXC message format.

Example JSON command

{ “command”: “rtc_configuration”, “unix_time”: “1448551563”}

To set the RTC to the correct unix time on Windows Git Bash, Mac, or Linux shell, use the following command:

```
% TIME=$(date +%s); openxc-control set --time $TIME
```

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.

The payload format of the commands is specified in the OpenXC Message Format and will not be repeated here.

The tools from the OpenXC Python library can send all of these commands and are a good way to get started. Examples of the terminal commands to use are included here.

6.1.1 Simple Vehicle Message Writes

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

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

6.1.2 CAN Message Writes

The 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.

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

6.1.3 Diagnostic Requests

The diagnostic request 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.

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

6.1.4 Version Query

```
openxc-control version
```

6.1.5 Device ID Query

```
openxc-control id
```

6.1.6 Platform Query

```
openxc-control platform
```

6.1.7 Set CAN Message Passthrough Status

```
openxc-control set --passthrough --bus 1
```

6.1.8 Set CAN Acceptance Filter Bypass

```
openxc-control set --af-bypass --bus 1
```

6.1.9 Set Payload Format

```
openxc-control set --new-payload-format protobuf
```

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 `\0` 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 `\0` 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 `\0`.

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

Transfer request type: `0x83`

The VI accepts USB control requests on the standard endpoint 0 where the payload is a standard OpenXC message format command meessage (e.g. version, device ID, or diagnostic request, etc).

The responses are injected into the normal output data stream usig the same format as the *version query*, *device ID query*, etc.

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 `DEFAULT_LOGGING_OUTPUT` config option, 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
```

7.5 Functional Test Suite

For a complete functional test of the firmware code on the support hardware platforms, the repository includes an automated script to program a VI and run a self-test. The VIs are configured to self-receive every CAN message they send, allowing the complete firmware stack to be tested without any physical CAN configuration. The test script is written in Python and uses the OpenXC Python library send commands to the VI, read responses and receive the test vehicle data.

The functional test suite is currently supported on the:

- Ford Reference VI
- chipKIT VI

7.5.1 Running the Suite

Make sure you have the following connected to your development machine:

- Reference VI via USB
- USB JTAG programming connected to the refernece VI
- chipKIT micro-USB (for OpenXC interface)
- chipKIT mini-USB (for programming)

Then run the automated functional test suite:

```
vi-firmware/ $ fab auto_functional_test
```

which will program each of the VIs and run the tests on them independently. The process takes a couple of minutes.

7.5.2 Other Hardware Tests

`TEST_MODE_ONLY` is `1` build configuration enables user to test hardware functionality on the device such as the BLE radio, RTC and SD card. Supported on CrossChasm C5 hardware this option creates a firmware build that will check that hardware and glow an onboard LED depending on the result. The method for programming the test firmware is same as that for the default settings. If all tests were a sucess then the green LED will glow on the device. Failure will result in blinking of red LED after which it will glow continuously red. To repeat the test one has to unplug the device and power cycle the board.

```
Blink times = 1,  SD Card could not be mounted.
Blink times = 2,  RTC could not be mounted.
Blink times = 3,  Bluetooth low energy radio failed to initialize.
```

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.

8.4 Related Projects

8.4.1 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 Python page.

8.4.2 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 Android section of the OpenXC website.

CHAPTER 9

License Disclosure

The OpenXC vehicle interface firmware is an open source project, and in turn depends on a few other open source projects. If you are building from source, or have downloaded a pre-compiled binary firmware, the result may contain source code covered by the following licenses:

vi-firmware

Copyright (c) 2012 Ford Motor Company All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the <organization> nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL <COPYRIGHT HOLDER> BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

emhashmap

Copyright (c) 2013, Ford Motor Company All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

emlist

Copyright (c) 2013, Ford Motor Company All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

emqueue

Copyright (c) 2013, Ford Motor Company All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CON-

TRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

LPCUSB

LPCUSB, an USB device driver for LPC microcontrollers Copyright (C) 2006 Bertrik Sikken (bertrik@sikken.nl)

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Extended NXP Board Support Package

Copyright(C) NXP Semiconductors, 2011 All rights reserved.

Software that is described herein is for illustrative purposes only which provides customers with programming information regarding theLPC products. This software is supplied "AS IS" without any warranties of any kind, and NXP Semiconductors and its licensor disclaim any and all warranties, express or implied, including all implied warranties of merchantability, fitness for a particular purpose and non-infringement of intellectual property rights. NXP Semiconductors assumes no responsibility or liability for the use of the software, conveys no license or rights under any patent, copyright, mask work right, or any other intellectual property rights in or to any products. NXP Semiconductors reserves the right to make changes in the software without notification. NXP Semiconductors also makes no representation or warranty that such application will be suitable for the specified use without further testing or modification.

Permission to use, copy, modify, and distribute this software and its documentation is hereby granted, under NXP Semiconductors' and its licensor's relevant copyrights in the software, without fee, provided that it is used in conjunction with NXP Semiconductors microcontrollers. This copyright, permission, and disclaimer notice must appear in all copies of this code.

NXP Common Driver Library for LPCxxx

Copyright(C) 2012, NXP Semiconductor All rights reserved. Software that is described herein is for illustrative purposes only which provides customers with programming information regarding the products. This software is supplied "AS IS" without any warranties. NXP Semiconductors assumes no responsibility or liability for the use of the software, conveys no license or title under any patent, copyright, or mask work right to the product. NXP Semiconductors reserves the right to make changes in the software without notification. NXP Semiconductors also make no representation or warranty that such application will be suitable for the specified use without further testing or modification.

cJSON

cJSON is distributed under the MIT open source license. Copyright (c) 2009 Dave Gamble

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

nxpUSBlib

Copyright(C) NXP Semiconductors, 2012 All rights reserved.

Software that is described herein is for illustrative purposes only which provides customers with programming information regarding the LPC products. This software is supplied "AS IS" without any warranties of any kind, and NXP Semiconductors and its licensor disclaim any and all warranties, express or implied, including all implied warranties of merchantability, fitness for a particular purpose and non-infringement of intellectual property rights. NXP Semiconductors assumes no responsibility or liability for the use of the software, conveys no license or rights under any patent, copyright, mask work right, or any other intellectual property rights in or to any products. NXP Semiconductors reserves the right to make changes in the software without notification. NXP Semiconductors also makes no representation or warranty that such application will be suitable for the specified use without further testing or modification.

Permission to use, copy, modify, and distribute this software and its documentation is hereby granted, under NXP Semiconductors' and its licensor's relevant copyrights in the software, without fee, provided that it is used in conjunction with NXP Semiconductors microcontrollers. This copyright, permission, and disclaimer notice must appear in all copies of this code.

http-parser

http_parser.c is based on src/http/ngx_http_parse.c from NGINX copyright Igor Sysoev.

Additional changes are licensed under the same terms as NGINX and copyright Joyent, Inc. and other Node contributors. All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

md5

This software was written by Alexander Peslyak in 2001. No copyright is claimed, and the software is hereby placed in the public domain. In case this attempt to disclaim copyright and place the software in the public domain is deemed null and void, then the software is Copyright (c) 2001 Alexander Peslyak and it is hereby released to the general public under the following terms:

Redistribution and use in source and binary forms, with or without modification, are permitted.

There's ABSOLUTELY NO WARRANTY, express or implied.

(This is a heavily cut-down "BSD license".)

This differs from Colin Plumb's older public domain implementation in that no exactly 32-bit integer data type is required (any 32-bit or wider unsigned integer data type will do), there's no compile-time endianness configuration, and the function prototypes match OpenSSL's. No code from Colin Plumb's implementation has been reused; this comment merely compares the properties of the two independent implementations.

The primary goals of this implementation are portability and ease of use. It is meant to be fast, but not as fast as possible. Some known optimizations are not included to reduce source code size and avoid compile-time configuration.

cmp

The MIT License (MIT)

Copyright (c) 2014 Charles Gunyon

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

STBTLE Blue NRG support files from STMicroelectronics

By using this Licensed Software, You are agreeing to be bound by the terms and conditions of this License Agreement. Do not use the Licensed Software until You have read and agreed to the following terms and conditions. The use of the Licensed Software implies automatically the acceptance of the following terms and conditions. Please indicate your acceptance or NON-acceptance by selecting "I ACCEPT" or "I DO NOT ACCEPT" as indicated below in the media.

DEFINITIONS Licensed Software: means the enclosed BlueNRG, BlueNRG-MS DK and all the related documentations and design tools licensed in the form of object and/or source code as the case maybe.

Product: means a product or a system that includes or incorporates solely and exclusively an executable version of the Licensed Software and provided further that such Licensed Software executes solely and exclusively with BlueNRG, BlueNRG-MS network processor devices manufactured by or for ST.

LICENSE STMicroelectronics ("ST") grants You a non-exclusive, worldwide, non-transferable (whether by assignment or otherwise) non sub-licensable (except as expressly provided hereunder), revocable, royalty-free limited license to: (i) make copies, prepare derivatives works, of the source code version of the Licensed Software for the sole and exclusive purpose of developing executable versions of such Licensed Software only for use with the Product; (ii) make copies, prepare derivatives works of the object code versions of the Licensed Software for the sole purpose of designing, developing and manufacturing the Products;(iii) make, use, sell, offer to sell, import or otherwise distribute Products.

You have the right to grant sub-licenses of the rights set forth in this section in order to execute all of the rights set forth herein, provided that you will impose to the sub-licensee the same terms and conditions set forth in this Agreement and that you will remain fully liable to indemnify ST against all and any loss, liability, costs, damages, expenses (including the fees of lawyers and other professionals), suffered, incurred or sustained as a result of or in relation to the breach of the terms of this Agreement.

OWNERSHIP AND COPYRIGHT Title to the Licensed Software, related documentation and all copies thereof remain with ST and/or its licensors. You may not remove the copyrights notices from the Licensed Software and to any copies of the Licensed Software. You agree to prevent any unauthorized copying of the Licensed Software and related documentation.

RESTRICTIONS Unless otherwise explicitly stated in this Agreement, You may not sell, assign, sublicense, lease, rent or otherwise distribute the Licensed for commercial purposes, in whole or in part.

You acknowledge and agree that any use, adaptation translation or transcription of the Licensed Software or any portion or derivative thereof, for use with processors manufactured by or for an entity other than ST is a material breach of this Agreement and requires a separate license from ST.

No source code and/or object code relating to and/or based upon Licensed Software is to be made available by You unless expressly permitted under the section “License”.

You acknowledge and agree that the protection of the source code of the Licensed Software warrants the imposition of reasonable security precautions In the event ST demonstrates to You a reasonable belief that the source code of the Licensed Software has been used or distributed in violation of this Agreement, ST may by written notification request certification as to whether such unauthorized use or distribution has occurred. You shall cooperate and assist ST in its determination of whether there has been unauthorized use or distribution of the source code of the Licensed Software and will take appropriate steps to remedy any unauthorized use or distribution.

NO WARRANTY The Licensed Software is provided “as is” and “with all faults” without warranty of any kind expressed or implied. ST and its licensors expressly disclaim all warranties, expressed, implied or otherwise, including without limitation, warranty of merchantability, fitness for a particular purpose and non-infringement of intellectual property rights. ST does not warrant that the use in whole or in part of the Licensed Software will be interrupted or error free, will meet your requirements, or will operate with the combination of hardware and software selected by You.

You are responsible for determining whether the Licensed Software will be suitable for your intended use or application or will achieve your intended results.

ST has not authorized anyone to make any representation or warranty for the Licensed Software, and any technical, applications or design information or advice, quality characterization, reliability data or other services provided by ST shall not constitute any representation or warranty by ST or alter this disclaimer or warranty, and in no additional obligations or liabilities shall arise from ST’s providing such information or services. ST does not assume or authorize any other person to assume for it any other liability in connection with its Licensed Software.

Nothing contained in this Agreement will be construed as (i) a warranty or representation by ST to maintain production of any ST device or other hardware or software with which the Licensed Software may be used or to otherwise maintain or support the Licensed Software in any manner; and (ii)a commitment from ST and/or its licensors to bring or prosecute actions or suits against third parties for infringement of any of the rights licensed hereby, or conferring any rights to bring or prosecute actions or suits against third parties for infringement. However, ST has the right to terminate this Agreement immediately upon receiving notice of any claim, suit or proceeding that alleges that the Licensed Software or your use or distribution of the Licensed Software infringes any third party intellectual property rights.

All other warranties, conditions or other terms implied by law are excluded to the fullest extent permitted by law.

LIMITATION OF LIABILITIES In no event ST or its licensors shall be liable to You or any third party for any indirect, special, consequential, incidental , punitive damages or other damages (including but not limited to, the cost of labor, re-qualification, delay, loss of profits, loss of revenues, loss of data, costs of procurement of substitute goods or services or the like) whether based on contract, tort, or any other legal theory, relating to or in connection with

the Licensed Software, the documentation or this Agreement, even if ST has been advised of the possibility of such damages.

In no event shall ST's liability to You or any third party under this Agreement, including any claim with respect of any third party intellectual property rights, for any cause of action exceed 100 US$. This section does not apply to the extent prohibited by law. For the purposes of this section, any liability of ST shall be treated in the aggregate.

TERMINATION ST may terminate this license at any time if You are in breach of any of its terms and conditions. Upon termination, You will immediately destroy or return all copies of the software and documentation to ST.

APPLICABLE LAW AND JURISDICTION In case of dispute and in the absence of an amicable settlement, the only competent jurisdiction shall be the Courts of Geneva, Switzerland. The applicable law shall be the law of Switzerland.

SEVERABILITY If any provision of this agreement is or becomes, at any time or for any reason, unenforceable or invalid, no other provision of this agreement shall be affected thereby, and the remaining provisions of this agreement shall continue with the same force and effect as if such unenforceable or invalid provisions had not been inserted in this Agreement.

WAIVER The waiver by either party of any breach of any provisions of this Agreement shall not operate or be construed as a waiver of any other or a subsequent breach of the same or a different provision.

RELATIONSHIP OF THE PARTIES Nothing in this Agreement shall create, or be deemed to create, a partnership or the relationship of principal and agent or employer and employee between the Parties. Neither Party has the authority or power to bind, to contract in the name of or to create a liability for the other in any way or for any purpose

Microchip Library for Applications

License Rev. No. 01-101910

IMPORTANT: MICROCHIP IS WILLING TO LICENSE THE ACCOMPANYING SOFTWARE AND DOCUMENTATION TO YOU ONLY ON THE CONDITION THAT YOU ACCEPT ALL OF THE FOLLOWING TERMS. TO ACCEPT THE TERMS OF THIS LICENSE, CLICK "I ACCEPT" AND PROCEED WITH THE DOWNLOAD OR INSTALL. IF YOU DO NOT ACCEPT THESE LICENSE TERMS, CLICK "I DO NOT ACCEPT," AND DO NOT DOWNLOAD OR INSTALL THIS SOFTWARE. NON-EXCLUSIVE SOFTWARE LICENSE AGREEMENT FOR ACCOMPANYING MICROCHIP SOFTWARE AND DOCUMENTATION INCLUDING, BUT NOT LIMITED TO: GRAPHICS LIBRARY SOFTWARE, IrDA STACK SOFTWARE, MCHPFSUSB STACK SOFTWARE, MEMORY DISK DRIVE FILE SYSTEM SOFTWARE, mTOUCH™ CAPACITIVE LIBRARY SOFTWARE, PC PROGRAMS, SMART CARD LIBRARY SOFTWARE, TCP/IP STACK SOFTWARE, AND/OR MiWi™ DE SOFTWARE IMPORTANT - READ CAREFULLY.

This Nonexclusive Software License Agreement ("Agreement") is a contract between you, your heirs, successors and assigns ("Licensee") and Microchip Technology Incorporated, a Delaware corporation, with a principal place of business at 2355 W. Chandler Blvd., Chandler, AZ 85224-6199, and its subsidiary, Microchip Technology (Barbados) Incorporated (collectively, "Microchip") for the accompanying Microchip software including, but not limited to, Graphics Library Software, IrDA Stack Software, MCHPFSUSB Stack Software, Memory Disk Drive File System Software, mTouch™ Capacitive Library Software, PC programs, Smart Card Library Software, TCP/IP Stack Software, MiWi™ DE Software, and any updates thereto (collectively, the "Software"), and accompanying documentation, including images and any other graphic resources provided by Microchip ("Documentation"). The Software and Documentation are licensed under this Agreement and not sold. U.S. copyright laws, international copyright treaties, and other intellectual property laws and treaties protect the Software and Documentation. Microchip reserves all rights not expressly granted to Licensee in this Agreement. 1. License and Sublicense Grant. (a) Definitions. As used in this Agreement, the following capitalized terms will have the meanings defined below: (i) "Microchip Products" means Microchip microcontrollers, Microchip digital signal controllers, or other Microchip products that use or implement the Software. (ii) "Licensee Products" means Licensee products that use or incorporate Microchip Products. (iii) "Object Code" means the Software computer programming code that is in binary form (including related documentation, if any), and error corrections, improvements, modifications, and updates. (iv) "Source Code" means the Software computer programming code that may be printed out or displayed in human readable form (including related programmer comments and documentation, if any), and error corrections, improvements, modifications, and

updates. (v) "Third Party" means Licensee's agents, representatives, consultants, clients, customers, or contract manufacturers. (vi) "Third Party Products" means Third Party products that use or incorporate Microchip Products. (b) Software License Grant. Microchip grants strictly to Licensee a non-exclusive, non-transferable, worldwide license to use the Software, as described below: (i) Graphics Library Software, MCHPFSUSB Stack Software, Memory Disk Drive File System Software, mTouch™ Capacitive Library Software, Smart Card Library, TCP/IP Stack Software, and other accompanying Microchip software not otherwise expressly addressed in Section 1(b)(ii) or Section 1(b)(iii) below: License Rev. No. 01-101910 (1) Licensee may use, modify, copy, and distribute the Software identified in the subheading of this Section 1(b)(i) when such Software is embedded in a Microchip Product that is either integrated into Licensee Product or Third Party Product pursuant to Section 1(d) below. (2) Further, with respect to the TCP/IP Stack Software, Licensee may only port the ENC28J60.c, ENC28J60.h, ENCX24J600.c, and ENCX24J600.h driver source files to a non-Microchip device used in conjunction with a Microchip ethernet controller for the sole purpose of interfacing with such ethernet controller. (ii) IrDA Stack Software: (1) Licensee may install the IrDA Stack Software on a single computer and use with Microchip Products; and (2) Use, modify, copy, and distribute the device driver source files of the IrDA Stack Software ("Device Drivers"); provided that (x) such Device Drivers are only used with Microchip Products, and (y) no Open Source Components (defined in Section 2(c) below) are incorporated into such Device Drivers. (iii) PC Programs: (1) Licensee may install, use, modify (if Microchip provides Source Code), copy, and distribute the PC programs on an unlimited number of computers; provided that (x) such PC programs are used with Microchip products, and (y) the following conditions are met when Licensee redistributes any PC programs: "Copyright (c) Microchip Technology Inc. All rights reserved. Microchip PC programs are provided for your use with Microchip products only. Redistributions and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. * Neither the name of Microchip nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED BY MICROCHIP ''AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT WILL MICROCHIP BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE." (iv) MiWi™ DE Software: Licensee may use, modify, copy and distribute the Software only when embedded on a Microchip Product and used with either a Microchip radio frequency transceiver or UBEC UZ2400 radio frequency transceiver, which are integrated into Licensee Products or Third Party Products pursuant to Section 1(d) below. For purposes of clarity, Licensee may NOT embed the Software identified in Section 1(b)(i) (including derivatives or modifications thereof) on a non-Microchip Product, except as described in Section 1(b)(i)(2). Licensee may NOT use, License Rev. No. 01-101910 copy, modify or distribute the IrDA Stack Software, except as described in Section 1(b)(ii)(2) above (relating to Device Drivers) and Section 1(d) below (relating to sublicenses). Licensee may NOT distribute Source Code or Object Code of the Software on a standalone basis, except as described in Section 1(b)(iii) above (relating to PC programs) and Section 1(d) below (relating to sublicense rights). For the MiWi™ DE Software, Licensee may NOT embed the Software (including derivatives or modifications thereof) on a non-Microchip Product, use the Software with a radio frequency transceiver other than the Microchip and UBEC transceivers listed in Section 1(b)(iv), or distribute the Software (in Source Code or Object Code) except as described in Section 1(d) below. (c) Documentation License Grant. Microchip grants strictly to Licensee a non-exclusive, non-transferable, worldwide license to use the Documentation in support of Licensee's authorized use of the Software. (d) Sublicense Grants. Licensee may grant a sublicense to a Third Party to use the Software as described in this subparagraph (d) provided that Licensee first obtains such Third Party's agreement in writing to comply with the terms of this Agreement: (i) Graphics Library Software, MCHPFSUSB Stack Software, Memory Disk Drive File System Software, mTouch™ Capacitive Library Software, Smart Card Library, TCP/IP Stack Software, MiWi™ DE Software, and other accompanying Microchip software not otherwise expressly addressed in Section 1(b)(iii) above or Section 1(d)(ii) below: With respect to Software identified in this Section 1(d)(i), Third Party may: (1) Modify Source Code for Licensee. (2) Program Software into Microchip Products for Licensee. (3) Use the Software to develop

and/or manufacture Licensee Products. (4) Use the Software to develop and/or manufacture Third Party Products where either: (x) the sublicensed Software contains Source Code modified or otherwise optimized by Licensee for integration into Third Party Products; and/or (y) the sublicensed Software is programmed into Microchip Products by Licensee on behalf of such Third Party. (5) Use the Documentation in support of Third Party's authorized use of the Software in conformance with this Section 1(d)(i). (ii) IrDA Stack Software: With respect to IrDA Stack Software, Third Party may: (1) Modify ONLY Device Drivers for Licensee. (2) Program Software into Microchip Products for Licensee. (3) Use the Documentation in support of Third Party's authorized use of the Software in conformance with this Section 1(d)(ii). (e) Audit. Microchip's authorized representatives will have the right to reasonably inspect, announced or unannounced, and its sole and absolute discretion, Licensee's premises and to audit Licensee's records and inventory of Licensee Products, whether located on Licensee's premises or elsewhere at any time, in order to ensure Licensee's adherence to the terms of this Agreement. 2. Third Party Requirements; Open Source Components. Licensee acknowledges that it is Licensee's responsibility to comply with any third party license terms or requirements applicable to the use of such third party software, specifications, systems, or tools. Microchip is not responsible and will not be held responsible in any manner for Licensee's failure to comply with such applicable terms or requirements. (a) IrDA Stack Software. With respect to the IrDA Stack Software, Licensee further acknowledges that it is Licensee's responsibility to obtain a copy of, to familiarize itself fully with, and comply with the requirements of the Infrared Data Association, especially regarding the use of IrDA specifications and development of IrDA-compliant products. At the time of this Software release, it is Microchip's understanding that Licensee may obtain a copy of IrDA specifications from the IrDA website by either paying an access fee or becoming a member of the Infrared Data Association. For more information go to www.irda.org. License Rev. No. 01-101910 (b) Memory Disk Drive File System Software. With respect to the Memory Disk Drive File Systems Software, Licensee further acknowledges that it is Licensee's responsibility to obtain a copy of, familiarize itself fully with, and comply with the requirements and licensing obligations applicable to the use of flash-based media and FAT files systems available from Compact Flash Association, SD Card Association, Multi Media Card Association, and Microsoft Corporation. (c) Open Source Components. Notwithstanding the license grant in Section 1 above, Licensee further acknowledges that certain components of the Software may be covered by so-called "open source" software licenses ("Open Source Components"). Open Source Components means any software licenses approved as open source licenses by the Open Source Initiative or any substantially similar licenses, including without limitation any license that, as a condition of distribution of the software licensed under such license, requires that the distributor make the software available in source code format. To the extent required by the licenses covering Open Source Components, the terms of such license will apply in lieu of the terms of this Agreement, and Microchip hereby represents and warrants that the licenses granted to such Open Source Components will be no less broad than the license granted in Section 1. To the extent the terms of the licenses applicable to Open Source Components prohibit any of the restrictions in this Agreement with respect to such Open Source Components, such restrictions will not apply to such Open Source Component. For purposes of clarity: QT Compiled PC Program: The MCHPFSUSB Stack Software is accompanied by a certain PC program developed by Microchip that was developed using the QT 3 compiler ("QT Compiled PC Program"). Such QT 3 compiler is governed by the GNU General Public License version 2 ("GPL"). Accordingly, Microchip offers such QT Compiled PC Program, including Source Code for such program, under the terms of GPL version 2 as well. For copies of the GPL version 2, please go to http://www.fsf.org. JPEG Software: Independent JPEG Group's software and accompanying third party files including, but not limited to, GIF compressed format files are governed by the terms described in IJG's license. This license is posted in the "IJG License.pdf" file located at: <Installed Folder>MicrochipImage Decoders. (d) Google Static Maps API: Google Static Maps API and Google server usage are governed by the legal notices, terms and conditions posted on the Google website at www.google.com, including but not limited to the Static Maps API Developer Guide, the Legal Notices for Google Maps/Google Earth and Google Maps/Google Earth API, the Google Maps/Google Earth Terms of Service, and the Google Maps/Google Earth API Terms of Service. 3. Licensee Obligations. (a) Licensee will ensure Third Party compliance with the terms of this Agreement, and will be liable for any breach of this Agreement committed by such Third Party. (b) Licensee will not: (i) engage in unauthorized use, modification, disclosure or distribution of Software or Documentation, or its derivatives; (ii) use all or any portion of the Software, Documentation, or its derivatives except in conjunction with Microchip Products or Third Party Products; or (iii) reverse engineer (by disassembly, decompilation or otherwise) Software or any portion thereof. (c) Licensee may not remove or alter any Microchip copyright or other proprietary rights notice posted in any portion of the Software or Documentation. (d) Licensee will defend, indemnify and hold Microchip and its subsidiaries harmless from and against any and all claims, costs, damages, expenses (including reasonable attorney's fees), liabilities, and losses, including without limitation product liability claims, directly or indirectly arising from

or related to: (i) the use, modification, disclosure or distribution of the Software, Documentation, or any intellectual property rights related thereto; (ii) the use, sale and distribution of Licensee Products or Third Party Products; and (iii) breach of of this Agreement. THIS SECTION 3(d) STATES THE SOLE AND EXCLUSIVE LIABILITY OF THE PARTIES FOR INTELLECTUAL PROPERTY INFRINGEMENT. License Rev. No. 01-101910 4. Confidentiality. (a) Licensee agrees that the Software (including but not limited to the Source Code, Object Code and library files) and its derivatives, Documentation and underlying inventions, algorithms, know-how and ideas relating to the Software and the Documentation are proprietary information belonging to Microchip and its licensors ("Proprietary Information"). Except as expressly and unambiguously allowed herein, Licensee will hold in confidence and not use or disclose any Proprietary Information and will similarly bind its employees and Third Party(ies) in writing. Proprietary Information will not include information that: (i) is in or enters the public domain without breach of this Agreement and through no fault of the receiving party; (ii) the receiving party was legally in possession of prior to receiving it; (iii) the receiving party can demonstrate was developed by the receiving party independently and without use of or reference to the disclosing party's Proprietary Information; or (iv) the receiving party receives from a third party without restriction on disclosure. If Licensee is required to disclose Proprietary Information by law, court order, or government agency, such disclosure will not be deemed a breach of this Agreement provided that Licensee: (x) gives Microchip prompt notice of such requirement in order to allow Microchip to object or limit such disclosure; (y) cooperates with Microchip to protect Proprietary Information; and (z) Licensee complies with any protective order in place and discloses only the information required by process of law. (b) Licensee agrees that the provisions of this Agreement regarding unauthorized use and nondisclosure of the Software, Documentation and related Proprietary Rights are necessary to protect the legitimate business interests of Microchip and its licensors and that monetary damages alone cannot adequately compensate Microchip or its licensors if such provisions are violated. Licensee, therefore, agrees that if Microchip alleges that Licensee or Third Party has breached or violated such provision then Microchip will have the right to petition for injunctive relief, without the requirement for the posting of a bond, in addition to all other remedies at law or in equity. 5. Ownership of Proprietary Rights. (a) Microchip and its licensors retain all right, title and interest in and to the Software and Documentation ("Proprietary Rights") including, but not limited to: (i) patent, copyright, trade secret and other intellectual property rights in the Software, Documentation, and underlying technology; (ii) the Software as implemented in any device or system, including all hardware and software implementations of the Software technology (expressly excluding Licensee and Third Party code developed and used in conformance with this Agreement solely to interface with the Software and Licensee Products and/or Third Party Products); and (iii) all copies and derivative works thereof (by whomever produced). Further, copies and derivative works will be considered works made for hire with ownership vesting in Microchip on creation. To the extent such modifications and derivatives do not qualify as a "work for hire," Licensee hereby irrevocably transfers, assigns and conveys the exclusive copyright thereof to Microchip, free and clear of any and all liens, claims or other encumbrances, to the fullest extent permitted by law. Licensee and Third Party use of such modifications and derivatives is limited to the license rights described in Sections 1 and 2 above. (b) Licensee will have no right to sell, assign or otherwise transfer all or any portion of the Software, Documentation, or any related intellectual property rights except as expressly set forth in this Agreement. 6. Termination of Agreement. Without prejudice to any other rights, this Agreement terminates immediately, without notice by Microchip, upon a failure by Licensee or Third Party to comply with any provision of this Agreement. Further, Microchip may also terminate this Agreement upon reasonable belief that Licensee or Third Party failed to comply with this Agreement. Upon termination, Licensee and Third Party will immediately stop using the Software, Documentation, and derivatives thereof, and immediately destroy all such copies. 7. Warranties and Disclaimers. THE SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY, TITLE, NON-INFRINGEMENT AND FITNESS FOR A PARTICULAR PURPOSE. MICROCHIP AND ITS LICENSORS ASSUME NO RESPONSIBILITY FOR THE ACCURACY, RELIABILITY OR APPLICATION OF THE SOFTWARE OR DOCUMENTATION. MICROCHIP AND ITS LICENSORS DO NOT WARRANT THAT THE SOFTWARE WILL MEET REQUIREMENTS OF LICENSEE OR THIRD PARTY, BE UNINTERRUPTED OR ERROR-FREE. MICROCHIP AND ITS LICENSORS HAVE NO OBLIGATION TO CORRECT ANY DEFECTS IN THE SOFTWARE. LICENSEE AND THIRD PARTY ASSUME THE ENTIRE RISK ARISING OUT OF USE OR PERFORMANCE OF THE SOFTWARE AND DOCUMENTATION PROVIDED UNDER THIS AGREEMENT. License Rev. No. 01-101910 8. Limited Liability. IN NO EVENT WILL MICROCHIP OR ITS LICENSORS BE LIABLE OR OBLIGATED UNDER CONTRACT, NEGLIGENCE, STRICT LIABILITY, CONTRIBUTION, BREACH OF WARRANTY, OR OTHER LEGAL OR EQUITABLE THEORY FOR ANY DIRECT OR INDIRECT DAMAGES OR EXPENSES INCLUDING BUT NOT LIMITED TO INCIDENTAL,

SPECIAL, INDIRECT, PUNITIVE OR CONSEQUENTIAL DAMAGES, LOST PROFITS OR LOST DATA, COST OF PROCUREMENT OF SUBSTITUTE GOODS, TECHNOLOGY, SERVICES, OR ANY CLAIMS BY THIRD PARTIES (INCLUDING BUT NOT LIMITED TO ANY DEFENSE THEREOF), OR OTHER SIMILAR COSTS. The aggregate and cumulative liability of Microchip and its licensors for damages hereunder will in no event exceed $1000 or the amount Licensee paid Microchip for the Software and Documentation, whichever is greater. Licensee acknowledges that the foregoing limitations are reasonable and an essential part of this Agreement. 9. General. (a) Governing Law, Venue and Waiver of Trial by Jury. THIS AGREEMENT WILL BE GOVERNED BY AND CONSTRUED UNDER THE LAWS OF THE STATE OF ARIZONA AND THE UNITED STATES WITHOUT REGARD TO CONFLICTS OF LAWS PROVISIONS. Licensee agrees that any disputes arising out of or related to this Agreement, Software or Documentation will be brought in the courts of the State of Arizona. The parties agree to waive their rights to a jury trial in actions relating to this Agreement. (b) Attorneys' Fees. If either the Microchip or Licensee employs attorneys to enforce any rights arising out of or relating to this Agreement, the prevailing party will be entitled to recover its reasonable attorneys' fees, costs and other expenses. (c) Entire Agreement. This Agreement will constitute the entire agreement between the parties with respect to the subject matter hereof. It will not be modified except by a written agreement signed by an authorized representative of the Microchip. (d) Severability. If any provision of this Agreement will be held by a court of competent jurisdiction to be illegal, invalid or unenforceable, that provision will be limited or eliminated to the minimum extent necessary so that this Agreement will otherwise remain in full force and effect and enforceable. (e) Waiver. No waiver of any breach of any provision of this Agreement will constitute a waiver of any prior, concurrent or subsequent breach of the same or any other provisions hereof, and no waiver will be effective unless made in writing and signed by an authorized representative of the waiving party. (f) Export Regulation. Licensee agrees to comply with all export laws and restrictions and regulations of the Department of Commerce or other United States or foreign agency or authority. (g) Survival. The indemnities, obligations of confidentiality, and limitations on liability described herein, and any right of action for breach of this Agreement prior to termination, will survive any termination of this Agreement. (h) Assignment. Neither this Agreement nor any rights, licenses or obligations hereunder, may be assigned by Licensee without the prior written approval of Microchip except pursuant to a merger, sale of all assets of Licensee or other corporate reorganization, provided that assignee agrees in writing to be bound by the Agreement. (i) Restricted Rights. Use, duplication or disclosure by the United States Government is subject to restrictions set forth in subparagraphs (a) through (d) of the Commercial Computer-Restricted Rights clause of FAR 52.227-19 when applicable, or in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, and in similar clauses in the NASA FAR Supplement. Contractor/manufacturer is Microchip Technology Inc., 2355 W. Chandler Blvd., Chandler, AZ 85225-6199. If Licensee has any questions about this Agreement, please write to Microchip Technology Inc., 2355 W. Chandler Blvd., Chandler, AZ 85224-6199 USA. ATTN: Marketing. Copyright © 2010 Microchip Technology Inc. All rights reserved. License Rev. No. 01-101910

CHAPTER 10

License

Copyright (c) 2012-2016 Ford Motor Company

Licensed under the BSD license.

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