Porting an Arduino library to CircuitPython: VL6180X Distance Sensor

Created by Tony DiCola

https://learn.adafruit.com/porting-an-arduino-library-to-circuitpython-vl6180x-distance-sensor

Last updated on 2023-08-29 03:27:27 PM EDT
# Table of Contents

## Overview

## Hardware
- Parts
- Wiring

## Datasheet & Registers
- Datasheet
- I2C Registers
- Accessing Registers

## Template
- cookiecutter Install
- CircuitPython Template

## Driver Code

## Simplify Driver
- Initializer Simplification
- Properties

## Optimizations
- Internal Constant Values
- Memory Usage

## Gotchas
Overview

So you're up and running with a nifty CircuitPython board, but what can you connect to it to control hardware? There are lots of guides on drivers that are ported from Arduino to CircuitPython, but what if you have hardware with no CircuitPython driver yet? If you're willing to get your hands a bit dirty you can actually convert Arduino code to CircuitPython and Python without a lot of trouble. Both languages support the same basic hardware access, and in fact with Python you might even find the driver code is simpler and easier to write and use. This guide will explore how to port an Arduino driver for the VL6180X distance sensor to CircuitPython. We'll walk through the entire process of converting and testing the Python driver code.

Before you get started you'll want to be aware that this is an advanced tutorial. You'll want to be familiar with the structure and use of Arduino libraries and Arduino code in general. You'll also want to be familiar with Python code, like the basics of control, loops, etc. In addition it will greatly help to be familiar with CircuitPython code for talking to hardware--check out these guides:

- CircuitPython Basics: Digital I/O
- CircuitPython Basics: Analog I/O
- CircuitPython Basics: I2C & SPI

Continue on to learn about the hardware used in this guide!
Parts

You'll need the following parts to follow this guide:

CircuitPython board. This guide focuses on the ESP8266 (I) and Feather M0/SAMD21-based boards (http://adafruit.it/2772), but any CircuitPython board that supports I2C should work.

If your board doesn't come with CircuitPython running on it already then check out your board's guide for how to load CircuitPython firmware. For example the Feather M0 express guide () is a good reference.

VL6180X time-of-flight distance sensor. () This is a handy sensor that uses a laser to detect the distance of an object in front of it (up to a few inches away). For this guide you'll see how to take code written for Arduino to talk to this sensor and convert it to a module for CircuitPython.
Breadboard (http://adafru.it/64) and jumper wires (http://adafru.it/153). These are useful to prototype and connect the sensor to your board.

Soldering tools (http://adafru.it/136). You'll need to solder headers to the boards. Check out the guide to excellent soldering () if you're new to soldering.

Before you get started you'll want to follow the VL6180X product guide () to assemble the board (just solder on a row of headers). In addition it's highly recommended to follow the guide and test the board with its Arduino-based examples (). This will help you confirm the board is assembled correctly and able to communicate with your board. It's much easier to troubleshoot issues with communication in isolation like this before you get deep into porting the code!

Wiring

Like the VL6180X guide mentions you'll want to wire it to your board using a standard I2C connection with a clock, data, power, and ground connection:

- VL6180X SCL to board SCL / I2C clock.
• VL6180X SDA to board SDA / I2C data.
• VL6180X VIN to board 3.3V power.
• VL6180X ground/GND to board ground/GND.

Once you have the hardware assembled and wired up, continue on to learn how to read the datasheet and start interacting with registers on the device.

Datasheet & Registers

Datasheet

Before you try porting or creating a library for hardware you'll need to get very familiar with how the hardware works. Typically you'll need to download the device's datasheet which is a document that describes in very great detail how the hardware works. Grab the VL6180X datasheet below:

VL6180X Datasheet

Datasheets can be somewhat intimidating, but don't worry you don't usually need to read or understand the entire document in excruciating detail (but it can't hurt!). First start by skimming the document and you'll see high level sections that most datasheets contain:

• Overview / Summary - Typically at the start datasheets will have a high level description of the device, important specifications, and how it communicates. For this device a very important thing to note in the overview is that this device uses the I2C protocol for communication. As a driver creator you'll need to understand what protocol is used to communicate with the device--typically I2C, SPI, serial, or even an analog or digital I/O signal. For this guide and example we'll focus on I2C as it's a very common protocol for sensors.
• Description - After the overview there's usually a deeper description of how the device functions. This is important to read as it helps you understand the capabilities and behavior of the device.
• Performance - Most sensors and devices have certain physical characteristics, like the precision and accuracy of their measurements or limits on environmental factors like temperature. This is an important section to read if you're really getting deep into the usage of a device. However if you're just porting existing code for a board that's already built you might not need to focus on these details.
• I2C / Protocol Specification - Typically there’s a section in the datasheet that describes the communication protocol for the device. For this sensor the I2C section describes how to talk to the device using the I2C protocol. Other devices might describe different protocols like SPI, serial, etc. For an I2C device be sure to watch for uncommon behavior, like if the device mentions it needs clock stretching or repeated stop bits. Some hardware doesn’t support these uncommon I2C protocol behaviors and can be more difficult to port or work with. Save yourself some future headaches by looking for these characteristics from the start! In addition be sure to look for the I2C address of a device as you’ll need to know it to talk to it later. For the VL6180X its datasheet shows the I2C communication is fairly typical and doesn’t use clock stretching, repeated stops, or other uncommon behavior. In addition the address of the device is 0x29.

• Electrical Characteristics - This section describes electrical characteristics and limits for the device. It’s good to skim this section and look for potential areas of concern like if the device works with 3.3V or 5V, how it behaves across a range of voltage inputs, etc. As a driver writer you’re probably less interested in this information, but it’s good to skim for any warnings or gotchas.

• Package Information - This is the least useful section for you as a driver writer. Typically the mechanical and physical descriptions of the device’s packages are found here. Things like CAD and mechanical drawings of the footprint, layout, etc. of the hardware. Since you’re probably using a breakout that was already designed to hold the board you’re less interested in this information and can skim or skip it.

I2C Registers

For an I2C device like the VL6180X you’ll typically find the device exposes information through registers. These are just like registers in your computer’s processor—they’re locations in memory that store data, like an integer or fractional number. Most datasheets have an entire table or section devoted to describing the registers of the device, for example the VL6180X has:
<table>
<thead>
<tr>
<th>Offset</th>
<th>Register name</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000</td>
<td>IDENTIFICATION_MODEL_ID</td>
<td>Section 6.2.1 on page 51</td>
</tr>
<tr>
<td>0x001</td>
<td>IDENTIFICATION_MODEL_REV_MAJOR</td>
<td>Section 6.2.2 on page 51</td>
</tr>
<tr>
<td>0x002</td>
<td>IDENTIFICATION_MODEL_REV_MINOR</td>
<td>Section 6.2.3 on page 51</td>
</tr>
<tr>
<td>0x003</td>
<td>IDENTIFICATION_MODULE_REV_MAJOR</td>
<td>Section 6.2.4 on page 52</td>
</tr>
<tr>
<td>0x004</td>
<td>IDENTIFICATION_MODULE_REV_MINOR</td>
<td>Section 6.2.5 on page 52</td>
</tr>
<tr>
<td>0x006</td>
<td>IDENTIFICATION_DATE_HI</td>
<td>Section 6.2.6 on page 52</td>
</tr>
<tr>
<td>0x007</td>
<td>IDENTIFICATION_DATE_LO</td>
<td>Section 6.2.7 on page 53</td>
</tr>
<tr>
<td>0x008:0x009</td>
<td>IDENTIFICATION_TIME</td>
<td>Section 6.2.8 on page 53</td>
</tr>
<tr>
<td>0x010</td>
<td>SYSTEM_MODE_GPIO0</td>
<td>Section 6.2.9 on page 54</td>
</tr>
<tr>
<td>0x011</td>
<td>SYSTEM_MODE_GPIO1</td>
<td>Section 6.2.10 on page 55</td>
</tr>
<tr>
<td>0x012</td>
<td>SYSTEM_HISTORY_CTRL</td>
<td>Section 6.2.11 on page 56</td>
</tr>
<tr>
<td>0x014</td>
<td>SYSTEM_INTERRUPT_CONFIG_GPIO</td>
<td>Section 6.2.12 on page 57</td>
</tr>
<tr>
<td>0x015</td>
<td>SYSTEM_INTERRUPT_CLEAR</td>
<td>Section 6.2.13 on page 57</td>
</tr>
<tr>
<td>0x016</td>
<td>SYSTEM_FRESH_OUT_OF_RESET</td>
<td>Section 6.2.14 on page 58</td>
</tr>
<tr>
<td>0x017</td>
<td>SYSTEM_GROUPED_PARAMETER_HOLD</td>
<td>Section 6.2.15 on page 58</td>
</tr>
<tr>
<td>0x018</td>
<td>SYSRANGE_START</td>
<td>Section 6.2.16 on page 59</td>
</tr>
<tr>
<td>0x019</td>
<td>SYSRANGE_THRESH_HIGH</td>
<td>Section 6.2.17 on page 59</td>
</tr>
<tr>
<td>0x01A</td>
<td>SYSRANGE_THRESH_LOW</td>
<td>Section 6.2.18 on page 60</td>
</tr>
<tr>
<td>0x01B</td>
<td>SYSRANGE_INTERMEASUREMENT_PERIOD</td>
<td>Section 6.2.19 on page 60</td>
</tr>
<tr>
<td>0x01C</td>
<td>SYSRANGE_MAX_CONVERGENCE_TIME</td>
<td>Section 6.2.20 on page 60</td>
</tr>
<tr>
<td>0x01E</td>
<td>SYSRANGE_CROSSTALK_COMPENSATION_RATE</td>
<td>Section 6.2.21 on page 61</td>
</tr>
<tr>
<td>0x021</td>
<td>SYSRANGE_CROSSTALK_VALID_HEIGHT</td>
<td>Section 6.2.22 on page 61</td>
</tr>
<tr>
<td>0x022</td>
<td>SYSRANGE_EARLY_CONVERGENCE_ESTIMATE</td>
<td>Section 6.2.23 on page 61</td>
</tr>
<tr>
<td>0x024</td>
<td>SYSRANGE_PART_TO_PART_RANGE_OFFSET</td>
<td>Section 6.2.24 on page 62</td>
</tr>
<tr>
<td>0x025</td>
<td>SYSRANGE_RANGE_IGNORE_VALIDATE</td>
<td>Section 6.2.25 on page 62</td>
</tr>
<tr>
<td>0x026</td>
<td>SYSRANGE_RANGE_IGNORE_THRESHOLD</td>
<td>Section 6.2.26 on page 62</td>
</tr>
<tr>
<td>0x02C</td>
<td>SYSRANGE_MAX_AMBIENT_LEVEL_MULTI</td>
<td>Section 6.2.27 on page 63</td>
</tr>
<tr>
<td>0x02D</td>
<td>SYSRANGE_RANGE_CHECK_ENABLES</td>
<td>Section 6.2.27 on page 63</td>
</tr>
<tr>
<td>0x02E</td>
<td>SYSRANGE_VHV_RECALIBRATE</td>
<td>Section 6.2.29 on page 64</td>
</tr>
<tr>
<td>0x031</td>
<td>SYSRANGE_VHV_REPEAT_RATE</td>
<td>Section 6.2.30 on page 64</td>
</tr>
<tr>
<td>0x038</td>
<td>SYSAI5S_START</td>
<td>Section 6.2.31 on page 65</td>
</tr>
<tr>
<td>0x03A</td>
<td>SYSAI5S_THRESH_HIGH</td>
<td>Section 6.2.32 on page 65</td>
</tr>
<tr>
<td>0x03C</td>
<td>SYSAI5S_THRESH_LOW</td>
<td>Section 6.2.33 on page 66</td>
</tr>
</tbody>
</table>

You can see this table lists the offset, or address, of each register and then a description and link to more details about it. A simple register like the IDENTIFICATION_MODEL_ID at the top has an offset of 0x000 (be careful to note these addresses are 16-bit and not just 8-bit!) and links to more details about its contents:

### 6.2.1 IDENTIFICATION_MODEL_ID

<table>
<thead>
<tr>
<th>Address: 0x000</th>
<th>Type: R/W</th>
<th>Reset: 0xB4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong></td>
<td>[7:0] Identification_model_id: Device model identification number. 0xB4 = VL6180X</td>
<td></td>
</tr>
</tbody>
</table>
By reading the register description you can understand the data exposed by the chip. In this case the model ID register contains a byte of data (8 bits) with a model identification number inside. According to the datasheet this ID should be the value 0xB4.

**Accessing Registers**

As a quick test let's try reading the register to confirm we see this value. Before you get started be sure to read the [CircuitPython I2C guide](https://circuitpython.org/tutorial/i2c) for details on using the I2C protocol with CircuitPython. Then with the VL6180X wired to your connect to your board's serial REPL and initialize the I2C bus with code:

```python
import board
import busio as io
i2c = io.I2C(board.SCL, board.SDA)
```

Note if your board doesn't support hardware I2C (like the ESP8266) you'll need to use the bitbangio module instead:

```python
import board
import bitbangio as io
i2c = io.I2C(board.SCL, board.SDA)
```

Next let's scan for device addresses to make sure we see the VL6180X at the expected address (0x29). Like the I2C guide mentions you can run code like this to scan for devices and print their address in hex:

```python
while not i2c.try_lock():
    pass
[hex(x) for x in i2c.scan()]
```

```plaintext
>>> while not i2c.try_lock():
...    pass
...
>>> [hex(x) for x in i2c.scan()]
['0x29']
```

If everything is connected and working correctly you should see the expected device address printed, 0x29. If you don't see the address or see a different value carefully check your wiring and solder connections!

Now let's read the model ID register, again following the [code from the I2C guide](https://circuitpython.org/tutorial/i2c) but modifying it to specify register address 0x000 (from the register table above):
i2c.writeto(0x29, bytes([0x00, 0x00]), stop=False)
result = bytearray(1)
i2c.readfrom_into(0x29, result)
print(hex(result[0]))

Success! Notice the value 0xb4 is printed and matches the value we expect to see from the datasheet. If you notice in the code that read the register value the `i2c.writeto` function is used to request the register to read--in this case the device with address 0x29 (the VL6180X sensor) and register 0x000. Be careful to note for this device register addresses take two bytes since they are 16-bit values, hence the list [0x00, 0x00] to represent the 2 bytes of register address 0x000. The number of bytes to read from the register are specified by the size of the buffer that will hold results, in this case a bytearray of size 1 (i.e. one byte).

Finally like the I2C guide mentions there’s an easier way to talk to I2C devices using the Adafruit bus device module and its `I2CDevice` class. Follow the I2C guide to see how to install this module on your board (it does not currently come built in to CircuitPython builds and must be installed separately). Then connect to the board and initialize I2C as before, but now run this to create an `I2CDevice` instance for the VL6180X:

```python
from adafruit_bus_device.i2c_device import I2CDevice
device = I2CDevice(i2c, 0x29)
```

Now you can use the device class in a context manager to simplify the locking, unlocking and access to registers. For example to read the model ID register again you can run:

```python
with device:
    device.write(bytes([0x00, 0x00]), stop=False)
    result = bytearray(1)
    device.read_into(result)
    print(hex(result[0]))
```

©Adafruit Industries
Again you should see the model ID 0xb4 printed, but this time notice how the code to talk to the device is simpler. Instead of sending the device address and locking/unlocking the I2C bus you can use the I2CDevice class to do this automatically. When you port over a library it will greatly help to use the bus device module and its I2C and SPI device classes!

Now that you've seen how to talk to the device with CircuitPython continue on to learn about using a template to create some of the boilerplate code for a module.

---

**Template**

When creating a module for CircuitPython you typically want to include information like documentation, continuous integration build scripts, etc. which can be a lot of boilerplate code to create and copy. Luckily there's a handy tool called [cookiecutter](https://cookiecutter.org/) that uses a template to bootstrap the creation of a driver. Follow these steps to install and use cookiecutter to start creating a VL6180X driver.

### cookiecutter Install

First you'll need to make sure you have Python and cookiecutter installed. Download and install Python from [python.org](https://python.org) (be sure you select to have the pip package manager installed too and Python added to your system path if installing on Windows). Then in a command prompt use pip (either pip if using Python 2.7.x or pip3 if using Python 3.x) to install cookiecutter:

```
pip install cookiecutter
```

Note on some systems like mac OSX or Debian/Ubuntu Linux you might need to run the command above as root with sudo. If you run into trouble installing cookiecutter be sure to read its documentation.

### CircuitPython Template

Once cookiecutter is installed you can use it to bootstrap the creation of a driver and its boilerplate code. You'll use this special cookiecutter template that's made for creating CircuitPython drivers. From a terminal run:

```
cookiecutter gh:adafruit/cookiecutter-adafruit-circuitpython
```
You should see cookiecutter run and prompt you for information about the driver you're creating. For the VC6180X you can use information like:

```
library_name: VL6180x
depends_on_bus_device []: True
depends_on_register []:
author: ladyada
github_user [adafruit]:
library_prefix []: adafruit
company []: Adafruit Industries
```

Notice we specified True for the `depends_on_bus_device` question. This means cookiecutter will do extra work to include the dependency on the bus device module automatically. Since we saw in the previous page that bus device simplifies access to the I2C device we'll be sure to include it here.

The `depends_on_register` question we left blank to specify false, or no dependency, on the register module. The `register module` provides some handy abstractions for accessing 8-bit device registers but we aren't using it here because the VL6180X uses 16-bit register addresses and isn't complex enough to need the extra dependency. You might examine the register module and consider using it when writing your own drivers for other devices!

After the cookiecutter template finishes running you should see a `vl6180x` directory created, and inside will be files like:

```
cookiecutter created these files and populated them with information to bootstrap the driver creation process. Here's what you'll find in the files:

- `.travis.yml` - This is a configuration for the [Travis-CI build system](https://www.travis-ci.com) to automatically generate `.mpy` files for code stored on Adafruit's GitHub account. Open this file and carefully read the comments to see how to setup Travis CI (specifically setup steps you need to follow on Travis' website to enter your Github access token). Setting up Travis-CI is optional and this file can be ignored if you don't want automatic .mpy generation for Github releases.
- `adafruit_vl6180x.py` - This is the main source file for your driver code. The cookiecutter template will create this by taking the name of your library and appending `adafruit_` to it so it matches the convention for other Adafruit
libraries. When users import your library they'll do so using the exact name of this file.

- api.rst - This is the API documentation that will automatically be generated by Read The Docs / sphinx tools. You don't typically need to add or change anything here—the comments and docstrings from your source file will be parsed and turned into documentation automatically.

- CODE_OF_CONDUCT.md - This is populated with a good code of conduct for potential contributors to your library.

- conf.py - This is a configuration for the sphinx documentation engine used by Read The Docs. You don't need to change anything here.

- LICENSE - By default this is populated with the MIT license.

- README.rst - A small readme is automatically generated to match the format of other drivers. You should fill in the introduction and usage section TODOs before publishing the code.

- readthedocs.yml - This is a configuration file that Read The Docs uses to check that a repository has documentation it should process. You don't need to change this file.

- requirements.txt - This is a Python package configuration file that lists the dependencies of the driver. Although CircuitPython / MicroPython don't use this file, tools like Read The Docs will use it to determine what packages need to be installed to build documentation. You don't need to change this file.

In the next section we'll start filling in the code for adafruit_vl6180x.py by converting from Arduino to Python & CircuitPython code.

Driver Code

Now that you have a template for starting the driver let's fill in the code to convert from the Arduino library to a Python & CircuitPython module. First open the adafruit_vl6180x.py created in the previous step and notice it's populated with license information and a documentation template, but otherwise blank and ready to be populated with Python code:

```python
# The MIT License (MIT)
#
# Copyright (c) 2017 Tony DiCola for Adafruit Industries
#
# 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.
#```

©Adafruit Industries
Adding a description is a good idea as it will be parsed by tools like Read The Docs to generate documentation:

```
`adafruit_vl6180x`

This is a CircuitPython driver for the VL6180X time-of-flight distance sensor.

* Author(s): Tony DiCola
```

Now for the actual code, to start you'll want to very carefully review the Arduino library you're porting. For the [VL6180X library](https://arduinolibrary.com) it's a simple Arduino library that's defined by a single header and cpp file. The header file defines register addresses and other values, but most importantly the interface for the VL6180X class:

```
class Adafruit_VL6180X {
  public:
    Adafruit_VL6180X();
    boolean begin(void);
    uint8_t readRange(void);
    float readLux(uint8_t gain);
    uint8_t readRangeStatus(void);

  private:
    void loadSettings(void);
    void write8(uint16_t address, uint8_t data);
    void write16(uint16_t address, uint16_t data);
    uint16_t read16(uint16_t address);
    uint8_t read8(uint16_t address);
    uint8_t _i2caddr;
};
```

You can almost directly convert this to a stub of a Python class:

```
class Adafruit_VL6180X:
    def __init__(self):
        pass
```
def begin(self):
    return True

def read_range(self):
    return 0

def read_lux(self, gain):
    return 0

def read_range_status(self):
    return 0

def _load_settings(self):
    pass

def _write_8(self, address, data):
    pass

def _write_16(self, address, data):
    pass

def _read_8(self, address):
    return 0

def _read_16(self, address):
    return 0

All of the same functions as the Arduino library are created but slightly changed to more closely file Python's PEP8 style conventions (underscores to separate words instead of camel casing, a _ prefix to denote 'private' functions).

There are also a set of #define values at the top of the header which we'll most likely need to include in our driver. Here's the Arduino code:

```c
#define VL6180X_DEFAULT_I2C_ADDR 0x29
#define VL6180X_REG_IDENTIFICATION_MODEL_ID 0x000
#define VL6180X_REG_SYSTEM_INTERRUPT_CONFIG 0x014
#define VL6180X_REG_SYSTEM_INTERRUPT_CLEAR 0x015
#define VL6180X_REG_SYSTEM_FRESH_OUT_OF_RESET 0x016
#define VL6180X_REG_SYSRANGE_START 0x018
#define VL6180X_REG_SYSALS_START 0x038
#define VL6180X_REG_SYSALS_ANALOGUE_GAIN 0x03F
#define VL6180X_REG_SYSALS_INTEGRATION_PERIOD_HI 0x040
#define VL6180X_REG_SYSALS_INTEGRATION_PERIOD_LO 0x041
#define VL6180X_REG_RESULT_ALS_VAL 0x050
#define VL6180X_REG_RESULT_RANGE_VAL 0x062
#define VL6180X_REG_RESULT_RANGE_STATUS 0x04d
#define VL6180X_REG_RESULT_INTERRUPT_STATUS_GPIO 0x04f
#define VL6180X_ALS_GAIN_1 0x06
#define VL6180X_ALS_GAIN_1_25 0x05
#define VL6180X_ALS_GAIN_1_67 0x04
#define VL6180X_ALS_GAIN_2_5 0x03
#define VL6180X_ALS_GAIN_5 0x02
#define VL6180X_ALS_GAIN_10 0x01
#define VL6180X_ALS_GAIN_20 0x00
#define VL6180X_ALS_GAIN_40 0x07
#define VL6180X_ERROR_NONE 0
#define VL6180X_ERROR_SYSERR_1 1
#define VL6180X_ERROR_SYSERR_5 5
```

©Adafruit Industries
Python doesn't have a concept of compile-time defines since it's an interpreted language. This means each of these defines should be created as variables--typically module-level variables with all capitol names (to follow PEP8 convention). Be aware that each of these variables will take a small amount of memory so ideally you specify them as const in case the variables can be better optimized.

Here's how the Python version of these values would look, and they should be placed before the class definition:

```python
VL6180X_DEFAULT_I2C_ADDR = const(0x29)
VL6180X_REG_IDENTIFICATION_MODEL_ID = const(0x000)
VL6180X_REG_SYSTEM_INTERRUPT_CONFIG = const(0x014)
VL6180X_REG_SYSTEM_INTERRUPT_CLEAR = const(0x015)
VL6180X_REG_SYSTEM_FRESH_OUT_OF_RESET = const(0x016)
VL6180X_REG_SYSRANGE_START = const(0x018)
VL6180X_REG_SYSALS_START = const(0x038)
VL6180X_REG_SYSALS_ANALOGUE_GAIN = const(0x03F)
VL6180X_REG_SYSALS_INTEGRATION_PERIOD_HI = const(0x040)
VL6180X_REG_SYSALS_INTEGRATION_PERIOD_LO = const(0x041)
VL6180X_REG_RESULT_ALS_VAL = const(0x050)
VL6180X_REG_RESULT_RANGE_VAL = const(0x062)
VL6180X_REG_RESULT_RANGE_STATUS = const(0x04d)
VL6180X_REG_RESULT_INTERRUPT_STATUS_GPIO = const(0x04f)
VL6180X_REG_RESULT_INTERRUPT_STATUS_GPIO_B = const(0x0f)
VL6180X_ALS_GAIN_1 = const(0x06)
VL6180X_ALS_GAIN_1_25 = const(0x05)
VL6180X_ALS_GAIN_1_67 = const(0x04)
VL6180X_ALS_GAIN_2_5 = const(0x03)
VL6180X_ALS_GAIN_5 = const(0x02)
VL6180X_ALS_GAIN_10 = const(0x01)
VL6180X_ALS_GAIN_20 = const(0x00)
VL6180X_ALS_GAIN_40 = const(0x07)
VL6180X_ERROR_NONE = const(0)
VL6180X_ERROR_SYSERR_1 = const(1)
VL6180X_ERROR_SYSERR_5 = const(5)
VL6180X_ERROR_ECEFAIL = const(6)
VL6180X_ERROR_NOCONVERGE = const(7)
VL6180X_ERROR_RANGEIGNORE = const(8)
VL6180X_ERROR_SNR = const(11)
VL6180X_ERROR_RAWUFLOW = const(12)
VL6180X_ERROR_RAWOFLOW = const(13)
VL6180X_ERROR_RANGEUFLOW = const(14)
VL6180X_ERROR_RANGEOFLOW = const(15)
```

```python
class Adafruit_VL6180X:
...
```
Now let's start filling in function implementations. We'll start with the initializer which is like the constructor in Arduino/C++ code. Code in the initializer is responsible for creating the object and setting its internal state to a default mode. If you look at the constructor in the Arduino library you'll see it doesn't actually need to do anything:

```c
Adafruit_VL6180X::Adafruit_VL6180X(void) {
}
```

With Arduino it can access the I2C bus from global objects and doesn't need to keep a reference to the bus. This is in contrast to CircuitPython and MicroPython where you typically need to keep a unique reference to the bus. The initializer is a good place to do this, and in particular an I2CDevice instance from the bus device module can be created. The typical convention is for a driver to take in a reference to the I2C bus as a parameter and optionally any specific or overridden address value (some devices allow modifying their I2C address).

Modify the Python initializer to look like:

```python
def __init__(self, i2c, address=VL6180X_DEFAULT_I2C_ADDR):
    self._device = I2CDevice(i2c, address)
```

And since you're using the I2CDevice class and its associated module be sure to add an import at the top of the file after the comments:

```python
from adafruit_bus_device.i2c_device import I2CDevice
```

For many Arduino libraries they simplify access to registers with explicit read and write functions, like the write8 and read8 functions in the VL6180X library. Let's start by filling in those functions first as they're the core functions used by all other functions to interact with the device. The Arduino versions look like:

```c
// Read 1 byte from the VL6180X at 'address'
uint8_t Adafruit_VL6180X::read8(uint16_t address) {
    uint8_t data;
    Wire.beginTransmission(_i2caddr);
    Wire.write(address>>8);
    Wire.write(address);
    Wire.endTransmission();
    Wire.requestFrom(_i2caddr, (uint8_t)1);
    uint8_t r = Wire.read();
    #if defined(I2C_DEBUG)
    Serial.print("\t$"); Serial.print(address, HEX); Serial.print(": 0x");
    Serial.println(r, HEX);
    #endif
    return r;
}
```
// Read 2 byte from the VL6180X at 'address'
uint16_t Adafruit_VL6180X::read16(uint16_t address)
{
    uint16_t data;

    Wire.beginTransmission(_i2caddr);
    Wire.write(address>>8);
    Wire.write(address);
    Wire.endTransmission();

    Wire.requestFrom(_i2caddr, (uint8_t)2);
    while(!Wire.available());
    data = Wire.read();
    data <<= 8;
    while(!Wire.available());
    data |= Wire.read();

    return data;
}

// write 1 byte
void Adafruit_VL6180X::write8(uint16_t address, uint8_t data)
{
    Wire.beginTransmission(_i2caddr);
    Wire.write(address>>8);
    Wire.write(address);
    Wire.write(data);
    Wire.endTransmission();

    #if defined(I2C_DEBUG)
    Serial.print("\t$"); Serial.print(address, HEX); Serial.print(" = 0x");
    Serial.println(data, HEX);
    #endif
}

// write 2 bytes
void Adafruit_VL6180X::write16(uint16_t address, uint16_t data)
{
    Wire.beginTransmission(_i2caddr);
    Wire.write(address>>8);
    Wire.write(address);
    Wire.write(data>>8);
    Wire.write(data);
    Wire.endTransmission();

You'll want to be a bit familiar with Arduino's I2C interface to understand this code, but at a high level the comments describe what's happening in each function. For a given 16-bit register address (remember the VL6180X is somewhat special and uses these larger addresses) one or two bytes of data are read or written to the device. We can convert these to Python code using the I2CDevice class as follows:

```python
def _write_8(self, address, data):
    # Write 1 byte of data from the specified 16-bit register address.
    with self._device:
        self._device.write(bytes([(address >> 8) & 0xFF, address & 0xFF, data]))

def _write_16(self, address, data):
    # Write a 16-bit big endian value to the specified 16-bit register
```
The Python implementations all rely on the I2CDevice to perform low level reads and writes, just like you saw manually at the REPL in the beginning of this guide. However one big difference with Python and Arduino code is that numeric and byte values in Python don’t have types and you sometimes need to manipulate bytes to pack and unpack large values.

For example all the registers of the VL6180X are specified as 16-bit big endian values (the most significant byte first). A number like 0x123 in Python has to be converted into two bytes using bit shifts and masks like:

```python
>>> value = 0x123
>>> high_byte = (value >> 8) & 0xFF
>>> low_byte = value & 0xFF
>>> hex(high_byte)
'0x1'
>>> hex(low_byte)
'0x23'
```

The masking operations (& 0xFF) are very important as they tell Python you mean to convert from its arbitrarily long numeric values to a single byte value. If you miss this step your code will at best throw an error that a non-byte value was found and at worst silently fail with a completely unexpected (usually negative) value!

Likewise when reading a 16-bit big-endian value from the VL6180X it has to be packed back into a single numeric value with bitshifts and boolean operations:
Another option for performing these conversions is to use the `struct` module, however it adds some complexity and memory usage that you might not desire. For a simple case of packing or unpacking a few 16-bit values doing it in place with boolean operations and shifts as above is the easiest option.

With the core read and write operations implemented let's move on to implementing the other functions. The begin function is typically where initialization happens and looks like:

```c
boolean Adafruit_VL6180X::begin(void) {
  _i2caddr = VL6180X_DEFAULT_I2C_ADDR;
  Wire.begin();
  if (read8(VL6180X_REG_IDENTIFICATION_MODEL_ID) != 0xB4) {
    return false;
  }
  //if (read8(VL6180X_REG_SYSTEM_FRESH_OUT_OF_RESET) == 0x01) {
    loadSettings();
  //}
  write8(VL6180X_REG_SYSTEM_FRESH_OUT_OF_RESET, 0x00);
  return true;
}
```

This is easy to convert to Python using the `_read_8` and `_write_8` functions we filled in previously:

```python
def begin(self):
    """Initialize access to the sensor. Returns True if successful.""
    if self._read_8(VL6180X_REG_IDENTIFICATION_MODEL_ID) != 0xB4:
      return False
    self._load_settings()
    self._write_8(VL6180X_REG_SYSTEM_FRESH_OUT_OF_RESET, 0x00)
    return True
```

We don't need to save the I2C address or initialize the I2C/Wire module since that was done in the Python class' initializer when it created the I2CDevice instance.

Notice also a docstring was added with a small description of what the function does. This string will be read by the documentation engine and automatically added to API docs! In Python it's a best practice to add docstrings to all public/user-facing functions as it creates the core of user documentation.

Let's fill in `_load_settings` next, first look at the Arduino version:
void Adafruit_VL6180X::loadSettings(void) {
    // load settings!
    // private settings from page 24 of app note
    write8(0x0207, 0x01);
    write8(0x0208, 0x01);
    write8(0x0096, 0x00);
    write8(0x0097, 0xfd);
    write8(0x00e3, 0x00);
    write8(0x00e4, 0x04);
    write8(0x00e5, 0x02);
    write8(0x00e6, 0x01);
    write8(0x00e7, 0x03);
    write8(0x00f5, 0x02);
    write8(0x00d9, 0x05);
    write8(0x00db, 0xce);
    write8(0x00dc, 0x03);
    write8(0x00dd, 0xf8);
    write8(0x009f, 0x00);
    write8(0x00a3, 0x3c);
    write8(0x00b7, 0x00);
    write8(0x00bb, 0x3c);
    write8(0x00b2, 0x09);
    write8(0x00ca, 0x09);
    write8(0x0098, 0x01);
    write8(0x001b0, 0x17);
    write8(0x001ad, 0x00);
    write8(0x00ff, 0x05);
    write8(0x0100, 0x05);
    write8(0x0199, 0x05);
    write8(0x01a6, 0x1b);
    write8(0x0030, 0x00);
    // Recommended : Public registers - See data sheet for more detail
    write8(0x0011, 0x10);  // Enables polling for 'New Sample ready'
    // when measurement completes
    write8(0x010a, 0x30);  // Set the averaging sample period
    // (compromise between lower noise and
    // increased execution time)
    write8(0x003f, 0x46);  // Sets the light and dark gain (upper
    // nibble). Dark gain should not be
    // changed.
    write8(0x0031, 0xFF);  // sets the # of range measurements after
    // which auto calibration of system is
    // performed
    write8(0x0040, 0x63);  // Set ALS integration time to 100ms
    write8(0x002e, 0x01);  // perform a single temperature calibration
    // of the ranging sensor
    // Optional: Public registers - See data sheet for more detail
    write8(0x001b, 0x24);  // Configures interrupt on 'New Sample
    // Ready threshold event'
}

This is all just a set of _write_8 calls that we can directly convert to Python:

```python
def _load_settings(self):
    # private settings from page 24 of app note
    self._write_8(0x0207, 0x01)
    self._write_8(0x0208, 0x01)```
Now we can fill in the functions people will use, like read_range. First the Arduino version:

```c
uint8_t Adafruit_VL6180X::readRange(void) {
  // wait for device to be ready for range measurement
  while (! (read8(VL6180X_REG_RESULT_RANGE_STATUS) & 0x01));

  // Start a range measurement
  write8(VL6180X_REG_SYSRANGE_START, 0x01);

  // Poll until bit 2 is set
  while (! (read8(VL6180X_REG_RESULT_INTERRUPT_STATUS_GPIO) & 0x04));

  // read range in mm
  return read8(VL6180X_REG_RESULT_RANGE_MM);}
```
```c
uint8_t range = read8(VL6180X_REG_RESULT_RANGE_VAL);

// clear interrupt
write8(VL6180X_REG_SYSTEM_INTERRUPT_CLEAR, 0x07);

return range;
```

Nothing looks particularly complicated with this code, it's just a few loops waiting for a specific register result and then read and write calls to manipulate the registers. The Python version would look like:

```python
def read_range(self):
    """Read the range of an object in front of sensor and return it in mm."""
    # wait for device to be ready for range measurement
    while not (self._read_8(VL6180X_REG_RESULT_RANGE_STATUS) & 0x01):
        pass
    # Start a range measurement
    self._write_8(VL6180X_REG_SYSRANGE_START, 0x01);
    # Poll until bit 2 is set
    while not (self._read_8(VL6180X_REG_RESULT_INTERRUPT_STATUS_GPIO) & 0x04):
        pass
    # read range in mm
    range_ = self._read_8(VL6180X_REG_RESULT_RANGE_VAL)
    # clear interrupt
    self._write_8(VL6180X_REG_SYSTEM_INTERRUPT_CLEAR, 0x07)
    return range_
```

The Python code is almost exactly like the Arduino code—they both just use the underlying read and write functions to talk to the hardware. The only gotcha is to be careful of variables with names like range as those are built-in Python function names and might cause problems. The typical convention in Python is to add an underscore to denote the variable as a local instance vs. the built-in function or object.

We can go on to read_range_status, a very simple one register read. In Arduino:

```c
uint8_t Adafruit_VL6180X::readRangeStatus(void) {
    return (read8(VL6180X_REG_RESULT_RANGE_STATUS) >> 4);
}
```

And the Python version:

```python
def read_range_status(self):
    """Retrieve the status/error from a previous range read."""
    return self._read_8(VL6180X_REG_RESULT_RANGE_STATUS) >> 4
```

Finally the read_lux function, first the Arduino version:

```c
float Adafruit_VL6180X::readLux(uint8_t gain) {
    uint8_t reg;
    reg = read8(VL6180X_REG_SYSTEM_INTERRUPT_CONFIG);
    reg &==~ 0x38;
```
reg |= (0x4 << 3); // IRQ on ALS ready
write8(VL6180X_REG_SYSTEM_INTERRUPT_CONFIG, reg);

// 100 ms integration period
write8(VL6180X_REG_SYSALS_INTEGRATION_PERIOD_HI, 0);
write8(VL6180X_REG_SYSALS_INTEGRATION_PERIOD_LO, 100);

// analog gain
if (gain > VL6180X_ALS_GAIN_40) {
    gain = VL6180X_ALS_GAIN_40;
}
write8(VL6180X_REG_SYSALS_ANALOGUE_GAIN, 0x40 | gain);

// start ALS
write8(VL6180X_REG_SYSALS_START, 0x1);

// Poll until "New Sample Ready threshold event" is set
while (4 != ((read8(VL6180X_REG_RESULT_INTERRUPT_STATUS_GPIO) >> 3) & 0x7));

// read lux!
float lux = read16(VL6180X_REG_RESULT_ALS_VAL);

// clear interrupt
write8(VL6180X_REG_SYSTEM_INTERRUPT_CLEAR, 0x07);

lux *= 0.32; // calibrated count/lux
switch(gain) {
    case VL6180X_ALS_GAIN_1:
        break;
    case VL6180X_ALS_GAIN_1_25:
        lux /= 1.25;
        break;
    case VL6180X_ALS_GAIN_1_67:
        lux /= 1.76;
        break;
    case VL6180X_ALS_GAIN_2_5:
        lux /= 2.5;
        break;
    case VL6180X_ALS_GAIN_5:
        lux /= 5;
        break;
    case VL6180X_ALS_GAIN_10:
        lux /= 10;
        break;
    case VL6180X_ALS_GAIN_20:
        lux /= 20;
        break;
    case VL6180X_ALS_GAIN_40:
        lux /= 20;
        break;
}
lux *= 100;
lux /= 100; // integration time in ms
return lux;

There's a bit more going on in this function, but if you carefully read it you'll see it's only using the basic read and write functions with a bit of math and other bit operations. A Python version would look like:

```python
def read_lux(self, gain):
    """Read the lux (light value) from the sensor and return it."
    """
reg = self._read_8(VL6180X_REG_SYSTEM_INTERRUPT_CONFIG)
reg &= ~0x38
reg |= (0x4 << 3) # IRQ on ALS ready
self._write_8(VL6180X_REG_SYSTEM_INTERRUPT_CONFIG, reg)
# 100 ms integration period
self._write_8(VL6180X_REG_SYSALS_INTEGRATION_PERIOD_HI, 0)
self._write_8(VL6180X_REG_SYSALS_INTEGRATION_PERIOD LO, 100)
# analog gain
if gain &gt; VL6180X_ALS_GAIN_40:
    gain = VL6180X_ALS_GAIN_40
self._write_8(VL6180X_REG_SYSALS_ANALOGUE_GAIN, 0x40 | gain)
# start ALS
self._write_8(VL6180X_REG_SYSALS_START, 0x1)
# Poll until "New Sample Ready threshold event" is set
while 4 != ((self._read_8(VL6180X_REG_RESULT_INTERRUPT_STATUS_GPIO)
            >> 3) & 0x7):
    pass
# read lux!
lux = self._read_16(VL6180X_REG_RESULT_ALS_VAL)
# clear interrupt
self._write_8(VL6180X_REG_SYSTEM_INTERRUPT_CLEAR, 0x07);lux *= 0.32 # calibrated count/lux
if gain == VL6180X_ALS_GAIN_1:
    pass
elif gain == VL6180X_ALS_GAIN_1_25:
    lux /= 1.25
elif gain == VL6180X_ALS_GAIN_1_67:
    lux /= 1.76
elif gain == VL6180X_ALS_GAIN_2_5:
    lux /= 2.5
elif gain == VL6180X_ALS_GAIN_5:
    lux /= 5
elif gain == VL6180X_ALS_GAIN_10:
    lux /= 10
elif gain == VL6180X_ALS_GAIN_20:
    lux /= 20
elif gain == VL6180X_ALS_GAIN_40:
    lux /= 20
lux *= 100
lux /= 100 # integration time in ms
return lux

Not much changes, however notice Python doesn't support switch statements and a
cascade of if and elif statements is necessary.

Phew, that's it as far as the porting of codes goes! For reference here's the complete
adafruit_vl6180x.py file:

# The MIT License (MIT)
#
# Copyright (c) 2017 Tony DiCola for Adafruit Industries
#
# 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,
This is a CircuitPython driver for the VL6180X time-of-flight distance sensor.

* Author(s): Tony DiCola

from adafruit_bus_device.i2c_device import I2CDevice

VL6180X_DEFAULT_I2C_ADDR = const(0x29)

VL6180X_REG_IDENTIFICATION_MODEL_ID = const(0x000)
VL6180X_REG_SYSTEM_INTERRUPT_CONFIG = const(0x014)
VL6180X_REG_SYSTEM_INTERRUPT_CLEAR = const(0x015)
VL6180X_REG_SYSTEM_FRESH_OUT_OF_RESET = const(0x016)
VL6180X_REG_SYSRANGE_START = const(0x018)
VL6180X_REG_SYSALS_START = const(0x038)
VL6180X_REG_SYSALS_ANALOGUE_GAIN = const(0x03F)
VL6180X_REG_SYSALS_INTEGRATION_PERIOD_HI = const(0x040)
VL6180X_REG_SYSALS_INTEGRATION_PERIOD_LO = const(0x041)
VL6180X_REG_RESULT_ALS_VAL = const(0x050)
VL6180X_REG_RESULT_RANGE_VAL = const(0x062)
VL6180X_REG_RESULT_INTERRUPT_STATUS_GPIO = const(0x04f)

VL6180X_ALS_GAIN_1 = const(0x06)
VL6180X_ALS_GAIN_1_25 = const(0x05)
VL6180X_ALS_GAIN_1_67 = const(0x04)
VL6180X_ALS_GAIN_2_5 = const(0x03)
VL6180X_ALS_GAIN_5 = const(0x02)
VL6180X_ALS_GAIN_10 = const(0x01)
VL6180X_ALS_GAIN_20 = const(0x00)
VL6180X_ALS_GAIN_40 = const(0x07)

VL6180X_ERROR_NONE = const(0)
VL6180X_ERROR_SYSERR_1 = const(1)
VL6180X_ERROR_SYSERR_5 = const(5)
VL6180X_ERROR_ECEFAIL = const(6)
VL6180X_ERROR_NOCONVERGE = const(7)
VL6180X_ERROR_RANGEIGNORE = const(8)
VL6180X_ERROR_SNR = const(11)
VL6180X_ERROR_RAWUFLOW = const(12)
VL6180X_ERROR_RAWOFLOW = const(13)
VL6180X_ERROR_RANGEUFLOW = const(14)
VL6180X_ERROR_RANGEOFLOW = const(15)

class Adafruit_VL6180X:
    """
    Initialize access to the sensor. Returns True if successful."""
    def __init__(self, i2c, address=VL6180X_DEFAULT_I2C_ADDR):
        self._device = I2CDevice(i2c, address)
        self._load_settings()
        self._write_8(VL6180X_REG_SYSTEM_FRESH_OUT_OF_RESET, 0x00)
        return True

    def begin(self):
        """Initialize access to the sensor. Returns True if successful."""
        if self._read_8(VL6180X_REG_IDENTIFICATION_MODEL_ID) != 0xB4:
            return False
        return True

    def read_range(self):
        """Read the range of an object in front of sensor and return it in mm."""

©Adafruit Industries
# wait for device to be ready for range measurement
while not (self._read_8(VL6180X_REG_RESULT_RANGE_STATUS) & 0x01):
    pass

# Start a range measurement
self._write_8(VL6180X_REG_SYSRANGE_START, 0x01);

# Poll until bit 2 is set
while not (self._read_8(VL6180X_REG_RESULT_INTERRUPT_STATUS_GPIO) & 0x04):
    pass

# read range in mm
range_ = self._read_8(VL6180X_REG_RESULT_RANGE_VAL)

# clear interrupt
self._write_8(VL6180X_REG_SYSTEM_INTERRUPT_CLEAR, 0x07)

return range_

def read_lux(self, gain):
    """Read the lux (light value) from the sensor and return it.""

    reg = self._read_8(VL6180X_REG_SYSTEM_INTERRUPT_CONFIG)
    reg &= ~0x38
    reg |= (0x4 << 3) # IRQ on ALS ready

    self._write_8(VL6180X_REG_SYSTEM_INTERRUPT_CONFIG, reg)

    # 100 ms integration period
    self._write_8(VL6180X_REG_SYSALS_INTEGRATION_PERIOD_HI, 0)
    self._write_8(VL6180X_REG_SYSALS_INTEGRATION_PERIOD_LO, 100)

    # analog gain
    if gain &gt; VL6180X_ALS_GAIN_40:
        gain = VL6180X_ALS_GAIN_40
        self._write_8(VL6180X_REG_SYSALS_ANALOGUE_GAIN, 0x40 | gain)
    
    # start ALS
    self._write_8(VL6180X_REG_SYSALS_START, 0x1)

    # Poll until "New Sample Ready threshold event" is set
    while 4 != ((self._read_8(VL6180X_REG_RESULT_INTERRUPT_STATUS_GPIO) &lt;&lt; 3) & 0x7):
        pass

    # read lux!
    lux = self._read_16(VL6180X_REG_RESULT_ALS_VAL)

    # clear interrupt
    self._write_8(VL6180X_REG_SYSTEM_INTERRUPT_CLEAR, 0x07);

    lux *= 0.32 # calibrated count/lux

    if gain == VL6180X_ALS_GAIN_1:
        pass
    elif gain == VL6180X_ALS_GAIN_1_25:
        lux /= 1.25
    elif gain == VL6180X_ALS_GAIN_1_67:
        lux /= 1.76
    elif gain == VL6180X_ALS_GAIN_2_5:
        lux /= 2.5
    elif gain == VL6180X_ALS_GAIN_5:
        lux /= 5
    elif gain == VL6180X_ALS_GAIN_10:
        lux /= 10
    elif gain == VL6180X_ALS_GAIN_20:
        lux /= 20
    elif gain == VL6180X_ALS_GAIN_40:
        lux /= 20
    lux *= 100
    lux /= 100 # integration time in ms
    return lux

def read_range_status(self):
    """Retrieve the status/error from a previous range read.""
    return self._read_8(VL6180X_REG_RESULT_RANGE_STATUS) &gt;&gt; 4

def _load_settings(self):
    # private settings from page 24 of app note
    self._write_8(0x0207, 0x01)
    self._write_8(0x0208, 0x01)
    self._write_8(0x0096, 0x00)
    self._write_8(0x0097, 0xFD)
def _write_8(self, address, data):
    # Write 1 byte of data from the specified 16-bit register address.
    with self._device:
        self._device.write(bytes([(address >> 8) & 0xFF, address & 0xFF, data]))

def _write_16(self, address, data):
    # Write a 16-bit big endian value to the specified 16-bit register address.
    with self._device:
        self._device.write(bytes([(address >> 8) & 0xFF, address & 0xFF, (data >> 8) & 0xFF, data & 0xFF]))

def _read_8(self, address):
    # Read and return a byte from the specified 16-bit register address.
    with self._device:
Let's try testing it out on the hardware. Copy the adafruit_vl6180x.py to your board's filesystem. Then connect to the REPL and initialize I2C as you did before:

```python
import board
import busio
i2c = busio.I2C(board.SCL, board.SDA)
```

Now import the module you created and try creating an instance of the Adafruit_VL6180X class. Remember you need to pass it the I2C bus instance:

```python
import adafruit_vl6180x
vl6180x = adafruit_vl6180x.Adafruit_VL6180X(i2c)
```

Note if your driver is very large or complex you might run into an error that your board ran out of memory. Unfortunately memory is a big constraint with small microcontrollers and it can be tricky to optimize and reduce memory usage. One simple thing to do is to convert your driver to a .mpy file () and try loading it again. Remember every time you update the driver code you must regenerate and copy over the new .mpy file!

Next remember you should call begin to initialize the library and verify it can talk to the chip. If begin succeeds it will return True:

```python
vl6180x.begin()
```

Now try reading the range in millimeters:
You should see a value of 255 if something is too far away for the sensor to detect (it only can read up to about 6 inches in front of itself). As you move an object closer or further away try calling the function again to see the value change, for example moving a hand close and then further away results in values like:

```python
>>> vl6180x.read_range()
255
>>> vl6180x.read_range()
12
>>> vl6180x.read_range()
56
>>> vl6180x.read_range()
76
>>> vl6180x.read_range()
115
>>> vl6180x.read_range()
136
>>> vl6180x.read_range()
183
>>> vl6180x.read_range()
```

You can read the status and lux values too:

```python
vl6180x.read_range_status()
```

```python
>>> vl6180x.read_range_status()
0
```

```python
vl6180x.read_lux(adafruit_vl6180x.VL6180X_ALS_GAIN_1)
```

```python
>>> vl6180x.read_lux(adafruit_vl6180x.VL6180X_ALS_GAIN_1)
233.6
```

Notice you need to pass a gain value to the read_lux function. The global variables at the top of the module are what you can use in place of the #define values typically used by Arduino libraries. Remember to reference these values with the module name preceding them!

That's all there is to the basics of porting an Arduino library to Python and CircuitPython. Continue on to the next page to learn how to simplify the Python code so it's more friendly and easy to use.

**Simplify Driver**

In the previous section we saw how to directly convert the VL6180X Arduino library into a version that works as a CircuitPython module. You can follow those steps to get most libraries ported over and working with Python code. However you might want to go further and simplify the driver a bit so it's more friendly and easy to use.
with Python code. In this section we'll go through a few small simplifications that make using the driver easier.

First remember the CircuitPython driver code:

```python
# The MIT License (MIT)
#
# Copyright (c) 2017 Tony DiCola for Adafruit Industries
#
# 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.
#
""" 
'adafruit_vl6180x'
================================================================

This is a CircuitPython driver for the VL6180X time-of-flight distance sensor.

* Author(s): Tony DiCola

```
VL6180X_ERROR_NOCONVERGE = const(7)
VL6180X_ERROR_RANGEIGNORE = const(8)
VL6180X_ERROR_SNR = const(11)
VL6180X_ERROR_RAWUFLOW = const(12)
VL6180X_ERROR_RAWOFLOW = const(13)
VL6180X_ERROR_RANGEUFLOW = const(14)
VL6180X_ERROR_RANGEOFLOW = const(15)

class Adafruit_VL6180X:
    def __init__(self, i2c, address=VL6180X_DEFAULT_I2C_ADDR):
        self._device = I2CDevice(i2c, address)
    
    def begin(self):
        """Initialize access to the sensor. Returns True if successful."""
        if self._read_8(VL6180X_REG_IDENTIFICATION_MODEL_ID) != 0xB4:
            return False
        self._load_settings()
        self._write_8(VL6180X_REG_SYSTEM_FRESH_OUT_OF_RESET, 0x00)
        return True

    def read_range(self):
        """Read the range of an object in front of sensor and return it in mm."""
        while not (self._read_8(VL6180X_REG_RESULT_RANGE_STATUS) & 0x01):
            pass
        self._write_8(VL6180X_REG_SYSRANGE_START, 0x01)
        while not (self._read_8(VL6180X_REG_RESULT_INTERRUPT_STATUS_GPIO) & 0x04):
            pass
        range_ = self._read_8(VL6180X_REG_RESULT_RANGE_VAL)
        self._write_8(VL6180X_REG_SYSTEM_INTERRUPT_CLEAR, 0x07)
        return range_

    def read_lux(self, gain):
        """Read the lux (light value) from the sensor and return it."
        reg = self._read_8(VL6180X_REG_SYSTEM_INTERRUPT_CONFIG)
        reg &= ~0x38
        if gain > VL6180X_ALS_GAIN_40:
            gain = VL6180X_ALS_GAIN_40
            self._write_8(VL6180X_REG_SYSALS_ANALOGUE_GAIN, 0x40 | gain)
        self._write_8(VL6180X_REG_SYSALS_INTEGRATION_PERIOD_HI, 0)
        self._write_8(VL6180X_REG_SYSALS_INTEGRATION_PERIOD_LO, 100)
        self._write_8(VL6180X_REG_SYSALS_START, 0x1)
        while 4 != ((self._read_8(VL6180X_REG_RESULT_INTERRUPT_STATUS_GPIO) >> 3) & 0x7):
            pass
        lux = self._read_16(VL6180X_REG_RESULT_ALS_VAL)
        lux *= 0.32 # calibrated count/lux
        if gain == VL6180X_ALS_GAIN_1:
            pass
        elif gain == VL6180X_ALS_GAIN_1_25:
            lux /= 1.25
        elif gain == VL6180X_ALS_GAIN_1_67:
            lux /= 1.76
elif gain == VL6180X_ALS_GAIN_2_5:
    lux /= 2.5
elif gain == VL6180X_ALS_GAIN_5:
    lux /= 5
elif gain == VL6180X_ALS_GAIN_10:
    lux /= 10
elif gain == VL6180X_ALS_GAIN_20:
    lux /= 20
elif gain == VL6180X_ALS_GAIN_40:
    lux /= 20
lux *= 100
lux /= 100 # integration time in ms
return lux

def read_range_status(self):
    """Retrieve the status/error from a previous range read."""
    return self._read_8(VL6180X_REG_RESULT_RANGE_STATUS) >> 4

def _load_settings(self):
    # private settings from page 24 of app note
    self._write_8(0x0207, 0x01)
    self._write_8(0x0208, 0x01)
    self._write_8(0x0096, 0x00)
    self._write_8(0x0097, 0xfd)
    self._write_8(0x00e3, 0x00)
    self._write_8(0x00e4, 0x04)
    self._write_8(0x00e5, 0x02)
    self._write_8(0x00e6, 0x01)
    self._write_8(0x00e7, 0x03)
    self._write_8(0x00e8, 0x00f5, 0x02)
    self._write_8(0x00e9, 0x00d9, 0x05)
    self._write_8(0x00eb, 0x00db, 0xce)
    self._write_8(0x00ed, 0x00dc, 0x03)
    self._write_8(0x00ee, 0x00dd, 0xf8)
    self._write_8(0x00ef, 0x009f, 0x00)
    self._write_8(0x00f0, 0x00a3, 0x3c)
    self._write_8(0x00f1, 0x00b7, 0x00)
    self._write_8(0x00f2, 0x00bb, 0x3c)
    self._write_8(0x00f3, 0x00b2, 0x09)
    self._write_8(0x00f4, 0x00ca, 0x09)
    self._write_8(0x00f5, 0x0019, 0x00)
    self._write_8(0x00f6, 0x001b, 0x17)
    self._write_8(0x00f7, 0x001ad, 0x00)
    self._write_8(0x00f8, 0x00ff, 0x05)
    self._write_8(0x00f9, 0x0100, 0x05)
    self._write_8(0x00fa, 0x0199, 0x05)
    self._write_8(0x00fb, 0x01a6, 0x1b)
    self._write_8(0x00fc, 0x01ac, 0x3e)
    self._write_8(0x00fd, 0x01a7, 0x1f)
    self._write_8(0x00fe, 0x0030, 0x00)
    # Recommended : Public registers - See data sheet for more detail
    self._write_8(0x0011, 0x10)   # Enables polling for 'New Sample ready'
    # when measurement completes
    self._write_8(0x001a, 0x30)   # Set the averaging sample period
    # (compromise between lower noise and
    # increased execution time)
    self._write_8(0x003f, 0x46)   # Sets the light and dark gain (upper
    # nibble). Dark gain should not be
    # changed.
    self._write_8(0x0031, 0xFF)   # sets the # of range measurements after
    # which auto calibration of system is
    # performed
    self._write_8(0x0040, 0x63)   # Set ALS integration time to 100ms
    self._write_8(0x002e, 0x01)   # perform a single temperature calibration
    # of the ranging sensor
    # Optional: Public registers - See data sheet for more detail
    self._write_8(0x001b, 0x09)   # Set default ranging inter-measurement
    # period to 100ms
self._write_8(0x003e, 0x31)   # Set default ALS inter-measurement period 
          # to 500ms
self._write_8(0x0014, 0x24)   # Configures interrupt on 'New Sample 
          # Ready threshold event'

def _write_8(self, address, data):
    # Write 1 byte of data from the specified 16-bit register address.
    with self._device:
        self._device.write(bytes([((address &gt;&gt; 8) &amp; 0xFF, 
                    address &amp; 0xFF, 
                    data)])

def _write_16(self, address, data):
    # Write a 16-bit big endian value to the specified 16-bit register
    # address.
    with self._device:
        self._device.write(bytes([((address &gt;&gt; 8) &amp; 0xFF, 
                       address &amp; 0xFF, 
                       (data &gt;&gt; 8) &amp; 0xFF, 
                       data &amp; 0xFF)])

def _read_8(self, address):
    # Read and return a byte from the specified 16-bit register address.
    with self._device:
        self._device.write(bytes([((address &gt;&gt; 8) &amp; 0xFF, 
                       address &amp; 0xFF, 
                       0, 
                       0]),
                              stop=False)
        result = bytearray(1)
        self._device.read_into(result)
        return result[0]

def _read_16(self, address):
    # Read and return a 16-bit unsigned big endian value read from the
    # specified 16-bit register address.
    with self._device:
        self._device.write(bytes([((address &gt;&gt; 8) &amp; 0xFF, 
                       address &amp; 0xFF, 
                       0, 
                       0]),
                              stop=False)
        result = bytearray(2)
        self._device.read_into(result)
        return (result[0] &lt;&lt; 8) | result[1]

Initializer Simplification

One easy thing you can do to simplify the Python driver is combine the initializer (the
__init__ function in the class) and the begin function. In Arduino begin is typically
used because constructors (the equivalent of an initializer in Arduino/C++) can be
problematic when they talk to hardware and might fail in some way. If a failure occurs
in a constructor it's hard to respond with an error--typically you need to throw an
exception, but some Arduino or C++ code might not handle the exception well!

Luckily in Python it's easy to manage and catch exceptions so throwing them in an
initializer is a good option. This means we can combine both the begin and __init__
function:

def __init__(self, i2c, address=VL6180X_DEFAULT_I2C_ADDR):
    self._device = I2CDevice(i2c, address)
    if self._read_8(VL6180X_REG_IDENTIFICATION_MODEL_ID) != 0xB4:

©Adafruit Industries
raise RuntimeError('Could not find VL6180X, is it connected and powered?')
self._load_settings()
self._write_8(VL6180X_REG_SYSTEM_FRESH_OUT_OF_RESET, 0x00)

Notice the initializer will now both create the I2CDevice (the core of communication with the sensor) and now do a few checks to verify the chip is present and load default settings. If an error occurs, like when verifying the chip model ID register value, now an exception is thrown back to the user. This way when the user’s Python code creates the VL6180X instance it will immediately see an exception thrown if there’s a problem communicating with the chip. More importantly though the user doesn’t need to call begin anymore--there’s one less function to learn about and call!

Properties

Properties in Python are functions on a class that you interact with as if they were attributes--that is to say instead of using parenthesis to tell Python you’re calling code, you just reference the property by name and Python knows to call some code internally. Another great simplification for your driver is to look where you can turn functions into properties, both to read and write data.

For example the read_range function looks like this right now when a user calls it:

```python
>>> vl6180x.read_range()
163
```

However as a property you can expose the range like this:

```python
>>> vl6180x.range
163
```

Notice there’s no need to call the code as a function, instead you just refer to the range property on the object.

If you haven’t used them be sure to read more about properties to understand how and why to use them.

For this driver we can convert a couple functions into properties, specifically read_range and read_range_status. For the lux value it’s a little trickier because it needs to be told what gain to use as a parameter. Properties aren’t like functions and can’t be given parameters, so the lux function is a good case where keeping it as a function that takes the necessary gain parameter is the best option.

For read_range change its function to look like this to expose it as a property:
@property
def range(self):
    """Read the range of an object in front of sensor and return it in mm."""
    # wait for device to be ready for range measurement
    while not (self._read_8(VL6180X_REG_RESULT_RANGE_STATUS) & 0x01):
        pass
    # Start a range measurement
    self._write_8(VL6180X_REG_SYSRANGE_START, 0x01);
    # Poll until bit 2 is set
    while not (self._read_8(VL6180X_REG_RESULT_INTERRUPT_STATUS_GPIO) &
               0x04):
        pass
    # read range in mm
    range_ = self._read_8(VL6180X_REG_RESULT_RANGE_VAL)
    # clear interrupt
    self._write_8(VL6180X_REG_SYSTEM_INTERRUPT_CLEAR, 0x07)
    return range_

Notice none of the code inside the function changed, only the definition. The name
was changed from read_range to just range--generally you don't want verbs in
properties whereas they can be handy in functions. More importantly the @property
decorator was added before the function definition, this is the key that tells Python to
expose the function as a property.

You can convert the read_range_status in the same way:

@property
def range_status(self):
    """Retrieve the status/error from a previous range read."""
    return self._read_8(VL6180X_REG_RESULT_RANGE_STATUS) >> 4

Again only the function name and @property decorator are changed--the code inside
the property/function doesn't need to change.

Now try copying the updated adafruit_vl6180x.py to your board and creating an
instance of it:

import board
import busio
i2c = busio.I2C(board.SCL, board.SDA)
import adafruit_vl6180x
vl6180x = adafruit_vl6180x.Adafruit_VL6180X(i2c)
vl6180x.range
vl6180x.range_status

©Adafruit Industries
You should see the properties work just like the functions did before. However the code is a bit simpler and easier since you don't need to add the parenthesis like when calling a function.

Although it's not shown here you can also add a setter to a property which enables you to write data using property syntax. See this page about Python properties for more details on creating a setter for a property. Typically if you have an attribute that can be both read and written to (like a configuration value or register) you'll want to expose it as a property with both a getter and setter.

As a review here's the final simplified driver file:

```python
>>> import board
>>> import busio
>>> i2c = busio.I2C(board.SCL, board.SDA)
>>> import adafruit_vl6180x
>>> vl6180x = adafruit_vl6180x.Adafruit_VL6180X(i2c)
>>> vl6180x.range
255
>>> vl6180x.range
19
>>> vl6180x.range
35
>>> vl6180x.range
53
>>> vl6180x.range
63
>>> vl6180x.range
131
>>> vl6180x.range_status
0
```

```
# The MIT License (MIT)
#
# Copyright (c) 2017 Tony DiCola for Adafruit Industries
#
# 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.
```

`adafruit_vl6180x`

`====================================================`

This is a CircuitPython driver for the VL6180X time-of-flight distance sensor.

* Author(s): Tony DiCola

```python
from adafruit_bus_device.i2c_device import I2CDevice
```
VL6180X_DEFAULT_I2C_ADDR = const(0x29)

VL6180X_REG_IDENTIFICATION_MODEL_ID = const(0x000)
VL6180X_REG_SYSTEM_INTERRUPT_CONFIG = const(0x014)
VL6180X_REG_SYSTEM_INTERRUPT_CLEAR = const(0x015)
VL6180X_REG_SYSTEM_FRESH_OUT_OF_RESET = const(0x016)
VL6180X_REG_SYSRANGE_START = const(0x018)
VL6180X_REG_SYSALS_START = const(0x038)
VL6180X_REG_SYSALS_ANALOGUE_GAIN = const(0x03F)
VL6180X_REG_SYSALS_INTEGRATION_PERIOD_HI = const(0x040)
VL6180X_REG_SYSALS_INTEGRATION_PERIOD_LO = const(0x041)
VL6180X_REG_RESULT_ALS_VAL = const(0x050)
VL6180X_REG_RESULT_RANGE_VAL = const(0x062)
VL6180X_REG_RESULT_RANGE_STATUS = const(0x04d)
VL6180X_REG_RESULT_INTERRUPT_STATUS_GPIO = const(0x04f)

VL6180X_ALS_GAIN_1 = const(0x06)
VL6180X_ALS_GAIN_1_25 = const(0x05)
VL6180X_ALS_GAIN_1_67 = const(0x04)
VL6180X_ALS_GAIN_2_5 = const(0x03)
VL6180X_ALS_GAIN_5 = const(0x02)
VL6180X_ALS_GAIN_10 = const(0x01)
VL6180X_ALS_GAIN_20 = const(0x00)
VL6180X_ALS_GAIN_40 = const(0x07)

class Adafruit_VL6180X:
    def __init__(self, i2c, address=VL6180X_DEFAULT_I2C_ADDR):
        self._device = I2CDevice(i2c, address)
        if self._read_8(VL6180X_REG_IDENTIFICATION_MODEL_ID) != 0xB4:
            raise RuntimeError('Could not find VL6180X, is it connected and
powered?')
        self._load_settings()
        self._write_8(VL6180X_REG_SYSTEM_FRESH_OUT_OF_RESET, 0x00)

    @property
    def range(self):
        """Read the range of an object in front of sensor and return it in mm."""
        # wait for device to be ready for range measurement
        while not (self._read_8(VL6180X_REG_RESULT_RANGE_STATUS) & 0x01):
            pass
        # Start a range measurement
        self._write_8(VL6180X_REG_SYSRANGE_START, 0x01);
        # Poll until bit 2 is set
        while not (self._read_8(VL6180X_REG_RESULT_INTERRUPT_STATUS_GPIO) &amp;
0x04):
            pass
        # read range in mm
        range_ = self._read_8(VL6180X_REG_RESULT_RANGE_VAL)
        # clear interrupt
        self._write_8(VL6180X_REG_SYSTEM_INTERRUPT_CLEAR, 0x07)
        return range_

    def read_lux(self, gain):
        """Read the lux (light value) from the sensor and return it."""
reg = self._read_8(VL6180X_REG_SYSTEM_INTERRUPT_CONFIG)
reg &= ~0x38
reg |= (0x4 << 3)  # IRQ on ALS ready
self._write_8(VL6180X_REG_SYSTEM_INTERRUPT_CONFIG, reg)

# 100 ms integration period
self._write_8(VL6180X_REG_SYSALS_INTEGRATION_PERIOD_HI, 0)
self._write_8(VL6180X_REG_SYSALS_INTEGRATION_PERIOD_LO, 100)
# analog gain
if gain > VL6180X_ALS_GAIN_40:
    gain = VL6180X_ALS_GAIN_40
self._write_8(VL6180X_REG_SYSALS_ANALOGUE_GAIN, 0x40 | gain)
# start ALS
self._write_8(VL6180X_REG_SYSALS_START, 0x1)

while 4 != ((self._read_8(VL6180X_REG_RESULT_INTERRUPT_STATUS_GPIO)
            >> 3) & 0x7):
    pass

# read lux!
lux = self._read_16(VL6180X_REG_RESULT_ALS_VAL)
# clear interrupt
self._write_8(VL6180X_REG_SYSTEM_INTERRUPT_CLEAR, 0x07);
lux *= 0.32  # calibrated count/lux
if gain == VL6180X_ALS_GAIN_1:
    pass
elif gain == VL6180X_ALS_GAIN_1_25:
    lux /= 1.25
elif gain == VL6180X_ALS_GAIN_1_67:
    lux /= 1.76
elif gain == VL6180X_ALS_GAIN_2_5:
    lux /= 2.5
elif gain == VL6180X_ALS_GAIN_5:
    lux /= 5
elif gain == VL6180X_ALS_GAIN_10:
    lux /= 10
elif gain == VL6180X_ALS_GAIN_20:
    lux /= 20
elif gain == VL6180X_ALS_GAIN_40:
    lux /= 20
lux *= 100
lux /= 100  # integration time in ms
return lux

@property
def range_status(self):
    """Retrieve the status/error from a previous range read."""
    return self._read_8(VL6180X_REG_RESULT_RANGE_STATUS)

def _load_settings(self):
    # private settings from page 24 of app note
    self._write_8(0x0207, 0x01)
    self._write_8(0x0208, 0x01)
    self._write_8(0x0096, 0x00)
    self._write_8(0x0097, 0xfd)
    self._write_8(0x00e3, 0x00)
    self._write_8(0x00e4, 0x04)
    self._write_8(0x00e5, 0x02)
    self._write_8(0x00e6, 0x01)
    self._write_8(0x00e7, 0x03)
    self._write_8(0x00f5, 0x02)
    self._write_8(0x00d9, 0x05)
    self._write_8(0x00db, 0xce)
    self._write_8(0x00dd, 0x0b)
    self._write_8(0x00e7, 0x03)
    self._write_8(0x00e7, 0x03)
    self._write_8(0x00f5, 0x02)
    self._write_8(0x00d9, 0x05)
    self._write_8(0x00db, 0xce)
    self._write_8(0x00dd, 0x0b)
self._write_8(0x0198, 0x01)
self._write_8(0x01b0, 0x17)
self._write_8(0x01ad, 0x00)
self._write_8(0x00ff, 0x05)
self._write_8(0x0100, 0x05)
self._write_8(0x0199, 0x05)
self._write_8(0x01a6, 0x1b)
self._write_8(0x01ac, 0x3e)
self._write_8(0x01a7, 0x1f)
self._write_8(0x0030, 0x00)

# Recommended : Public registers - See data sheet for more detail
self._write_8(0x0011, 0x10)   # Enables polling for 'New Sample ready'
                       # when measurement completes
self._write_8(0x0010a, 0x30) # Set the averaging sample period
                       # (compromise between lower noise and
                       # increased execution time)
self._write_8(0x003f, 0x46)   # Sets the light and dark gain (upper
                       # nibble). Dark gain should not be
                       # changed.
self._write_8(0x0031, 0xFF)   # sets the # of range measurements after
                       # which auto calibration of system is
                       # performed
self._write_8(0x0040, 0x63)   # Set ALS integration time to 100ms
self._write_8(0x003e, 0x31)   # Set default ALS inter-measurement period
                       # to 500ms
self._write_8(0x0014, 0x24)   # Configures interrupt on 'New Sample
                       # Ready threshold event'

def _write_8(self, address, data):
    # Write 1 byte of data from the specified 16-bit register address.
    with self._device:
        self._device.write(bytes([(address >> 8) & 0xFF, 
                            address & 0xFF, 
                            data]))

def _write_16(self, address, data):
    # Write a 16-bit big endian value to the specified 16-bit register
    # address.
    with self._device:
        self._device.write(bytes([(address >> 8) & 0xFF, 
                            address & 0xFF, 
                            (data >> 8) & 0xFF, 
                            data & 0xFF]))

def _read_8(self, address):
    # Read and return a byte from the specified 16-bit register address.
    with self._device:
        self._device.write(bytes([(address >> 8) & 0xFF, 
                            address & 0xFF, 
                            0xFF, 
                            data & 0xFF]),
                        stop=False)
        result = bytearray(1)
        self._device.read_into(result)
        return result[0]

def _read_16(self, address):
    # Read and return a 16-bit unsigned big endian value read from the
    # specified 16-bit register address.
    with self._device:
        self._device.write(bytes([(address >> 8) & 0xFF, 
                            address & 0xFF, 
                            0xFF, 
                            data & 0xFF]),
                        stop=False)
        result = bytearray(2)
        self._device.read_into(result)
        return result[0]

©Adafruit Industries
With these small changes to simplify the begin & initializer, and switch some functions to properties the users of the driver will need to write less code--awesome!

That's all there is to the basics of porting and simplifying an Arduino driver to work with Python and CircuitPython. Continue on for some helpful tips that you should consider if your port a driver of your own.

Optimizations

After simplifying a driver there are a few optimizations you might consider making to help reduce the memory usage of the driver. In general memory usage is a struggle for small microcontrollers and as a driver write you want to ensure your driver uses as little memory as possible. This allows your users to build more complex and interesting programs without running out of memory. This page describes a few useful tips to reduce memory usage.

Internal Constant Values

One optimization is the use of const values for integers or numbers that don't ever change. There's a special const function added to MicroPython & CircuitPython that allows you to tell the interpreter and other tools that a value will never change. When tools know the value is constant they can optimize its usage, like inlining the value where it's used instead of creating a variable that takes precious memory. As your saw in the [driver code page](#) it makes sense to set register and other fixed values as consts.

However one extra optimization is to make 'internal' constants marked as private module members with an underscore. These are global values in a module that are prefixed with an underscore in their name and tell tools that nothing outside the module will reference them. By creating an internal constant you can be absolutely sure the value will be inlined and use as little memory as possible.

The big gotcha with internal constant values is that they can only be used inside your driver or module. Your users can never reference these values! This means you typically only want to use internal constants for things like register addresses that only your driver code will read and write. For example the constants in our driver look like:
Some of these values are only used inside the driver and users of the driver are not expected to ever reference them. For example the VL6180X_DEFAULT_I2C_ADDR, and VL6180X_REG_* values are addresses which the driver code references--users never need to specify these values.

Some of the values your driver user might need to reference. For example the VL6180X_ALS_* and VL6180X_ERROR_* values are constants which users might need to send to the driver (like when setting gain interpreting an error).

Let's make the register and address values internal constants by adding an underscore in front of their name. We'll keep the ALS and error constants as-is so they remain as module-level values that a user can access:

```python
_VL6180X_DEFAULT_I2C_ADDR = const(0x29)
_VL6180X_REG_IDENTIFICATION_MODEL_ID = const(0x000)
_VL6180X_REG_SYSTEM_INTERRUPT_CONFIG = const(0x014)
_VL6180X_REG_SYSTEM_INTERRUPT_CLEAR = const(0x015)
_VL6180X_REG_SYSTEM_FRESH_OUT_OF_RESET = const(0x016)
_VL6180X_REG_SYSRANGE_START = const(0x018)
_VL6180X_REG_SYSALS_START = const(0x038)
_VL6180X_REG_SYSALS_ANALOGUE_GAIN = const(0x03F)
_VL6180X_REG_SYSALS_INTEGRATION_PERIOD_HI = const(0x040)
_VL6180X_REG_SYSALS_INTEGRATION_PERIOD_LO = const(0x041)
_VL6180X_REG_RESULT_ALS_VAL = const(0x050)
_VL6180X_REG_RESULT_RANGE_VAL = const(0x062)
_VL6180X_REG_RESULT_RANGE_STATUS = const(0x04d)
_VL6180X_REG_RESULT_INTERRUPT_STATUS_GPIO = const(0x04f)
_VL6180X_ALS_GAIN_1 = const(0x06)
_VL6180X_ALS_GAIN_1_25 = const(0x05)
_VL6180X_ALS_GAIN_1_67 = const(0x04)
_VL6180X_ALS_GAIN_2_5 = const(0x03)
_VL6180X_ALS_GAIN_5 = const(0x02)
_VL6180X_ALS_GAIN_10 = const(0x01)
_VL6180X_ALS_GAIN_20 = const(0x00)
_VL6180X_ALS_GAIN_40 = const(0x07)
_VL6180X_ERROR_NONE = const(0)
_VL6180X_ERROR_SYSERR_1 = const(1)
_VL6180X_ERROR_SYSERR_5 = const(5)
_VL6180X_ERROR_ECEFAIL = const(6)
_VL6180X_ERROR_NOCONVERGE = const(7)
_VL6180X_ERROR_RANGIGNORE = const(8)
_VL6180X_ERROR_SNR = const(11)
_VL6180X_ERROR_RAWUFLOW = const(12)
_VL6180X_ERROR_RAWOFLOW = const(13)
_VL6180X_ERROR_RANGEUFLOW = const(14)
_VL6180X_ERROR_RANGEOFLOW = const(15)
```
Then in your driver code be sure to update all the references to the registers to use the new underscore name. The rest of the driver now looks like:

class Adafruit_VL6180X:

def __init__(self, i2c, address=_VL6180X_DEFAULT_I2C_ADDR):
    self._device = I2CDevice(i2c, address)
    if self._read_8(_VL6180X_REG_IDENTIFICATION_MODEL_ID) != 0xB4:
        raise RuntimeError('Could not find VL6180X, is it connected and powered?')
    self._load_settings()
    self._write_8(_VL6180X_REG_SYSTEM_FRESH_OUT_OF_RESET, 0x00)

@property
def range(self):
    """Read the range of an object in front of sensor and return it in mm."
    # wait for device to be ready for range measurement
    while not (self._read_8(_VL6180X_REG_RESULT_RANGE_STATUS) & 0x01):
        pass
    # Start a range measurement
    self._write_8(_VL6180X_REG_SYSRANGE_START, 0x01);
    # Poll until bit 2 is set
    while not (self._read_8(_VL6180X_REG_RESULT_INTERRUPT_STATUS_GPIO) & 0x04):
        pass
    # read range in mm
    range_ = self._read_8(_VL6180X_REG_RESULT_RANGE_VAL)
    # clear interrupt
    self._write_8(_VL6180X_REG_SYSTEM_INTERRUPT_CLEAR, 0x07)
    return range_

def read_lux(self, gain):
    """Read the lux (light value) from the sensor and return it."
    reg = self._read_8(_VL6180X_REG_SYSTEM_INTERRUPT_CONFIG) & 0x38
    reg |= (0x4 &lt;&lt; 3) # IRQ on ALS ready
    self._write_8(_VL6180X_REG_SYSTEM_INTERRUPT_CONFIG, reg)
    # 100 ms integration period
self._write_8(_VL6180X_REG_SYSALS_INTEGRATION_PERIOD_HI, 0)
self._write_8(_VL6180X_REG_SYSALS_INTEGRATION_PERIOD_LO, 100)

# analog gain
if gain > VL6180X_ALS_GAIN_40:
    gain = VL6180X_ALS_GAIN_40
self._write_8(_VL6180X_REG_SYSALS_ANALOGUE_GAIN, 0x40 | gain)
# start ALS
self._write_8(_VL6180X_REG_SYSALS_START, 0x1)
# Poll until "New Sample Ready threshold event" is set
while 4 != ((self._read_8(_VL6180X_REG_RESULT_INTERRUPT_STATUS_GPIO)
            >> 3) & 0x7):
    pass
# read lux!
lux = self._read_16(_VL6180X_REG_RESULT_ALS_VAL)
# clear interrupt
self._write_8(_VL6180X_REG_SYSTEM_INTERRUPT_CLEAR, 0x07);
if gain == VL6180X_ALS_GAIN_1:
    pass
elif gain == VL6180X_ALS_GAIN_1_25:
    lux /= 1.25
elif gain == VL6180X_ALS_GAIN_1_67:
    lux /= 1.76
elif gain == VL6180X_ALS_GAIN_2_5:
    lux /= 2.5
elif gain == VL6180X_ALS_GAIN_5:
    lux /= 5
elif gain == VL6180X_ALS_GAIN_10:
    lux /= 10
elif gain == VL6180X_ALS_GAIN_20:
    lux /= 20
elif gain == VL6180X_ALS_GAIN_40:
    lux /= 20
lux *= 100
lux /= 100 # integration time in ms
return lux

@property
def range_status(self):
    """Retrieve the status/error from a previous range read."
    return self._read_8(_VL6180X_REG_RESULT_RANGE_STATUS)

def _load_settings(self):
    """private settings from page 24 of app note"
    self._write_8(0x0207, 0x01)
    self._write_8(0x0208, 0x01)
    self._write_8(0x0096, 0x00)
    self._write_8(0x0097, 0xfd)
    self._write_8(0x00e3, 0x00)
    self._write_8(0x00e4, 0x04)
    self._write_8(0x00e5, 0x02)
    self._write_8(0x00e6, 0x01)
    self._write_8(0x00e7, 0x02)
    self._write_8(0x00f5, 0x02)
    self._write_8(0x0100, 0x05)
    self._write_8(0x01b0, 0x17)
    self._write_8(0x01ad, 0x00)
    self._write_8(0x00ff, 0x05)
    self._write_8(0x0100, 0x05)
self._write_8(0x0199, 0x05)  
self._write_8(0x01a6, 0x1b)  
self._write_8(0x01ac, 0x3e)  
self._write_8(0x01a7, 0x1f)  
self._write_8(0x0030, 0x00)  
# Recommended : Public registers - See data sheet for more detail
self._write_8(0x0011, 0x10)   # Enables polling for 'New Sample ready'  
when measurement completes
self._write_8(0x010a, 0x30)   # Set the averaging sample period  
# compromise between lower noise and  
# increased execution time
self._write_8(0x003f, 0x46)   # Sets the light and dark gain (upper  
# nibble). Dark gain should not be  
# changed.
self._write_8(0x0031, 0xFF)   # sets the # of range measurements after  
# which auto calibration of system is  
# performed
self._write_8(0x0040, 0x63)   # Set ALS integration time to 100ms
self._write_8(0x002e, 0x01)   # perform a single temperature calibration  
# of the ranging sensor
self._write_8(0x001b, 0x09)   # Set default ranging inter-measurement  
# period to 100ms
self._write_8(0x003e, 0x31)   # Set default ALS inter-measurement period  
# to 500ms
self._write_8(0x0014, 0x24)   # Configures interrupt on 'New Sample  
# Ready threshold event'

def _write_8(self, address, data):
    # Write 1 byte of data from the specified 16-bit register address.
    with self._device:
        self._device.write(bytes([(address >> 8) & 0xFF,
                                   address & 0xFF,
                                   data]))

def _write_16(self, address, data):
    # Write a 16-bit big endian value to the specified 16-bit register  
    # address.
    with self._device:
        self._device.write(bytes([(address >> 8) & 0xFF,
                                   address & 0xFF,
                                   (data >> 8) & 0xFF,
                                   data & 0xFF]))

def _read_8(self, address):
    # Read and return a byte from the specified 16-bit register address.
    with self._device:
        self._device.write(bytes([(address >> 8) & 0xFF,
                                   address & 0xFF]),
                            stop=False)
        result = bytearray(1)
        self._device.read_into(result)
        return result[0]

def _read_16(self, address):
    # Read and return a 16-bit unsigned big endian value read from the  
    # specified 16-bit register address.
    with self._device:
        self._device.write(bytes([(address >> 8) & 0xFF,
                                   address & 0xFF]),
                            stop=False)
        result = bytearray(2)
        self._device.read_into(result)
        return (result[0] << 8) | result[1]
Notice the register and other values use the underscore-prefixed names. This ensures the internal constants will be as efficient as possible with memory usage and inlined in binary versions of the code where possible.

Remember you need to understand how your users will use the driver code before you change values to internal constants. Once a value is an internal constant it can never be accessed by your driver users! Internal constants are intended only for internal values that the driver uses--not values like errors or settings that a user might need to pass to a function.

Memory Usage

In Python you don't typically need to manage memory as the interpreter is smart about understanding when memory is used and unused. However there is still a limited amount of memory available and this can cause problems for your driver code. As mentioned before at some point (typically 200-300 lines of code) a pure text .py file becomes too big for your board to load and process in memory. For this issue you typically want to convert the file to a binary .mpy version ()--this file will use less memory when loaded by the interpreter.

On Linux and OSX it's easy to compile mpy-cross and use it yourself as mentioned in the guide ()--this is because these operating systems support a POSIX build environment (once you install Xcode Command Line Tools on OSX ()). Simply clone the CircuitPython repository (being careful to choose the right branch as the master branch might be on a later version of the MPY format than the released CircuitPython builds) and build inside the mpy-cross folder:

```
git clone --recursive https://github.com/adafruit/circuitpython.git
cd circuitpython/mpy-cross
make
```

Now you can use mpy-cross just like from within the Vagrant-based VM ()

For Windows it's a bit trickier and you can either use the Vagrant-based VM () in the mentioned guide (this is recommended to ensure you get the latest mpy-cross built) or you can download a pre-built binary below. NOTE: This binary is built for CircuitPython 2.x / MicroPython 1.9.2 only. Earlier or later versions of either might not work with binaries built using this tool! This tool was built using the mingw 32-bit toolchain and should run on all versions of Windows 7 and above, 32-bit and 64-bit (with WOW32 emulation layer available).

```
mpy-cross.exe (for CircuitPython 2.x / MicroPython 1.9.2 only)
```
In addition you can find other binary builds of mpy-cross on the CircuitPython releases page, but be aware they can only be guaranteed to work with the associated release of CircuitPython and might not work with earlier or later versions! In addition be careful on OSX and Linux or Raspbian builds as binary dependencies might differ between your system and the built version--if you run into trouble stick with manually building your own version of mpy-cross.

To use mpy-cross, as the guide mentions you just need to invoke it in a terminal and pass in the path to a python .py file. The tool will generate a .mpy version (if there are no syntax errors or other problems with the input file) which you can copy to your board and use in place of the .py version. The .mpy version will use much less memory when being loaded by CircuitPython.

For example to mpy-cross a file called test.py you might run:

```
mpy-cross test.py
```

Make sure mpy-cross.exe (on Windows) or other suitable executable is in the same directory or in your terminal path. If the tool succeeds there will be no output generated and you can find the test.mpy file in the same location.

Run mpy-cross with the --help option to see more details on advanced usage!

Within your driver code you might run into memory issues too, like if you're creating very large buffers or lots of variables. There's no easy fix for this--memory is finite and you typically can't add more. If you need more memory you need to find ways to reduce other memory usage. Sometimes you can remove unused variables, functions, etc. Sometimes you might need to remove functionality entirely and strip a driver down to just the core features. There's no easy fix but be aware memory can be a significant issue as you port complex drivers and code to Python!

---

**Gotchas**

If you're converting code from Arduino/C or C++ to Python there are a few subtle and not-so-subtle issues to worry about. One major difference between these languages is that the type of a variable is always known in Arduino and it can do certain optimizations and checks automatically for you. In Python a variable can be any type and freely change between them--this flexibility makes Python easier to program, but can cause some trouble if you're moving over code that expects rigid and fixed types. In particular manipulating numbers and numeric values in Python sometimes has to be done a bit different than Arduino.
Check out the following tips below:

**Unsigned Integers**

One major stumbling block in converting Arduino to Python code is that there is no concept of an unsigned (or only positive) integer in Python. In Python all numbers are arbitrarily large positive & negative values. In Arduino you can choose if a number should only be positive values from 0-255 (a byte or uint8_t type), 0-65535 (an unsigned 16-bit value), and more. You can run into very subtle bugs, like this code in Arduino:

```c
uint8_t foo = 255;
Serial.println(foo + 1);
```

Would not behave the same as this code in Python:

```python
foo = 255
print(foo + 1)
```

Try it out to see for yourself! The Python code will print 256 whereas the Arduino code will print 0! If you're blindly copying Arduino code to Python this is a very subtle bug that could mean your arithmetic and math doesn't work the way you expect. At best you might run into an error or unexpected exception, and at worst you might spend days and weeks chasing down impossible to understand issues!

How can you simulate the behavior of an unsigned type like the above in Python? It turns out if you mask off the bits you care about then Python will get a result that you expect, so the equivalent Python code would be:

```python
foo = 255
print((foo + 1) & 0xFF)
```

The "& 0xFF" bit is a boolean and operation which tells Python to only look at the lower 8 bits of the numeric value and ignore everything else. Once Python looks at the low bits of the value 256 it will see the value 0 which is what you would expect when 255 overflows with the addition of 1. Be very careful to watch for all uses of unsigned types in Arduino code and make sure your Python code masks out the appropriate size bits after it performs operations on them.
Struct Module

Another issue with converting from Arduino to Python is that again Python doesn't have a concept of fixed numeric types. This means it can be difficult to take the bytes of a register that you read from a device and interpret them as a fixed size number (like a 16-bit unsigned integer, 8-bit signed integer, 32-bit floating point value, etc.). Luckily there's an entire module in Python that's built to do this conversion for you, the struct module. In CircuitPython and MicroPython the struct module is available (sometimes called ustruct in older builds) and you can use its pack and unpack functions to convert bytes to and from different size numeric types.

Division Is Floating Point

Another issue that cause very subtle bugs in converting Arduino to Python code is the user of division. Consider this line in Arduino:

```cpp
int foo = 10;
Serial.println(foo / 3);
```

And the similar lines of code in Python:

```python
foo = 10
print(foo / 3)
```

If you run both you'll get very different results! In Arduino you'll see 3 and in Python you'll see 3.33333! Although both numbers are similar, the Python version is performing division with floating point by default. This means you can get fractional values where you might not have expected them otherwise. At best you might get an exception (like if performing division to manipulate bytes and bits) and at worst you'll create code that runs without error but gives you the completely wrong result.

Luckily in Python you can fix this by telling Python to explicitly perform integer division. Do this by specifying the double divide operator, //, like:

```python
foo = 10
print(foo // 3)
```

Now you'll see the expected result of 3 from the Python code. The double divide will force Python to perform integer division. As you port Arduino code to Python
you need to be very careful to see where the Arduino code is performing division on integer types (even implicitly!) and use the double divide operator. Don't use double divide everywhere though because in some cases you actually want floating point division. Carefully review each use of division to see where integer vs. floating point is necessary.

---

**Timing Critical Code**

Another potential stumbling block when converting Arduino to Python code is dealing with blocks of code that are timing critical. That is when a section of code has to operate within a very fast or small window of time. For example when telling NeoPixels what color to light up a very fast signal needs to be generated. This signal is so fast it's actually too slow to try doing it in an interpreted language like CircuitPython and MicroPython. The overhead of the Python interpreter converting the Python code into machine code is just too high and the performance can't be matched compared to compiled languages like Arduino.

So what can you do when you run into timing critical code? Unfortunately there's no single fix or magic bullet to solve the problem. You'll need to understand what the code is doing and if there are ways it might be written to depend less on performance (are there peripherals or other parts of the hardware that can do the same job with less performance demands?). You can also consider modifying the Python firmware to include C functions that don't have as many timing constraints, but this is an advanced topic and not something that's easy or possible to do purely in a driver (it requires building an entirely new CircuitPython or MicroPython firmware).

The key thing about performance is to look for where it matters before you get deep into porting driver code. Carefully review the Arduino code and look for sections which mention critical sections, timing constraints, etc. Also review the datasheet and if your device isn't using a standard protocol like SPI, I2C, etc. which most hardware will natively support, then you might run into performance issues trying to emulate or 'bit bang' a protocol in Python code. However some devices don't care about speed and even slow Python code is just fine to talk to them—every device is different so be prepared to spend time investigating the devices you're porting and their requirements for speed and timing critical code.

---

**Memory Usage**

In Python you don't typically need to manage memory as the interpreter is smart about understanding when memory is used and unused. However there is still a limited amount of memory available and this can cause problems for your driver.
code. As mentioned before at some point (typically 200-300 lines of code) a pure text .py file becomes too big for your board to load and process in memory. For this issue you typically want to convert the file to a binary .mpy version --this file will use less memory when loaded by the interpreter.

Within your driver code you might run into memory issues too, like if you’re creating very large buffers or lots of variables. There's no easy fix for this--memory is finite and you typically can't add more. If you need more memory you need to find ways to reduce other memory usage. Sometimes you can remove unused variables, functions, etc. Sometimes you might need to remove functionality entirely and strip a driver down to just the core features. There's no easy fix but be aware memory can be a significant issue as you port complex drivers and code to Python!