

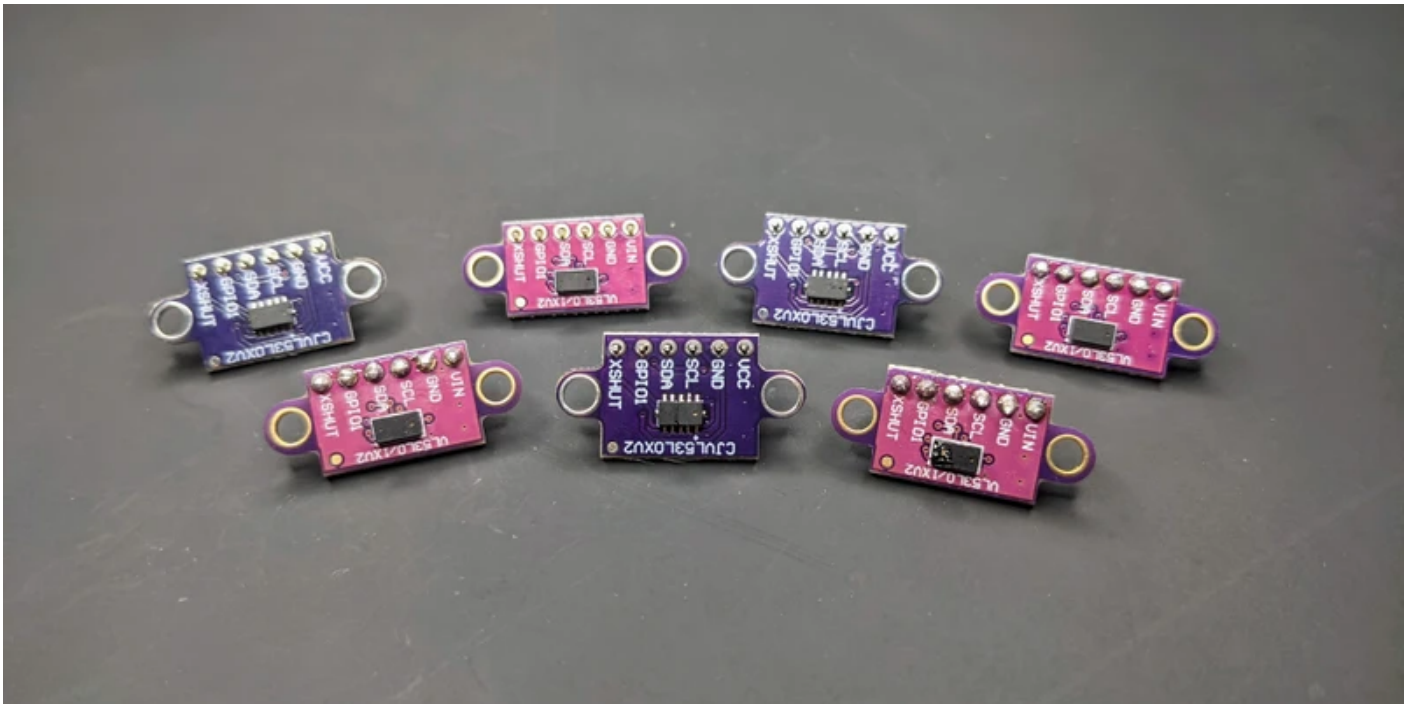
## Artful Bytes

A blog about building and programming hardware.

[Articles](#) [Projects](#) [Tools](#) [About](#)

# Writing a driver for the tiny range sensor VL53L0X (0-200 cm)

August 29, 2021 • 10 min read



## ► Contents

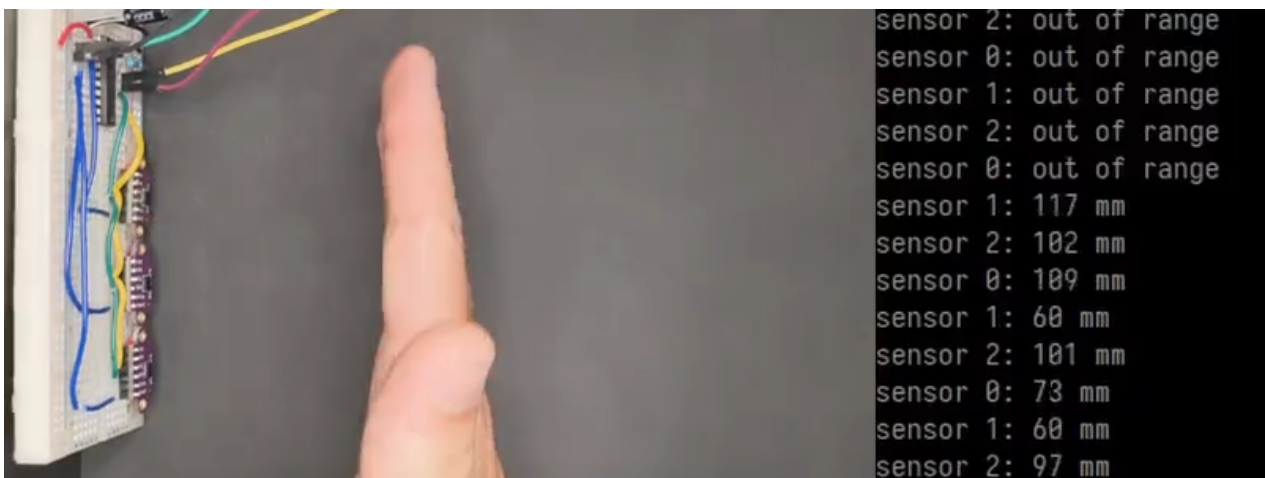
In [my previous post](#), I went into details on how to write a driver for the VL6180X. In this post, I want to continue by implementing a driver for another time-of-flight sensor from ST, the VL53L0X. I won't go into details about I2C in this post. Instead, I will re-use the I2C layer as for the VL6180X and only focus on the VL53L0X-specific parts.

As I began implementing the driver for VL53L0X, I quickly realized it was harder to get working than the VL6180X because it's a more complex and configurable sensor. On top of that, [ST has focused on providing a "not-so-good" API driver instead of comprehensive documentation](#).

The quick route to get the VL53L0X working is to pull in the entire [API driver](#) from ST and only implement the lower I2C drivers specific to your

microcontroller. The downside of this is that you have to include their bloaty (and potentially buggy) code, which is a problem if you are trying to minimize the memory footprint. Therefore, it's often preferable to implement your driver and only use the vendor code as a reference.

In this post, I will create a minimal driver to initialize and do a single range measurement with the VL53L0X. This will be helpful to those of you who are trying to write a custom driver for the VL53L0X. I will also add support for multiple VL53L0X, and give you some more information to help you tailor the VL53L0X for your application.



## The code

All of the code is available at [GitHub](#), and I also share parts of it as gists in this post. The easiest way to follow along is to pull down the repo from GitHub and then use interactive git rebase (*git rebase -i --root*) because I have basically made a new commit after each section.

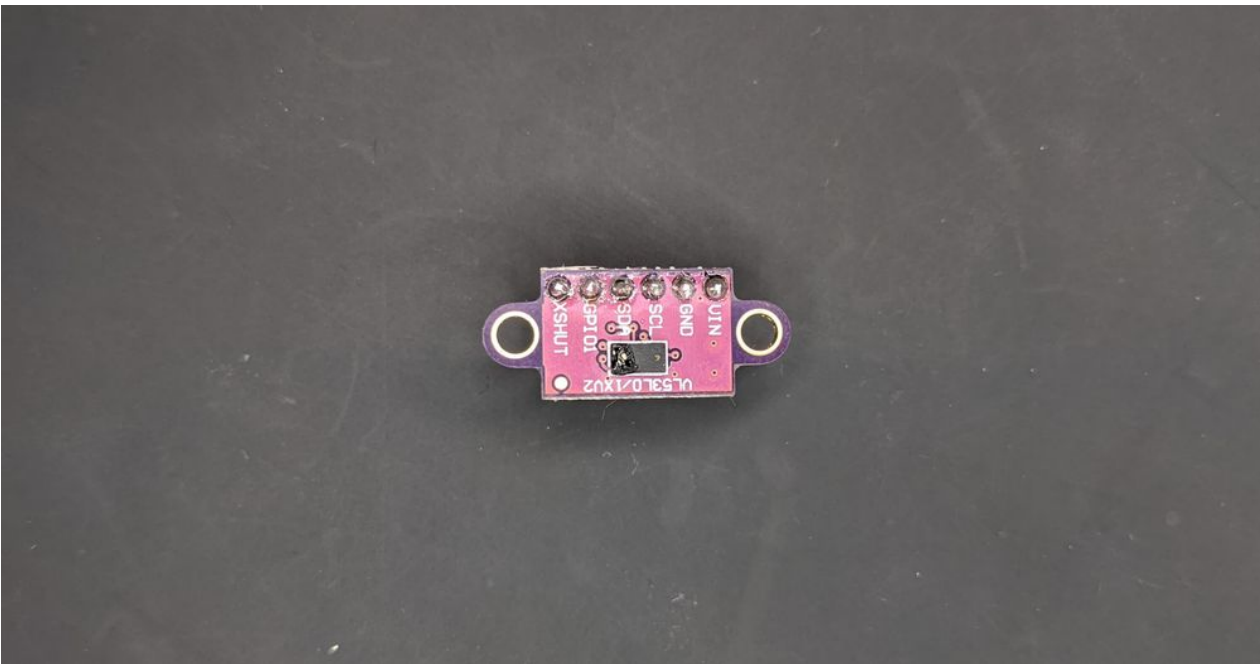
## About VL53L0X

The VL53L0X measures range by emitting light from a laser and measuring the light reflected back on its array of SPADs.

Like the VL6180X, the VL53L0X has a microcontroller on board, handling the computation and configuration, and interfaced with I2C.

It measures a range between ~0-2 meters at up to 50 Hz, but the exact performance depends on your configuration and operating conditions.

See [ST's documentation](#) for more details.



**NOTE:** *Be careful when you solder the header pins, so you don't melt the sensor case as I've done in the image above.*

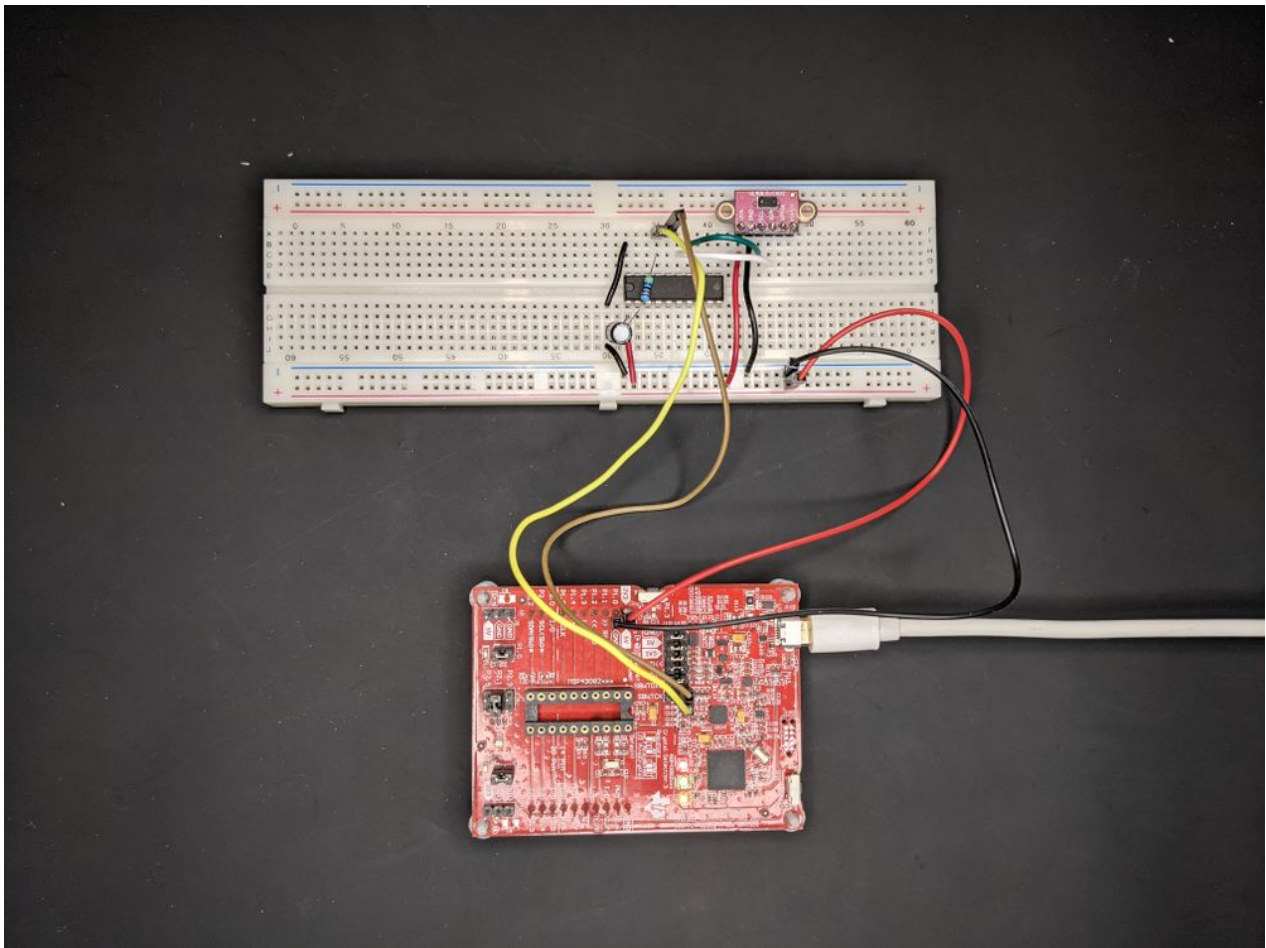
## How to hook up the VL53L0X

Just as in [my previous post](#), I will use a VL53L0X breakout board and hook it up to a [MSP430 LaunchPad](#). All the breakout boards that I've seen have the same pinout:

Pin	Description
VIN	Support 5 V or 3.3 V (double-check your breakout board to make sure)
XSHUT (GPIO0)	Set sensor in reset (useful for multi-sensor support, otherwise can be left unconnected)
GPIO1	Interrupt pin (can be left unconnected if polling)

Pin	Description
GND	Ground
SDA	I2C data line
SCL	I2C clock line

At minimum - when using a single sensor and no interrupt - you must connect four pins: SDA, SCL, VIN, and GND. My setup looks like this:



As you can see, I've moved the MSP430 DIP package from the launchpad to the solderless board. I find this makes the prototyping a bit cleaner. If you do this, don't forget to connect the reset pull-up resistor to stop the MSP430 from endlessly rebooting. You should also connect a bypass capacitor ( $\sim 0.1 \mu\text{F}$ ) to prevent the current surge from the VL53L0X from resetting the MSP430.

# Tip: Verify the VL53L0X with an Arduino

There are several Arduino libraries for the VL53L0X, and they only take 5 minutes to set up. If you have an Arduino, please use it to sanity-check that your VL53L0X works before writing your own driver to avoid wasting time on a faulty sensor.

- [Pololu's Arduino library](#).
- [Adafruit's Arduino library](#).

## Verify the I2C communication

Building on top of the I2C driver from [my previous post](#), we should create a function to verify that we can talk to the VL53L0X. The simplest way is to read a register with a known value. The datasheet suggests several registers:

### 3.2 I<sup>2</sup>C interface - reference registers

The registers shown in the table below can be used to validate the user I<sup>2</sup>C interface.

**Table 4. Reference registers**

Address	(After fresh reset, without API loaded)
0xC0	0xEE
0xC1	0xAA
0xC2	0x10
0x51	0x0099
0x61	0x0000

The first one, 0xC0, contains the device model id, which is always 0xEE for VL53L0X. If we read the value 0xEE from this register, we know that the VL53L0X is up and that the I2C communication works.

```

1  #define VL53L0X_EXPECTED_DEVICE_ID (0xEE)
2
3  /**
4   * We can read the model id to confirm that the device is booted.
5   * (There is no fresh_out_of_reset as on the vl6180x)
6   */
7  static bool device_is_booted()
8  {
9      uint8_t device_id = 0;
10     if (!i2c_read_addr8_data8(REG_IDENTIFICATION_MODEL_ID, &device_id)) {
11         return false;
12     }
13     return device_id == VL53L0X_EXPECTED_DEVICE_ID;
14 }

```

vl53l0x.c hosted with ❤ by GitHub

[view raw](#)

Running in debug mode, you should see:

Expression	Type	Value	Address
 device_id	unsigned char	238 '\xee' 	0x03FD

**NOTE:** The registers on the VL53L0X are 8-bit addressed (NOT 16-bit addressed as on the VL6180X). Make sure you use the 8-bit version of the I2C functions.

## A minimal driver

The VL53L0X is a complex sensor with many configurable registers. I won't cover them all, instead, I aim to provide you with a minimal driver example that is capable of doing a single range measurement.

Since ST's documentation lacks vital information, I've had to study the code they provide as well as the library by Pololu, and after some trial and error extracted the necessary bits.

Similar to the VL6180X driver, I put all of the code for the VL53L0X driver in two files *drivers/vl53l0x.h* and *drivers/vl53l0x.c*.



# How to initialize the VL53L0X

There are several steps to initialize the VL53L0X, and the datasheet provides an overview of them:

UM2039

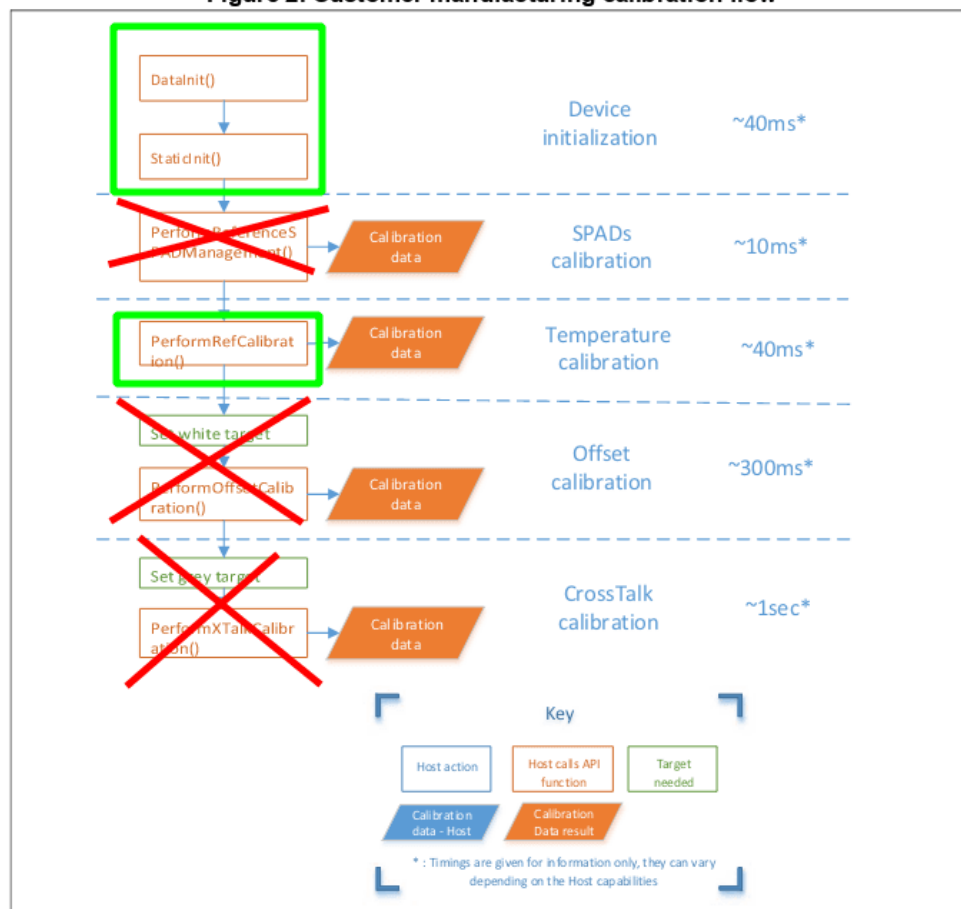
Initial customer manufacturing calibration

## 2 Initial customer manufacturing calibration

There is an initial, once only, calibration step required that should be applied at customer level during the manufacturing process. This flow takes into account all parameters (cover glass, temperature & voltage) from the application.

The customer manufacturing calibration flow is described in [Figure 2](#) below.

**Figure 2. Customer manufacturing calibration flow**



The necessary steps are *device initialization* and *temperature calibration*. We can skip the *SPAD*, *Offset*, and *Crosstalk calibration* for our minimal driver because the default values are good enough. These extra calibration steps are necessary when you use a cover glass. Though, you may want to do *SPAD calibration* later on even if you don't have a cover glass (for more info, see [SPAD management](#)).

Once again, it's helpful to define the registers at the top of the C file. ST unfortunately doesn't provide a complete register map for the VL53L0X, so instead, we have to extract the values from the code they provide.

```
1  #define REG_IDENTIFICATION_MODEL_ID (0xC0)
2  #define REG_VHV_CONFIG_PAD_SCL_SDA_EXTSUP_HV (0x89)
3  #define REG_MSRC_CONFIG_CONTROL (0x60)
4  #define REG_FINAL_RANGE_CONFIG_MIN_COUNT_RATE_RTN_LIMIT (0x44)
5  #define REG_SYSTEM_SEQUENCE_CONFIG (0x01)
6  #define REG_DYNAMIC_SPAD_REF_EN_START_OFFSET (0x4F)
7  #define REG_DYNAMIC_SPAD_NUM_REQUESTED_REF_SPAD (0x4E)
8  #define REG_GLOBAL_CONFIG_REF_EN_START_SELECT (0xB6)
9  #define REG_SYSTEM_INTERRUPT_CONFIG_GPIO (0x0A)
10 #define REG_GPIO_HV_MUX_ACTIVE_HIGH (0x84)
11 #define REG_SYSTEM_INTERRUPT_CLEAR (0x0B)
12 #define REG_RESULT_INTERRUPT_STATUS (0x13)
13 #define REG_SYSRANGE_START (0x00)
14 #define REG_GLOBAL_CONFIG_SPAD_ENABLER_REF_0 (0xB0)
15 #define REG_RESULT_RANGE_STATUS (0x14)
```

vl53l0x.c hosted with ❤ by GitHub

[view raw](#)

## Device initialization

The *device initialization* is divided into two steps, *data initialization* and *static initialization*.

*Data initialization* sets the voltage mode (1v8 or 2v8), the I2C mode (standard or fast), and retrieves the value for the stop variable. The stop variable is used for initiating the stop sequence when doing range measurement. It's not obvious why this stop variable exists.

I set the voltage mode to 2v8 because that's what my breakout board is designed for (and probably most others). I set the I2C mode to standard because the I2C driver is configured for standard mode (100 kHz).



```
1  static uint8_t stop_variable = 0;
2
3  /**
4   * One time device initialization
5   */
6  static bool data_init()
7  {
8      bool success = false;
9
10     /* Set 2v8 mode */
11     uint8_t vhw_config_scl_sda = 0;
12     if (!i2c_read_addr8_data8(REG_VHW_CONFIG_PAD_SCL_SDA_EXTSUP_HW, &vhw_config_scl_sda))
13         return false;
14     }
15     vhw_config_scl_sda |= 0x01;
16     if (!i2c_write_addr8_data8(REG_VHW_CONFIG_PAD_SCL_SDA_EXTSUP_HW, vhw_config_scl_sda))
17         return false;
18     }
19
20     /* Set I2C standard mode */
21     success = i2c_write_addr8_data8(0x88, 0x00);
22
23     success &= i2c_write_addr8_data8(0x80, 0x01);
24     success &= i2c_write_addr8_data8(0xFF, 0x01);
25     success &= i2c_write_addr8_data8(0x00, 0x00);
26     success &= i2c_read_addr8_data8(0x91, &stop_variable);
27     success &= i2c_write_addr8_data8(0x00, 0x01);
28     success &= i2c_write_addr8_data8(0xFF, 0x00);
29     success &= i2c_write_addr8_data8(0x80, 0x00);
30
31     return success;
32 }
```

vl53l0x.c hosted with ❤ by GitHub


[view raw](#)

*Static initialization* loads the default tuning settings, enables interrupt, and sets the sequence steps.

ST provides the default tuning settings (without explanation) in their API driver.

```
1  /**
2   * Load tuning settings (same as default tuning settings provided by ST api code)
3   */
4  static bool load_default_tuning_settings()
5  {
6      bool success = i2c_write_addr8_data8(0xFF, 0x01);
7      success &= i2c_write_addr8_data8(0x00, 0x00);
8      success &= i2c_write_addr8_data8(0xFF, 0x00);
9      success &= i2c_write_addr8_data8(0x09, 0x00);
10     success &= i2c_write_addr8_data8(0x10, 0x00);
11     success &= i2c_write_addr8_data8(0x11, 0x00);
12     success &= i2c_write_addr8_data8(0x24, 0x01);
13     success &= i2c_write_addr8_data8(0x25, 0xFF);
14     success &= i2c_write_addr8_data8(0x75, 0x00);
15     success &= i2c_write_addr8_data8(0xFF, 0x01);
16     success &= i2c_write_addr8_data8(0x4E, 0x2C);
17     success &= i2c_write_addr8_data8(0x48, 0x00);
18     success &= i2c_write_addr8_data8(0x30, 0x20);
19     success &= i2c_write_addr8_data8(0xFF, 0x00);
20     success &= i2c_write_addr8_data8(0x30, 0x09);
21     success &= i2c_write_addr8_data8(0x54, 0x00);
22     success &= i2c_write_addr8_data8(0x31, 0x04);
23     success &= i2c_write_addr8_data8(0x32, 0x03);
24     success &= i2c_write_addr8_data8(0x40, 0x83);
25     success &= i2c_write_addr8_data8(0x46, 0x25);
26     success &= i2c_write_addr8_data8(0x60, 0x00);
27     success &= i2c_write_addr8_data8(0x27, 0x00);
28     success &= i2c_write_addr8_data8(0x50, 0x06);
29     success &= i2c_write_addr8_data8(0x51, 0x00);
30     success &= i2c_write_addr8_data8(0x52, 0x96);
31     success &= i2c_write_addr8_data8(0x56, 0x08);
32     success &= i2c_write_addr8_data8(0x57, 0x30);
33     success &= i2c_write_addr8_data8(0x61, 0x00);
34     success &= i2c_write_addr8_data8(0x62, 0x00);
35     success &= i2c_write_addr8_data8(0x64, 0x00);
36     success &= i2c_write_addr8_data8(0x65, 0x00);
37     success &= i2c_write_addr8_data8(0x66, 0xA0);
38     success &= i2c_write_addr8_data8(0xFF, 0x01);
39     success &= i2c_write_addr8_data8(0x22, 0x32);
40     success &= i2c_write_addr8_data8(0x47, 0x14);
41     success &= i2c_write_addr8_data8(0x49, 0xFF);
42     success &= i2c_write_addr8_data8(0x4A, 0x00);
43     success &= i2c_write_addr8_data8(0xFF, 0x00);
44     success &= i2c_write_addr8_data8(0x7A, 0x0A);
45     success &= i2c_write_addr8_data8(0x7B, 0x00);
46     success &= i2c_write_addr8_data8(0x78, 0x21);
47     success &= i2c_write_addr8_data8(0xFF, 0x01);
```

```
48     success &= i2c_write_addr8_data8(0x23, 0x34);
49     success &= i2c_write_addr8_data8(0x42, 0x00);
50     success &= i2c_write_addr8_data8(0x44, 0xFF);
51     success &= i2c_write_addr8_data8(0x45, 0x26);
52     success &= i2c_write_addr8_data8(0x46, 0x05);
53     success &= i2c_write_addr8_data8(0x40, 0x40);
54     success &= i2c_write_addr8_data8(0x0E, 0x06);
55     success &= i2c_write_addr8_data8(0x20, 0x1A);
56     success &= i2c_write_addr8_data8(0x43, 0x40);
57     success &= i2c_write_addr8_data8(0xFF, 0x00);
58     success &= i2c_write_addr8_data8(0x34, 0x03);
59     success &= i2c_write_addr8_data8(0x35, 0x44);
60     success &= i2c_write_addr8_data8(0xFF, 0x01);
61     success &= i2c_write_addr8_data8(0x31, 0x04);
62     success &= i2c_write_addr8_data8(0x4B, 0x09);
63     success &= i2c_write_addr8_data8(0x4C, 0x05);
64     success &= i2c_write_addr8_data8(0x4D, 0x04);
65     success &= i2c_write_addr8_data8(0xFF, 0x00);
66     success &= i2c_write_addr8_data8(0x44, 0x00);
67     success &= i2c_write_addr8_data8(0x45, 0x20);
68     success &= i2c_write_addr8_data8(0x47, 0x08);
69     success &= i2c_write_addr8_data8(0x48, 0x28);
70     success &= i2c_write_addr8_data8(0x67, 0x00);
71     success &= i2c_write_addr8_data8(0x70, 0x04);
72     success &= i2c_write_addr8_data8(0x71, 0x01);
73     success &= i2c_write_addr8_data8(0x72, 0xFE);
74     success &= i2c_write_addr8_data8(0x76, 0x00);
75     success &= i2c_write_addr8_data8(0x77, 0x00);
76     success &= i2c_write_addr8_data8(0xFF, 0x01);
77     success &= i2c_write_addr8_data8(0x0D, 0x01);
78     success &= i2c_write_addr8_data8(0xFF, 0x00);
79     success &= i2c_write_addr8_data8(0x80, 0x01);
80     success &= i2c_write_addr8_data8(0x01, 0xF8);
81     success &= i2c_write_addr8_data8(0xFF, 0x01);
82     success &= i2c_write_addr8_data8(0x8E, 0x01);
83     success &= i2c_write_addr8_data8(0x00, 0x01);
84     success &= i2c_write_addr8_data8(0xFF, 0x00);
85     success &= i2c_write_addr8_data8(0x80, 0x00);
86     return success;
87 }
```

vl53l0x.c hosted with  by GitHub[view raw](#)

Interrupt settings are explained a bit more in section 6.8 of [UM2039](#). Even if we are not using the interrupt pin, we must still enable interrupt to poll the interrupt

register. I also configure the pin to be active low since my breakout board pulls the pin up by default. Lastly, we should ensure the interrupt is cleared.

```
1  static bool configure_interrupt()
2  {
3      /* Interrupt on new sample ready */
4      if (!i2c_write_addr8_data8(REG_SYSTEM_INTERRUPT_CONFIG_GPIO, 0x04)) {
5          return false;
6      }
7
8      /* Configure active low since the pin is pulled-up on most breakout boards */
9      uint8_t gpio_hv_mux_active_high = 0;
10     if (!i2c_read_addr8_data8(REG_GPIO_HV_MUX_ACTIVE_HIGH, &gpio_hv_mux_active_high))
11         return false;
12     }
13     gpio_hv_mux_active_high &= ~0x10;
14     if (!i2c_write_addr8_data8(REG_GPIO_HV_MUX_ACTIVE_HIGH, gpio_hv_mux_active_high))
15         return false;
16     }
17
18     if (!i2c_write_addr8_data8(REG_SYSTEM_INTERRUPT_CLEAR, 0x01)) {
19         return false;
20     }
21     return true;
22 }
```

vl53l0x.c hosted with ❤ by GitHub

[view raw](#)

The range measuring process is divided into a sequence of steps, and we should explicitly enable the steps we want to use. They have not explained in detail anywhere, but there are five of them:

Sequence step	Description (my guess)
Minimum signal rate check (MSRC)	Check if the signal rate is below a certain threshold, return error value if it is
Target CentreCheck (TCC)	Check to see if the target is centered?

Sequence step	Description (my guess)
Dynamic SPAD selection (DSS)	Dynamically select the active SPADs to avoid saturating from too much light
Pre Range	First stage of range measuring (I don't know what it does)
Final Range	Second stage of range measuring (I don't know what it does)

Following what ST does, I enable everything but MSRC and TCC.

```
1  #define RANGE_SEQUENCE_STEP_TCC (0x10) /* Target CentreCheck */
2  #define RANGE_SEQUENCE_STEP_MSRC (0x04) /* Minimum Signal Rate Check */
3  #define RANGE_SEQUENCE_STEP_DSS (0x28) /* Dynamic SPAD selection */
4  #define RANGE_SEQUENCE_STEP_PRE_RANGE (0x40)
5  #define RANGE_SEQUENCE_STEP_FINAL_RANGE (0x80)
6
7  /**
8   * Enable (or disable) specific steps in the sequence
9   */
10 static bool set_sequence_steps_enabled(uint8_t sequence_step)
11 {
12     return i2c_write_addr8_data8(REG_SYSTEM_SEQUENCE_CONFIG, sequence_step);
13 }
```

vl53l0x.c hosted with ❤ by GitHub

[view raw](#)

And the full static\_init function:

```
1  /**
2   * Basic device initialization
3   */
4  static bool static_init()
5  {
6      if (!load_default_tuning_settings()) {
7          return false;
8      }
9
10     if (!configure_interrupt()) {
11         return false;
12     }
13
14     if (!set_sequence_steps_enabled(RANGE_SEQUENCE_STEP_DSS +
15                                     RANGE_SEQUENCE_STEP_PRE_RANGE +
16                                     RANGE_SEQUENCE_STEP_FINAL_RANGE)) {
17         return false;
18     }
19
20     return true;
21 }
```

vl53l0x.c hosted with ❤ by GitHub

[view raw](#)

## Temperature calibration

Apart from the basic initialization, we need to calibrate to compensate for temperature, which is explained in the [datasheet](#):

### 2.4 Ref (temperature) calibration

Ref calibration is the calibration of two parameters (VHV and phase cal) which are temperature dependent. These two parameters are used to set the device sensitivity.

Ref calibration allows the adjustment of the device sensitivity when temperature varies.

Ref calibration must be performed during initial manufacturing calibration, it should be performed again when temperature varies more than 8degC compared to the initial calibration temperature.

If temperature does not vary, the ref calibration data can be loaded without re-performing the calibration procedure.

Glancing at the ST API driver once again, I created the corresponding code:

```
1  typedef enum
2  {
3      CALIBRATION_TYPE_VHV,
4      CALIBRATION_TYPE_PHASE
5  } calibration_type_t;
6
7  static bool perform_single_ref_calibration(calibration_type_t calib_type)
8  {
9      uint8_t sysrange_start = 0;
10     uint8_t sequence_config = 0;
11     switch (calib_type)
12     {
13     case CALIBRATION_TYPE_VHV:
14         sequence_config = 0x01;
15         sysrange_start = 0x01 | 0x40;
16         break;
17     case CALIBRATION_TYPE_PHASE:
18         sequence_config = 0x02;
19         sysrange_start = 0x01 | 0x00;
20         break;
21     }
22     if (!i2c_write_addr8_data8(REG_SYSTEM_SEQUENCE_CONFIG, sequence_config)) {
23         return false;
24     }
25     if (!i2c_write_addr8_data8(REG_SYSRANGE_START, sysrange_start)) {
26         return false;
27     }
28     /* Wait for interrupt */
29     uint8_t interrupt_status = 0;
30     bool success = false;
31     do {
32         success = i2c_read_addr8_data8(REG_RESULT_INTERRUPT_STATUS, &interrupt_status)
33     } while (success && ((interrupt_status & 0x07) == 0));
34     if (!success) {
35         return false;
36     }
37     if (!i2c_write_addr8_data8(REG_SYSTEM_INTERRUPT_CLEAR, 0x01)) {
38         return false;
39     }
40
41     if (!i2c_write_addr8_data8(REG_SYSRANGE_START, 0x00)) {
42         return false;
43     }
44     return true;
45 }
46
47 /**
```



```
48  * Temperature calibration needs to be run again if the temperature changes by
49  * more than 8 degrees according to the datasheet.
50  */
51  static bool perform_ref_calibration()
52  {
53      if (!perform_single_ref_calibration(CALIBRATION_TYPE_VHV)) {
54          return false;
55      }
56      if (!perform_single_ref_calibration(CALIBRATION_TYPE_PHASE)) {
57          return false;
58      }
59      /* Restore sequence steps enabled */
60      if (!set_sequence_steps_enabled(RANGE_SEQUENCE_STEP_DSS +
61                                     RANGE_SEQUENCE_STEP_PRE_RANGE +
62                                     RANGE_SEQUENCE_STEP_FINAL_RANGE)) {
63          return false;
64      }
65      return true;
66  }
```

vl53l0x.c hosted with ❤ by GitHub

[view raw](#)

Finally, we should define the "public" init function, which calls all of the init functions we have defined.

```
1  bool vl53l0x_init()
2  {
3      if (!device_is_booted()) {
4          return false;
5      }
6      if (!data_init()) {
7          return false;
8      }
9      if (!static_init()) {
10         return false;
11     }
12     if (!perform_ref_calibration()) {
13         return false;
14     }
15     return true;
16 }
```

vl53l0x.c hosted with ❤ by GitHub

[view raw](#)

```
1  #ifndef VL53L0X_H
2  #define VL53L0X_H
3
4  #include <stdbool.h>
5  #include <stdint.h>
6
7  bool vl53l0x_init(void);
8
9  #endif /* VL53L0X_H */
```

vl53l0x.h hosted with ❤ by GitHub

[view raw](#)

After calling this function, the sensor is ready for range measuring.

For the complete code until this point, look at [this commit](#).

## Single range measurement

Once the sensor is initialized, it only takes a few lines of code to do a single range measurement. These are the steps:

1. Stop any ongoing range measuring
2. Trigger a new range measurement
3. Wait for it to start

4. Poll interrupt
5. Read range
6. Clear the interrupt

And in code:

```
1  bool vl53l0x_read_range_single(uint16_t *range)
2  {
3      bool success = i2c_write_addr8_data8(0x80, 0x01);
4      success &= i2c_write_addr8_data8(0xFF, 0x01);
5      success &= i2c_write_addr8_data8(0x00, 0x00);
6      success &= i2c_write_addr8_data8(0x91, stop_variable);
7      success &= i2c_write_addr8_data8(0x00, 0x01);
8      success &= i2c_write_addr8_data8(0xFF, 0x00);
9      success &= i2c_write_addr8_data8(0x80, 0x00);
10     if (!success) {
11         return false;
12     }
13
14     if (!i2c_write_addr8_data8(REG_SYSRANGE_START, 0x01)) {
15         return false;
16     }
17
18     uint8_t sysrange_start = 0;
19     do {
20         success = i2c_read_addr8_data8(REG_SYSRANGE_START, &sysrange_start);
21     } while (success && (sysrange_start & 0x01));
22     if (!success) {
23         return false;
24     }
25
26     uint8_t interrupt_status = 0;
27     do {
28         success = i2c_read_addr8_data8(REG_RESULT_INTERRUPT_STATUS, &interrupt_status);
29     } while (success && ((interrupt_status & 0x07) == 0));
30     if (!success) {
31         return false;
32     }
33
34     if (!i2c_read_addr8_data16(REG_RESULT_RANGE_STATUS + 10, range)) {
35         return false;
36     }
37
38     if (!i2c_write_addr8_data8(REG_SYSTEM_INTERRUPT_CLEAR, 0x01)) {
39         return false;
40     }
41
42     /* 8190 or 8191 may be returned when obstacle is out of range. */
43     if (*range == 8190 || *range == 8191) {
44         *range = VL53L0X_OUT_OF_RANGE;
45     }
46
47     return true;
```

48 }

vl53l0x.c hosted with ❤ by GitHub

[view raw](#)

```
1  #ifndef VL53L0X_H
2  #define VL53L0X_H
3
4  #include <stdbool.h>
5  #include <stdint.h>
6
7  #define VL53L0X_OUT_OF_RANGE (8190)
8
9  bool vl53l0x_init(void);
10
11 /**
12  * Does a single range measurement
13  * @param range contains the measured range or VL53L0X_OUT_OF_RANGE
14  *         if out of range.
15  * @return True if success, False if error
16  * @note    Polling-based
17  */
18 bool vl53l0x_read_range_single(uint16_t *range);
19
20 #endif /* VL53L0X_H */
```

vl53l0x.h hosted with ❤ by GitHub

[view raw](#)

The complete code until this point.

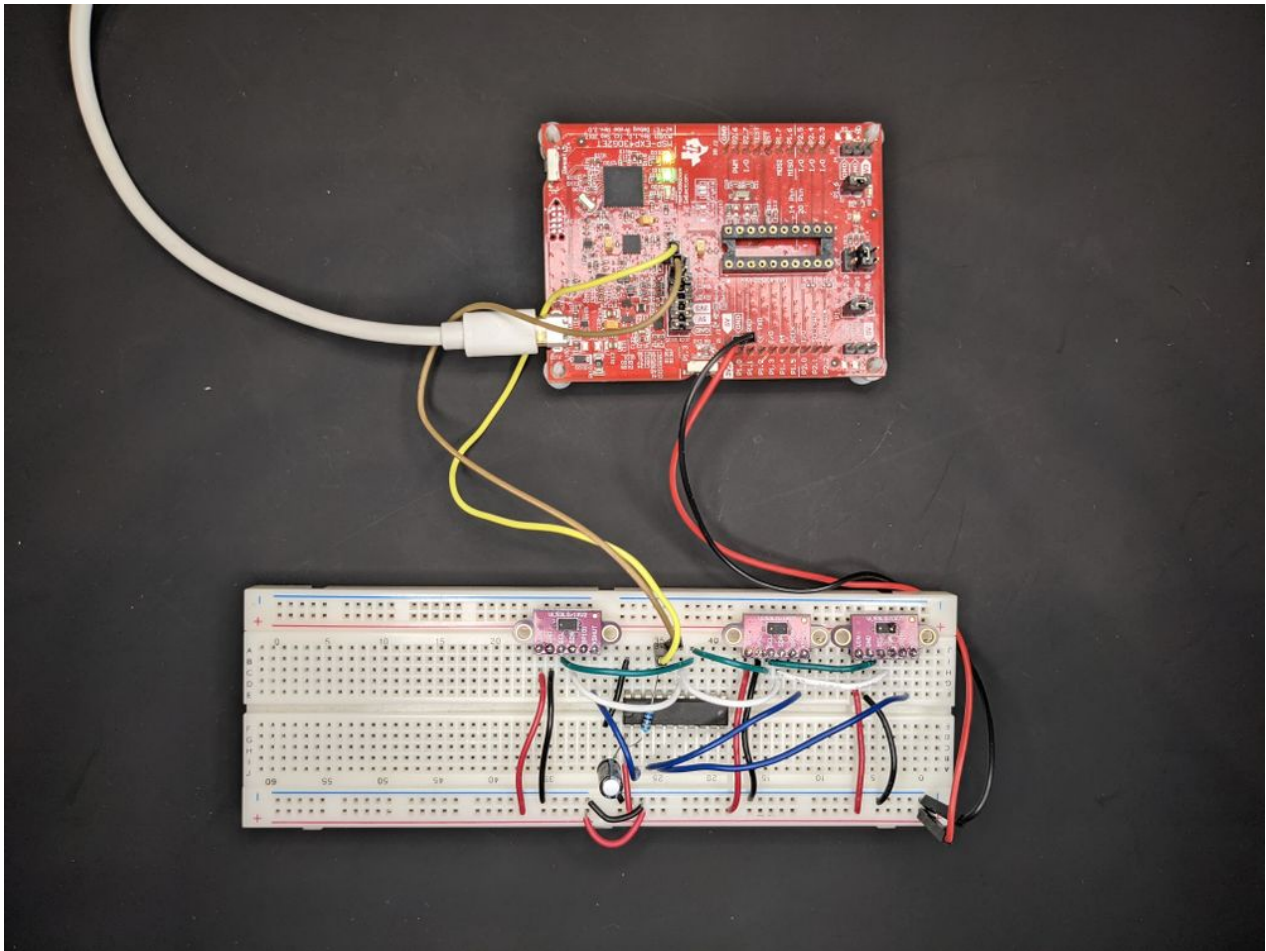
## Multi-sensor support

We got a single sensor working, but let's also add multi-sensor support.

When using multiple VL53L0X on the same I2C bus, we have the same problem as with the VL6180X, we get bus collision because they share the same default I2C address. We solve it the same way: put all sensors in hardware standby and then power them up and configure their address individually. This solution is described in the application note [AN4846](#).

## How to hook up multiple VL53L0X

Besides connecting the sensors to the VCC, GND, SDA, and SCL pin, we should also connect the XSHUT (GPIO0) pin for hardware standby. I will be using three sensors as an example and connect their XSHUT to p1.0, p1.1 and p1.2 of the MSP430.



**NOTE:** Consider the pull-up resistance if you use many breakout boards. The effective resistance may become too low if you use many breakout boards, and you may need to desolder the pull-ups from some of them. Though three boards worked fine for me (~3.3 kOhm pull-up).

## Modify the driver to support multiple sensors

The code is almost identical to VL6180X, so please read [my other post](#) for more details. One difference is that there is no "fresh out of reset" register, so we read the device id register instead. Another is that VL53L0X returns 8190 or 8191 when nothing is blocking them, but this is by default. Note, I'm not sure, but the limit checks may affect this return value (see [Ranging profiles](#)).

The new code (probably easier to study [the commit on GitHub](#)):

```
1  #define VL53L0X_DEFAULT_ADDRESS (0x29)
2
3  static const vl53l0x_info_t vl53l0x_infos[] =
4  {
5      [VL53L0X_IDX_FIRST] = { .addr = 0x30, .xshut_gpio = GPIO_XSHUT_FIRST },
6  #ifdef VL53L0X_SECOND
7      [VL53L0X_IDX_SECOND] = { .addr = 0x31, .xshut_gpio = GPIO_XSHUT_SECOND },
8  #endif
9  #ifdef VL53L0X_THIRD
10     [VL53L0X_IDX_THIRD] = { .addr = 0x32, .xshut_gpio = GPIO_XSHUT_THIRD },
11 #endif
12 };
13
14 static bool configure_address(uint8_t addr)
15 {
16     /* 7-bit address */
17     return i2c_write_addr8_data8(REG_SLAVE_DEVICE_ADDRESS, addr & 0x7F);
18 }
19
20 /**
21  * Sets the sensor in hardware standby by flipping the XSHUT pin.
22  */
23 static void set_hardware_standby(vl53l0x_idx_t idx, bool enable)
24 {
25     gpio_set_output(vl53l0x_infos[idx].xshut_gpio, !enable);
26 }
27
28 /**
29  * Configures the GPIOs used for the XSHUT pin.
30  * Output low by default means the sensors will be in
31  * hardware standby after this function is called.
32  *
33  * NOTE: The pins are hard-coded to P1.0, P1.1, and P1.2.
34  */
35 static void configure_gpio()
36 {
37     gpio_init();
38     gpio_set_output(GPIO_XSHUT_FIRST, false);
39     gpio_set_output(GPIO_XSHUT_SECOND, false);
40     gpio_set_output(GPIO_XSHUT_THIRD, false);
41 }
42
43 static bool configure_address(uint8_t addr)
44 {
45     /* 7-bit address */
46     return i2c_write_addr8_data8(REG_SLAVE_DEVICE_ADDRESS, addr & 0x7F);
47 }
```



```
48
49  /**
50   * Sets the sensor in hardware standby by flipping the XSHUT pin.
51   */
52  static void set_hardware_standby(vl53l0x_idx_t idx, bool enable)
53  {
54      gpio_set_output(vl53l0x_infos[idx].xshut_gpio, !enable);
55  }
56
57  /**
58   * Configures the GPIOs used for the XSHUT pin.
59   * Output low by default means the sensors will be in
60   * hardware standby after this function is called.
61   *
62   * NOTE: The pins are hard-coded to P1.0, P1.1, and P1.2.
63   */
64  static void configure_gpio()
65  {
66      gpio_init();
67      gpio_set_output(GPIO_XSHUT_FIRST, false);
68      gpio_set_output(GPIO_XSHUT_SECOND, false);
69      gpio_set_output(GPIO_XSHUT_THIRD, false);
70  }
71
72  /* Sets the address of a single VL53L0X sensor.
73   * This functions assumes that all non-configured VL53L0X are still
74   * in hardware standby. */
75  static bool init_address(vl53l0x_idx_t idx)
76  {
77      set_hardware_standby(idx, false);
78      i2c_set_slave_address(VL53L0X_DEFAULT_ADDRESS);
79
80      /* The datasheet doesn't say how long we must wait to leave hw standby,
81       * but using the same delay as vl6180x seems to work fine. */
82      __delay_cycles(400);
83
84      if (!device_is_booted()) {
85          return false;
86      }
87
88      if (!configure_address(vl53l0x_infos[idx].addr)) {
89          return false;
90      }
91      return true;
92  }
93
94  /**
95   * Initializes the sensors by putting them in hw standby and then
```

```
96  * waking them up one-by-one as described in AN4846.
97  */
98  static bool init_addresses()
99  {
100     /* Puts all sensors in hardware standby */
101     configure_gpio();
102
103     /* Wake each sensor up one by one and set a unique address for each one */
104     if (!init_address(VL53L0X_IDX_FIRST)) {
105         return false;
106     }
107     #ifdef VL53L0X_SECOND
108     if (!init_address(VL53L0X_IDX_SECOND)) {
109         return false;
110     }
111     #endif
112     #ifdef VL53L0X_THIRD
113     if (!init_address(VL53L0X_IDX_THIRD)) {
114         return false;
115     }
116     #endif
117
118     return true;
119 }
120
121 static bool init_config(vl53l0x_idx_t idx)
122 {
123     i2c_set_slave_address(vl53l0x_infos[idx].addr);
124     if (!data_init()) {
125         return false;
126     }
127     if (!static_init()) {
128         return false;
129     }
130     if (!perform_ref_calibration()) {
131         return false;
132     }
133     return true;
134 }
135
136 bool vl53l0x_init()
137 {
138     if (!init_addresses()) {
139         return false;
140     }
141     if (!init_config(VL53L0X_IDX_FIRST)) {
142         return false;
143     }
144     if (!init_config(VL53L0X_IDX_SECOND)) {
145         return false;
146     }
147     if (!init_config(VL53L0X_IDX_THIRD)) {
148         return false;
149     }
150 }
```

```
144     #ifdef VL53L0X_SECOND
145         if (!init_config(VL53L0X_IDX_SECOND)) {
146             return false;
147         }
148     #endif
149     #ifdef VL53L0X_THIRD
150         if (!init_config(VL53L0X_IDX_THIRD)) {
151             return false;
152         }
153     #endif
154     return true;
155 }
156
157 bool vl53l0x_read_range_single(vl53l0x_idx_t idx, uint16_t *range)
158 {
159     i2c_set_slave_address(vl53l0x_infos[idx].addr);
160     bool success = i2c_write_addr8_data8(0x80, 0x01);
161     success &= i2c_write_addr8_data8(0xFF, 0x01);
162     success &= i2c_write_addr8_data8(0x00, 0x00);
163     success &= i2c_write_addr8_data8(0x91, stop_variable);
164     success &= i2c_write_addr8_data8(0x00, 0x01);
165     success &= i2c_write_addr8_data8(0xFF, 0x00);
166     success &= i2c_write_addr8_data8(0x80, 0x00);
167     if (!success) {
168         return false;
169     }
170
171     if (!i2c_write_addr8_data8(REG_SYSRANGE_START, 0x01)) {
172         return false;
173     }
174
175     uint8_t sysrange_start = 0;
176     do {
177         success = i2c_read_addr8_data8(REG_SYSRANGE_START, &sysrange_start);
178     } while (success && (sysrange_start & 0x01));
179     if (!success) {
180         return false;
181     }
182
183     uint8_t interrupt_status = 0;
184     do {
185         success = i2c_read_addr8_data8(REG_RESULT_INTERRUPT_STATUS, &interrupt_status);
186     } while (success && ((interrupt_status & 0x07) == 0));
187     if (!success) {
188         return false;
189     }
190
```

```
191     if (!i2c_read_addr8_data16(REG_RESULT_RANGE_STATUS + 10, range)) {
192         return false;
193     }
194
195     if (!i2c_write_addr8_data8(REG_SYSTEM_INTERRUPT_CLEAR, 0x01)) {
196         return false;
197     }
198
199     /* 8190 or 8191 may be returned when obstacle is out of range. */
200     if (*range == 8190 || *range == 8191) {
201         *range = VL53L0X_OUT_OF_RANGE;
202     }
203
204     return true;
205 }
```

vl53l0x.c hosted with ❤ by GitHub

[view raw](#)

```
1  #ifndef VL53L0X_H
2  #define VL53L0X_H
3
4  #include <stdbool.h>
5  #include <stdint.h>
6
7  #define VL53L0X_OUT_OF_RANGE (8190)
8
9  /* Comment these out if not connected */
10 #define VL53L0X_SECOND
11 #define VL53L0X_THIRD
12
13 typedef enum
14 {
15     VL53L0X_IDX_FIRST,
16     #ifdef VL53L0X_SECOND
17     VL53L0X_IDX_SECOND,
18     #endif
19     #ifdef VL53L0X_THIRD
20     VL53L0X_IDX_THIRD,
21     #endif
22 } vl53l0x_idx_t;
23
24 /**
25  * Initializes the sensors in the vl53l0x_idx_t enum.
26  * @note Each sensor must have its XSHUT pin connected.
27  */
28 bool vl53l0x_init(void);
29
30 /**
```

```
31  * Does a single range measurement
32  * @param idx selects specific sensor
33  * @param range contains the measured range or VL53L0X_OUT_OF_RANGE
34  *         if out of range.
35  * @return True if success, False if error
36  * @note    Polling-based
37  */
38  bool vl53l0x_read_range_single(vl53l0x_idx_t idx, uint16_t *range);
39
40  #endif /* VL53L0X_H */
```

vl53l0x.h hosted with ❤ by GitHub

[view raw](#)

## Additional notes

This post aimed to give you a minimal code example for VL53L0X. There is much more you can configure to better adapt the VL53L0X for your application. I won't go into everything, but I do want to talk about some of it in this section.

## SPAD management

One thing that ST doesn't explain in detail is the configuration of the SPAD array.

The single-photon avalanche diodes (SPADs) array is the set of diodes that measures the reflected light that's emitted by the laser. The VL53L0X has  $16 \times 16 = 256$  of them, but only a handful of them should be used to achieve a good signal rate. Exactly which SPADs to enable depends on the particular VL53L0X, operating conditions, and whether a cover glass is used or not.

During production, ST calibrates and generates a good reference SPAD map, which marks the best diodes (I think) among the 256 diodes. This map is different for each VL53L0X unit, and ST saves it to the non-volatile memory (NVM).

From what I've extracted, we basically have three options for configuring the SPADs:

**Option 1: Do nothing:** By default, the SPADs are enabled according to the good reference SPAD map

**Option 2: Enable a subset of the good reference map:** Only a subset of the good reference SPAD map should be enabled for optimal signal rate. ST also saves the number and type of SPADs to enable from the good reference SPAD map, which is based on the SPAD calibration they run in production. We can retrieve these values from NVM to reconfigure the SPAD map accordingly.

**Option 3: Run SPAD management calibration:** We can run the SPAD management calibration ourselves to determine the best SPADs to enable from the good reference SPAD map.

I quickly compared all three using a breakout board (no cover glass) and in relatively normal operating conditions. In my case, the difference is small, but **option 2** gives slightly better readings than **option 1**, and **option 3** slightly better than **option 2**. The difference is likely much more noticeable with a cover glass.

The code we have written so far relies on **option 1**. I will go ahead and also implement **option 2**, but then only discuss **option 3**.

## SPADs

Before moving on, there are a few things you should know about the SPADs.

There are two types of SPADs, *aperture* and *non-aperture*, and only one type of SPAD should be enabled simultaneously. ST doesn't explain the difference between these two, but if it's related to the aperture concept of a digital camera, it probably means that the aperture SPADs are covered to capture less light.

In total, there are  $16 \times 16 = 256$  SPADs divided into four quadrants where the third quadrant is non-aperture and the rest is aperture. And if I've interpreted ST's code right, we should only enable SPADs within an area of 44 SPADs. ST has also hardcoded the start of this area to the 180th diode, meaning the area only spans the third and fourth quadrant.

## Option 2: Load SPAD config from NVM

To load the SPAD configuration from the NVM, we must set up the registers to read the data from NVM, and then create and write the SPAD configuration accordingly. We should do this as part of the *static initialization*.

Below I've extracted the required parts from the ST driver and tried to simplify it as much as possible:



```
1  /* There are two types of SPAD: aperture and non-aperture. My understanding
2   * is that aperture ones let it less light (they have a smaller opening), similar
3   * to how you can change the aperture on a digital camera. Only 1/4 th of the
4   * SPADs are of type non-aperture. */
5  #define SPAD_TYPE_APERTURE (0x01)
6  /* The total SPAD array is 16x16, but we can only activate a quadrant spanning 44 SPAD
7   * a time. In the ST api code they have (for some reason) selected 0xB4 (180) as a sta
8   * point (lies in the middle and spans non-aperture (3rd) quadrant and aperture (4th)
9  #define SPAD_START_SELECT (0xB4)
10 /* The total SPAD map is 16x16, but we should only activate an area of 44 SPADs at a t
11 #define SPAD_MAX_COUNT (44)
12 /* The 44 SPADs are represented as 6 bytes where each bit represents a single SPAD.
13  * 6x8 = 48, so the last four bits are unused. */
14 #define SPAD_MAP_ROW_COUNT (6)
15 #define SPAD_ROW_SIZE (8)
16 /* Since we start at 0xB4 (180), there are four quadrants (three aperture, one aperture
17  * and each quadrant contains 256 / 4 = 64 SPADs, and the third quadrant is non-aperture
18  * offset to the aperture quadrant is (256 - 64 - 180) = 12 */
19 #define SPAD_APERTURE_START_INDEX (12)
20
21 /**
22  * Wait for strobe value to be set. This is used when we read values
23  * from NVM (non volatile memory).
24  */
25 static bool read_strobe()
26 {
27     bool success = false;
28     uint8_t strobe = 0;
29     if (!i2c_write_addr8_data8(0x83, 0x00)) {
30         return false;
31     }
32     do {
33         success = i2c_read_addr8_data8(0x83, &strobe);
34     } while (success && (strobe == 0));
35     if (!success) {
36         return false;
37     }
38     if (!i2c_write_addr8_data8(0x83, 0x01)) {
39         return false;
40     }
41     return true;
42 }
43
44 /**
45  * Gets the spad count, spad type och "good" spad map stored by ST in NVM at
46  * their production line.
47  * .
```

```

48  * According to the datasheet, ST runs a calibration (without cover glass) and
49  * saves a "good" SPAD map to NVM (non volatile memory). The SPAD array has two
50  * types of SPADs: aperture and non-aperture. By default, all of the
51  * good SPADs are enabled, but we should only enable a subset of them to get
52  * an optimized signal rate. We should also only enable either only aperture
53  * or only non-aperture SPADs. The number of SPADs to enable and which type
54  * are also saved during the calibration step at ST factory and can be retrieved
55  * from NVM.
56  */
57  static bool get_spad_info_from_nvm(uint8_t *spad_count, uint8_t *spad_type, uint8_t g
58  {
59      bool success = false;
60      uint8_t tmp_data8 = 0;
61      uint32_t tmp_data32 = 0;
62
63      /* Setup to read from NVM */
64      success = i2c_write_addr8_data8(0x80, 0x01);
65      success &= i2c_write_addr8_data8(0xFF, 0x01);
66      success &= i2c_write_addr8_data8(0x00, 0x00);
67      success &= i2c_write_addr8_data8(0xFF, 0x06);
68      success &= i2c_read_addr8_data8(0x83, &tmp_data8);
69      success &= i2c_write_addr8_data8(0x83, tmp_data8 | 0x04);
70      success &= i2c_write_addr8_data8(0xFF, 0x07);
71      success &= i2c_write_addr8_data8(0x81, 0x01);
72      success &= i2c_write_addr8_data8(0x80, 0x01);
73      if (!success) {
74          return false;
75      }
76
77      /* Get the SPAD count and type */
78      success &= i2c_write_addr8_data8(0x94, 0x6b);
79      if (!success) {
80          return false;
81      }
82      if (!read_strobe()) {
83          return false;
84      }
85      success &= i2c_read_addr8_data32(0x90, &tmp_data32);
86      if (!success) {
87          return false;
88      }
89      *spad_count = (tmp_data32 >> 8) & 0x7f;
90      *spad_type = (tmp_data32 >> 15) & 0x01;
91
92      /* Since the good SPAD map is already stored in REG_GLOBAL_CONFIG_SPAD_ENABLES_RE
93      * we can simply read that register instead of doing the below */
94      #if 0
95      /* Get the first part of the SPAD map */

```

```
96     if (!i2c_write_addr8_data8(0x94, 0x24)) {
97         return false;
98     }
99     if (!read_strobe()) {
100         return false;
101     }
102     if (!i2c_read_addr8_data32(0x90, &tmp_data32)) {
103         return false;
104     }
105     good_spad_map[0] = (uint8_t)((tmp_data32 >> 24) & 0xFF);
106     good_spad_map[1] = (uint8_t)((tmp_data32 >> 16) & 0xFF);
107     good_spad_map[2] = (uint8_t)((tmp_data32 >> 8) & 0xFF);
108     good_spad_map[3] = (uint8_t)(tmp_data32 & 0xFF);
109
110     /* Get the second part of the SPAD map */
111     if (!i2c_write_addr8_data8(0x94, 0x25)) {
112         return false;
113     }
114     if (!read_strobe()) {
115         return false;
116     }
117     if (!i2c_read_addr8_data32(0x90, &tmp_data32)) {
118         return false;
119     }
120     good_spad_map[4] = (uint8_t)((tmp_data32 >> 24) & 0xFF);
121     good_spad_map[5] = (uint8_t)((tmp_data32 >> 16) & 0xFF);
122
123     #endif
124
125     /* Restore after reading from NVM */
126     success &= i2c_write_addr8_data8(0x81, 0x00);
127     success &= i2c_write_addr8_data8(0xFF, 0x06);
128     success &= i2c_read_addr8_data8(0x83, &tmp_data8);
129     success &= i2c_write_addr8_data8(0x83, tmp_data8 & 0xfb);
130     success &= i2c_write_addr8_data8(0xFF, 0x01);
131     success &= i2c_write_addr8_data8(0x00, 0x01);
132     success &= i2c_write_addr8_data8(0xFF, 0x00);
133     success &= i2c_write_addr8_data8(0x80, 0x00);
134
135     /* When we haven't configured the SPAD map yet, the SPAD map register actually
136      * contains the good SPAD map, so we can retrieve it straight from this register
137      * instead of reading it from the NVM. */
138     if (!i2c_read_addr8_bytes(REG_GLOBAL_CONFIG_SPAD_ENABLES_REF_0, good_spad_map, 6))
139         return false;
140 }
141 return success;
142 }
```

```

144  /**
145   * Sets the SPADs according to the value saved to NVM by ST during production. Assuming
146   * similar conditions (e.g. no cover glass), this should give reasonable readings and
147   * can avoid running ref spad management (tedious code).
148   */
149  static bool set_spads_from_nvm()
150  {
151      uint8_t spad_map[SPAD_MAP_ROW_COUNT] = { 0 };
152      uint8_t good_spad_map[SPAD_MAP_ROW_COUNT] = { 0 };
153      uint8_t spads_enabled_count = 0;
154      uint8_t spads_to_enable_count = 0;
155      uint8_t spad_type = 0;
156      volatile uint32_t total_val = 0;
157
158      if (!get_spad_info_from_nvm(&spads_to_enable_count, &spad_type, good_spad_map)) {
159          return false;
160      }
161
162      for (int i = 0; i < 6; i++) {
163          total_val += good_spad_map[i];
164      }
165
166      bool success = i2c_write_addr8_data8(0xFF, 0x01);
167      success &= i2c_write_addr8_data8(REG_DYNAMIC_SPAD_REF_EN_START_OFFSET, 0x00);
168      success &= i2c_write_addr8_data8(REG_DYNAMIC_SPAD_NUM_REQUESTED_REF_SPAD, 0x2C);
169      success &= i2c_write_addr8_data8(0xFF, 0x00);
170      success &= i2c_write_addr8_data8(REG_GLOBAL_CONFIG_REF_EN_START_SELECT, SPAD_START);
171      if (!success) {
172          return false;
173      }
174
175      uint8_t offset = (spad_type == SPAD_TYPE_APERTURE) ? SPAD_APERTURE_START_INDEX : 0;
176
177      /* Create a new SPAD array by selecting a subset of the SPADs suggested by the good_spad_map.
178       * The subset should only have the number of type enabled as suggested by the readings from
179       * the NVM (spads_to_enable_count and spad_type). */
180      for (int row = 0; row < SPAD_MAP_ROW_COUNT; row++) {
181          for (int column = 0; column < SPAD_ROW_SIZE; column++) {
182              int index = (row * SPAD_ROW_SIZE) + column;
183              if (index >= SPAD_MAX_COUNT) {
184                  return false;
185              }
186              if (spads_enabled_count == spads_to_enable_count) {
187                  /* We are done */
188                  break;
189              }
190              if (index < offset) {

```

```
191         continue;
192     }
193     if ((good_spad_map[row] >> column) & 0x1) {
194         spad_map[row] |= (1 << column);
195         spads_enabled_count++;
196     }
197 }
198 if (spads_enabled_count == spads_to_enable_count) {
199     /* To avoid looping unnecessarily when we are already done. */
200     break;
201 }
202 }
203
204 if (spads_enabled_count != spads_to_enable_count) {
205     return false;
206 }
207
208 /* Write the new SPAD configuration */
209 if (!i2c_write_addr8_bytes(REG_GLOBAL_CONFIG_SPAD_ENABLES_REF_0, spad_map, SPAD_M
210     return false;
211 }
212
213 return true;
214 }
215
216 /**
217  * Basic device initialization
218  */
219 static bool static_init()
220 {
221     if (!set_spads_from_nvm()) {
222         return false;
223     }
224
225     if (!load_default_tuning_settings()) {
226         return false;
227     }
228
229     if (!configure_interrupt()) {
230         return false;
231     }
232
233     if (!set_sequence_steps_enabled(RANGE_SEQUENCE_STEP_DSS +
234                                     RANGE_SEQUENCE_STEP_PRE_RANGE +
235                                     RANGE_SEQUENCE_STEP_FINAL_RANGE)) {
236         return false;
237     }
238 }
```

```
239     return true;  
240 }
```

vl53l0x.c hosted with ❤ by GitHub

[view raw](#)

**NOTE:** *To save some lines of code, we can read the good reference SPAD map directly from the SPAD configuration register instead of from the NVM.*

## Option 3: Reference SPAD calibration

To get the best SPAD configuration, you should do SPAD calibration, which ST further explains:

### Reference SPADs calibration

In order to optimize the dynamic of the system, the reference SPADs have to be calibrated.

This step is performed on the bare modules during Final Module Test at STMicroelectronics, and the calibration data (SPAD numbers and type) are stored into the device NVM.

In case a cover glass is used on top of VL53L0X, the reference SPADs have to be re-calibrated by the customer.

Reference SPAD calibration needs to be done only once during the initial manufacturing calibration, calibration data should then be stored on the Host.

I won't implement it, but if you are interested, you should refer to `VL53L0X_PerformRefSpadManagement` in ST's API code. It takes some tedious code to get this working, so before you do, measure the difference with an Arduino first (Adafruit's library has support). You could also take a shortcut by extracting the resulting SPAD map using the Arduino and hard-code those values in your code.

**NOTE:** *SPAD calibration doesn't produce a new reference SPAD map. It only finds an optimal subset of SPADs to activate from the good reference map already stored in the NVM.*

**NOTE:** *Every sensor unit is unique, so you must run the SPAD calibration for each sensor.*

## Ranging profiles

Apart from configuring the SPAD map, you can configure the VL53L0X with different ranging profiles for a trade-off between speed, accuracy, and range. ST provides four example profiles:

7

Example API range profiles

There are 4 range profiles available via API example code.

Table 6. Example API range profiles			
	Timing budget	Typical max range	Typical application
Default mode	30ms	1.2m (white target)	standard
High accuracy	200ms	1.2m (white target)	precise measurement
Long range	33ms	2m (white target)	long ranging, only for dark conditions
High Speed	20ms	1.2m (white target)	high speed where accuracy is not priority

The API document explains what you need to do to configure each example profile. If you want something other than the default profile, you should read the API documentation and refer to these ST API functions:

*VL53L0X\_SetLimitCheckValue*: Change the limit checks, i.e., longer range at the cost of more false positives

*VL53L0X\_SetMeasurementTiming BudgetMicroSeconds*: Set the total time allowed for single range measurement, i.e., shorter time at the cost of accuracy.

*VL53L0X\_SetVcseIPulsePeriod*: Change the laser pulse period. A longer pulse period potentially gives a longer range.

Continuous (timed) mode

The VL53L0X can operate in two modes: *single* and *continuous*. In *single mode*, which we used in previous sections, we must start the range measuring each time. In *continuous mode*, range measurements run automatically one



after another, immediately or with a set interval (intermeasurement period). If you want to use *continuous mode* you should refer to the following ST API functions:

`VL53L0X_StartMeasurement()`

`VL53L0X_SetInterMeasurementPeriod MilliSeconds()`

`VL53L0X_GetInterMeasurementPeriod MilliSeconds()`

## Other things to look into

**Improve initialization time:** It's unnecessary to do the calibration each power-up. You can cut down startup time by saving the values to persistent memory

**Interrupt:** It's generally a good idea to wait for an interrupt rather than polling a register, especially if you configure the sensor to operate in continuous mode.

**Error handling:** I've kept error reporting to a bare minimum. You may want to report and handle errors better in your application.

**Cover glass:** If you use a cover glass, you must do the additional calibration.

**Many sensors:** A tip is to add a GPIO expander if you use many sensors and run out of pins on your microcontroller. You may also want to use different range profiles for the individual sensors.

**Gesture recognition:** VL53L0X is not only for range measurement, it also supports gesture recognition.

## Recommended reading

First of all, have a look at [the previous post](#).

The bulk of the documentation is to be found at [ST's page for VL53L0X](#):

- [User manual UM2039](#)

- [UM2039 API manual](#)
- [AN4846 Using multiple sensors](#)
- [A power point describing the technology in VL53L0X, VL6180X, and VL53L1X](#)

Have a look at the Arduino libraries as well, especially the one from Pololu because it doesn't rely on ST's API code.

- [Pololu's Arduino library](#)
- [Adafruit's Arduino library](#)

## Final words

I must admit it was a hassle to get this sensor working. On paper, the VL53L0X is a powerful range sensor in a small package for an affordable price, but its complexity and incomplete documentation makes it time-consuming to set up. ST does provide an extensive driver layer, but many times, you don't want to pull in an entire third-party driver layer into your project.

I have provided you with a minimal and portable code example for getting the VL53L0X up and running. In addition, I have given you information to help you better configure the VL53L0X for your application.

[The complete code here.](#)



### Niklas Nilsson

*I'm an embedded systems engineer from Sweden currently working at [Hasselblad](#).*

## Newsletter

G

Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 

?

Name

♥

1

Share

BestNewestOldest

Z

Zinahe Asnake

a year ago

Thank you. Thank you. Thank you.

10Reply • Share ›

S

shulong li

5 months ago edited

Thank you so much! It is very helpful. Do you have plan to do same research on VL53L1X?

00Reply • Share ›

K

Kevin Wolfe

a year ago

This is a fantastic blog post. Many thanks as well.

00Reply • Share ›