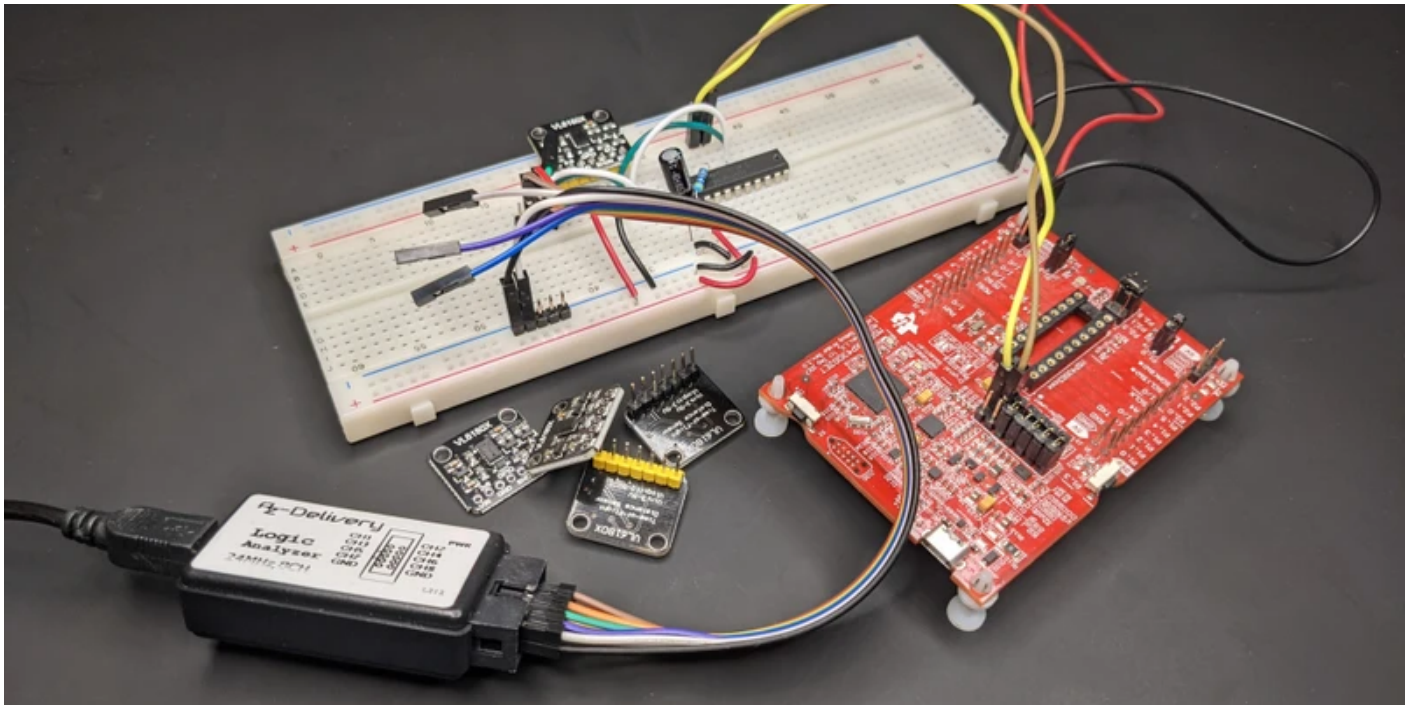
How to write a microcontroller driver for an I2C device? (MSP430 and VL6180X)

August 28, 2021 • 20 min read



▶ **Contents**

I recently had to write an I2C driver for talking to the range sensors on my sumobot. I2C is a standard protocol, one that every embedded developer should know, and the best way to learn it is to implement a driver for it, therefore, I thought it would be valuable to take you through my implementation.

What you should know

To follow along, you don't need to know the ins and outs of I2C, but you should have some familiarity - watch a youtube video or skim the specification sheet. You should also be familiar with the C programming language.

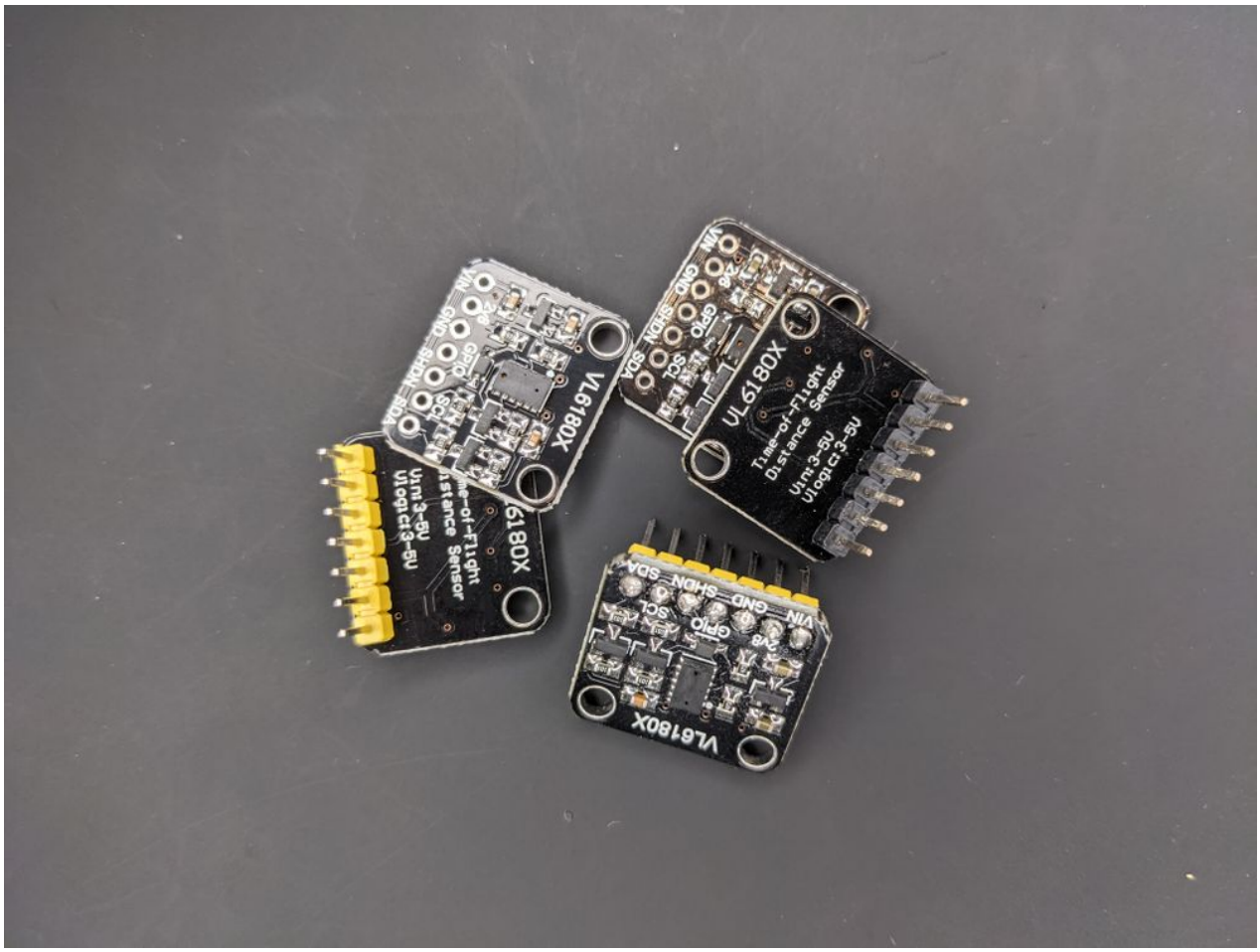
Hardware

I will write a driver on the microcontroller MSP430 for the range sensor VL6180X, and some of the code will obviously be tied to this hardware. Still, you should find this post helpful, even if you are writing a driver for something else.

NOTE: *I ended up using VL53L0X on my sumobot, but I wrote a driver for the VL6180X first. The VL53L0X is more complicated than VL6180X so I broke it into a second blog post.*

About VL6180X

The VL6180X is a range sensor by ST, which is part of a larger series of time-of-flight sensors, and has been on the market for a few years now. Using a technique called FlightSense, it measures the distance travelled by the light reflected from the target. In contrast to the more common approach - measuring the amount of light - it's independent of the target color and surface.



Besides measuring range (0-~20 cm), it can measure ambient light and recognize gestures, but I will only measure range.

For more information, have a look at ST's documentation

I picked this sensor for my project because it's tiny, cheap, and performant. It's a big step up from the bulkier range sensors you typically see in hobbyist projects like the Sharp series (e.g. GP2Y0A21YK0F) or the HC-SR04.

Where to get the VL6180X?

The VL6180X is available in a reflowable package, but the easiest way to test it is to get a breakout board. The breakout boards I've seen have the same pinout and go for about $5-15.

VL6180X pinout

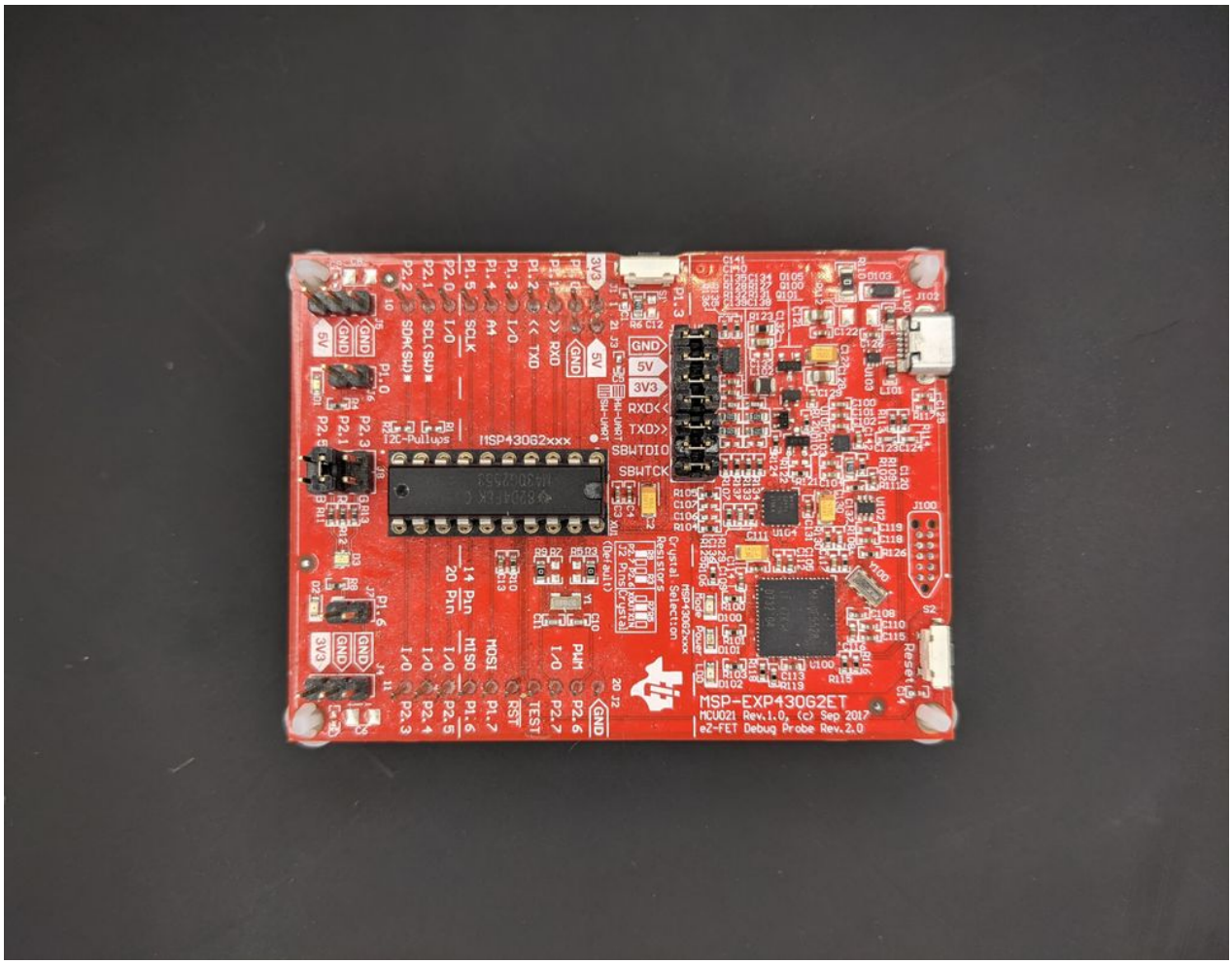
The typical VL6180X breakout board provides you with 7 pins:

Pin	Description/Usage
VIN	5V or 3.3V (but check your breakout board to make sure)
2v8	Just 2.8V, can normally be left unconnected
XSHUT	Used for putting the sensor in reset (needed for setting addr in multi-sensor) otherwise leave unconnected
GPIO	Used for receiving interrupt (can leave unconnected if polling)
GND	Ground
SDA	I2C data line
SCL	I2C clock line

That's a lot of pins, which is something to consider if you need to use several VL6180Xs because you may need to get a microcontroller package with a larger pin count or add a GPIO expander.

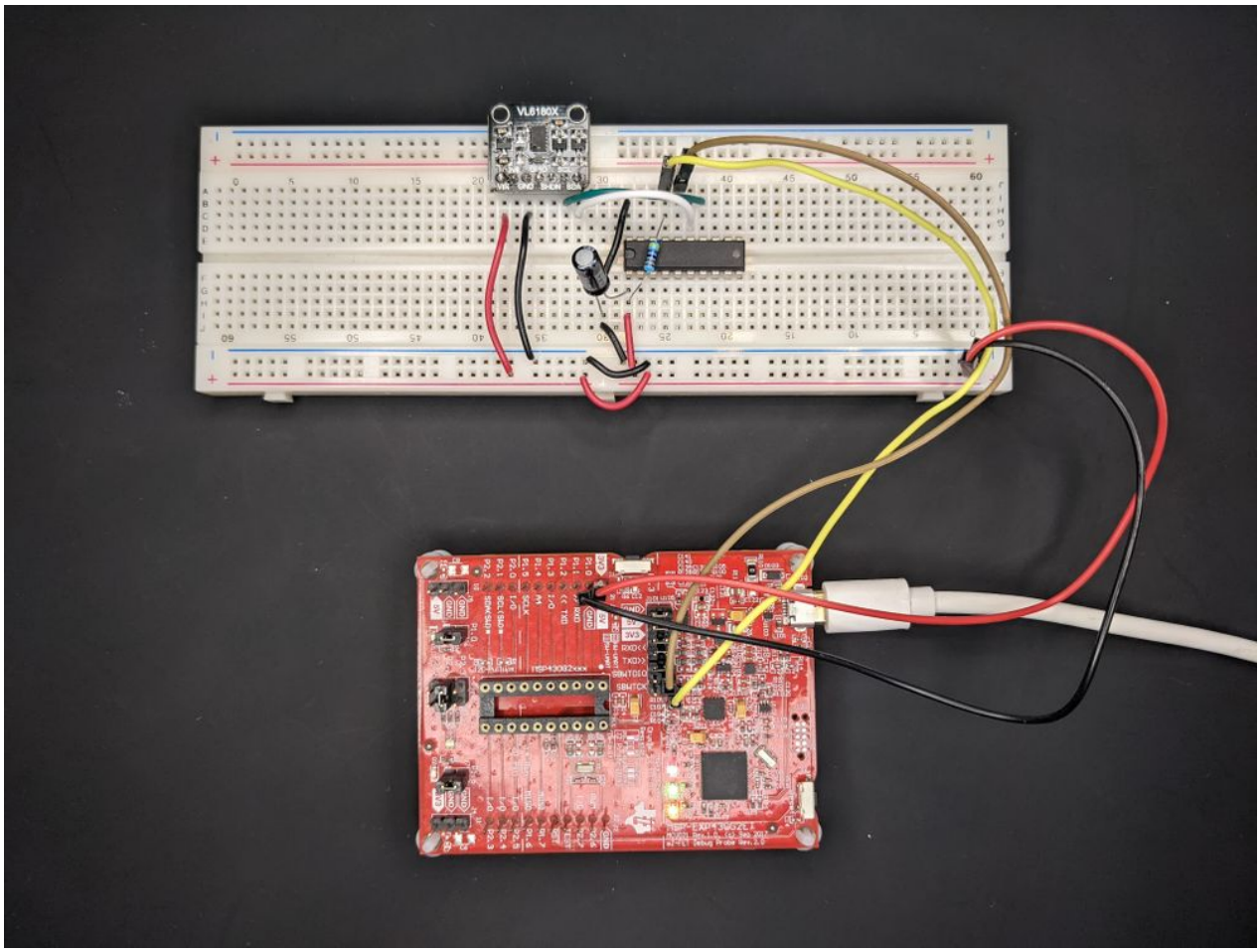
MSP430

I will write the code for an MSP430G2553 using the MSP430 LaunchPad. If you are new to embedded systems, the MSP430 is a great starting point because it has good tooling and documentation and a good set of peripherals.



How to hook up the VL6180X?

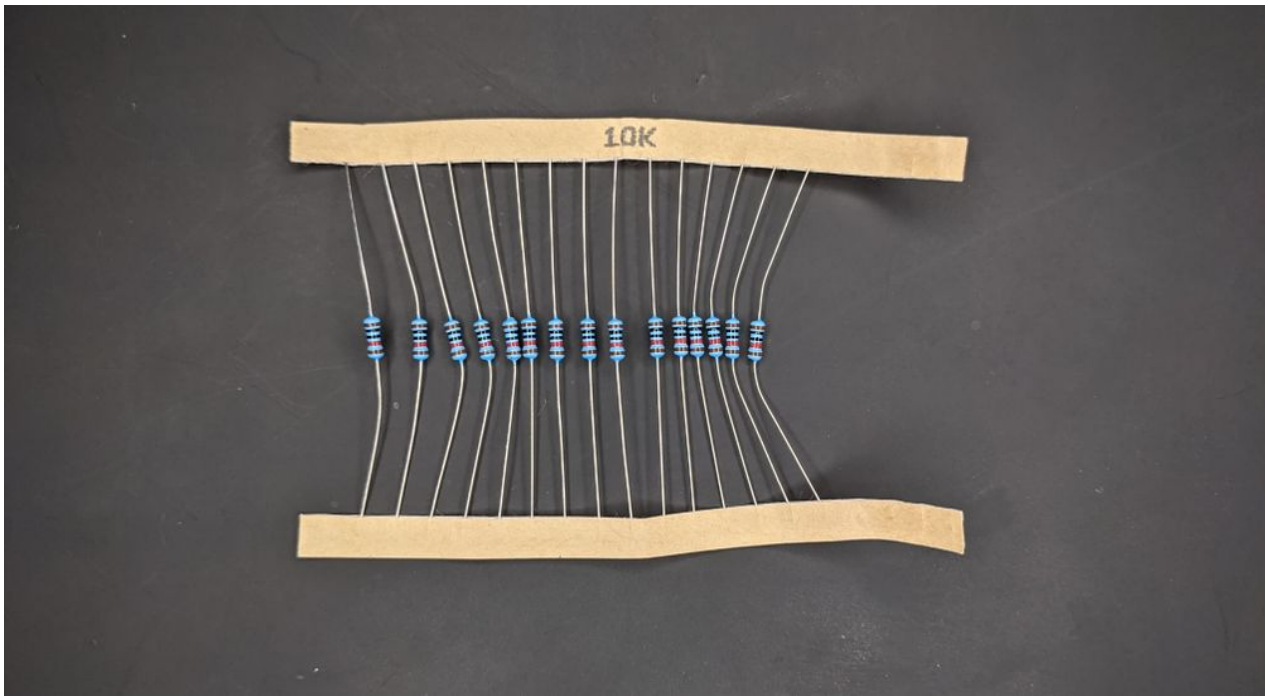
The bare-minimum you must connect to talk with a single VL53L0X are the power pins (VIN and GND) and the I2C pins (SDA and SCL). When we implement multi-sensor support in a later section, we also need to connect the XSHUT pin. We are only polling, so we can leave the GPIO (interrupt pin) unconnected.



NOTE: *I often move the MSP430 DIP package from the launchpad to a solderless board for easier prototyping and only use the launchpad for programming. If you do the same, make sure you add a 47 kOhm pull-up resistor on the reset line to prevent the MCU from endless reset. You should also add a bypass capacitor near the VCC pin to avoid voltage drop. Otherwise you are likely to experience intermittent reboots because of the current surges from the VL6180X.*

I2C pull-up resistors

A critical aspect of I2C is that it's designed to have the bus operate in open-drain, which means the I2C devices can only drive the bus lines low or not at all. They can't drive them high. Therefore, we need a mechanism to pull the lines high by default, and this is where the *pull-up resistors* come in. These resistors are connected between the I2C lines (SDA and SCL) and the VCC line (e.g. 3.3 V). If there are no pull-up resistors, there is no default logic level, so **the I2C communication will simply not work**.



What value should the pull-up resistors have?

Okay, so we need pull-up resistors, but what value should they have?

A too low value will make the pull-up too strong, preventing the I2C devices from pulling the lines low, which is why you can't connect the bus lines directly to VCC. It will also waste a lot of power.

A too-large value will make the pull-up too weak, making the lines go back to high state slower, limiting the communication speed. This is also why you typically don't use the internal pull-up (~100 kOhm) resistors of the microcontroller.

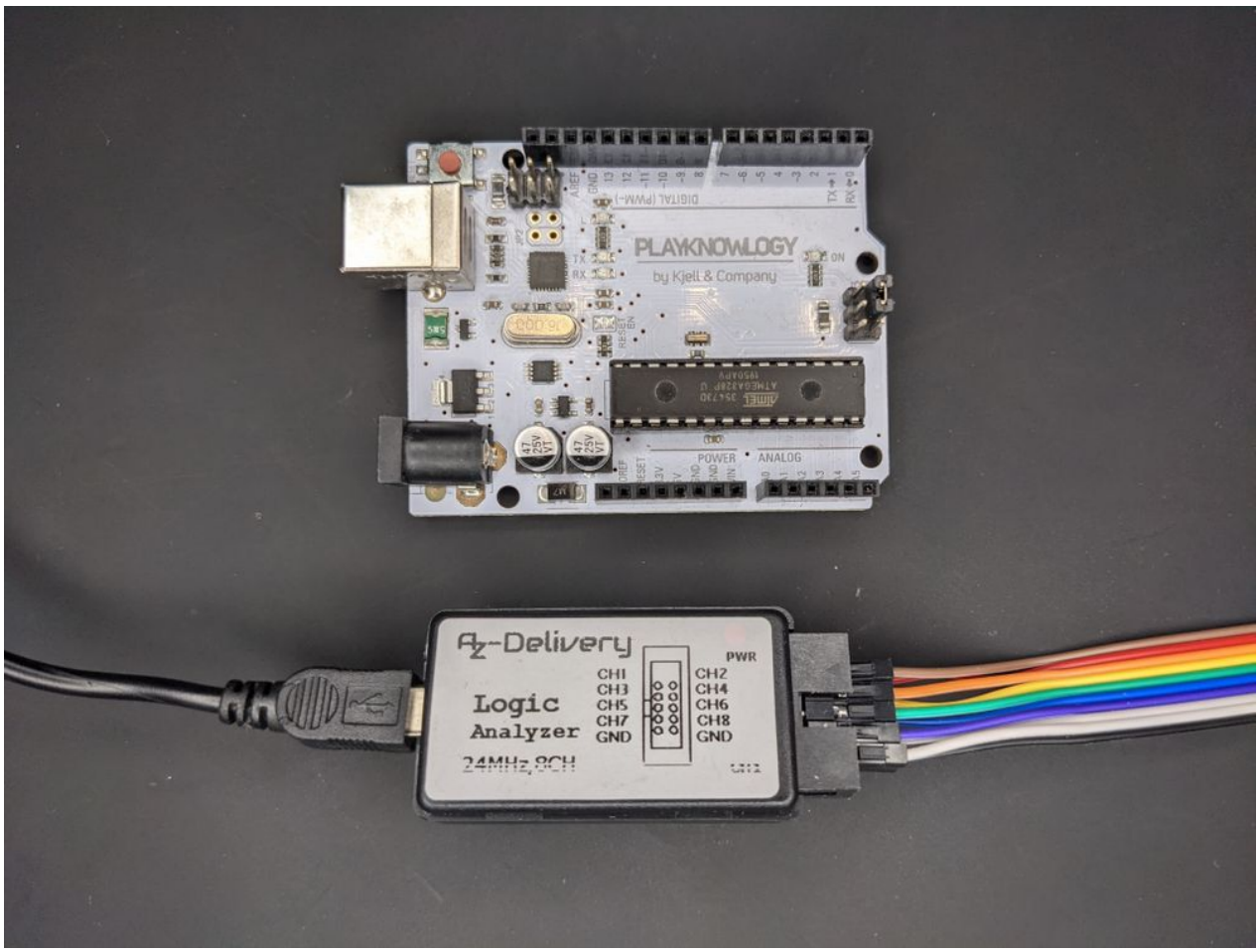
The optimal value depends on your setup, particularly the wire properties (bus capacitance) and the communication speed (rise time) you desire. I won't go into these details here. As a general rule of thumb, 4.7-10k typically works well.

NOTE: *Your breakout board will likely already have a pair of pull-ups - my VL6180X breakout board has 10k pull-ups. In that case, you don't need to think about it, but be cautious about connecting several breakout boards to the same I2C bus because then the effective pull-up resistance can become too low.*

Essential tools for debugging I2C

First, if you work with a protocol such as I2C, SPI, or UART, **you need a logic analyzer**. If you don't have one, get one. It's going to save you so much debugging time. Fair, you may not want to spend several hundred bucks on something like a Saleae, but at the very least get a cheap FX2-based analyzer (< 20 USD) and use it with sigrok.

Second, **get an Arduino**. Even if we are not coding for an Arduino, it's useful for verifying that your sensor works. There are good Arduino libraries that take 5 minutes to set up. Combined with a logic analyzer, it also gives you a reference on what I2C traffic to expect, which will save you plenty of time when writing your driver.



My approach

I will code in a bottom-up approach, starting with the I2C layer at the bottom and then move up to the VL6180X driver. The code will be polling-based (not

interrupt-driven) and have no sophisticated error handling or recovery. I have done this to simplify the code, but I do discuss these topics in another section.

NOTE: *If you stumble upon issues when implementing this yourself, have a look at Common issues.*

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.

Initialize the microcontroller

We have everything set up and the tools we need; It's time to start coding.

Starting from scratch, the first thing you need to do is create a new project in your editor of choice. I prefer to use Code Composer Studio and GCC for MSP430 development.

As with most microcontrollers, there is some initialization we must first do. On the MSP430, we at least need to configure the watchdog timer and the clock speed. The MSP430 goes up to 16 MHz, but configuring it for 1 MHz is enough here.

```c
#include <msp430.h>

static void msp430_init()
{
    WDTCTL = WDTPW | WDTHOLD; /* Stop watchdog timer */
    /* Configure the MCLK (and SMCLK) for 1 MHz using the Digitally controlled
     * oscillator (DCO) as source. */
    BCSCTL1 = CALBC1_1MHZ;
    DCOCTL = CALDCO_1MHZ;
}

int main()
{
    msp430_init();
    while(1);
}
```

main.c hosted with ❤ by GitHub view raw

If you are unsure about any of this, take a look in your microcontroller's datasheet or user's guide. Here is the one for MSP430.

I2C driver

Let's create two new files for the I2C driver *drivers/i2c.h* and *drivers/i2c.c*.

NOTE: *I like to keep my low-level drivers under a separate folder drivers/ and have a separate header and implementation file for each driver.*

Initialize I2C on MSP430

To use I2C, we must configure the pinout and I2C peripheral of the microcontroller. The GPIOs on a microcontroller can serve several functions, so we have to select the function explicitly.

The MSP430G2553 has 16 GPIO pins, and only pin 1.6 (SCL) and 1.7 (SDA) can be selected for I2C. We select I2C by writing to the select-register.

```
i2c_init() {
    /* Pinmux P1.6 (SCL) and P1.7 (SDA) to I2C peripheral  */
    P1SEL |= BIT6 + BIT7;
    P1SEL2 |= BIT6 + BIT7;
}
```

i2c.c hosted with ❤ by GitHub view raw

Next, we have to configure the I2C peripheral. On the MSP430, we do this by configuring the USCI module, which is a general module for I2C, SPI, and UART communication. In particular, we need to write to these registers:

17.4 USCI Registers: I²C Mode

The USCI registers applicable in I²C mode for USCI_B0 are listed in Table 17-2, and for USCI_B1 in Table 17-3.

Table 17-2. USCI_B0 Control and Status Registers

Register	Short Form	Register Type	Address	Initial State
USCI_B0 control register 0	UCB0CTL0	Read/write	068h	001h with PUC
USCI_B0 control register 1	UCB0CTL1	Read/write	069h	001h with PUC
USCI_B0 bit rate control register 0	UCB0BR0	Read/write	06Ah	Reset with PUC
USCI_B0 bit rate control register 1	UCB0BR1	Read/write	06Bh	Reset with PUC
USCI_B0 I²C interrupt enable register	UCB0I2CIE	Read/write	06Ch	Reset with PUC
USCI_B0 status register	UCB0STAT	Read/write	06Dh	Reset with PUC
USCI_B0 receive buffer register	UCB0RXBUF	Read	06Eh	Reset with PUC
USCI_B0 transmit buffer register	UCB0TXBUF	Read/write	06Fh	Reset with PUC

They are explained further in section 17.4 in MSP430 user's guide.

To configure I2C on the MSP430, you have to:

1. Put USCI module in reset

17.3.1 USCI Initialization and Reset

The USCI is reset by a PUC or by setting the UCSWRST bit. After a PUC, the UCSWRST bit is automatically set, keeping the USCI in a reset condition. To select I²C operation the UCMODEx bits must be set to 11. After module initialization, it is ready for transmit or receive operation. Clearing UCSWRST releases the USCI for operation.

Configuring and reconfiguring the USCI module should be done when UCSWRST is set to avoid unpredictable behavior. Setting UCSWRST in I²C mode has the following effects:

2. Select I2C, set synchronous single-master mode
   - The VL6180X is a slave device, and we act as master
   - I2C supports multiple masters, but we are a single master here
   - USCI supports asynchronous mode for UART/USART, but I2C is a synchronous protocol
3. Set the I2C clock rate

- I2C supports different clock rates (e.g. standard and fast mode)
- VL6180X can be configured for different rates
- I will use 100 kHz (Standard mode) here
- Note, if you configure a higher clock rate, you may need to adjust the pull-up resistors

4. Bring USCI out of reset
5. Set slave address
   - With I2C, the master always begins each communication by sending the address of the slave
   - The bus may have multiple slaves, and only the slave with the address should respond
   - The default address of VL6180X is 0x29

Writing this as a function we get:

```
1  void i2c_init()
2  {
3      /* Pinmux P1.6 (SCL) and P1.7 (SDA) to I2C peripheral  */
4      P1SEL |= BIT6 + BIT7;
5      P1SEL2 |= BIT6 + BIT7;
6
7      UCB0CTL1 |= UCSWRST; /* Enable SW reset */
8      UCB0CTL0 = UCMST + UCSYNC + UCMODE_3; /* Single master, synchronous mode, I2C mode
9      UCB0CTL1 |= UCSSEL_2; /* SMCLK */
10     UCB0BR0 = 10; /* SMCLK / 10 = ~100kHz */
11     UCB0BR1 = 0;
12     UCB0CTL1 &= ~UCSWRST; /* Clear SW */
13     UCB0I2CSA = 0x29;
14 }
```

i2c.c hosted with ❤ by GitHub view raw

Send a single byte to the slave

After initializing I2C, a good first step is to send a single byte from the master to confirm that our setup works.

I2C is master-driven, so the master is always the one initiating the communication. It sends a start condition followed by a byte containing the 7-bit slave address and a single R/W bit to indicate the transmit mode (RX/TX). Once a slave acks this byte, the master continues with sending the actual data.

To send a byte from the MSP430, we must

1. Configure TX mode and enable the start condition bit
2. Fill the transmit buffer with the byte we want to send
3. Wait for the start condition to be sent
4. Check if the slave acknowledged the address+start condition
5. If the slave acknowledged it, wait for the byte to be sent
6. Check if the slave acknowledged the byte
7. Send a stop condition to end the communication

NOTE: *You MUST fill the transmit buffer before waiting for the start condition (and address) to be sent because the start condition won't be sent otherwise, which is easy to miss if you don't read the datasheet carefully.*

The code:

```c
bool i2c_write_byte(uint8_t byte)
{
    bool success = false;

    UCB0CTL1 |= UCTXSTT + UCTR; /* Set up master as TX and send start condition */
    /* Note, when master is TX, we must write to TXBUF before waiting for UCTXSTT */
    UCB0TXBUF = byte; /* Fill the transmit buffer with the byte we want to send */

    while (UCB0CTL1 & UCTXSTT); /* Wait for start condition to be sent */
    success = !(UCB0STAT & UCNACKIFG);
    if (success) {
        while (!(IFG2 & UCB0TXIFG)); /* Wait for byte to be sent */
        success = !(UCB0STAT & UCNACKIFG);
    }

    UCB0CTL1 |= UCTXSTP; /* Send the stop condition */
    while (UCB0CTL1 & UCTXSTP); /* Wait for stop condition to be sent */
    success = !(UCB0STAT & UCNACKIFG);
    return success;
}
```

i2c.c hosted with ❤ by GitHub view raw

NOTE: *I only return false to indicate failure, which is naive. For example, if the slave hogs the bus for some reason, we may wait indefinitely for the start condition to be sent. I'm doing this to simplify the code. I discuss error handling more in a later section.*

We can call it from the main function in a loop and set a breakpoint with our debugger to verify that it works. Now is also a good time to pick up your logic analyzer.

For example, writing the value 9 repeatedly:

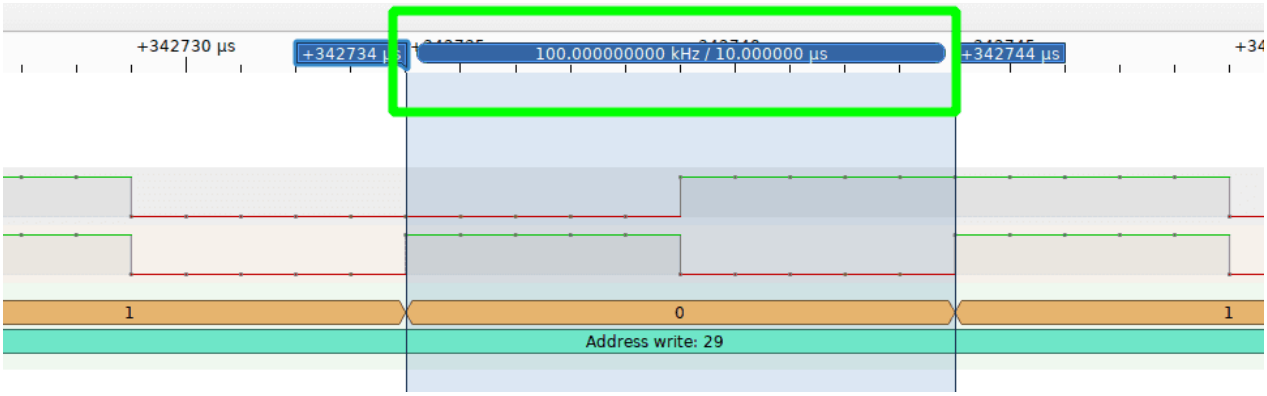
```c
int main(void)
{
    msp430_init();
    i2c_init();
    while (1) {
        i2c_write_byte(9);
    }
    return 0;
}
```

main.c hosted with ❤ by GitHub view raw

This is what you should see **without** the VL6180X connected:

This is what you should see **with** the VL6180X connected:

You can also verify the clock rate:



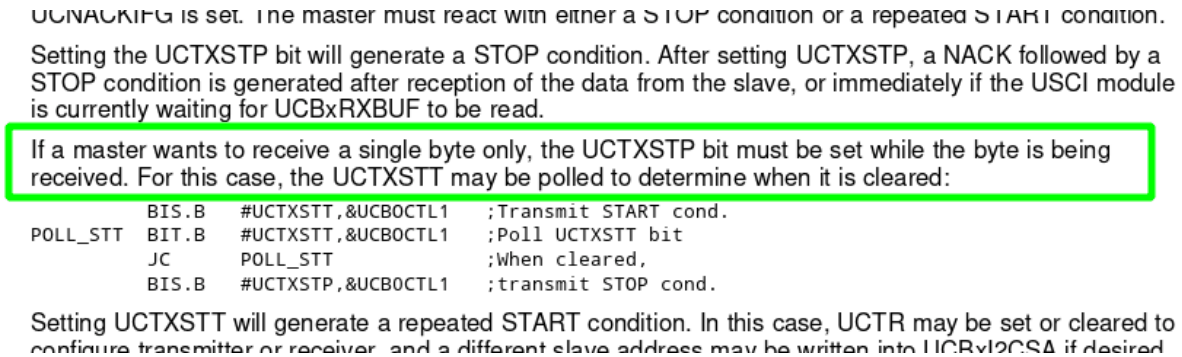
NOTE: *If you have any issues at this point, please take a look at the section Common issues.*

Receive a single byte from the slave

To read a byte from the slave, we follow a similar flow:

1. Configure RX mode and enable the start condition bit
2. Wait for the start condition to be sent
3. Check if the slave acknowledged the address+start condition
4. If the slave acknowledged it, send the stop condition immediately
5. Wait for the byte from the slave
6. Read the byte from the RX buffer

Also, when we want to receive a single byte, we should send the stop condition immediately after the start condition according to the MSP430 user's guide section 17.3.4.2.2:



UCNACKIFG is set. The master must react with either a STOP condition or a repeated START condition.

Setting the UCTXSTP bit will generate a STOP condition. After setting UCTXSTP, a NACK followed by a STOP condition is generated after reception of the data from the slave, or immediately if the USCI module is currently waiting for UCBxRXBUF to be read.

If a master wants to receive a single byte only, the UCTXSTP bit must be set while the byte is being received. For this case, the UCTXSTT may be polled to determine when it is cleared:

```
          BIS.B   #UCTXSTT,&UCB0CTL1   ;Transmit START cond.
POLL_STT  BIT.B   #UCTXSTT,&UCB0CTL1   ;Poll UCTXSTT bit
          JC      POLL_STT             ;When cleared,
          BIS.B   #UCTXSTP,&UCB0CTL1   ;transmit STOP cond.
```

Setting UCTXSTT will generate a repeated START condition. In this case, UCTR may be set or cleared to configure transmitter or receiver, and a different slave address may be written into UCBxI2CSA if desired

NOTE: *In practice, the master never tries to receive a byte before first transmitting something to the slave. It doesn't make sense to read a byte out of the blue because how can the slave know what the master wants if the master doesn't tell it first? Anyway, I do it here only as a demonstration.*

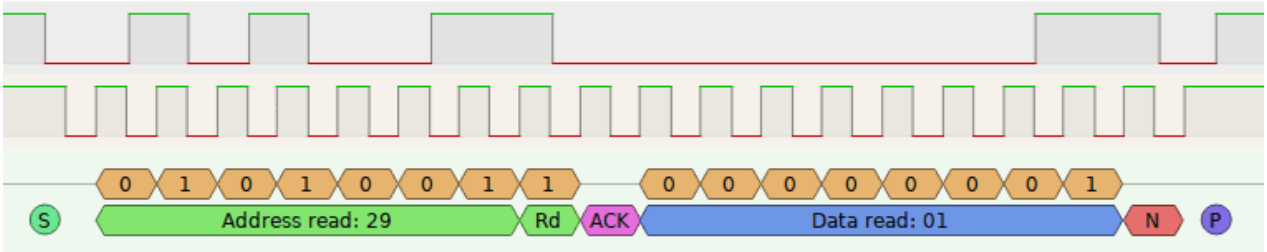
The code:

```
bool i2c_read_byte(uint8_t *received_byte)
{
    bool success = false;

    UCB0CTL1 &= ~UCTR; /* Set up master as RX */
    UCB0CTL1 |= UCTXSTT; /* Send start condition */
    while (UCB0CTL1 & UCTXSTT); /* Wait for start condition to be sent */
    success = !(UCB0STAT & UCNACKIFG);

    UCB0CTL1 |= UCTXSTP; /* Send stop condition because we only want to read one byte
    while (UCB0CTL1 & UCTXSTP);

    success &= !(UCB0STAT & UCNACKIFG);
    if (success) {
        while ((IFG2 & UCB0RXIFG) == 0); /* Wait for byte before reading the buffer */
        *received_byte = UCB0RXBUF; /* RX interrupt is cleared automatically afterward
    }
    return success;
}
```

i2c.c hosted with ❤ by GitHub view raw

I return the received byte via the function argument because I still want to use the return value to indicate success or failure.

If we call the read function similar to how we called the write function, the logic analyzer should show the following:



The slave (VL6180X) ACKs our start condition, but the read byte is NAKed. If you run the read function several times, you will also notice that the value differs each time, which we expect, because as I said the slave doesn't know what we want to read.

Read a register via I2C

The read function is useless on its own, so let's combine it with the write function to create a useful function that reads a register on the VL6180X. To read a register, we basically have to run the write and read function back-to-back:

1. Configure master for transmitting bytes (similar to the write function)
2. Transmit the address of the register we want to read
3. Reconfigure the master for reading bytes (similar to the read function)
4. Read the byte (register value)

In step 2, however, we have to write two bytes because the register addresses of VL6180X are 16-bit wide. Moreover, VL6180X expects us to write the most significant byte (MSB) first. All of this is explained in VL6180X's datasheet.

The code:

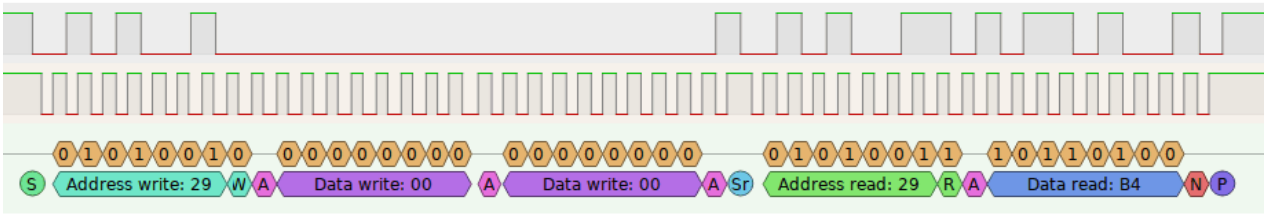
```
bool i2c_read_addr16_data8(uint16_t addr, uint8_t *data)
{
    bool success = false;

    UCB0CTL1 |= UCTXSTT + UCTR; /* Set up master as TX and send start condition */
    /* Note, when master is TX, we must write to TXBUF before waiting for UCTXSTT */
    UCB0TXBUF = (addr >> 8) & 0xFF; /* Send the most significant byte of the 16-bit ad

    while (UCB0CTL1 & UCTXSTT); /* Wait for start condition to be sent */
    success = !(UCB0STAT & UCNACKIFG);
    if (success) {
        while (!(IFG2 & UCB0TXIFG)); /* Wait for byte to be sent */
        success = !(UCB0STAT & UCNACKIFG);
    }
    if (success) {
        UCB0TXBUF = addr & 0xFF; /* Send the least significant byte of the 16-bit addr
        while (!(IFG2 & UCB0TXIFG)); /* Wait for byte to be sent */
        success = !(UCB0STAT & UCNACKIFG);
    }

    /* Address sent, now configure as receiver and get the value */
    if (success) {
        UCB0CTL1 &= ~UCTR; /* Set as a receiver */
        UCB0CTL1 |= UCTXSTT; /* Send (repeating) start condition (including address of
        while (UCB0CTL1 & UCTXSTT); /* Wait for start condition to be sent */
        success = !(UCB0STAT & UCNACKIFG);

        UCB0CTL1 |= UCTXSTP; /* Only receive one byte */
        while ((UCB0CTL1 & UCTXSTP));

        success &= !(UCB0STAT & UCNACKIFG);
        if (success) {
            while ((IFG2 & UCB0RXIFG) == 0); /* Wait for byte before reading the buffe
            *data = UCB0RXBUF; /* RX interrupt is cleared automatically afterwards */
        }
    } else {
        /* We should always end with a stop condition */
        UCB0CTL1 |= UCTXSTP; /* Send the stop condition */
        while (UCB0CTL1 & UCTXSTP); /* Wait for stop condition to be sent */
    }

    return success;
}
```

i2c.c hosted with ❤ by GitHub view raw

Most of the code is identical to the write and read function we previously created. The main difference is that we send two bytes (dividing the 16-bit address with bitwise operations). We also don't send a stop condition after the write, but instead, we send a so called *repeated* start condition.

To verify the read register function, we can read the register with address 0x00. This register contains the device id of VL6180X, which is always 180 (0xB4). Once again, they explain this in the datasheet.

Putting the logic analyzer to work, you should see the following traffic:



As you can see, the master writes the address (0x00+0x00=0x0000), then sends a repeating start condition, and finally reads the byte (0xB4). You may be surprised to see that the last byte is NAKed even though we receive the correct value, but this is entirely according to I2C specification:

transfer, or a repeated START condition to start a new transfer. There are five conditions that lead to the generation of a NACK:

1. No receiver is present on the bus with the transmitted address so there is no device to respond with an acknowledge.
2. The receiver is unable to receive or transmit because it is performing some real-time function and is not ready to start communication with the master.
3. During the transfer, the receiver gets data or commands that it does not understand.
4. During the transfer, the receiver cannot receive any more data bytes.
5. A master-receiver must signal the end of the transfer to the slave transmitter.

The master sends a NAK to signal the slave it has finished reading.

We can also verify we get 0xB4 (180) with the debugger in CCSTUDIO

Expression	Type	Value
(x)= data	unsigned char	180 '\xb4'

Great, now we got something useful.

Write to a register via I2C

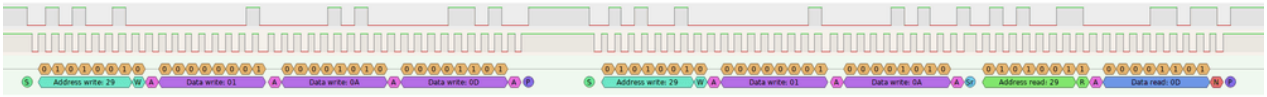
Similarly, we should create a function to write to a register on the VL6180X, which is easier than reading a byte because we can keep the master in transceiver mode.

```
1  bool i2c_write_addr16_data8(uint16_t addr, uint8_t data)
2  {
3      bool success = false;
4
5      UCB0CTL1 |= UCTXSTT + UCTR; /* Set up master as TX and send start condition */
6      /* Note, when master is TX, we must write to TXBUF before waiting for UCTXSTT */
7      UCB0TXBUF = (addr >> 8) & 0xFF; /* Send the most significant byte of the 16-bit ad
8
9      while (UCB0CTL1 & UCTXSTT); /* Wait for start condition to be sent */
10     success = !(UCB0STAT & UCNACKIFG);
11     if (success) {
12         while (!(IFG2 & UCB0TXIFG)); /* Wait for byte to be sent */
13         success = !(UCB0STAT & UCNACKIFG);
14     }
15     if (success) {
16         UCB0TXBUF = addr & 0xFF; /* Send the least significant byte of the 16-bit addr
17         while (!(IFG2 & UCB0TXIFG)); /* Wait for byte to be sent */
18         success = !(UCB0STAT & UCNACKIFG);
19     }
20     if (success) {
21         UCB0TXBUF = data; /* Send the value to write */
22         while (!(IFG2 & UCB0TXIFG)); /* Wait for byte to be sent */
23         success = !(UCB0STAT & UCNACKIFG);
24     }
25     UCB0CTL1 |= UCTXSTP; /* Send stop condition */
26     while (UCB0CTL1 & UCTXSTP); /* Wait for stop condition to be sent */
27     return success;
28 }
```

i2c.c hosted with ❤ by GitHub view raw

NOTE: *I name the I2C functions according to the address and data size. I prefer separate functions over having extra arguments because I think it makes the code cleaner. I add support for more sizes in the next section.*

The best way to confirm that it works is to first write to some register and then read from the same register. According to the datasheet, the register with address 0x10A is an 8-bit register with a valid range of 0-255. Let's write a 13 to this register and see if we get the same value back:



Look at this commit for the complete code until this point.

Support 8/16/32-bit address and data sizes

Being able to read and write a single byte is not enough to fully communicate with the VL6180X. We also need to read and write 16-bit and 32-bit data registers. Besides, it's a good idea to make the I2C layer flexible so we can re-use it for different devices (e.g. the VL53L0X is 8-bit addressed).

This is mostly a matter of refactoring the existing functions, adding extra helper functions, and introducing some nice constructs to make the code cleaner.

The new *drivers/i2c.h*:

```
#ifndef I2C_H
#define I2C_H

#include <stdint.h>
#include <stdbool.h>

void i2c_init(void);
void i2c_set_slave_address(uint8_t addr);

/**
 * Wrapper functions for reading from registers with different address
 * and data sizes.
 * @return True if success, False if error
 * NOTE: Reads the most significant byte first if multi-byte data.
 * NOTE: Polling-based
 */
bool i2c_read_addr8_data8(uint8_t addr, uint8_t *data);
bool i2c_read_addr8_data16(uint8_t addr, uint16_t *data);
bool i2c_read_addr16_data8(uint16_t addr, uint8_t *data);
bool i2c_read_addr16_data16(uint16_t addr, uint16_t *data);
bool i2c_read_addr8_data32(uint16_t addr, uint32_t *data);
bool i2c_read_addr16_data32(uint16_t addr, uint32_t *data);

/**
 * Read byte_count bytes from address addr
 * NOTE: Polling-based
 */
bool i2c_read_addr8_bytes(uint8_t start_addr, uint8_t *bytes, uint16_t byte_count);

/**
 * Wrapper functions for writing to registers with different address
 * and data sizes.
 * @return True if success, False if error
 * NOTE: Writes the most significant byte if multi-byte data.
 * NOTE: Polling-based
 */
bool i2c_write_addr8_data8(uint8_t addr, uint8_t data);
bool i2c_write_addr8_data16(uint8_t addr, uint16_t data);
bool i2c_write_addr16_data8(uint16_t addr, uint8_t data);
bool i2c_write_addr16_data16(uint16_t addr, uint16_t data);

/**
 * Write an array of bytes of size byte_count to address addr.
 * NOTE: Polling-based
 */
bool i2c_write_addr8_bytes(uint8_t start_addr, uint8_t *bytes, uint16_t byte_count);

```

```
48  #endif /* I2C_H */
```

i2c.h hosted with ❤ by GitHub view raw

The new *drivers/i2c.c*:

```
#include "i2c.h"
#include <msp430.h>

#define DEFAULT_SLAVE_ADDRESS (0x29)

typedef enum
{
    ADDR_SIZE_8BIT,
    ADDR_SIZE_16BIT
} addr_size_t;

typedef enum
{
    REG_SIZE_8BIT,
    REG_SIZE_16BIT,
    REG_SIZE_32BIT
} reg_size_t;

static bool start_transfer(addr_size_t addr_size, uint16_t addr)
{
    bool success = false;
    UCB0CTL1 |= UCTXSTT + UCTR; /* Set up master as TX and send start condition */

    /* Note, when master is TX, we must write to TXBUF before waiting for UCTXSTT */
    switch (addr_size) {
    case ADDR_SIZE_8BIT:
        UCB0TXBUF = addr & 0xFF;
        break;
    case ADDR_SIZE_16BIT:
        UCB0TXBUF = (addr >> 8) & 0xFF; /* Send the most significant byte of the 16-bi
        break;
    }

    while (UCB0CTL1 & UCTXSTT); /* Wait for start condition to be sent */
    success = !(UCB0STAT & UCNACKIFG);
    if (success) {
        while (!(IFG2 & UCB0TXIFG)); /* Wait for byte to be sent */
        success = !(UCB0STAT & UCNACKIFG);
    }

    if (success) {
        switch (addr_size) {
        case ADDR_SIZE_8BIT:
            break;
        case ADDR_SIZE_16BIT:
            UCB0TXBUF = addr & 0xFF; /* Send the least significant byte of the 16-bit
            while (!(IFG2 & UCB0TXIFG)); /* Wait for byte to be sent */
```

```
48              success = !(UCB0STAT & UCNACKIFG);
49              break;
50          }
51      }
52      return success;
53  }
54
55  static void stop_transfer()
56  {
57      UCB0CTL1 |= UCTXSTP; /* Send stop condition */
58      while (UCB0CTL1 & UCTXSTP); /* Wait for stop condition to be sent */
59  }
60
61  /* Read a register of size reg_size at address addr.
62   * NOTE: The bytes are read from MSB to LSB. */
63  static bool read_reg(addr_size_t addr_size, uint16_t addr, reg_size_t reg_size, uint8
64  {
65      bool success = false;
66
67      if (!start_transfer(addr_size, addr)) {
68          return false;
69      }
70
71      /* Address sent, now configure as receiver and get the data */
72      UCB0CTL1 &= ~UCTR; /* Set as a receiver */
73      UCB0CTL1 |= UCTXSTT; /* Send (repeating) start condition (including address of sl
74      while (UCB0CTL1 & UCTXSTT); /* Wait for start condition to be sent */
75      success = !(UCB0STAT & UCNACKIFG);
76      if (success) {
77          switch (reg_size) {
78          case REG_SIZE_8BIT:
79              break;
80          case REG_SIZE_16BIT:
81              /* Bytes are read from most to least significant */
82              while ((IFG2 & UCB0RXIFG) == 0); /* Wait for byte before reading the buff
83              data[1] = UCB0RXBUF; /* RX interrupt is cleared automatically afterwards
84              break;
85          case REG_SIZE_32BIT:
86              /* Bytes are read from most to least significant */
87              while ((IFG2 & UCB0RXIFG) == 0);
88              data[3] = UCB0RXBUF;
89              while ((IFG2 & UCB0RXIFG) == 0);
90              data[2] = UCB0RXBUF;
91              while ((IFG2 & UCB0RXIFG) == 0);
92              data[1] = UCB0RXBUF;
93              break;
94          }
95          stop_transfer();
```

```
96          while ((IFG2 & UCB0RXIFG) == 0); /* Wait for byte before reading the buffer */
97          data[0] = UCB0RXBUF; /* RX interrupt is cleared automatically afterwards */
98      }
99
100     return success;
101 }
102
103 static bool read_reg_bytes(addr_size_t addr_size, uint16_t addr, uint8_t *bytes, uint
104 {
105     bool success = false;
106     bool transfer_stopped = false;
107
108     if (!start_transfer(addr_size, addr)) {
109         return false;
110     }
111
112     /* Address sent, now configure as receiver and get the value */
113     UCB0CTL1 &= ~UCTR; /* Set as a receiver */
114     UCB0CTL1 |= UCTXSTT; /* Send (repeating) start condition (including address of sla
115     while (UCB0CTL1 & UCTXSTT); /* Wait for start condition to be sent */
116     success = !(UCB0STAT & UCNACKIFG);
117     if (success) {
118         for (int i = 0; i < byte_count; i++) {
119             if (i + 1 == byte_count) {
120                 stop_transfer();
121                 transfer_stopped = true;
122             }
123             success = !(UCB0STAT & UCNACKIFG);
124             if (success) {
125                 while ((IFG2 & UCB0RXIFG) == 0); /* Wait for byte before reading the
126                 bytes[i] = UCB0RXBUF; /* RX interrupt is cleared automatically afterwa
127             } else {
128                 break;
129             }
130         }
131     }
132     if (!transfer_stopped) {
133         stop_transfer();
134     }
135
136     return success;
137 }
138
139 bool i2c_read_addr8_data8(uint8_t addr, uint8_t *data)
140 {
141     return read_reg(ADDR_SIZE_8BIT, addr, REG_SIZE_8BIT, data);
142 }
```

```
144 bool i2c_read_addr8_data16(uint8_t addr, uint16_t *data)
145 {
146     return read_reg(ADDR_SIZE_8BIT, addr, REG_SIZE_16BIT, (uint8_t *)data);
147 }
148
149 bool i2c_read_addr16_data8(uint16_t addr, uint8_t *data)
150 {
151     return read_reg(ADDR_SIZE_16BIT, addr, REG_SIZE_8BIT, data);
152 }
153
154 bool i2c_read_addr16_data16(uint16_t addr, uint16_t *data)
155 {
156     return read_reg(ADDR_SIZE_16BIT, addr, REG_SIZE_16BIT, (uint8_t *)data);
157 }
158
159 bool i2c_read_addr8_data32(uint16_t addr, uint32_t *data)
160 {
161     return read_reg(ADDR_SIZE_8BIT, addr, REG_SIZE_32BIT, (uint8_t *)data);
162 }
163
164 bool i2c_read_addr16_data32(uint16_t addr, uint32_t *data)
165 {
166     return read_reg(ADDR_SIZE_16BIT, addr, REG_SIZE_32BIT, (uint8_t *)data);
167 }
168
169 bool i2c_read_addr8_bytes(uint8_t start_addr, uint8_t *bytes, uint16_t byte_count)
170 {
171     return read_reg_bytes(ADDR_SIZE_8BIT, start_addr, bytes, byte_count);
172 }
173
174 /* Write data to a register of size reg_size at address addr.
175  * NOTE: Writes the most significant byte (MSB) first. */
176 static bool write_reg(addr_size_t addr_size, uint16_t addr, reg_size_t reg_size, uint
177 {
178     bool success = false;
179
180     if (!start_transfer(addr_size, addr)) {
181         return false;
182     }
183
184     switch (reg_size) {
185     case REG_SIZE_8BIT:
186         success = true;
187         break;
188     case REG_SIZE_16BIT:
189         UCB0TXBUF = (data >> 8) & 0xFF; /* Start with the most significant byte */
190         while (!(IFG2 & UCB0TXIFG)); /* Wait for byte to be sent */
```

```
191             success = !(UCB0STAT & UCNACKIFG);
192             break;
193         case REG_SIZE_32BIT:
194             /* Not supported */
195             return false;
196         }
197
198         if (success) {
199             UCB0TXBUF = 0xFF & data; /* Send the least significant byte */
200             while (!(IFG2 & UCB0TXIFG)); /* Wait for byte to be sent */
201             success = !(UCB0STAT & UCNACKIFG);
202         }
203
204         stop_transfer();
205         return success;
206     }
207
208     static bool write_reg_bytes(addr_size_t addr_size, uint16_t addr, uint8_t *bytes, uin
209     {
210         bool success = false;
211
212         if (!start_transfer(addr_size, addr)) {
213             return false;
214         }
215
216         for (uint16_t i = 0; i < byte_count; i++) {
217             UCB0TXBUF = bytes[i];
218             while (!(IFG2 & UCB0TXIFG)); /* Wait for byte to be sent */
219             success = !(UCB0STAT & UCNACKIFG);
220             if (!success) {
221                 break;
222             }
223         }
224
225         stop_transfer();
226         return success;
227     }
228
229     bool i2c_write_addr8_data8(uint8_t addr, uint8_t value)
230     {
231         return write_reg(ADDR_SIZE_8BIT, addr, REG_SIZE_8BIT, value);
232     }
233
234     bool i2c_write_addr8_data16(uint8_t addr, uint16_t value)
235     {
236         return write_reg(ADDR_SIZE_8BIT, addr, REG_SIZE_16BIT, value);
237     }
238
```

```
239  bool i2c_write_addr16_data8(uint16_t addr, uint8_t value)
240  {
241      return write_reg(ADDR_SIZE_16BIT, addr, REG_SIZE_8BIT, value);
242  }
243
244  bool i2c_write_addr16_data16(uint16_t addr, uint16_t value)
245  {
246      return write_reg(ADDR_SIZE_16BIT, addr, REG_SIZE_16BIT, value);
247  }
248  bool i2c_write_addr8_bytes(uint8_t start_addr, uint8_t *bytes, uint16_t byte_count)
249  {
250      return write_reg_bytes(ADDR_SIZE_8BIT, start_addr, bytes, byte_count);
251  }
252
253  void i2c_set_slave_address(uint8_t addr)
254  {
255      UCB0I2CSA = addr;
256  }
257
258  void i2c_init()
259  {
260      /* Pinmux P1.6 (SCL) and P1.7 (SDA) to I2C peripheral  */
261      P1SEL |= BIT6 + BIT7;
262      P1SEL2 |= BIT6 + BIT7;
263
264      UCB0CTL1 |= UCSWRST; /* Enable SW reset */
265      UCB0CTL0 = UCMST + UCSYNC + UCMODE_3; /* Single master, synchronous mode, I2C mod
266      UCB0CTL1 |= UCSSEL_2; /* SMCLK */
267      UCB0BR0 = 10; /* SMCLK / 10 = ~100kHz */
268      UCB0BR1 = 0;
269      UCB0CTL1 &= ~UCSWRST; /* Clear SW */
270      i2c_set_slave_address(DEFAULT_SLAVE_ADDRESS);
271  }
```

i2c.c hosted with ❤ by GitHub view raw

I won't explain the code line-by-line as it's mostly self-explanatory, but there are some good practices the code exemplifies:

- Make helper functions *static* to limit their scope
- Typedef structs and enums
- Use portable type definitions (i.e. uint8_t, uint16_t, etc.)
- Use void for function declarations without arguments

After refactoring, don't forget to sanity-check that everything works by writing to and reading from a register again.

NOTE: *I didn't implement all the address and data sizes combinations, but the missing ones are easy to implement if you need them.*

NOTE: *I added two functions for reading and writing an array of bytes in preparation for the VL53L0X driver in my other blog post*

Look at this commit for the complete code until this point.

Writing the driver for VL6180X

Now that we have the I2C driver in place for our microcontroller, we are ready to move one layer up to write the driver for the I2C device. While the I2C device driver is tied to the microcontroller, the driver for the I2C device sits one level above and should ideally be independent of it. Doing it like this makes it easy to re-use the I2C driver for another device (which I do in my VL53L0X blog post) and to port the VL6180X driver to another microcontroller.

The VL6180X has a lot of functionality (and registers) to explore (VL53L0X is even worse). I won't go through it all, instead, I will focus on initialization and basic range measuring, which gives you a good starting point to explore the VL6180X further.

ST provides a "portable" API driver for the VL6180X, so technically, you don't need to write your driver. You only need to implement the lower-level I2C functions that are specific to your microcontroller. Though I find their code overly abstracted and challenging to read, and if you pull it in as is, you will end up with a lot of code you don't use.

I prefer to write my own driver (whenever it's practical) to keep the memory footprint small and simplify debugging. Still, it was great to use ST's API driver (and the Arduino libraries from Adafruit and Pololu) as a reference point.

How to initialize VL6180X

Similar to the I2C driver, create two new files under *drivers/*: *drivers/vl6180x.c* and *drivers/vl6180x.h*.

There are several steps to initializing the VL6180X. They are described in the appnote AN4545:

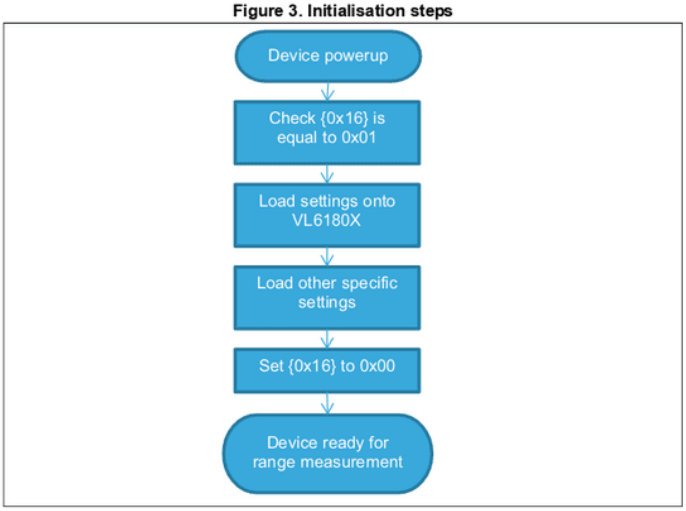
1.3 Initialisation

The latest Standard Ranging (SR) settings must be loaded onto VL6180X after the device has powered up. The following procedure is recommended for loading the settings into the VL6180X.

1. Check device has powered up (Optional)
   a) Check SYSTEM__FRESH_OUT_OF_RESET {0x16} register is equal to 0x01.
2. Load SR settings onto VL6180X
   a) See *Section 9* for the settings.
3. Apply other specific settings e.g. cross talk, GPIO, max convergence time etc. (Optional)
4. Write 0x00 to SYSTEM__FRESH_OUT_OF_RESET {0x16} (Optional)
   a) Help host determine if settings have been loaded.
5. VL6180X is ready to start a range measurement.

Note: *This procedure must be repeated if the VL6180X has been power cycled or if GPIO-0 has been toggled. SYSTEM__FRESH_OUT_OF_RESET {0x16} will reset to 0x01 if the VL6180X has been power cycled or if GPIO-0 was toggled.*

Figure 3. Initialisation steps



Writing out the code, we get:

```
#include "vl6180x.h"
#include "i2c.h"

#define REG_FRESH_OUT_OF_RESET (0x0016)
#define REG_AVERAGING_SAMPLE_PERIOD (0x010A)
#define REG_SYSALS_ANALOGUE_GAIN (0x003F)
#define REG_SYSRANGE_VHV_REPEAT_RATE (0x0031)
#define REG_SYSALS_INTEGRATION_PERIOD (0x0040)
#define REG_SYSRANGE_VHV_RECALIBRATE (0x002E)
#define REG_SYSRANGE_INTERMEASUREMENT_PERIOD (0x001B)
#define REG_SYSALS_INTERMEASUREMENT_PERIOD (0x003E)
#define REG_INTERRUPT_CONFIG_GPIO (0x014)
#define REG_MAX_CONVERGENCE_TIME (0x01C)
#define REG_INTERLEAVED_MODE_ENABLE (0x2A3)
#define REG_RANGE_START (0x018)
#define REG_RESULT_RANGE_STATUS (0x04d)
#define REG_INTERRUPT_STATUS_GPIO (0x04F)
#define REG_RANGE_VAL (0x062)
#define REG_INTERRUPT_CLEAR (0x015)
#define REG_SLAVE_DEVICE_ADDRESS (0x212)

/**
 * Waits for the device to be booted by reading the fresh out of reset
 * register.
 * NOTE: Blocks indefinitely if device is not freshly booted.
 */
static bool wait_device_booted()
{
    uint8_t fresh_out_of_reset = 0;
    bool success = false;
    do {
        success = i2c_read_addr16_data8(REG_FRESH_OUT_OF_RESET, &fresh_out_of_reset);
    } while (fresh_out_of_reset != 1 && !success);
    return success;
}

/**
 * Writes the settings recommended in the AN4545 application note.
 */
static bool write_standard_ranging_settings()
{
    bool success = i2c_write_addr16_data8(0x207, 0x01);
    success &= i2c_write_addr16_data8(0x208, 0x01);
    success &= i2c_write_addr16_data8(0x096, 0x00);
    success &= i2c_write_addr16_data8(0x097, 0xFD);
    success &= i2c_write_addr16_data8(0x0E3, 0x01);
    success &= i2c_write_addr16_data8(0x0E4, 0x03);
```

```
48      success &= i2c_write_addr16_data8(0x0E5, 0x02);
49      success &= i2c_write_addr16_data8(0x0E6, 0x01);
50      success &= i2c_write_addr16_data8(0x0E7, 0x03);
51      success &= i2c_write_addr16_data8(0x0F5, 0x02);
52      success &= i2c_write_addr16_data8(0x0D9, 0x05);
53      success &= i2c_write_addr16_data8(0x0DB, 0xCE);
54      success &= i2c_write_addr16_data8(0x0DC, 0x03);
55      success &= i2c_write_addr16_data8(0x0DD, 0xF8);
56      success &= i2c_write_addr16_data8(0x09F, 0x00);
57      success &= i2c_write_addr16_data8(0x0A3, 0x3C);
58      success &= i2c_write_addr16_data8(0x0B7, 0x00);
59      success &= i2c_write_addr16_data8(0x0BB, 0x3C);
60      success &= i2c_write_addr16_data8(0x0B2, 0x09);
61      success &= i2c_write_addr16_data8(0x0CA, 0x09);
62      success &= i2c_write_addr16_data8(0x198, 0x01);
63      success &= i2c_write_addr16_data8(0x1B0, 0x17);
64      success &= i2c_write_addr16_data8(0x1AD, 0x00);
65      success &= i2c_write_addr16_data8(0x0FF, 0x05);
66      success &= i2c_write_addr16_data8(0x100, 0x05);
67      success &= i2c_write_addr16_data8(0x199, 0x05);
68      success &= i2c_write_addr16_data8(0x1A6, 0x1B);
69      success &= i2c_write_addr16_data8(0x1AC, 0x3E);
70      success &= i2c_write_addr16_data8(0x1A7, 0x1F);
71      success &= i2c_write_addr16_data8(0x030, 0x00);
72      return success;
73  }
74
75  static bool configure_default()
76  {
77      bool success = false;
78      /* Default configuration recommended by application note AN4545 */
79      success = i2c_write_addr16_data8(REG_AVERAGING_SAMPLE_PERIOD, 0x30);
80      success &= i2c_write_addr16_data8(REG_SYSALS_ANALOGUE_GAIN, 0x46);
81      success &= i2c_write_addr16_data8(REG_SYSRANGE_VHV_REPEAT_RATE, 0xFF);
82      success &= i2c_write_addr16_data16(REG_SYSALS_INTEGRATION_PERIOD, 0x63);
83      success &= i2c_write_addr16_data8(REG_SYSRANGE_VHV_RECALIBRATE, 0x01);
84      success &= i2c_write_addr16_data8(REG_SYSRANGE_INTERMEASUREMENT_PERIOD, 0x09);
85      success &= i2c_write_addr16_data8(REG_SYSALS_INTERMEASUREMENT_PERIOD, 0x31);
86      success &= i2c_write_addr16_data8(REG_INTERRUPT_CONFIG_GPIO, 0x24);
87      if (!success) {
88          return false;
89      }
90
91      /* Abort measurement after 50 ms */
92      if (!i2c_write_addr16_data8(REG_MAX_CONVERGENCE_TIME, 0x31)) {
93          return false;
94      }
95
```

```
96      /* Disable interleaved mode */
97      success = i2c_write_addr16_data8(REG_INTERLEAVED_MODE_ENABLE, 0);
98      return success;
99  }
100
101 bool vl6180x_read_range_single(uint8_t *range)
102 {
103     bool success = false;
104
105     /* Wait device ready */
106     uint8_t result_range_status = 0;
107     do {
108         success = i2c_read_addr16_data8(REG_RESULT_RANGE_STATUS, &result_range_status
109     } while (success && !(result_range_status & 0x01));
110     if (!success) {
111         return false;
112     }
113
114     /* Start range measurement */
115     if (!i2c_write_addr16_data8(REG_RANGE_START, 0x01)) {
116         return false;
117     }
118
119     /* Wait for interrupt (polling) */
120     uint8_t interrupt_status = 0;
121     do {
122         success = i2c_read_addr16_data8(REG_INTERRUPT_STATUS_GPIO, &interrupt_status)
123     } while (success && (interrupt_status == 0));
124     if (!success) {
125         return false;
126     }
127
128     /* Read range result */
129     if (!i2c_read_addr16_data8(REG_RANGE_VAL, range)) {
130         return false;
131     }
132
133     /* Clear interrupt */
134     success = i2c_write_addr16_data8(REG_INTERRUPT_CLEAR, 0x07);
135     return success;
136 }
137
138 bool vl6180x_init()
139 {
140     bool success = false;
141
142     if (!wait_device_booted()) {
143         return false;
```

```
144     }
145
146     if (!write_standard_ranging_settings()) {
147         return false;
148     }
149
150     if (!configure_default()) {
151         return false;
152     }
153
154     /* No longer fresh out of reset */
155     success = i2c_write_addr16_data8(REG_FRESH_OUT_OF_RESET, 0);
156
157     return success;
158 }
```

vl6180x.c hosted with ❤ by GitHub view raw

I've added some defines at the top for the register addresses to make them easier to track. I also follow the good practice of enclosing the values with braces. You can find the complete register map in the datasheet.

The first thing we should do is ensure the device is freshly booted, or well, this is optional, but it's a good sanity check. Besides, we have to wait at least 1 ms for the device to be ready anyway. The datasheet also says that we should wait for 400 us after GPOI0 (XSHUT) pin goes high. On the breakout boards, the XSHUT pin is pulled high by default, so it's pulled immediately after power-up, and unless your microcontroller is super fast, you will already have spent 400 us before getting to this point.

Functional description **VL6180X**

From customer application point of view, the following sequence must be followed at the power-up stage

- Set GPIO0 to 0
- Set GPIO0 to 1
- Wait for a minimum of 400µs
- Call **VL6180x_WaitDeviceBooted()**[(b)] API function (or wait for 1ms to ensure device is ready).

We can wait for the VL6180X to boot by loop-reading the "fresh out of reset" register. **NOTE:** *As it's written, the wait_device_booted() will hang if VL6180X is not freshly booted. Consider adding a timeout.*

Once the device is booted, we should set some recommended standard ranging (SR) settings according to AN4545. To make the code less verbose, I *AND* (&) the values and only check the total return value in the end. See *write_standard_ranging_settings()*.

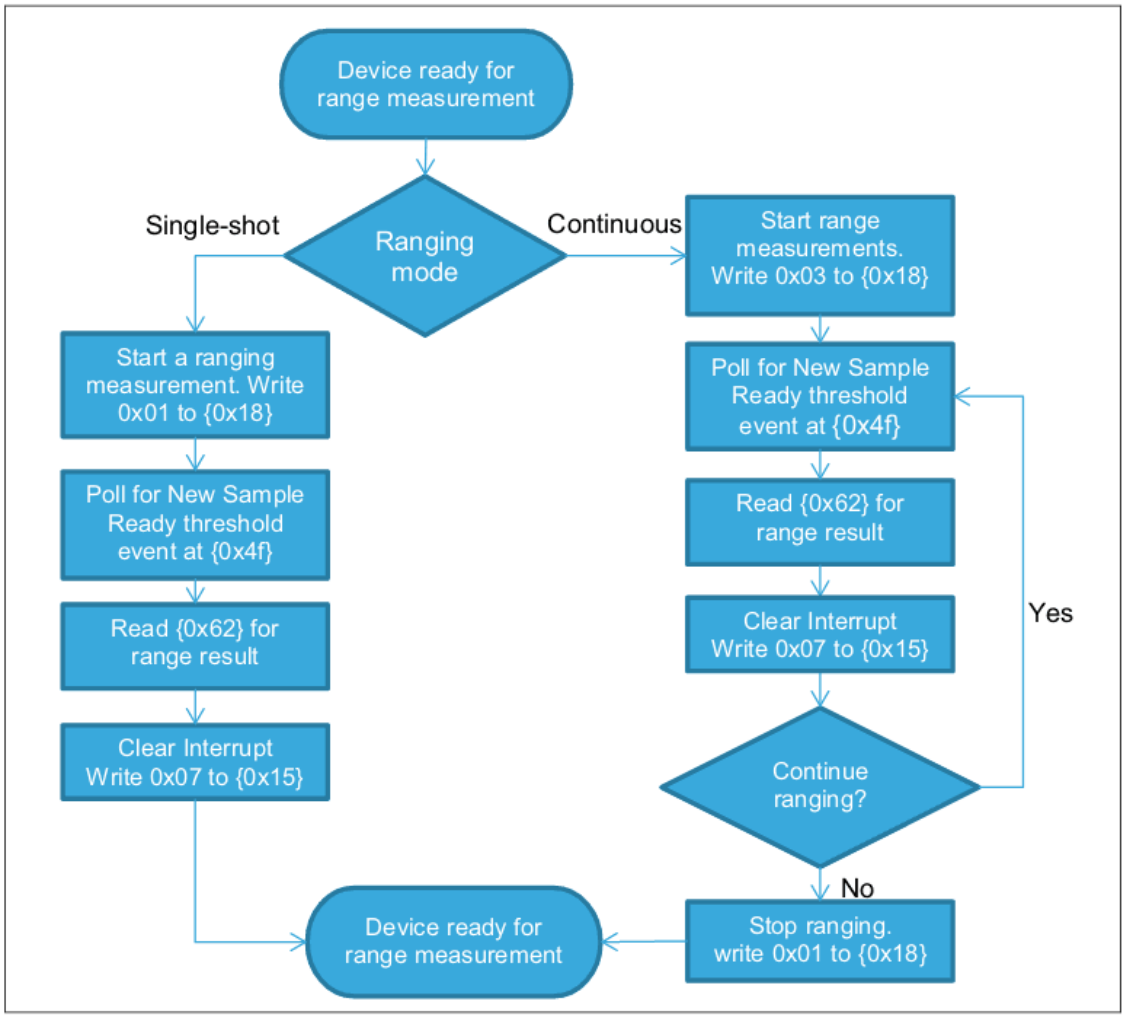
There are also some other recommended registers we should set (see *configure_default()*).

Once we are done with the configuration, we are no longer fresh out of reset, so we clear the "fresh out of reset" register.

How to measure distance with VL6180X

At this point, the sensor is initialized and ready for range measurement. The application note also explains the range measuring flow:

Figure 4. Flowchart for performing range measurements



There are two modes, *single* and *continuous*. In *single-shot mode*, we have to start the measurement each time, while in *continuos mode*, we can set up the sensor to measure at a given interval. I will only implement *single-shot mode*.

```c
bool vl6180x_read_range_single(uint8_t *range)
{
    bool success = false;

    /* Wait device ready */
    uint8_t result_range_status = 0;
    do {
        success = i2c_read_addr16_data8(REG_RESULT_RANGE_STATUS, &result_range_status)
    } while (success && !(result_range_status & 0x01));
    if (!success) {
        return false;
    }

    /* Start range measurement */
    if (!i2c_write_addr16_data8(REG_RANGE_START, 0x01)) {
        return false;
    }

    /* Wait for interrupt (polling) */
    uint8_t interrupt_status = 0;
    do {
        success = i2c_read_addr16_data8(REG_INTERRUPT_STATUS_GPIO, &interrupt_status);
    } while (success && (interrupt_status == 0));
    if (!success) {
        return false;
    }

    /* Read range result */
    if (!i2c_read_addr16_data8(REG_RANGE_VAL, range)) {
        return false;
    }

    /* Clear interrupt */
    success = i2c_write_addr16_data8(REG_INTERRUPT_CLEAR, 0x07);
    return success;
}
```

vl6180x.c hosted with ❤ by GitHub view raw

```c
#ifndef VL6180X_H
#define VL6180X_H

#include <stdbool.h>
#include <stdint.h>

#define VL6180X_OUT_OF_RANGE (255)

```

```
9   bool vl6180x_init(void);
10
11  /**
12   * Does a single range measurement
13   * @param range contains the measured range or VL6180X_OUT_OF_RANGE
14   *        if out of range.
15   * @return True if success, False if error
16   * NOTE: Polling-based
17   */
18  bool vl6180x_read_range_single(uint8_t *range);
19
20  #endif /* VL6180X_H */
```

vl6180x.h hosted with ❤ by GitHub view raw

I continue with polling (instead of interrupt-driven) for simplicity. I talk about interrupts in another section.

We can put a breakpoint with our debugger (or use our logic analyzer) to inspect the received value to confirm that we get a good measurement.

```
1   int main(void)
2   {
3       msp430_init();
4       i2c_init();
5       vl6180x_init();
6       uint8_t range = 0;
7
8       while (1) {
9           vl6180x_read_range_single(&range);
10      }
11
12      return 0;
13  }
```

main.c hosted with ❤ by GitHub view raw

For example, holding my hand close to the sensor:

Holding my hand further away:

No hand:

VL6180X reports 0xFF (255) when no obstacle is detected.

Look at this commit for the complete code until this point.

Multiple VL6180X on the same I2C bus

We got a single VL6180X working, but what if we want to use multiple ones on the same bus?

A big advantage with I2C is that additional slaves don't require extra pins (in contrast to other protocols like SPI). For this to work, each slave must have a unique address, but as we saw, the VL6180X always has a default address of 0x29, which means there will be a bus collision if we hook up several of them.

Fortunately, the address of VL6180X is configurable, but unfortunately, configuring it requires I2C communication. We can get around this catch-22 situation by using the GPIO0 (XSHUT) pin. The VL6180X won't respond to I2C when we pull the XSHUT pin low, and in turn, we can put all our VL6180X in standby and then bring them up and configure them one by one. This approach is explained in this appnote.

For example, with three sensors:

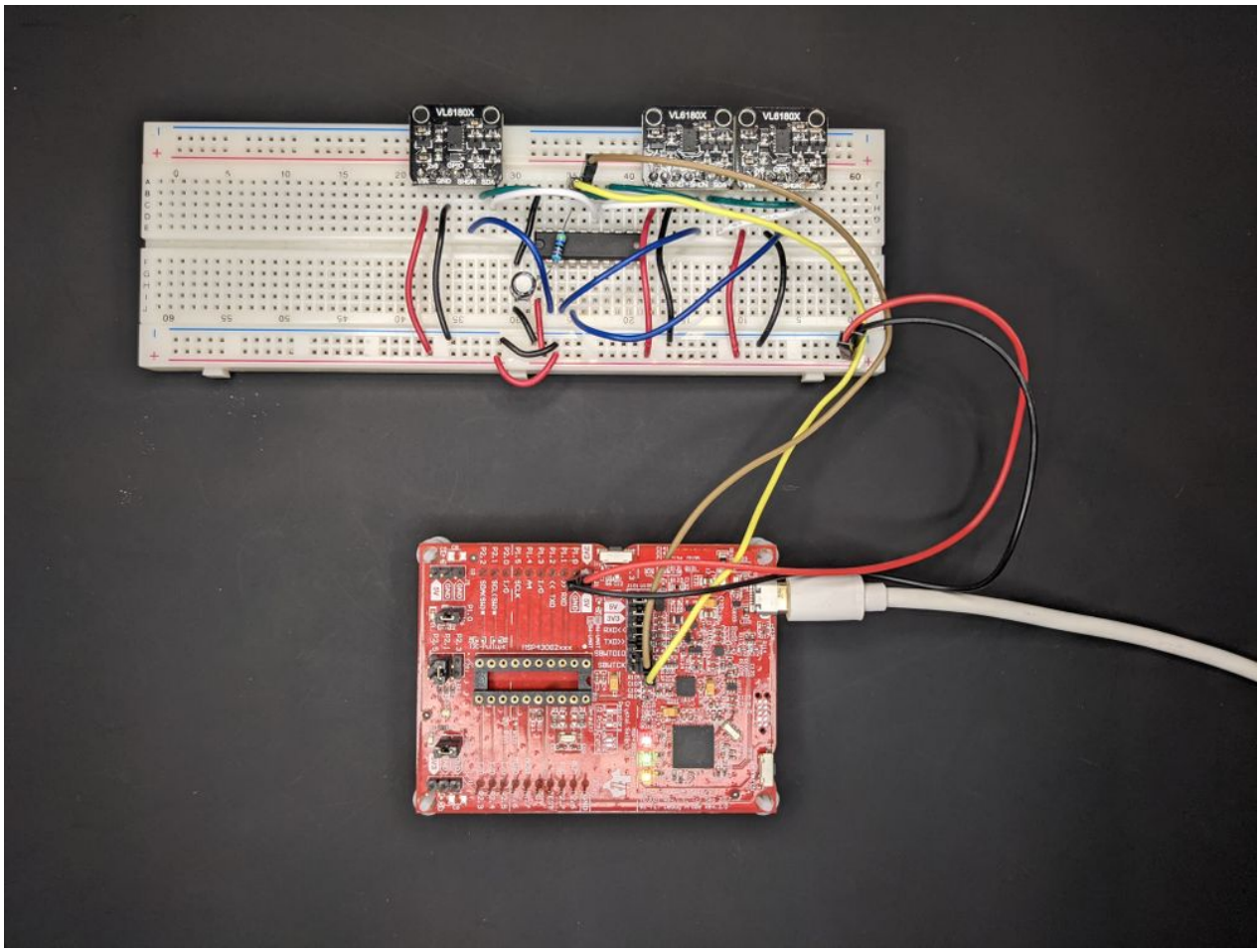
1. Put S1, S2, S3 in standby
2. Wake S1 and give it a unique address
3. Wake S2 and give it a unique address
4. Wake S3 and give it a unique address

The downside, of course, is that we then lose the "few pins"-advantage of I2C because we have to connect the XSHUT pin for each VL6180X.

How to connect multiple VL6180X

Connect the additional sensors to the same lines as with the first sensor (VCC, GND, SDA, and SCL), and then connect the XSHUT pins to some GPIO pins of your microcontroller.

I will use three sensors for demonstration and connect their XSHUT to pin P1.0, P1.1, and P1.2 on my MSP430.



NOTE: *When you connect multiple breakout boards that have pull-up resistors, the total pull-up resistance will decrease. In my case, the breakout boards have 10 kOhm resistors, which means the effective resistance becomes 10/3 ~= 3.3 kOhm. This still works for me, but it's a problem to be aware of.*

Extending the driver to support multiple VL6180X

Let's extend the existing functions and add a bunch of helper functions to support multiple VL6180X.

The new *drivers/vl6180x.h*:

```c
#ifndef VL6180X_H
#define VL6180X_H

#include <stdbool.h>
#include <stdint.h>

/* Comment these out if not connected */
#define VL6180X_SECOND
#define VL6180X_THIRD

#define VL6180X_OUT_OF_RANGE (255)

typedef enum
{
    VL6180X_IDX_FIRST,
#ifdef VL6180X_SECOND
    VL6180X_IDX_SECOND,
#endif
#ifdef VL6180X_THIRD
    VL6180X_IDX_THIRD,
#endif
} vl6180x_idx_t;

/**
 * Initializes the sensors in the vl6180x_idx_t enum.
 * NOTE: Each sensor must have its XSHUT pin connected.
 */
bool vl6180x_init(void);

/**
 * Does a single range measurement
 * @param idx selects specific sensor
 * @param range contains the measured range or VL6180X_OUT_OF_RANGE
 *        if out of range.
 * @return True if success, False if error
 * NOTE: Polling-based
 */
bool vl6180x_read_range_single(vl6180x_idx_t idx, uint8_t *range);

#endif /* VL6180X_H */
```

vl6180x.h hosted with ❤ by GitHub view raw

The new *drivers/vl6180x.c*:

```
#include "vl6180x.h"
#include "i2c.h"
#include "gpio.h"

#define REG_FRESH_OUT_OF_RESET (0x0016)
#define REG_AVERAGING_SAMPLE_PERIOD (0x010A)
#define REG_SYSALS_ANALOGUE_GAIN (0x003F)
#define REG_SYSRANGE_VHV_REPEAT_RATE (0x0031)
#define REG_SYSALS_INTEGRATION_PERIOD (0x0040)
#define REG_SYSRANGE_VHV_RECALIBRATE (0x002E)
#define REG_SYSRANGE_INTERMEASUREMENT_PERIOD (0x001B)
#define REG_SYSALS_INTERMEASUREMENT_PERIOD (0x003E)
#define REG_INTERRUPT_CONFIG_GPIO (0x014)
#define REG_MAX_CONVERGENCE_TIME (0x01C)
#define REG_INTERLEAVED_MODE_ENABLE (0x2A3)
#define REG_RANGE_START (0x018)
#define REG_RESULT_RANGE_STATUS (0x04d)
#define REG_INTERRUPT_STATUS_GPIO (0x04F)
#define REG_RANGE_VAL (0x062)
#define REG_INTERRUPT_CLEAR (0x015)
#define REG_SLAVE_DEVICE_ADDRESS (0x212)

#define VL6180X_DEFAULT_ADDRESS (0x29)

typedef struct vl6180x_info
{
    uint8_t addr;
    gpio_t xshut_gpio;
} vl6180x_info_t;

static const vl6180x_info_t vl6180x_infos[] =
{
    [VL6180X_IDX_FIRST] = { .addr = 0x30, .xshut_gpio = GPIO_XSHUT_FIRST },
#ifdef VL6180X_SECOND
    [VL6180X_IDX_SECOND] = { .addr = 0x31, .xshut_gpio = GPIO_XSHUT_SECOND },
#endif
#ifdef VL6180X_THIRD
    [VL6180X_IDX_THIRD] = { .addr = 0x32, .xshut_gpio = GPIO_XSHUT_THIRD },
#endif
};

/**
 * Waits for the device to be booted by reading the fresh out of reset
 * register.
 * NOTE: Blocks indefinitely if device is not freshly booted.
 * NOTE: Slave address must already be configured.
 */
```

```
48  static bool wait_device_booted()
49  {
50      uint8_t fresh_out_of_reset = 0;
51      bool success = false;
52      do {
53          success = i2c_read_addr16_data8(REG_FRESH_OUT_OF_RESET, &fresh_out_of_reset);
54      } while (fresh_out_of_reset != 1 && !success);
55      return success;
56  }
57
58  /**
59   * Writes the settings recommended in the AN4545 application note.
60   * NOTE: Slave address must already be configured.
61   */
62  static bool write_standard_ranging_settings()
63  {
64      bool success = i2c_write_addr16_data8(0x207, 0x01);
65      success &= i2c_write_addr16_data8(0x208, 0x01);
66      success &= i2c_write_addr16_data8(0x096, 0x00);
67      success &= i2c_write_addr16_data8(0x097, 0xFD);
68      success &= i2c_write_addr16_data8(0x0E3, 0x01);
69      success &= i2c_write_addr16_data8(0x0E4, 0x03);
70      success &= i2c_write_addr16_data8(0x0E5, 0x02);
71      success &= i2c_write_addr16_data8(0x0E6, 0x01);
72      success &= i2c_write_addr16_data8(0x0E7, 0x03);
73      success &= i2c_write_addr16_data8(0x0F5, 0x02);
74      success &= i2c_write_addr16_data8(0x0D9, 0x05);
75      success &= i2c_write_addr16_data8(0x0DB, 0xCE);
76      success &= i2c_write_addr16_data8(0x0DC, 0x03);
77      success &= i2c_write_addr16_data8(0x0DD, 0xF8);
78      success &= i2c_write_addr16_data8(0x09F, 0x00);
79      success &= i2c_write_addr16_data8(0x0A3, 0x3C);
80      success &= i2c_write_addr16_data8(0x0B7, 0x00);
81      success &= i2c_write_addr16_data8(0x0BB, 0x3C);
82      success &= i2c_write_addr16_data8(0x0B2, 0x09);
83      success &= i2c_write_addr16_data8(0x0CA, 0x09);
84      success &= i2c_write_addr16_data8(0x198, 0x01);
85      success &= i2c_write_addr16_data8(0x1B0, 0x17);
86      success &= i2c_write_addr16_data8(0x1AD, 0x00);
87      success &= i2c_write_addr16_data8(0x0FF, 0x05);
88      success &= i2c_write_addr16_data8(0x100, 0x05);
89      success &= i2c_write_addr16_data8(0x199, 0x05);
90      success &= i2c_write_addr16_data8(0x1A6, 0x1B);
91      success &= i2c_write_addr16_data8(0x1AC, 0x3E);
92      success &= i2c_write_addr16_data8(0x1A7, 0x1F);
93      success &= i2c_write_addr16_data8(0x030, 0x00);
94      return success;
95  }
```

```
96
97  /* NOTE: Slave address must already be configured. */
98  static bool configure_default()
99  {
100     bool success = false;
101     /* Default configuration recommended by application note AN4545 */
102     success = i2c_write_addr16_data8(REG_AVERAGING_SAMPLE_PERIOD, 0x30);
103     success &= i2c_write_addr16_data8(REG_SYSALS_ANALOGUE_GAIN, 0x46);
104     success &= i2c_write_addr16_data8(REG_SYSRANGE_VHV_REPEAT_RATE, 0xFF);
105     success &= i2c_write_addr16_data16(REG_SYSALS_INTEGRATION_PERIOD, 0x63);
106     success &= i2c_write_addr16_data8(REG_SYSRANGE_VHV_RECALIBRATE, 0x01);
107     success &= i2c_write_addr16_data8(REG_SYSRANGE_INTERMEASUREMENT_PERIOD, 0x09);
108     success &= i2c_write_addr16_data8(REG_SYSALS_INTERMEASUREMENT_PERIOD, 0x31);
109     success &= i2c_write_addr16_data8(REG_INTERRUPT_CONFIG_GPIO, 0x24);
110     if (!success) {
111         return false;
112     }
113
114     /* Abort measurement after 50 ms */
115     if (!i2c_write_addr16_data8(REG_MAX_CONVERGENCE_TIME, 0x31)) {
116         return false;
117     }
118
119     /* Disable interleaved mode */
120     success = i2c_write_addr16_data8(REG_INTERLEAVED_MODE_ENABLE, 0);
121     return success;
122 }
123
124 static bool configure_address(uint8_t addr)
125 {
126     /* 7-bit address */
127     return i2c_write_addr16_data8(REG_SLAVE_DEVICE_ADDRESS, addr & 0x7F);
128 }
129
130 /**
131  * Sets the sensor in hardware standby by flipping the XSHUT pin.
132  */
133 static void set_hardware_standby(vl6180x_idx_t idx, bool enable)
134 {
135     gpio_set_output(vl6180x_infos[idx].xshut_gpio, !enable);
136 }
137
138 /**
139  * Configures the GPIOs used for the XSHUT pin.
140  * Output low by default means the sensors will be in
141  * hardware standby after this function is called.
142  *
143  * NOTE: The pins are hard coded to P1_0, P1_1, and P1_2
```

```
143     * NOTE: The pins are hard-coded to P1.0, P1.1, and P1.2.
144     **/
145    static void configure_gpio()
146    {
147        gpio_init();
148        gpio_set_output(GPIO_XSHUT_FIRST, false);
149        gpio_set_output(GPIO_XSHUT_SECOND, false);
150        gpio_set_output(GPIO_XSHUT_THIRD, false);
151    }
152
153    /**
154     * Sets the address of a single VL6180X sensor.
155     * This functions assumes that all non-configured VL6180X are still
156     * in hardware standby.
157     **/
158    static bool init_address(vl6180x_idx_t idx)
159    {
160        set_hardware_standby(idx, false);
161        i2c_set_slave_address(VL6180X_DEFAULT_ADDRESS);
162
163        /* 400 us delay according to the vl6180x datasheet */
164        __delay_cycles(400);
165
166        if (!wait_device_booted()) {
167            return false;
168        }
169
170        if (!configure_address(vl6180x_infos[idx].addr)) {
171            return false;
172        }
173        return true;
174    }
175
176    /**
177     * Initializes the sensors by putting them in hw standby and then
178     * waking them up one-by-one as described in AN4478.
179     */
180    static bool init_addresses()
181    {
182        /* Puts all sensors in hardware standby */
183        configure_gpio();
184
185        /* Wake each sensor up one by one and set a unique address for each one */
186        if (!init_address(VL6180X_IDX_FIRST)) {
187            return false;
188        }
189    #ifdef VL6180X_SECOND
190        if (!init_address(VL6180X_IDX_SECOND)) {
```

```
191         return false;
192     }
193 #endif
194 #ifdef VL6180X_THIRD
195     if (!init_address(VL6180X_IDX_THIRD)) {
196         return false;
197     }
198 #endif
199 
200     return true;
201 }
202 
203 static bool init_config(vl6180x_idx_t idx)
204 {
205     i2c_set_slave_address(vl6180x_infos[idx].addr);
206     if (!write_standard_ranging_settings()) {
207         return false;
208     }
209     if (!configure_default()) {
210         return false;
211     }
212     /* No longer fresh out of reset */
213     if (!i2c_write_addr16_data8(REG_FRESH_OUT_OF_RESET, 0)) {
214         return false;
215     }
216     return true;
217 }
218 
219 bool vl6180x_init()
220 {
221     if (!init_addresses()) {
222         return false;
223     }
224     if (!init_config(VL6180X_IDX_FIRST)) {
225         return false;
226     }
227 #ifdef VL6180X_SECOND
228     if (!init_config(VL6180X_IDX_SECOND)) {
229         return false;
230     }
231 #endif
232 #ifdef VL6180X_THIRD
233     if (!init_config(VL6180X_IDX_THIRD)) {
234         return false;
235     }
236 #endif
237     return true;
238 }
```

```
239
240 bool vl6180x_read_range_single(vl6180x_idx_t idx, uint8_t *range)
241 {
242     bool success = false;
243     i2c_set_slave_address(vl6180x_infos[idx].addr);
244     /* Wait device ready */
245     uint8_t result_range_status = 0;
246     do {
247         success = i2c_read_addr16_data8(REG_RESULT_RANGE_STATUS, &result_range_status
248     } while (success && !(result_range_status & 0x01));
249     if (!success) {
250         return false;
251     }
252
253     /* Start range measurement */
254     if (!i2c_write_addr16_data8(REG_RANGE_START, 0x01)) {
255         return false;
256     }
257
258     /* Wait for interrupt (polling) */
259     uint8_t interrupt_status = 0;
260     do {
261         success = i2c_read_addr16_data8(REG_INTERRUPT_STATUS_GPIO, &interrupt_status)
262     } while (success && (interrupt_status == 0));
263     if (!success) {
264         return false;
265     }
266
267     /* Read range result */
268     if (!i2c_read_addr16_data8(REG_RANGE_VAL, range)) {
269         return false;
270     }
271
272     /* Clear interrupt */
273     success = i2c_write_addr16_data8(REG_INTERRUPT_CLEAR, 0x07);
274     return success;
275 }
```

vl6180x.c hosted with ❤ by GitHub view raw

I also added a separate GPIO handling layer to hide the microcontroller specifics from the sensor driver:

drivers/gpio.h:

```c
#ifndef GPIO_H
#define GPIO_H

#include <stdbool.h>

/**
 * Separate file for GPIO handling to avoid including MSP430.h directly
 * in the sensor drivers to make them more portable.
 */

typedef enum
{
    GPIO_XSHUT_FIRST,
    GPIO_XSHUT_SECOND,
    GPIO_XSHUT_THIRD,
} gpio_t;

void gpio_init(void);
void gpio_set_output(gpio_t gpio, bool enable);

#endif /* GPIO_H */
```

gpio.h hosted with ❤ by GitHub view raw

drivers/gpio.c:

```c
#include "gpio.h"
#include <msp430.h>
#include <stdint.h>

void gpio_init(void)
{
    /* P1.0 GPIO and output low */
    P1SEL &= ~BIT0;
    P1DIR |= BIT0;
    P1OUT &= ~BIT0;

    /* P1.1 GPIO and output low */
    P1SEL &= ~BIT1;
    P1DIR |= BIT1;
    P1OUT &= ~BIT1;

    /* P1.2 GPIO and output low */
    P1SEL &= ~BIT2;
    P1DIR |= BIT2;
    P1OUT &= ~BIT2;
}

void gpio_set_output(gpio_t gpio, bool enable)
{
    uint16_t bit = 0;
    switch (gpio)
    {
    case GPIO_XSHUT_FIRST:
        bit = BIT0;
        break;
    case GPIO_XSHUT_SECOND:
        bit = BIT1;
        break;
    case GPIO_XSHUT_THIRD:
        bit = BIT2;
        break;
    }

    if (enable) {
        P1OUT |= bit;
    } else {
        P1OUT &= ~bit;
    }
}
```

gpio.c hosted with ❤ by GitHub view raw

I added an enum to index the sensors, which makes it easy to add more sensors. The enum values are pretty ambiguous, so you may want to rename them to something more useful.

I also like to use an array with *designated initializers* to map the enum values to the data struct of each sensor.

I'm using addresses 0x30, 0x31, and 0x32 for no particular reason. You should be fine as long as the address is unique and 7-bit long.

As I mentioned earlier, the datasheet says that we have to wait for 400 us after pulling the XSHUT pin high. I use the *delay_cycles* function for this, which translates to a busy loop. I know we should avoid busy looping, but it's only for a short moment in this case. If you are using another microcontroller, you need to find some alternative function.

Once the addresses are configured, we do the same initialization steps as before for each sensor.

You can verify that it works by calling *vl6180x_read_range_single* for each sensor and inspect the value with the debugger or logic analyzer.

Look at this commit for the complete code until this point.

Improvements you can make

The code I've given you is simple, and there are many improvements you can make to it. I will mention the most obvious ones in this section.

Interrupts

One improvement to consider is to use interrupts instead of polling. Polling can be pretty wasteful because you let the CPU run while not doing anything useful. With interrupts, the MCU can do other things while waiting or sleep to reduce power consumption.

But remember, polling is not always a bad idea; it depends on your application. It may be unnecessary to add the extra layer of complexity of interrupts. For example, in my sumobot project, I mostly do polling because I can't really do anything useful while waiting for the data from my sensors, and the power consumption from the MSP430 is negligible in the scheme of things.

Even if the end goal is to use interrupts, it's often easier to get polling to work first.

You may also consider doing both. For example, if you have multiple sensors, you could start range measure for all of them but only wait for an interrupt from one of them and poll the rest. This reduces the amount of polling and saves you some interrupt pins.

Errors

I have barely considered error handling in the code examples above. In practice, many things can go wrong with I2C.

As a start, you could return different error codes depending on the error instead of a binary true or false. Together with some tracing to a console, this could make your debugging life easier.

You could also add a timeout at places where you risk getting stuck in an endless loop, for example, where we wait for the start condition to be sent, which can hang if the slave hogs the bus for some reason.

Taking it one step further, you should consider how to handle the errors. You may want to retry, ignore the failing device, or try to recover from the error.

As with most, how far you take depends on your application. If your system is critical and must work all the time, you have to think carefully about it. If not, then your efforts are better spent elsewhere.

I try to avoid adding too much error handling or recovery during development and instead let things go wild when errors occur because it makes it easier to

spot them. Your aim should always be to track down and fix your issues instead of band-aiding them with error handling.

Error handling is a big topic, and there are many resources to find on the internet. I suggest you look at them and study the open-source libraries out there for some practical examples.

DMA

Since direct memory acccess (DMA) is a common technique used when transferring data, it's natural to consider it for I2C. In practice, however, DMA is rarely used for I2C because the protocol is slow and most of its transactions are small (as exemplified here), making DMA cost more than it saves. Moreover, many DMA controllers don't fully support the I2C protocol, forcing you to baby sit the transactions, which in turn defeats the purpose of DMA: off-loading the CPU. Here is a take on I2C by an MSP430 expert.

Explore VL6180X further

There is more you can do with the VL6180X than I've shown here.

You can reconfigure it to better match your example. For example, you measure a longer range if you are okay with lower resolution and accuracy. You can use the continuous mode instead of the single-shot mode. You may also be interested in using it for gesture recognition or measuring ambient light (ALS).

Have a look at ST's documentation for more information.

Common issues

In this section, I list a few issues you may encounter.

"The I2C code get stuck"

This typically means the slave is in a bad state, which can happen when you reflash your microcontroller and interrupt the I2C transaction mid-through. If

you are unlucky, you interrupt during a read transaction, which causes the slave to hog the data line when the master is back up again. The solution is to reflash the MCU, put a breakpoint at the start, and reset the power (pull power pin) of the slave device before continuing. Alternatively, you can cut the power entirely of the MCU and sensor.

"I2C stuck on transmitting the start condition"

Another reason for getting stuck at the start condition (on the MSP430) is if you forgot to write to the TX buffer. When the master is in TX mode, you must write to the TX buffer before waiting on the start condition.

"VL6180X is not responding"

Triple-check that all lines are connected properly, especially if you are breadboarding. Check that you haven't mixed up SDA and SCL, and measure the power lines with a multi-meter. You can rule out that the unit is faulty by using an Arduino. Also, inspect the I2C traffic with a logic analyzer and read the I2C specification, so you know what to expect.

"My microcontroller reboots during range measurement"

The sensor can draw surges of current when operating. Make sure you use a bypass capacitor for the VCC pin of your microcontroller.

General tips

And some general driver development advice.

Read the datasheet

A golden rule in embedded systems development is "read the datasheet". You often overlook several things the first time you skim the datasheet and find the answer the second time around.

Isolate then integrate

Implement your driver in isolation (before you integrate it into your project) to reduce interference from other parts of your code.

Get it working first

It's easier to extend something that works than implementing everything at once. Just look at how we implemented the code in this post. We began by writing and reading a single byte, then writing and reading a register, and so on. Always try to implement in increments, from simple to complex.

Recommended reading

There are many good resources on I2C, VL6180X, and MSP430. Under this section, I will recommend the ones I found helpful.

The documentation for VL6180X is to be found on TI's official site for it. The most useful documents are the VL6180X datasheet and the application note AN4545.

General I2C docs:

- I2C specification
- A website solely dedicated to I2C
- Application report from TI explaining I2C

MSP430 resources:

- The user's guide
- MSP430 Microcontroller Basics by John H. Davies
- A lenghty blog post explaining I2C for MSP430
- Application report on common MSP430 I2C issues
- The MSP430 forum

Useful threads on the MSP430 forum:

- I2C hangs up on while (!(IFG2 & UCB0TXIFG));
- I2C hangs on while(UCB1CTL1 & UCTXSTT);
- What could cause an I2C transaction to fail to start...
- MSP430 hangs while checking uctxstt bit
- I2C master reads one byte more than needed

Useful posts on I2C error handling:

- Appnote on how to recover from I2C bus hang
- Stack overflow post about I2C bus hangs
- A great article on I2C error recovery
- Forum post, how to recover from I2C bus lockup

Code to study:

- The API driver by ST
- Arduino library by Pololu
- Arduino library by Adafruit.
- Open-source I2C library with MSP430 support

Closing words

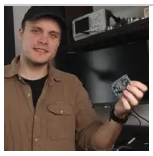
I2C is a widely used protocol and one to be familiar with as an embedded systems engineer. Like any other communication protocol, a good way to learn it is to implement a driver.

We wrote a driver for the VL6180X, a range sensor with an I2C interface. With MSP430 as the microcontroller, we wrote the driver in two layers, first the general I2C layer and then the VL6180X layer. This demonstrates a common pattern in embedded systems development where you layer the drivers to make them more reusable and portable.

VL6180X is an interesting range sensor because of its size and price, and decent documentation. For some applications, the range of < 20 cm will be far

too small, and in that case, you want to look at its big brother VL53L0X, which I cover in my next blog post.

Once again, all of the code is available at GitHub.



Niklas Nilsson

I'm an embedded systems engineer from Sweden currently working at Hasselblad.