```
Error: Can't find stylesheet to import.
  ╷
4 │ @import "gist";
  │         ^^^^^^
  ╵
  app/assets/stylesheets/application.pdf.scss 4:9  root stylesheet
```

Working with I2C Devices

Created by Carter Nelson

https://learn.adafruit.com/working-with-i2c-devices
Last updated on 2024-04-01 10:14:28 AM EDT

Table of Contents

Pull Up Resistors

- Pull Up To What?
- Where Are They?
- Pull Up Resistors on Host Controller
- Pull Up Resistors on Target Device
- Pull Up Resistors on Neither

Address Conflicts

- Set Alternate Addresses
- I2C Multiplexer
- Use Secondary I2C Port

Hot Swapping

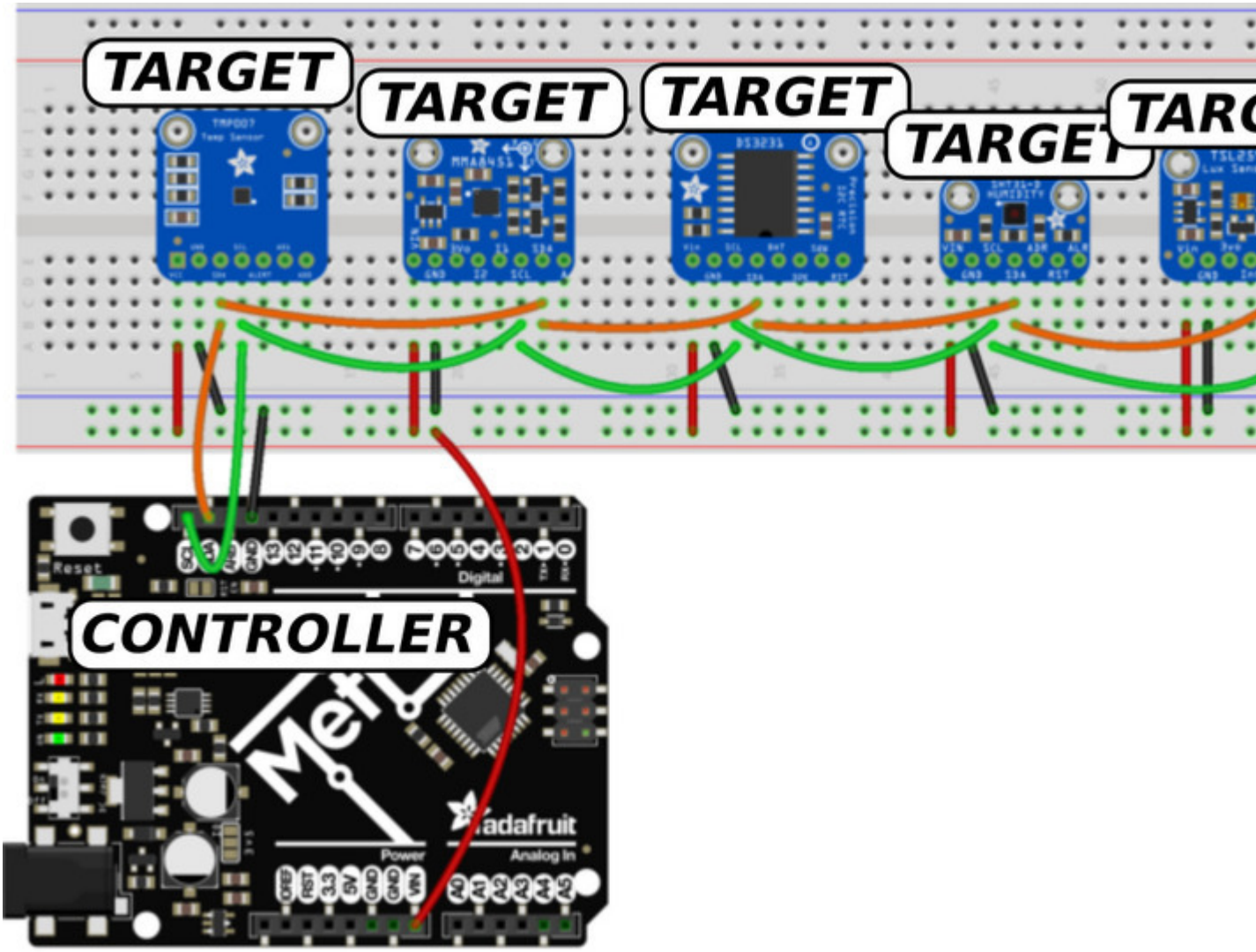
- Supporting Hot Swapping

Cable Length

- Suggested Approach

FAQ

Overview



Inter-Integrate Circuit, aka **I2C**, is an extremely popular method for connecting one or more peripheral devices, like sensor breakouts, to a host controller, like an Arduino board. It only requires two pins for data and clock and can support attaching over 100 devices. It's so popular that it even gets its own special, easy to use connector via Adafruit's STEMMA / STEMMA QT (https://adafru.it/YOE) and Sparkfun's Qwiic (https://adafru.it/Fpw). It's pretty much plug-and-play!

However, it's not all sunshine, rainbows, and lollipops when dealing with I2C. This guide goes into some of the main hiccups that one may encounter when dealing with I2C devices.

The information in this guide is lowish-level.

If you are just wanting to use I2C devices, you don't really need to read this guide. Ideally, you can just use an available library and follow its documentation. The library writer, to the extent possible, has taken care of

the low-level details. But the information here can still be helpful. It will give more context to what the device "address" is all about, etc.

If you are curious about the struggles a library developer deals with, or are wanting to write a library for an I2C device, then this guide hopefully provides some helpful background information on some of the main technical bumps.

Or just give it a read to increase your understanding of I2C, to better understand some of the lingo, etc. There's no project here. Just a grab bag of I2C related topics. Enjoy!

I2C Bus Specification

The original specification goes back to the 1980's and was created by what was then Phillips Semiconductors, which is now known as NXP Semiconductors. The specification is available from NXP's website here:

I2C Bus Specification and User Manual
https://adafru.it/YOF

Terminology

Here is a list of terminology used in I2C use.

BUS

The wires used to connect I2C devices from TARGETs to the CONTROLLER. The bus consists of the SDA and SCL wires and should also have a common ground wire. If distributing power, a power line may be added.

CONTROLLER

This is the host device and has most of the responsibility for controlling the I2C bus. Typically, there is only one controller. Examples are Arduino boards, CircuitPython boards, Raspberry Pi's, etc. **Your user code runs on the controller.**

TARGET

This is the target device being talked to by the controller. There can be more than one target on the I2C bus as long as each has a unique address. Examples are the various I2C **sensor breakouts**.

ADDRESS

All I2C target devices have an address. Typically, this is a 7 bit value. Each target on an I2C bus must have a unique address.

List of I2C Addresses

https://adafru.it/EnK

SDA

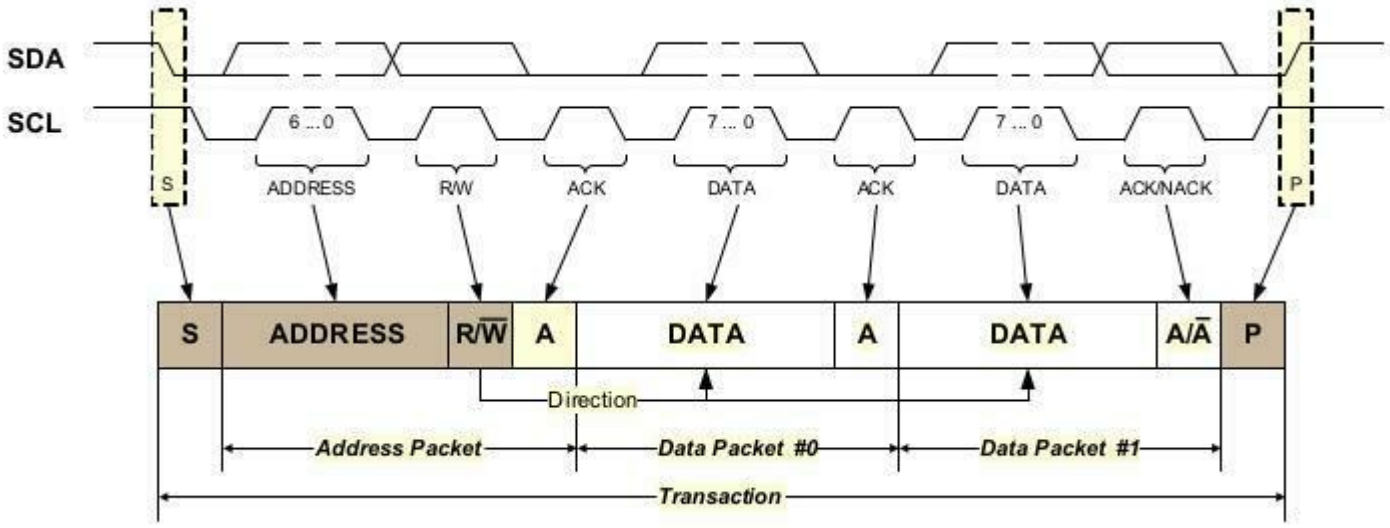
This is the **S**erial **DA**ta pin and/or signal. It carries the **data** being transferred. It is one of the two main I2C signals. The other being SCL.

SCL

This is the **S**erial **CL**ock pin and/or signal. It provides the **clock** signal used for synchronizing reading the data. It is one of the two main I2C signals. The other being SDA.

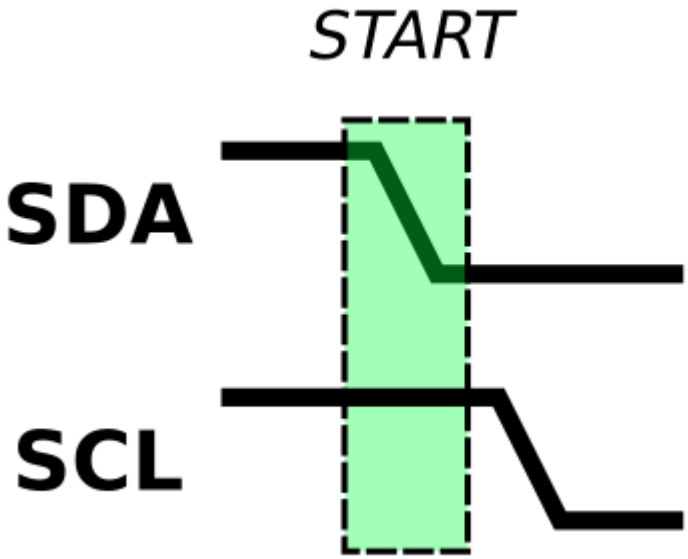
TRANSACTION

An I2C exchange between a controller and a target. It begins with a START and ends with STOP. In between is the actual exchange, the specifics for which can vary.



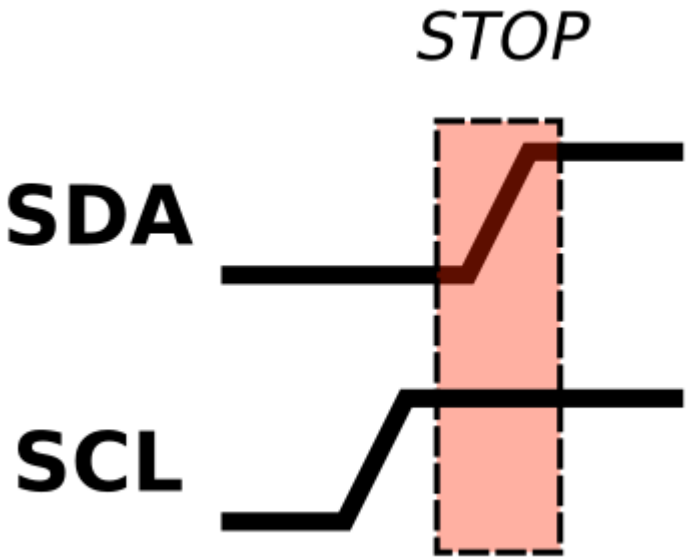
START

As the name implies, this is what starts an I2C transaction. A start condition is generated by transitioning SDA from HIGH to LOW while SCL is HIGH.



STOP

An I2C stop signals the end of an I2C transaction. It is generated by transitioning SDA from LOW to HIGH while SCL is HIGH.



REPEATED START

This is the same as a START. The only difference is that it occurs after an initial start and without a subsequent STOP.

ACK/NACK

Acknowledge (ACK) and Not Acknowledge (NACK). This is a single bit used within the I2C protocol to indicate various conditions. **It is sent on SDA after each byte has been transferred.** Generally:

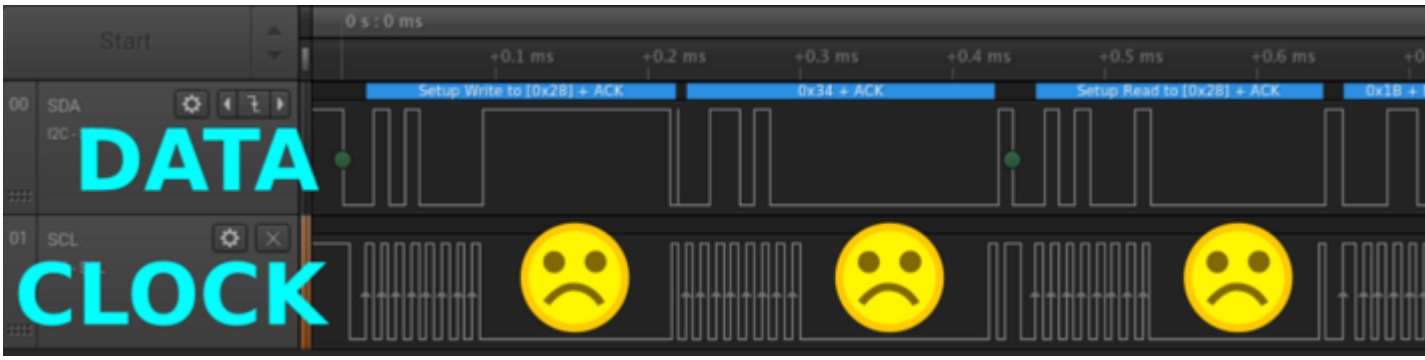
- ACK = SDA LOW
- NACK = SDA HIGH

PULL UP RESISTORS

These are the resistors required by the I2C bus. There are two. One between SDA and VCC. One between SCL and VCC.

This will be explained more as we go along.

Clock Stretching



Clock stretching exists to provide a simple solution to a simple problem. However, the actual results are endless confusion and issues dealing with real world details when trying to implement a reliable I2C interface to actually support clock stretching.

What Problem is Being Solved?

Imagine the general scenario of a controller (Arduino board, CircuitPython board, Raspberry Pi, etc.) talking to some I2C target device, like the BNO055 9-DOF sensor (http://adafru.it/2472).

The controller, over I2C, requests sensor values and expects the response to be the requested values. Pretty simple. But what if the sensor needs some time to obtain those values before they are available to be sent back out over I2C? Remember, the controller is what is **normally** driving the clock signal (SCL). So if the controller keeps blindly driving the clock after the initial request, and the sensor is not ready, what is the target device to do when the controller just keeps clocking?

Enter **clock stretching**.

The basic idea with clock stretching is the **target** (not the controller) takes over the clock signal and holds it **low** as way to indicate "please wait...i'm thinking...". As long as the target holds the clock line low, it is in control of the I2C bus. The host controller must accommodate this.

Why is That a Problem?

As simple as the concept is, numerous issues arise once you start working on implementing clock stretching and figuring out the details. This is mainly true on the host **controller** side.

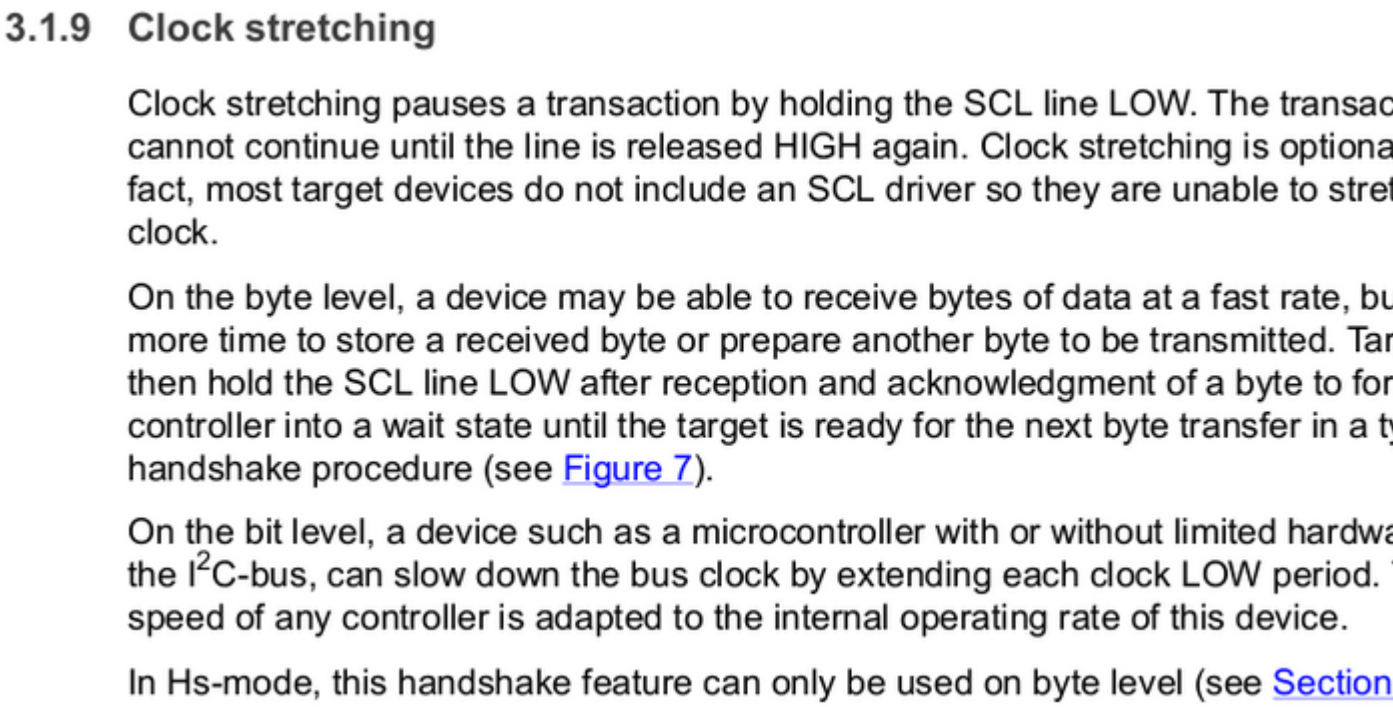
When exactly should the controller give up control of SCL and check for stretching? Each clock cycle? Before ACK/NACKing? After each byte? After a stop?

How long should the controller wait for the stretch to end? 1 millisecond? A couple of days? Five years?

Can't You Just Implement What The Standard Specifies?

The standard (https://adafru.it/YOF) is not specific. It only describes clock stretching in a general way. It doesn't say anything like "and yet shall not exceed 500 us of total clock stretch time".

Here is the entirety of the I2C clock stretch specification:



3.1.9 Clock stretching

Clock stretching pauses a transaction by holding the SCL line LOW. The transac
cannot continue until the line is released HIGH again. Clock stretching is optiona
fact, most target devices do not include an SCL driver so they are unable to stret
clock.

On the byte level, a device may be able to receive bytes of data at a fast rate, bu
more time to store a received byte or prepare another byte to be transmitted. Tar
then hold the SCL line LOW after reception and acknowledgment of a byte to for
controller into a wait state until the target is ready for the next byte transfer in a ty
handshake procedure (see Figure 7).

On the bit level, a device such as a microcontroller with or without limited hardwa
the I²C-bus, can slow down the bus clock by extending each clock LOW period.
speed of any controller is adapted to the internal operating rate of this device.

In Hs-mode, this handshake feature can only be used on byte level (see Section

So target devices can potentially do anything, and controller implementations need to accommodate that. This can be non-trivial.

A very prominent example of this exists on the Raspberry Pi. **Raspberry Pi's are all known to not handle I2C clock stretching well.** Further, this is a **hardware issue** with the I2C implementation in the Broadcom BCM28xx chips used on those models of Raspberry Pi. So it essentially cannot be fixed: it's baked into the silicon. At the time of this guide writing, the issue is almost 10 years old. Here are some (potentially dated) discussions on this issue.

- Raspberry Pi I2C clock-stretching bug (https://adafru.it/YPa) - Written 2013-08-17, this blog post is the most often cited and linked. It contains lots of technical details.
- RPI forums I2C clock stretching (https://adafru.it/YPb) - Discussion circa 2012 about this issue. There are 70 posts to the thread!
- I2C Broadcom bug workaround (https://adafru.it/YPc) - An old issue thread from 2013. There are 73 posts to the thread!
- I2C clock-stretching bug (https://adafru.it/YPd) - A more recent issue thread specific to the Pi 4. The saga continues!

How to Deal With Clock Stretching Issues

If you end up in a situation where a target device (sensor breakout, etc.) is clock stretching and the host controller (Raspberry Pi, etc.) is having issues coping, this can be very annoying. The target device creators will say the host is not handling clock stretching properly. The writers of the host I2C implementation will say the target device is clock stretching in a non-standard way. And you are left with a host/target combo that just won't work. So solutions tend to be somehwhat hack-ish.

Here are some commonly used approaches.

Slow Down The Clock

This is the simplest and easiest approach. The basic idea is to make the I2C clock speed slow enough that any potential clock stretching gets buried within the clock pulses. So, from the point of view of the host controller, it never sees the actual clock stretch. Or at least not all of it.

For example, on a Raspberry Pi, this is done by manually setting the I2C clock speed in **/boot/config.txt**.

I2C Clock Stretching on Raspberry Pi
https://adafru.it/C7x

The initial recommended slow down is a factor of 10! From 100kHz to 10kHz. However, unless you need to read from the target device blazingly fast, this should be fine.

Use Software I2C

Ideally, the majority of dealing with I2C can be left to a dedicated I2C hardware peripheral. For example, the peripheral will watch for an address

match - without any code being executed. This is the generally preferred approach. But, by their nature, I2C hardware peripherals are fairly fixed entities. If they have quirks - those are baked in (see the Raspberry Pi example above).

By contrast, a purely software I2C implementation (using two regular GPIO pins and code that emulates an I2C transaction) can be modified and updated - since it's just code.

On a Raspberry Pi, software I2C can be enabled by using the i2c-gpio device tree overlay (https://adafru.it/YPe).

i2c-gpio device tree overlay
https://adafru.it/YPe

Increase Timeout

This is more low level and only an option if you're actually writing I2C firmware on a controller. If the hardware I2C peripheral on the controller has such an ability, then try increasing it. This is easier for software implementations, since a timeout will likely be part of the base code.

Here's an example related to CircuitPython:

CircuitPython Pull Request to Increase Clock Stretching Timeout
https://adafru.it/YPf

As another example, on the BCM2711 chip used on the Pi 4, the Broadcom Serial Control (BSC) peripheral has a `CLKT` register with a `TOUT` setting. From the datasheet (https://adafru.it/YPA):

CLKT Register

Description

The clock stretch timeout register provides a timeout on how long the master waits for the slave to stret
before deciding that the slave has hung.
The TOUT field specifies the number I2C SCL clocks to wait after releasing SCL high and finding that th
low before deciding that the slave is not responding and moving the I2C machine forward. When a tim
the I2CS.CLKT bit is set.
Writing 0x0 to TOUT will result in the Clock Stretch Timeout being disabled.

Bits	Name	Description	Type	Reset
31:16	Reserved.	-	-	-
15:0	TOUT	Clock Stretch Timeout Value Number of SCL clock cycles to wait after the rising edge of SCL before deciding that the slave is not responding.	RW	0x0040

Source A Different Chip

If all else fails, consider simply using an entirely different chip. Luckily there is not just one I2C accelerometer, temperature, humidity, etc. sensor option. If there is no reason to focus on a specific device (do you really need a BNO055?), and that device is causing grief with its clock stretching, consider using a different chip.

Here's a list of chips that are known to be problematic:

Troublesome I2C Chips
https://adafru.it/YPB

Repeated Start

Here's a quote from NXP's I2C specification (https://adafru.it/YOF):

> All transactions begin with a START and are terminated by a STOP.

While a STOP is required to end a transaction, there is also the possibility to send another START before sending the final STOP. This is called a **repeated start**.

So a transaction won't always be as simple as START-STOP. There can also be START-START-STOP.

The repeated start signal is identical to the original start signal. It is just a special name given to the second one.

Why Do This?

Consider a typical I2C transaction for reading a register value from a target device. The read actually starts with a write - to send the address of the desired register. This is followed by the actual read to get the returned value. So there are actually two I2C transactions involved:

- I2C write - send the register address
- I2C read - receive the returned value

Each of these requires an initial START. But should there be a STOP between the two? Yes, no, maybe. It all depends. **The answer is target device specific and would be found in the target device datasheet.**

What's the Issue?

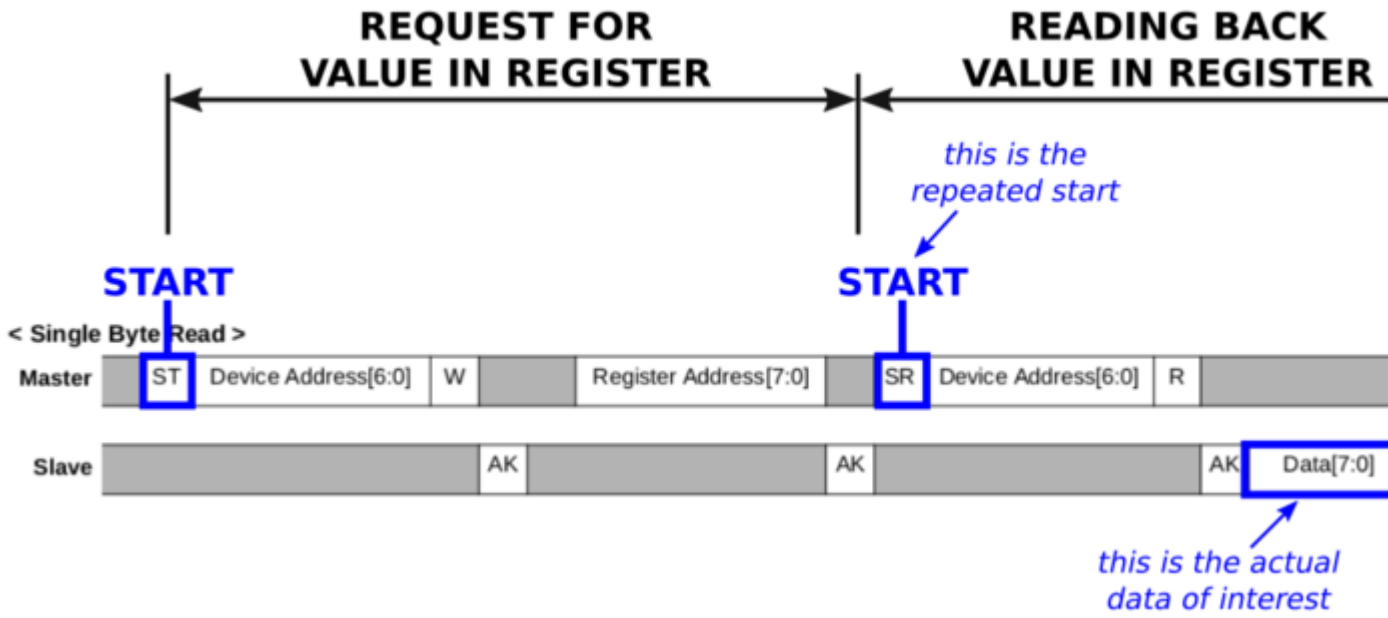
Some devices require a repeated start.

For these devices, a STOP is seen as sort of a "reset". They may throw away the register value from the initial write, so the subsequent read won't work.

Or they may do something like go back to sleep, which would also potentially corrupt the attempt to read.

In these cases, the STOP is suppressed until the very very end - when all the I2C communication is done. In between, STARTs are reused - these are called repeated starts.

Here's an example - the MMA8451 accelerometer (http://adafru.it/2019). Inside the datasheet (https://adafru.it/dLO), there's a section (5.11) that describes how to talk to the device over I2C. A **single byte read** is the most basic read interaction, where a request is made to return the single byte value (Data[7:0]) stored in a particular device register. Here is what the datasheet shows:



Note that there is no STOP after the requested register address is sent via a write (W) operation. Instead, another START is sent. The only difference is the read/write bit is now set to read (R). And then the value (Data[7:0]) is sent by the device and read back.

Then finally a STOP.

Why is That an Issue?

This is another detail that seems simple, but can cause some grief when actually implementing an I2C library. You probably are not doing this directly in your code, but are using an existing one, like the Wire (https://adafru.it/YPC) library provided by Arduino cores.

The basic START-STOP pattern is pretty easy to implement. But dealing with providing the option for suppressing a STOP can be tricky.

Arduino Example

The Arduino Wire library (https://adafru.it/YPC) provides for an optional parameter in endTransmission() (https://adafru.it/YPD) to control the behavior of sending a stop when **writing**. For example, the MMA8451 example from above would be done with:

```
Wire.beginTransmission(device_address);
Wire.write(register_address);
Wire.endTransmission(false); // !! NO STOP !!
Wire.requestFrom(device_address, 1);
register_data = Wire.read();
```

However, the degree to which that is supported can be core dependent. Remember, "Arduino" encompasses many boards with numerous architectures, like AVR, ESP8266, ESP32, ARM, etc. Each of these has their own completely separate Arduino "core" which implements the Arduino API. So there can be variability between cores.

A slightly different example might occur when attempting to transfer large amounts of data. If the amount of data exceeds the buffer being used, it can not be done with a single transfer. Instead, multiple transfers must be done. In that case, it can be useful to suppress the stop between each. For the case of a **read**, this would be done with option parameter in requestFrom() (https://adafru.it/YPE). But actually supporting this can be tricky:

ESP32 Repeated Start Issue
https://adafru.it/YPF

When this happens, it essentially means that platform and its I2C implementation **can not support repeated start**. So if you're trying to talk to an I2C target device that requires repeated start, you're sort of stuck.

Linux Example

The Linux kernel implementation also lacks the ability to suppress a stop for some situations.

Linux Kernel I2C Protocol
https://adafru.it/YQa

It provides the ability to do the following:

- A basic write with a stop
- A basic read with a stop
- A combined write followed by a read with a repeated start

Therefore, one can write with a stop, or, using the combined method, a write with no stop followed by a repeated start and a read. But there is no way to do just a write with no stop.

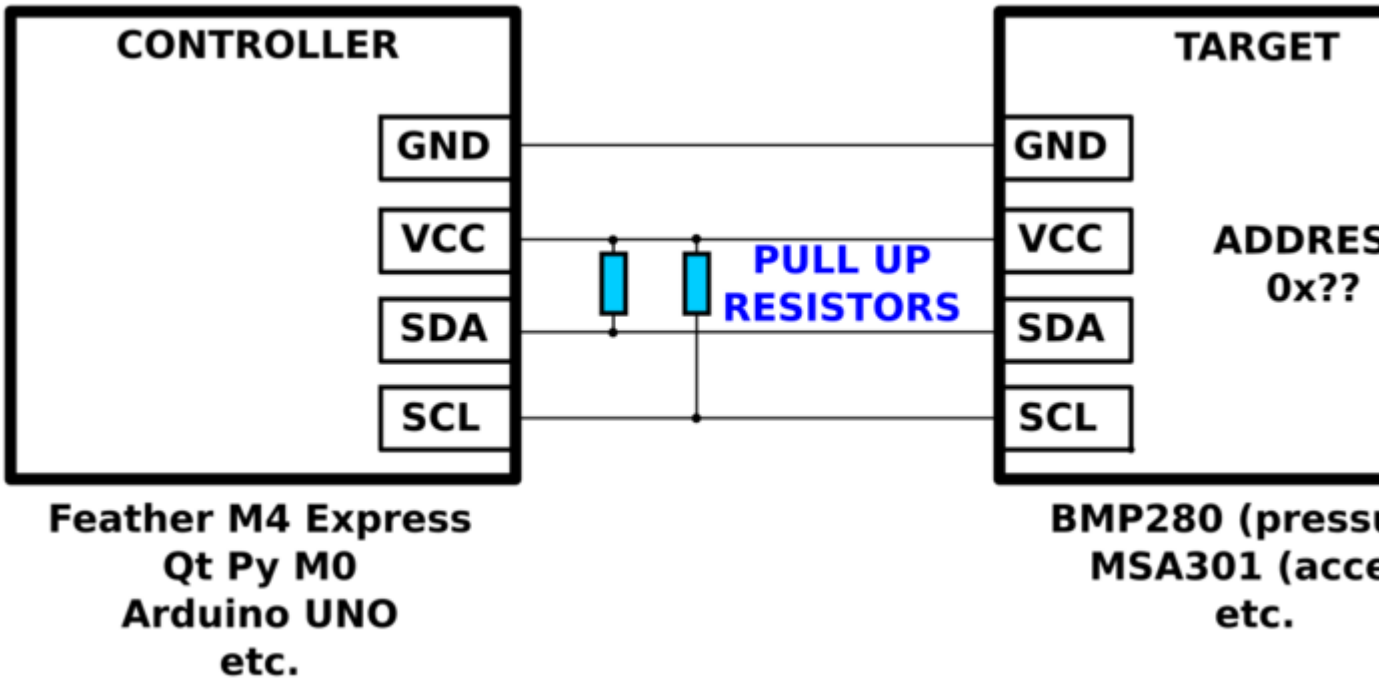
CircuitPython Example

This is related to the Linux example above and is a good example of how sometimes these "issues" come down to specific design decisions made by developers. While the code *could* be changed to potentially better support odd-ball scenarios, it is decided not to for other reasons (i.e., it will just introduce other issues).

It's a closed issue thread, but contains some good discussion on the rational:

[Expose control of stop bit on i2c write](https://adafru.it/YQb)
https://adafru.it/YQb

Pull Up Resistors



As part of the electrical interface of I2C, a pair of pull up resistors are required as shown in the diagram above. There are two separate resistors:

- One between **SDA** and **VCC**
- One between **SCL** and **VCC**

Where VCC is the logic level for the controller and target. Quite often, this is also power. So for example, the same 3.3V line that powers the target device.

Pull Up To What?

As mentioned above, the pull up resistors should be connected to the same voltage as the "logic level" for the device. This shouldn't be confused with the "power" voltage. The common values for both are 5V and 3.3V.

If "power" and "logic level" are the same, either 5V or 3.3V, then things are simple. There's generally only one voltage level in play.

Sometimes a board will have 5V "power" (like from a USB port) but the "logic level" (voltage on GPIO pins) is 3.3V. This can potentially lead to mixed voltage levels since there are now two voltages in play. Different processors have different levels of tolerance for this. For example, it has been noted that the SAMD51 really doesn't like the I2C lines pulled up higher than 3.3V.

Where Are They?

For a self contained setup, like an iPhone, the pull up resistors can be located anywhere convenient inside the iPhone itself. Everything - the host controller, all the target devices, wiring between them, etc. - are all in one blob. So the pull up resistors are just somewhere in that blob.

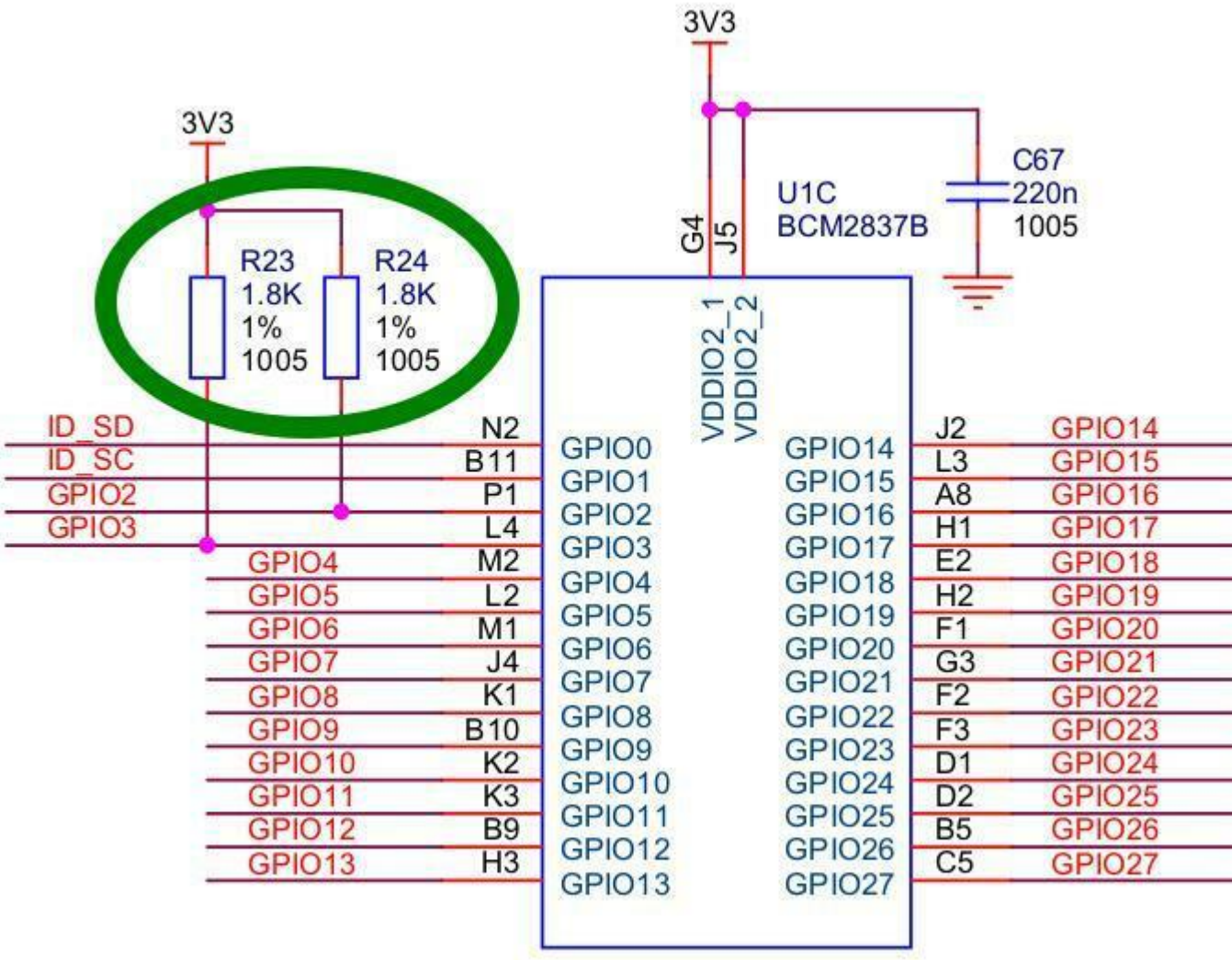
Things become a little trickier when the host controller and the target devices are separated. This is the situation that generally exists for Makers. There are numerous options for host controllers, for example a Raspberry Pi or an Arduino board. There are also lots of I2C based target devices, for example - all the STEMMA sensor breakouts. Where are the pull up resistors to be located in this scenario? Unfortunately, **there is no standard.** So it all depends. But there are generally only two options for where to locate the pull up resistors:

- on the host controller
- on the target device(s)

Pull Up Resistors on Host Controller

The Raspberry Pi is one example where the pull up resistors are located on the host controller. Here's an example showing the I2C pull up resistors on the schematic (https://adafru.it/YJB) for a Raspberry Pi 3 Model B+.

GPIO2 is SDA and GPIO3 is SCL.

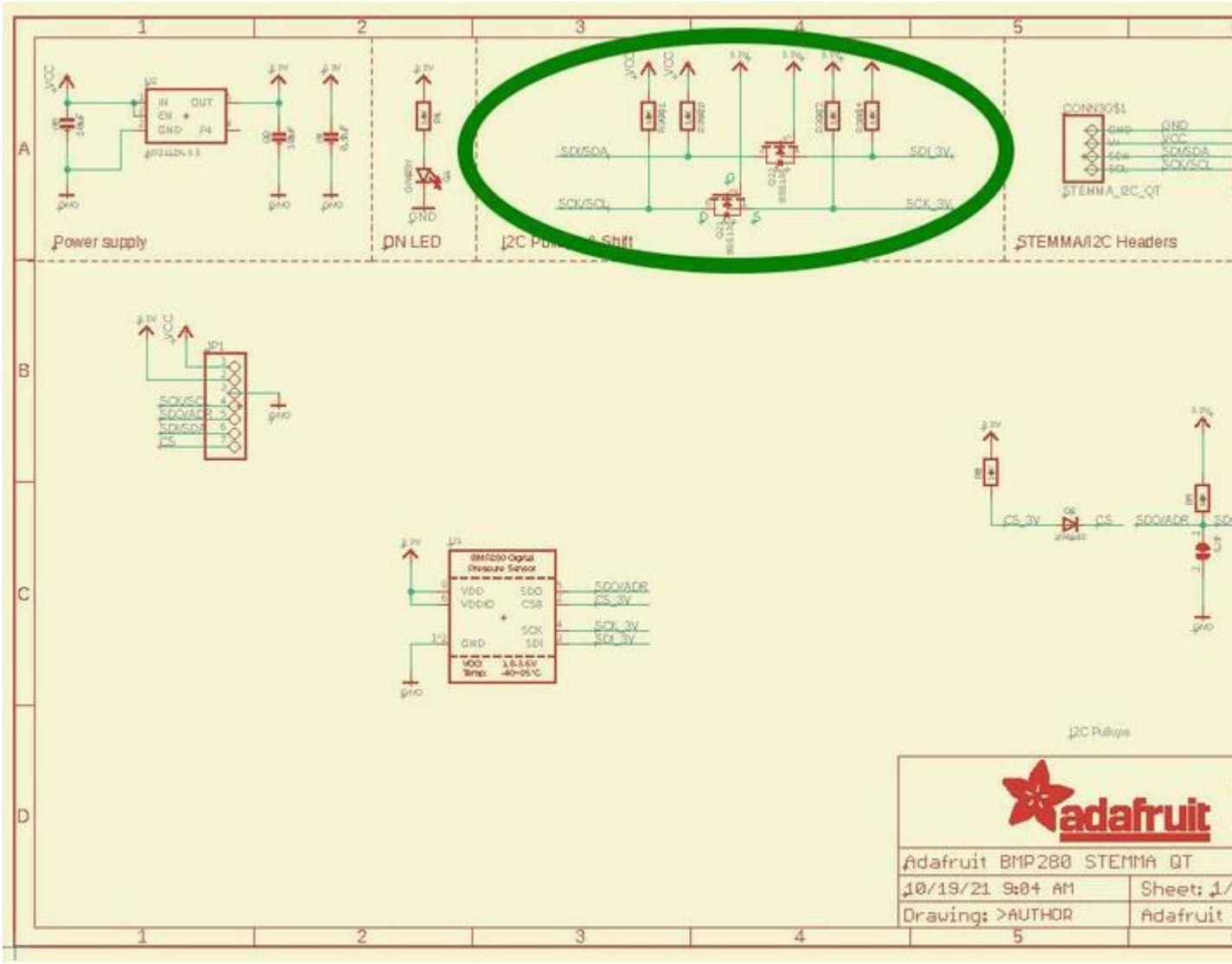


The down side to this is the pull up resistors are always present. So if one wanted to use the GPIO pins for something other than I2C, the pull up resistors would still be there, physically connected to the same pins, and could cause potential interference.

Pull Up Resistors on Target Device

This is the approach taken by pretty much all Adafruit I2C breakouts. Here's an example showing the pull up resistors for the STEMMA version of the BMP280 breakout (http://adafru.it/2651).

STEMMA breakouts also level shift I2C. That's why there are more than two pull up resistors in the schematic below.



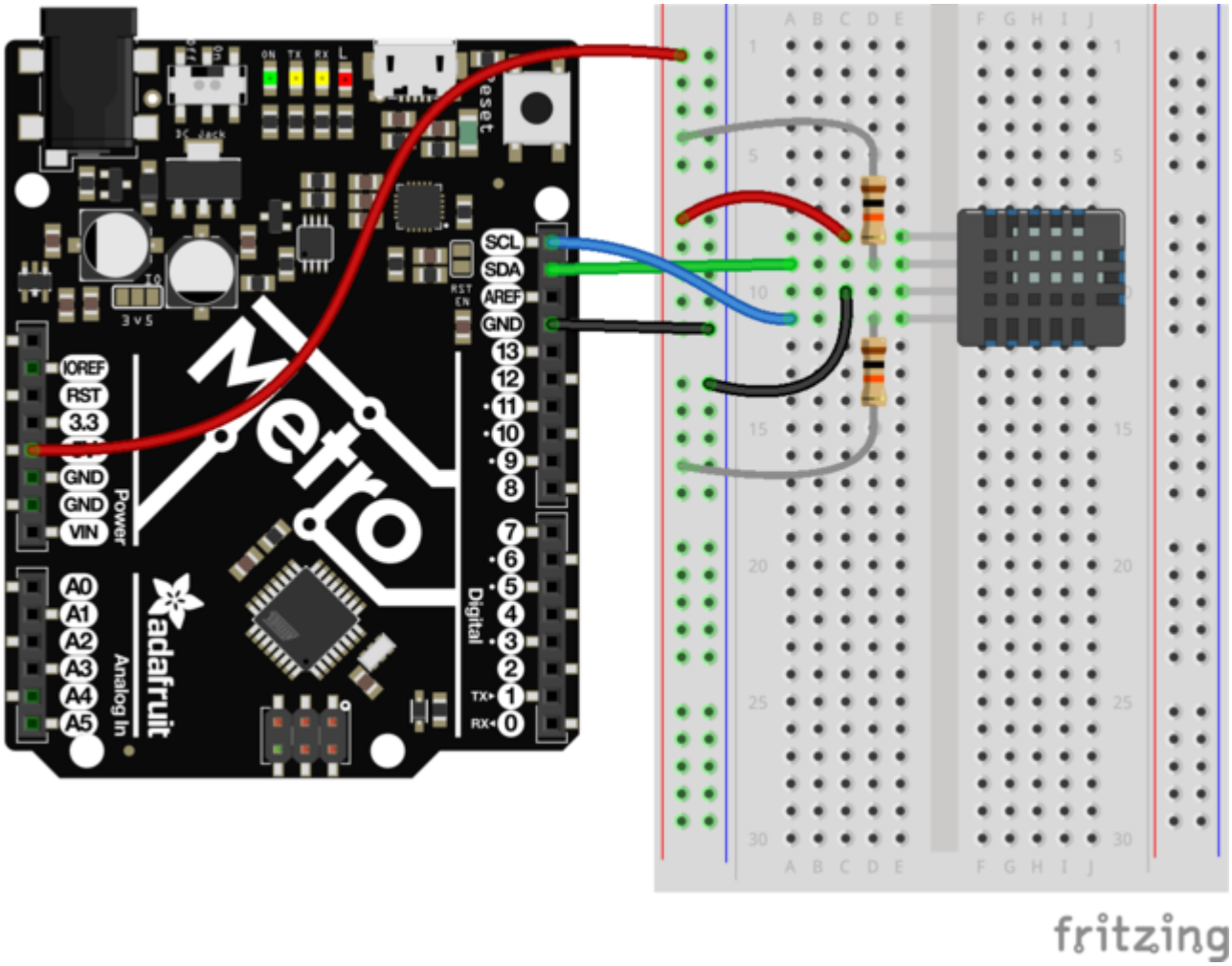
The down side to this approach is the varying effective pull up resistance value when multiple target devices are attached. The multiple sets of pull up resistors act in parallel and thus **reduce** the total pull up resistance as seen by the I2C bus. However, this is generally not an issue, since I2C will function over a fairly wide range of pull up resistance values.

Another minor point is the increased cost. The I2C bus only needs one set of pull up resistors, regardless of how many target devices are connected. So by putting the resistors on every breakout board, there is unnecessary hardware redundancy.

Pull Up Resistors on Neither

It is very possible to end up in a situation where there are no pull up resistors included at all - on either the controller or the target. Maybe a sensor breakout was designed for a controller, like a Raspberry Pi, that has pull ups. Or the exact resistor values were left to the end user to decide upon. So the resistors were left off. Then one tries to use such a breakout with a controller, like a Metro, that also does not have pull ups. **This will not work.**

In this situation, pull up resistors must be added somehow. If things are setup on a breadboard, this can be done fairly easy. For example, the AM2320 (http://adafru.it/3721) does not have pull up resistors inside its case. So they must be added externally as shown here:



Address Conflicts

Each device on an I2C bus needs to have a unique address. There can't be two devices with address 0x42 for example. Here's a list of I2C addresses for various target devices:

List of I2C Addresses
https://adafru.it/EnK

So what happens if more than one device has the same address? It's sort of undefined. It may be that all devices will respond the same, like "hey every 0x42 device, do a software reset". But since devices need to also write to the bus, there can be conflicts.

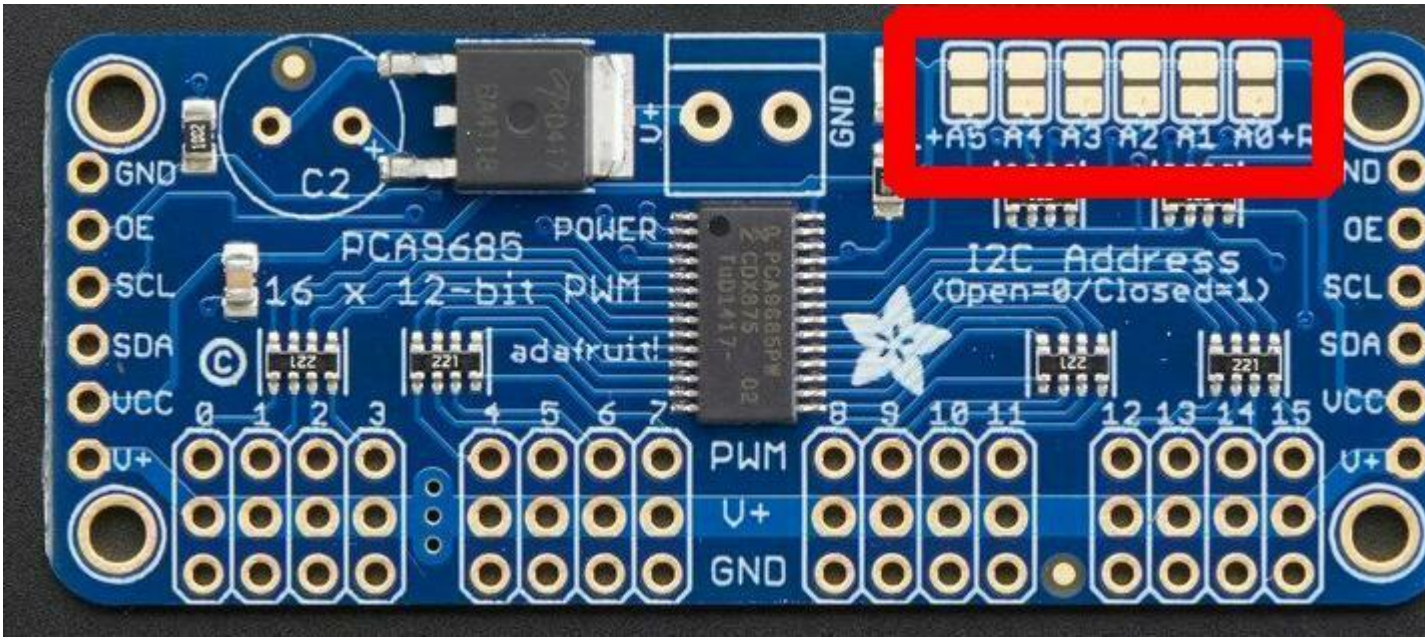
How do you deal with address conflicts? There are various approaches. Let's take a look at them, starting with the easiest option.

Set Alternate Addresses

Some, but not all, target devices allow for setting one or more alternate address. Some only have two options. Some have many more. And of course, some are fixed to just one address. Details vary, so one must consult datasheets for specifics.

Most devices that have this option will provide dedicated pins that configure the alternate I2C addresses. These may be broken out to the header pins, but a more common approach used on Adafruit breakouts is to provide solder pads. For example, here is the solder pad on the BMP280 breakout (http://adafru.it/2651) for setting the alternate address:

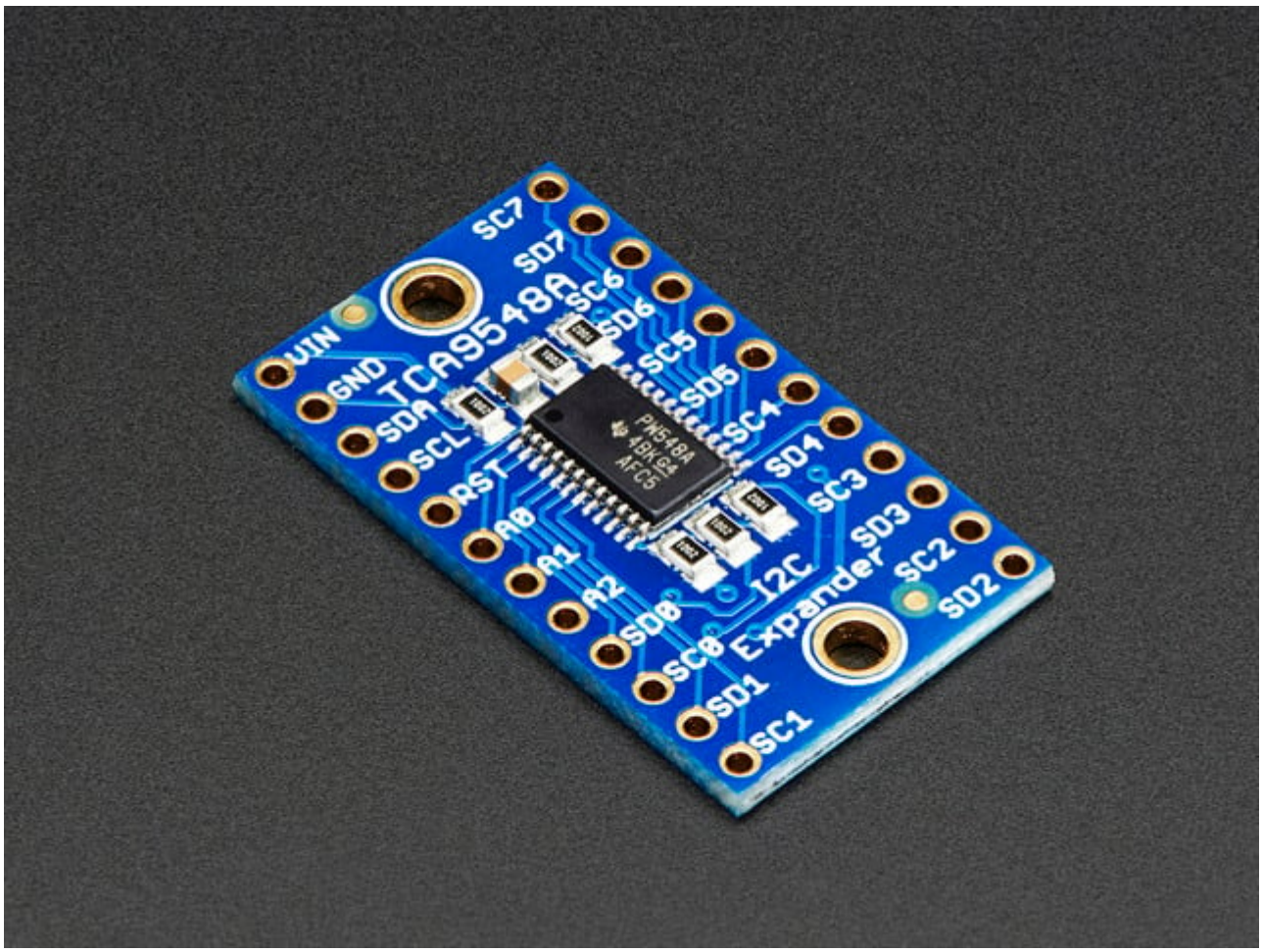
As another example, the PCA9685 PWM driver (http://adafru.it/815) allows setting up to 62 different addresses! This requires many more solder pads:



I2C Multiplexer

This is the simplest solution for dealing with I2C address conflicts if alternate addresses can not be set. An I2C multiplexer acts like a signal switcher. There is **one input** which gets connected to the controller's I2C port. Then, there are **multiple outputs** that get connected to the target devices. **Only one output is active at any given time.** Therefore, target devices with the same address can be place on the different outputs. And the host only sees one at a time. The output can be changed to talk to any of the other attached devices.

The TCA9548A is an example of an I2C multiplexer:



TCA954
Multiple
You just
perfect
sensor,
want to
two or t
more of
your Ar
when yo
"Uh oh,
has a fix
address
from...
https://
www.ad
product

Checkout this guide for example usage of a TCA9548A multiplexer:

Adafruit TCA9548A 1-to-8 I2C Multiplexer Breakout
https://adafru.it/ZlC

Use Secondary I2C Port

This is <u>not</u> a beginner friendly approach.

Some controllers may have more than one I2C bus. The Raspberry Pi Pico (https://adafru.it/YQc), for example, has two. So another idea is to use a

separate I2C port for each target device. While this generally works, it can be problematic. It does not scale well. And depending on how libraries are written, switching between buses may require extra work - i.e. more complex code.

Hot Swapping

In general, I2C was not designed to support hot swapping - i.e. plugging and unplugging devices while the system is powered up and code is running. This is in contrast to USB, for which hot swapping is one of its main design features.

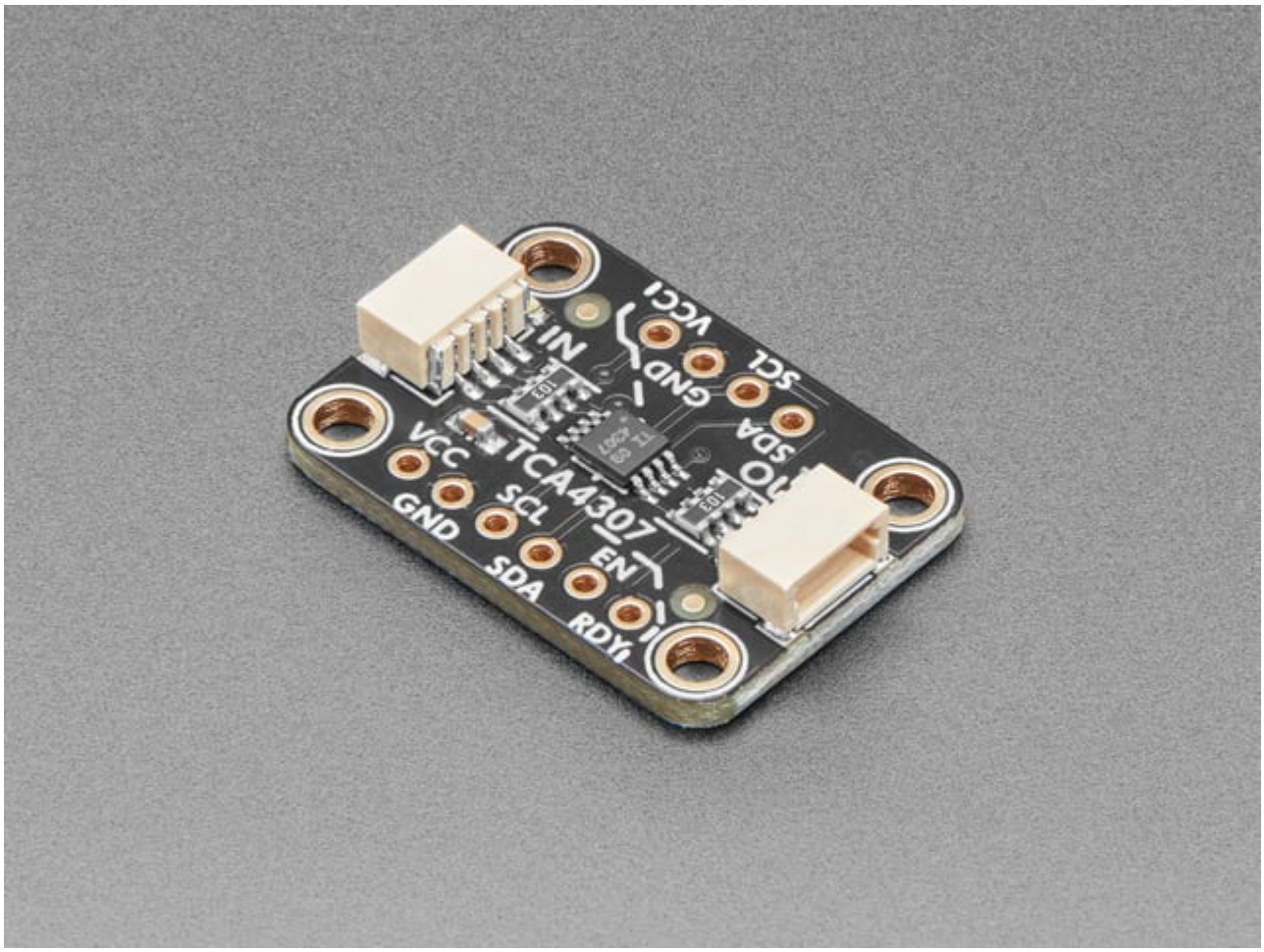
Additionally, the convenience of STEMMA / STEMMA QT (https://adafru.it/Ft4) cables and connectors gives the impression that I2C does allow hot swapping. And, yes, you may even see simple product demos where this is being done. But in general, this is not a supported feature of I2C.

Supporting Hot Swapping

If your application must support hot swapping, there are potential ways to attempt it. The biggest change will likely be with your code, which must accommodate an I2C target device being disconnected / reconnected at any given time in code execution.

Additionally, augmenting the I2C hardware (which was not designed for this) with something like a TCA4307 buffer is suggested.

Adafruit
Hot-Swa
Buffer w
Bus Rec
As we've
adding
QT conn
our brea
dev boa
folks...
https://
www.ad
product



Cable Length

I2C was originally designed for talking between devices separated by fairly short distances. For example, between all the chips inside an iPhone. So a total length of inches (centimeters), **not feet** (meters).

The total distance achievable depends on things like the strength of the pull up resistors, clock speed, external interference, quality of wires, etc. There is no one magic value. Here are some **very rough guide** line values.

THESE ARE NOT STRICT REQUIREMENTS. JUST ROUGH VALUES.

CLOCK SPEED	**TOTAL DISTANCE**
100kHz	12"
400kHz	6"
1MHz	3"

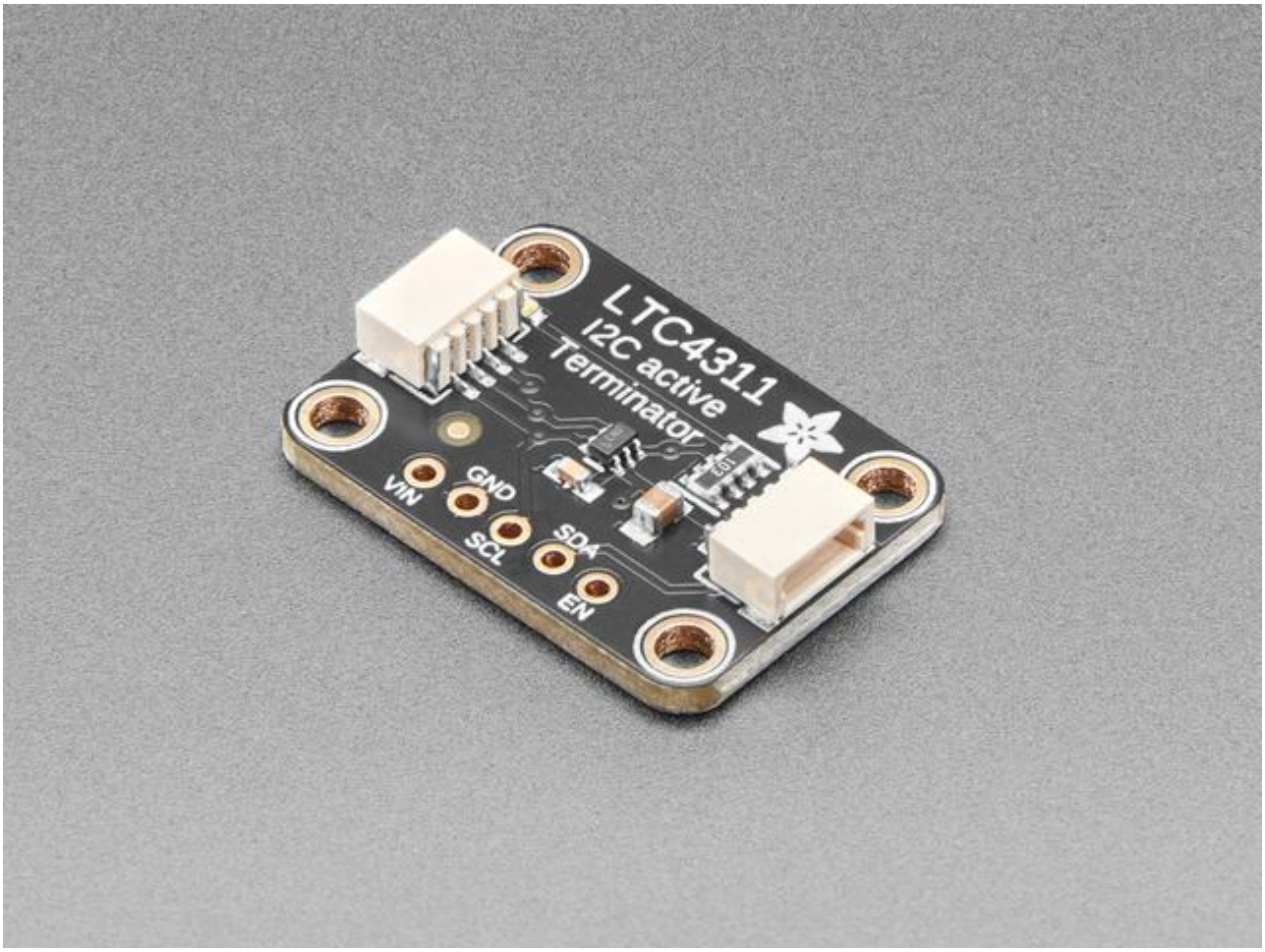
THESE ARE NOT STRICT REQUIREMENTS. JUST ROUGH VALUES.

Suggested Approach

So what can you get away with? In true hacker spirit - just try it and find out! So, yah, I2C was designed for short distances. BUT - you may find that for your application you're happily getting away with 10 feet of cabling. Great!

Not working reliably? Getting weird readings or random drop outs? Try shortening the cable length or reducing the I2C clock speed.

For applications that truly need long cable runs, using an active terminator is an option:



Adafruit
I2C Exte
Active T
I2C star
Inter-In
Circuit
commur
it's mea
short di
a PCB o
subasse
hey, we'
enginee
like to p
limits of
https://
www.ad
product

FAQ

What about multi-controller setups and arbitration?

That is beyond the scope of this guide. It is also not very common.

What about 10 bit addressing?

That is beyond the scope of this guide. It is also not very common.

Why Do Some Targets Have Two Addresses?

Most likely, the 7-bit address plus the 1-bit read/write are being combined and treated as two separate 8-bit addresses. One for reading. One for writing. It's just a different way of thinking of the first byte (8 bits) sent after the start condition.

For other cases, it may be multiple targets buried in one chip. Each target has its own address in the normal fashion.

Didn't the Raspberry Pi 4 fix the clock stretching issue?

Yes. No. Maybe? Checkout this issue thread. (https://adafru.it/YPd)

Is SMBus the same as I2C?

They're the "sameish". I2C came first, circa 1982. SMBus came along circa 1995 and added to it. SMBus stands for System Management Bus. They have some commonality at the protocol level, so there can be some intermixing.

See smbus.org (https://adafru.it/YQd) for more info. Also, this article (https://adafru.it/YQe) explains some key differences.