Communication Protocols etc…

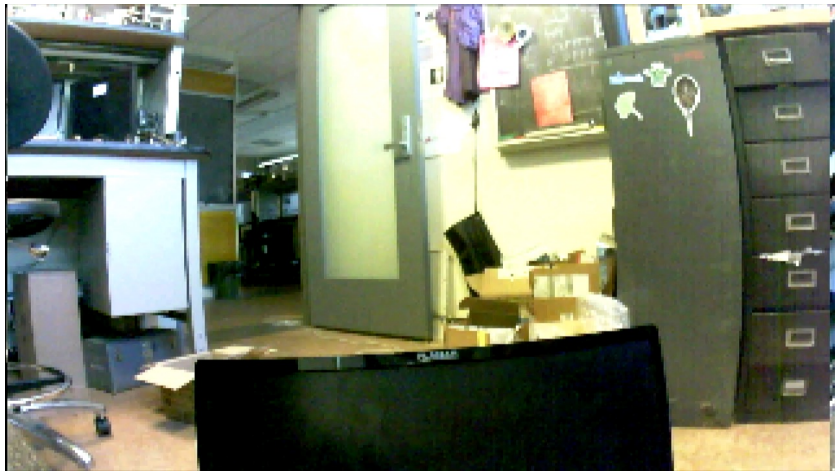
6.205

Week 7: Convolution

- Only have to write two/three modules
- Please start early


```
Phase 25 Post Router Timing
INFO: [Route 35-20] Post Routing Timing Summary | WNS=-0.059 | TNS=-0.319 | WHS=0.050  | THS=0.000  |
```

Light travels 17 mm in the time that my design initially failed by



Administrative

This is not good in lab 7. You do not want yours to look like this.

- Abstract due tomorrow
- Block Diagram report is due on Tue 10/28 at 5pm.
  - You'll also get feedback after
- Week/Lab 07 Due Wed 22nd
- Project presentation will take place after that. Details will come in a couple days about sign ups.

2-Bit Command FIFO

- Clever use of only two bit-width for command_fifo in lab 6.
- Just store the tlast signal rather than the whole address. Very smart.

Downside would be this precludes random access to memory, so for final projects this may not be a good idea depending on needs.

```
logic [1:0] cf_out2; // {read_response_addr, queued_command_write_enable} --> 24 bits + 1 bit
logic [1:0] cf_in2;   // {addr[24:0], write_enable[0]} --> 1 bits last flag + 1 bit write_enable
logic last_flag;
logic read_response_address_2;

assign last_flag = (memrequest_addr == MAX_ADDR);
assign cf_in2 = {last_flag, memrequest_write_enable};

command_fifo #(.DEPTH(64),.WIDTH(2)) mcf(
    // connect ports as appropriate.
    .clk(clk),
    .rst(rst),
    .write(cf_write),
    .command_in(cf_in2),
    .full(cf_full),
    .command_out(cf_out2),
    .read(cf_read),
    .empty(cf_empty)
);
```

Eileen Zu @zuleen

Interfacing with Devices

A great way to add complexity to final projects

Interfacing with Things

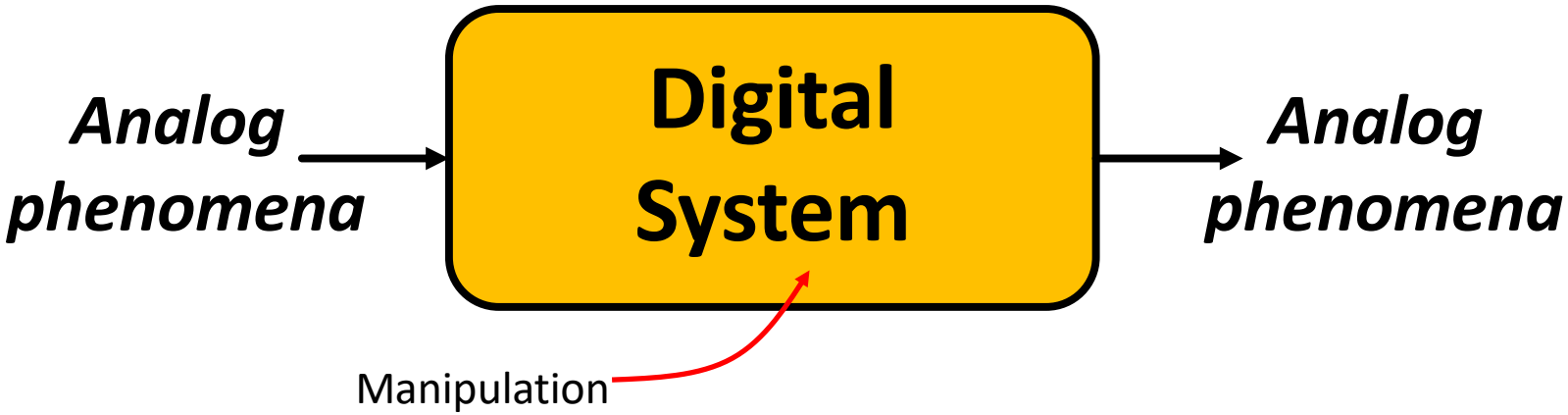
- Sensors
- Actuators
- Memory
- Microcontrollers
- Etc…
- We need ability/fluency to extract info from and work with them

How to get Access to the signals in first place?

- *Some* devices are analog out (can therefore read them with an A-to-D converter) (ADXL335 accelerometer…or the microphone we used in Lab 02, for example)
- These have limited functionality…and also it is analog so there's the whole noise issue....which is not nice
- *Most* modern sensors by-far are interfaced to in a digital form

The reason for this is signal integrity and is the same argument for why we do computation digitally

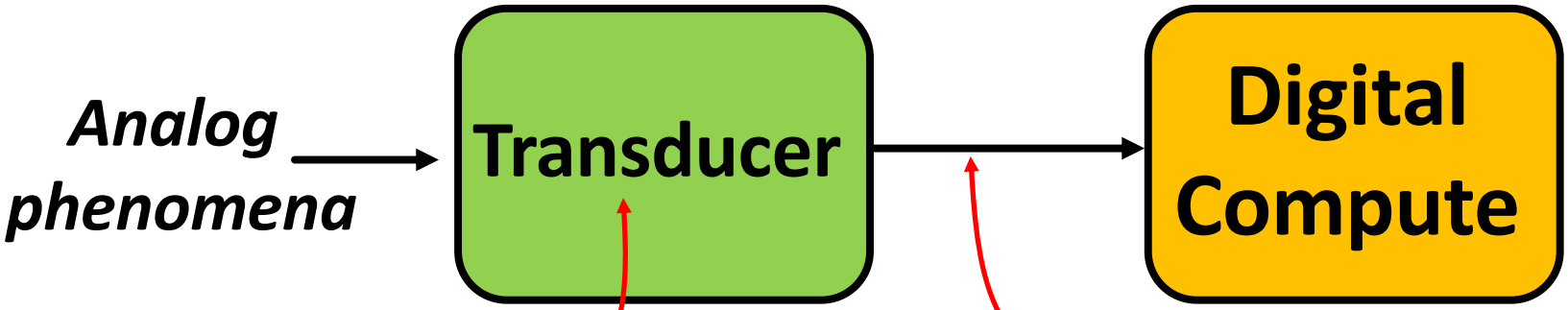
- It is true that most things we care about in terms of sensing and transducing are analog phenomena



- But Analog is inherently noisy…

Sensing…

- Why not keep analog until digital compute?



Converts analog energy
To electrical energy (voltage or current)

- Sound: microphone
- Light: camera/photosensor
- Temperature:
- Vibration
- Smell/air
- Etc…

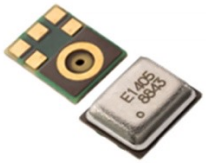
If this leg of the journey stays analog,
The likelihood of information getting lost becomes much higher

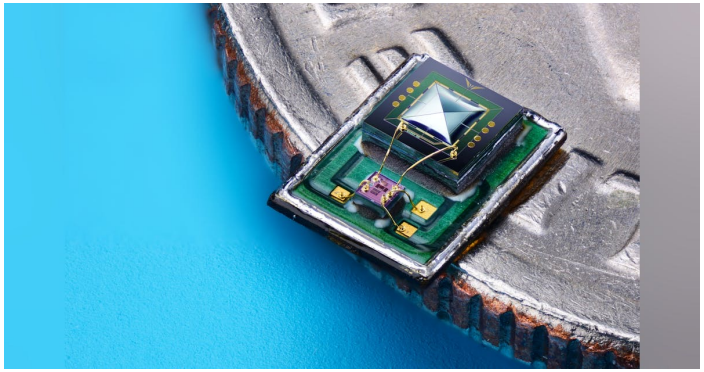
So most of the time asap in your signal chain you convert to digital

Microphones

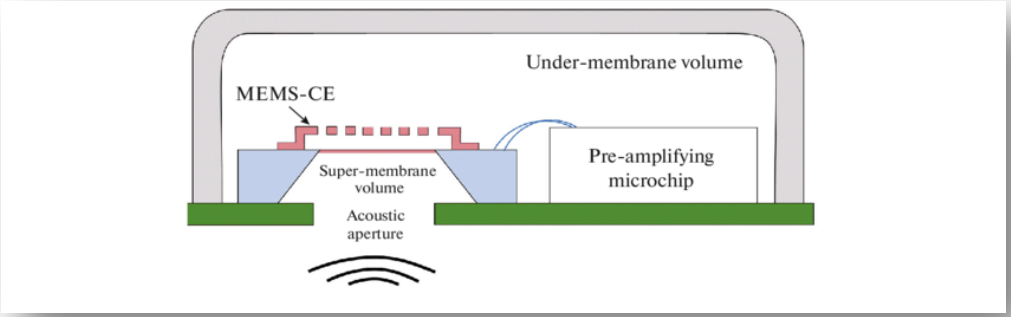
Older analog-out microphone module:

Modern MEMS microphone:
(digital out)





Cracked open sitting on a coin



https://www.researchgate.net/figure/The-design-of-a-MEMS-microphone_fig1_339839767

https://www.electronicdesign.com/technologies/analog/article/21808368/vesper-introduces-digital-mems-microphone-with-integrated-adc

Many sensors are so cheap now…

- …That multiple are used.
- The iPhone 15 has/had four microphones on it
- Airpods/most quasi-decent headphones now have six microphones in them (three for each side). Also have two accelerometers on each earbud for orientation and speech detection
- This pattern is happening a lot

Also many “sensors”…

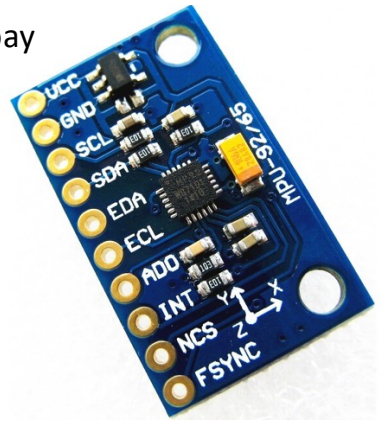
- …Have multiple sensors/transducers in them.

- So often “the microphone” is multiple microphones
- Or “the camera” is multiple cameras, etc…

- An example...

MPU-9250 "IMU"

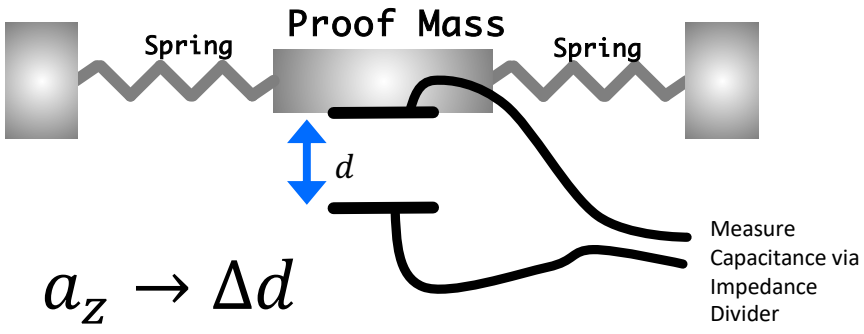
Board: $5.00 from Ebay
Chip: $1.00 in bulk

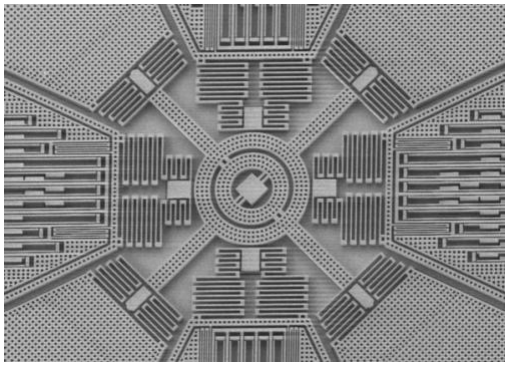


- **3-axis Accelerometer (16-bit readings)**
- **3-axis Gyroscope (16-bit readings)**
- **3-axis Magnetic Hall Effect Sensor (Compass) (16 bit readings)**
- SPI or I2C communication (!)…no analog out
- On-chip Filters (programmable)
- On-chip programmable offsets
- On-chip programmable scale!
- On-chip sensor fusion possible (with quaternion output)!
- Interrupt-out (for low-power applications!)
- On-chip sensor fusion and other calculations (can do orientation math on-chip or pedometry even)
- So cheap they usually aren't even counterfeited! ☺
- **Communicates using either I2C or SPI**

Accelerometers

- First MEMS accelerometer: 1979
- Position of a proof mass is capacitively sensed and decoded to provide acceleration data



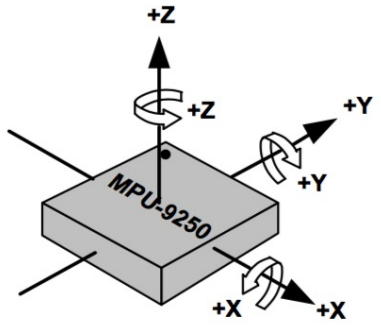


SEM of two-axis accelerometer

Uses of Acceleration Measurements:

- Acceleration can be used to detect motion
  - (pedometer, free-fall/drop detection):

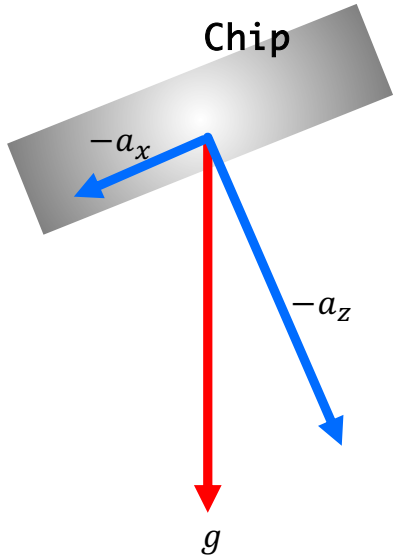
$$a_T = \sqrt{a_x^2 + a_y^2 + a_z^2}$$



Accelerometer directions
+X, +Y, +Z

- Use gravity and trig to find orientation:

$$\theta_y = \tan^{-1}\left(\frac{a_z}{a_x}\right)$$



Problems

- Accelerometers have huge amounts of high-frequency noise
- To fix, usually Low Pass Filter the raw signal (**Infinite Impulse Response*** approach shown below)
- This cuts down on frequency response though ☹

$$\theta_y[n] = \theta_y[n-1]\beta \ +(1-\beta)\tan^{-1}\left(\frac{a_z[n-1]}{a_x[n-1]}\right)$$

a_x X acceleration

a_z z acceleration

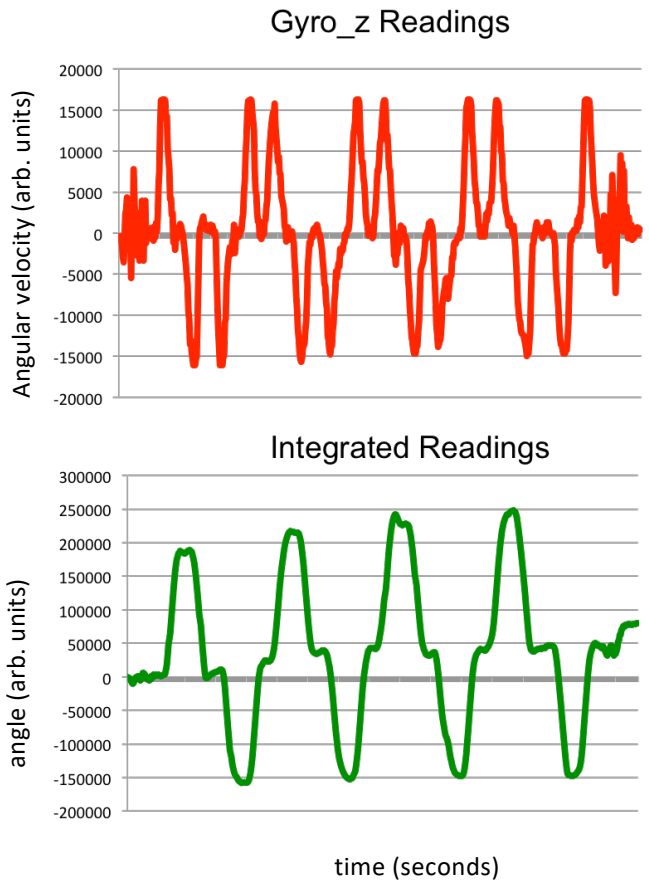
$0 < \beta < 1$ Filter Coefficient

θ_y Angle estimate around y axis

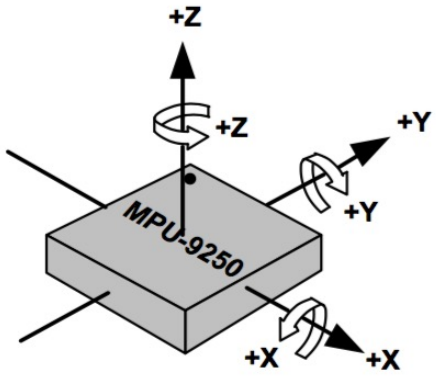
*from lecture 12

Bring in Gyroscopes

- Provide Direct **Angular Velocity** which we can integrate to get angle
- Very little high-frequency noise, but lots of low frequency noise (Gyros drift like crazy)

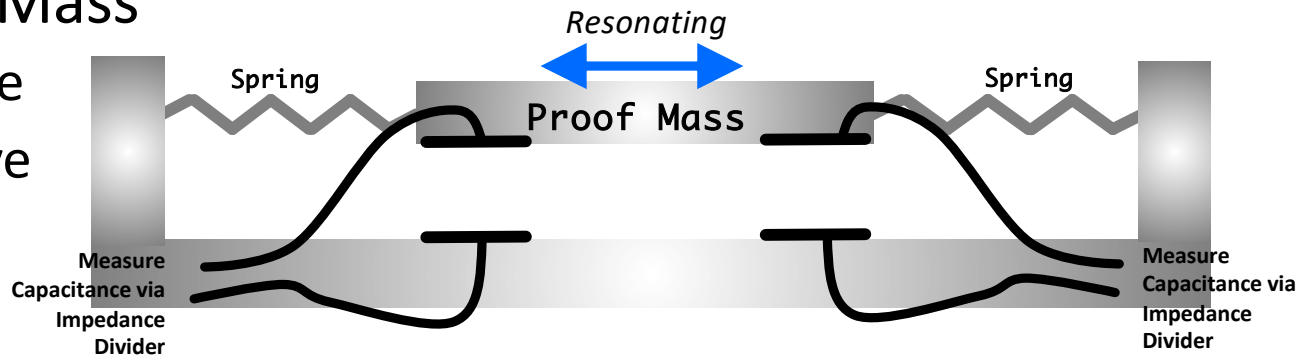


Gyro readings are "around" the axis they refer to (use right-hand rule):

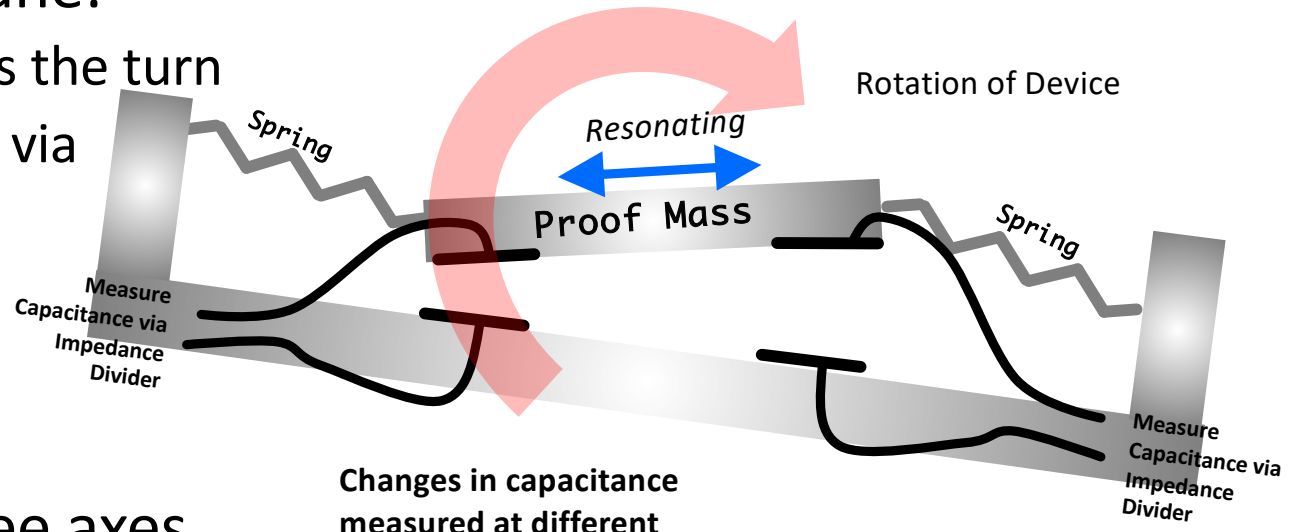


Gyro Operation

- Resonating Proof Mass
  - Electrostatic Drive
  - Piezoelectric Drive



- Turning out-of-plane:
  - Proof-mass fights the turn
  - Detect deviation via capacitance



- Do this for all three axes

Changes in capacitance measured at different points

Scale not accurate/nor design details

How to use Gyro Readings:

- Because of Drift (low frequency noise/offset) you want to avoid doing much long-term integration with a gyro reading
- Having beta less than unity ensures any angle that comes from gyro reading will eventually disappear, but in short term it will dominate
- Calculation per timestep: $\theta_g[n] = \beta\theta_g[n-1] + Tg_y[n-1]$

$0 < \beta < 1$ Filter Coefficient

$\beta \approx 0.95$ starting point

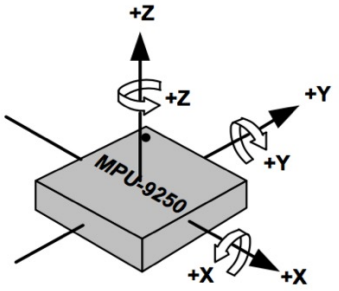
g_y Gyro y reading

T Time Step

What to do?

- Using only accelerometer, leaves us blind to motion/change in the short term but fine in the long-term
- Using only gyroscope, leaves us blind in the long term, but good in the short term
- What to do?

Merge the signals



- **Complementary Filter:**

$$\theta_y[n] = \beta\left(\theta_y[n-1] + T g_y[n-1]\right) + (1-\beta)\tan^{-1}\left(\frac{a_z[n-1]}{a_x[n-1]}\right)$$

$0 < \beta < 1$ Filter Coefficient

T Time Step

g_y Gyro y reading

$\beta \approx 0.95$ good starting point

a_x X acceleration reading

a_z z acceleration reading

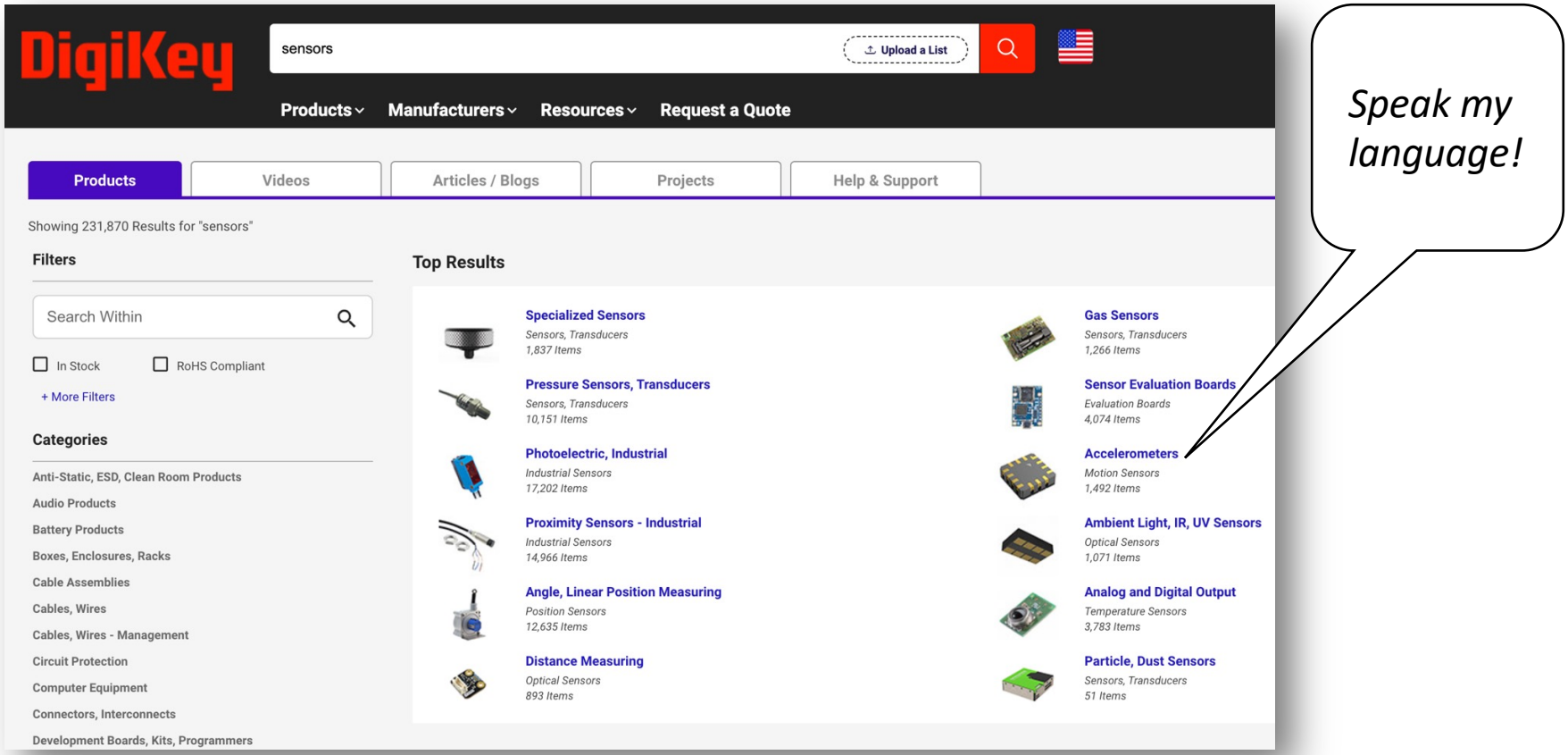
- Very simple form of **sensor fusion** (where you merge data from more than one sensor to build up model of what is going on)

Sensor Fusion

- Most modern sensors are used with other sensors:
- Can be incorporated open-loop (like complementary filter on previous page)
- Or incorporate into “learning” algorithms:
  - NLMS, Kalman, LQE, Baysean, Linear-Observer System
  - Estimate, compare to new data, correct, repeat…
  - These usually feature dynamic filters which learn how to filter the signal they care about

So a plethora of sensors out there

- But they all need to be communicated with…



Parallel vs. Serial in Wires

PARALLEL PROTOCOLS

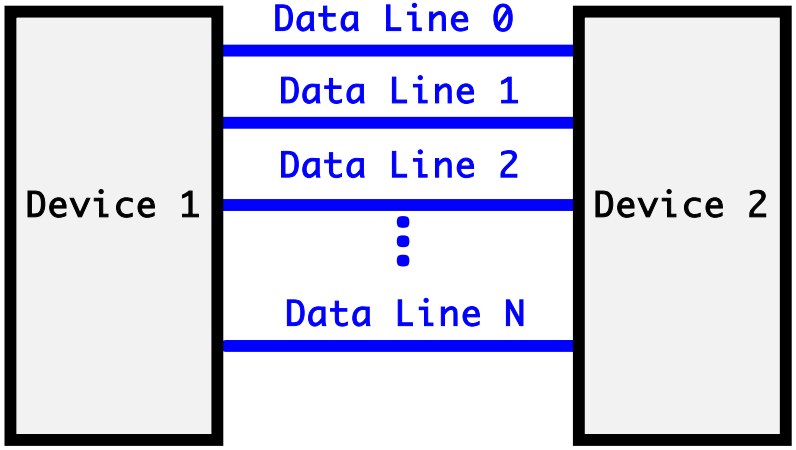
- **Parallel** (not so much on individual small devices)…mostly memory and things that need to send data at ***very*** high rates such as a camera, high-speed ADCs, etc…

SERIAL PROTOCOLS

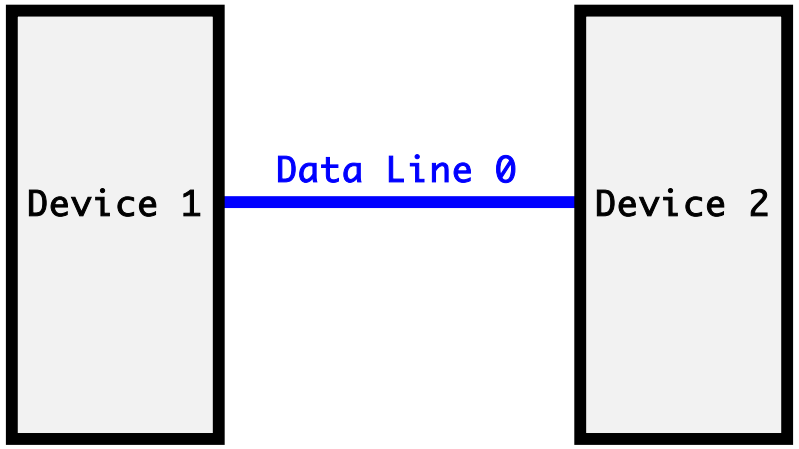
- **UART "Serial"** very common
- **SPI (Serial Peripheral Interface)** very common
- **I2C (Inter-Integrated Circuit Communication)** very common

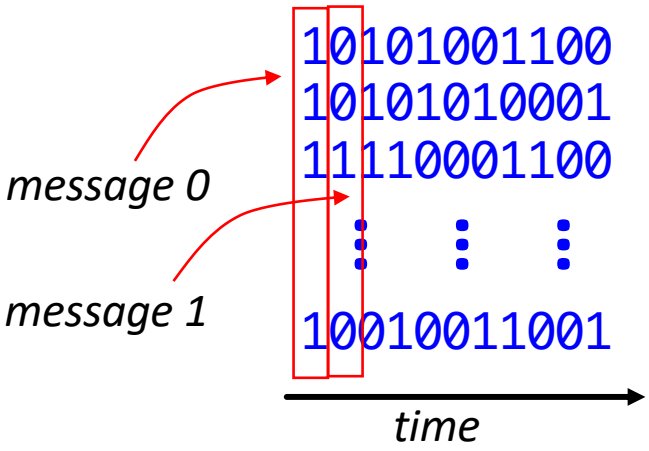
Data Transmission with Wires...

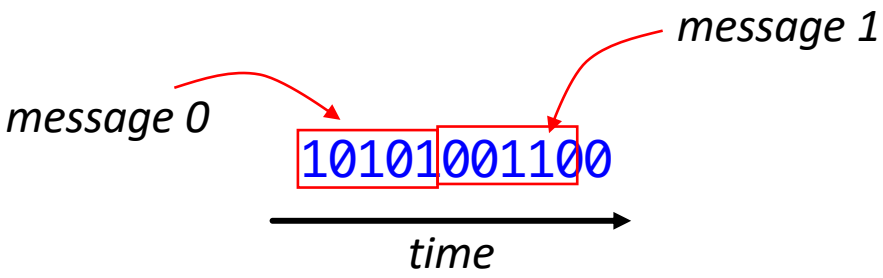
Parallel Link using Wires:



Serial Link using Wires:







Parallel vs. Serial in Wires

PARALLEL PROTOCOLS

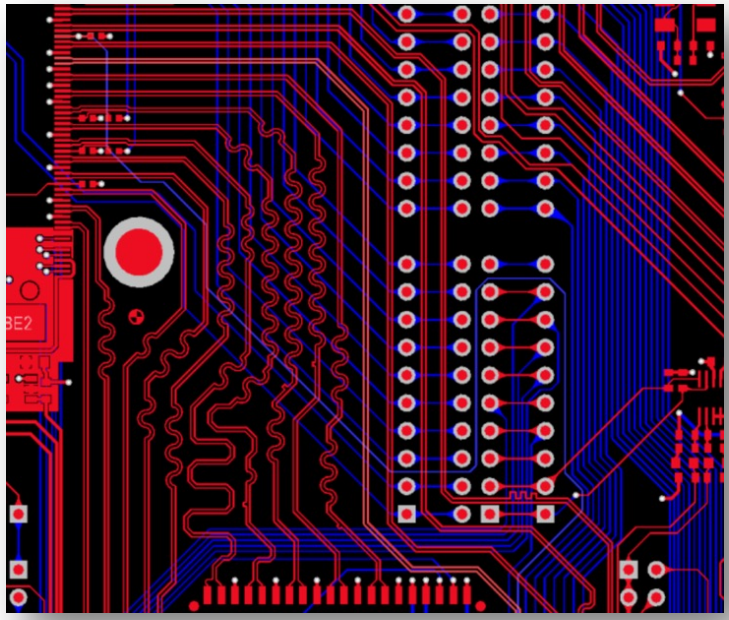
- **Parallel** (not so much on individual small devices)…mostly memory and things that need to send data at ***very*** high rates such as a camera, high-speed ADCs, etc…

SERIAL PROTOCOLS

- **UART "Serial"** very common
- **SPI (Serial Peripheral Interface)** very common
- **I2C (Inter-Integrated Circuit Communication)** very common

When Choose Parallel?

- When you need to transfer **large** amounts of data over short distances, parallel is a better choice.
- Data Transfer Rate will scale ~linearly with number of wires
- But Have to be careful of wiring length:
  - Ensure bits arrive same time
- Uses lots of space!!!



https://docs.toradex.com/102492-layout-design-guide.pdf

Where Have We Seen Parallel Data Transfer So Far?

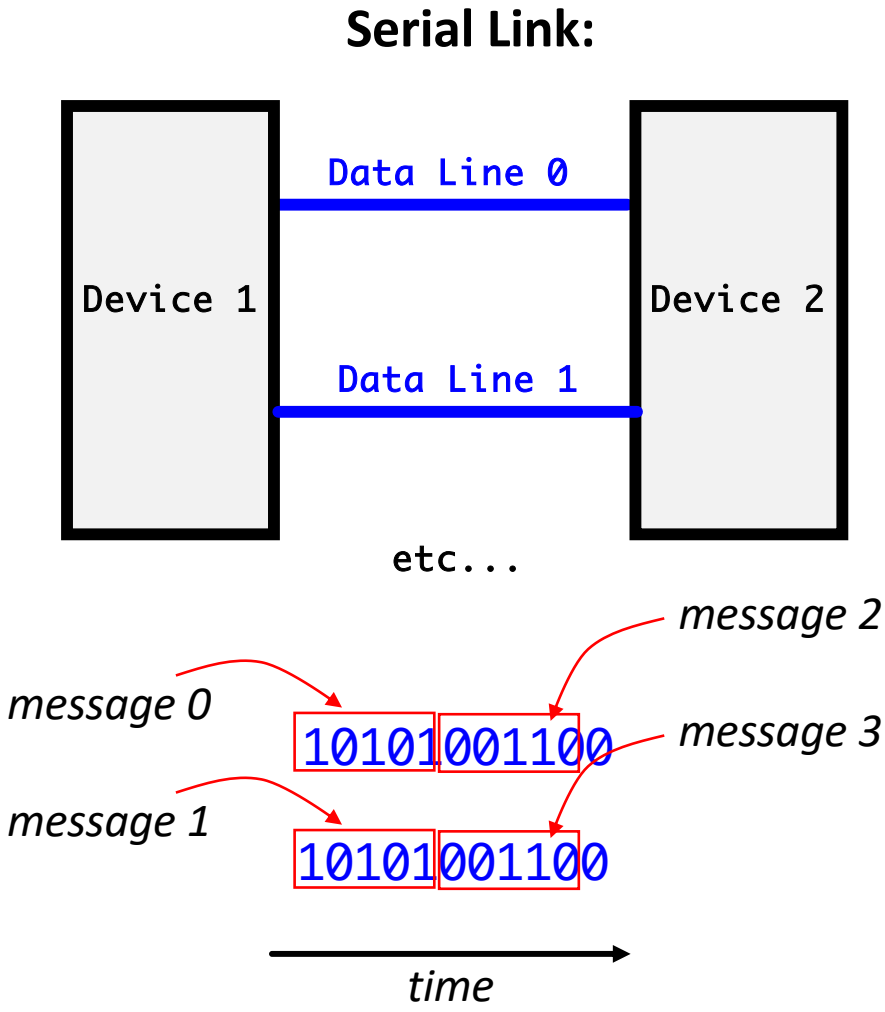
- Camera in labs 5-7 (moving in 200 to 400 Mbits per second across the 8 data pins which you then reassemble (or try to reassemble) using pixel_reconstruct

- DRAM (16 pins)

Communications Trends

- Serial: good for long distance (save on cable, pin and connector cost, easy synchronization). Requires "serializer" at sender, "deserializer" at receiver
- Parallel: issues with clock skew, crosstalk, interconnect density, pin count. Used to dominate for short-distances (eg, between chips).
- **BUT for *high data* movement, modern preference is for parallel, but independent serial links (eg, PCI-Express x1,x2,x4,x8,x16) as a hedge against link failures. Ethernet, USB, etc… these all follow that same pattern**

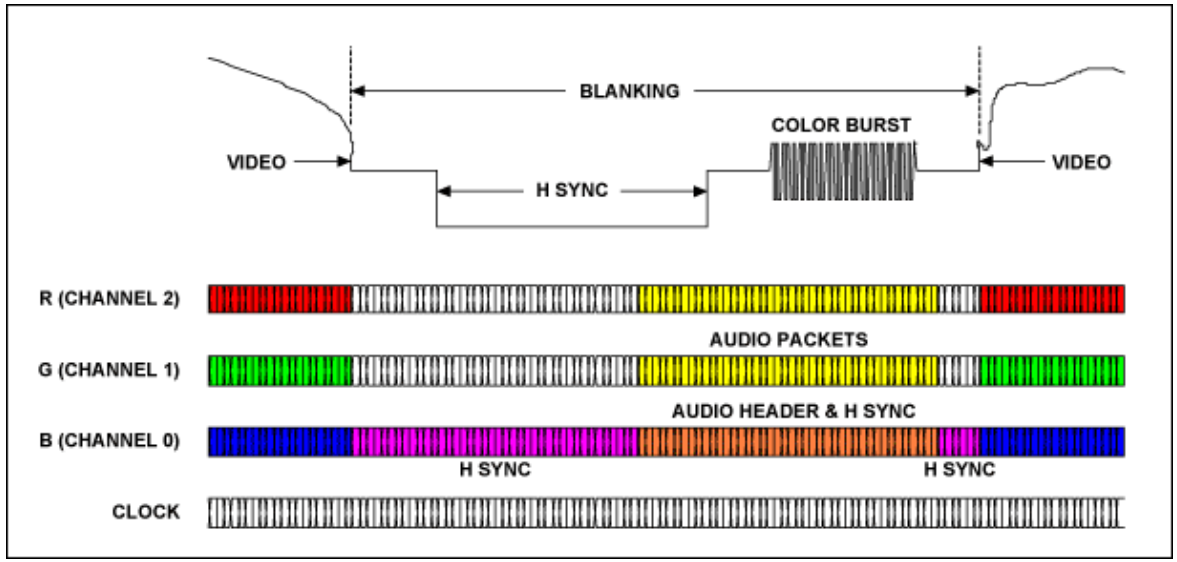
Multiple Serial Links in Parallel



- Multiple separate serial channels coexist.
- Generally data sent on each channel isn't intricately tied together (maybe separate packets/message)...n splitting bits across multiple wires

Where Have we Seen Multiple Serial Links?

- TMDS in DVI/HDMI!



- You'll also see LVDS and variants in a lot of higher-speed things like cameras

Parallel vs. Serial in Wires

PARALLEL PROTOCOLS

- **Parallel** (not so much on individual small devices)…mostly memory and things that need to send data at **very** high rates such as a camera, high-speed ADCs, etc…

SERIAL PROTOCOLS

- **UART "Serial"** very common
- **SPI (Serial Peripheral Interface)** very common
- **I2C (Inter-Integrated Circuit Communication)** very common

Serial Standards

- A zillion Serial standards
  - Asynchronous (no explicit clock) vs. Synchronous (CLK line in addition to DATA line).
  - Recent trend to reduce signaling voltages: save power, reduce transition times
  - Control/low-bandwidth Interfaces: SPI, I^2C, 1-Wire, PS/2, AC97, CAN, I2S,
  - Networking: RS232, Ethernet, T1, Sonet
  - Computer Peripherals: USB, FireWire, Fiber Channel, Infiniband, SATA, Serial Attached SCSI
  - Graphics: DVI, HDMI, DisplayPort

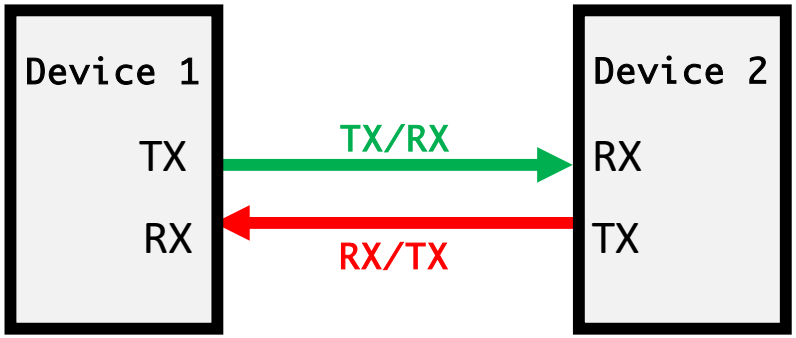
Common Chip-to-Chip Communication Protocols (not exhaustive)

- **Parallel** (not so much anymore)…mostly memory and things that need to send data at **very** high rates such as a camera, high-speed ADCs, etc…
- **UART "Serial"** (still common in random devices, reliable and easy to implement)
- **SPI (Serial Peripheral Interface)** very common
- **I2C (Inter-Integrated Circuit Communication)** very common
- **I2S (Inter-Integrated Circuit Sound Bus**) very common in audio-specific applications

Common Chip-to-Chip Communication Protocols (not exhaustive)

- **Parallel** (not so much anymore)…mostly memory and things that need to send data at **very** high rates such as a camera, high-speed ADCs, etc…
- **UART "Serial"** (still common in random devices, reliable and easy to implement)
- **SPI (Serial Peripheral Interface)** very common
- **I2C (Inter-Integrated Circuit Communication)** very common
- **I2S (Inter-Integrated Circuit Sound Bus**) very common in audio-specific applications

UART aka “Serial”



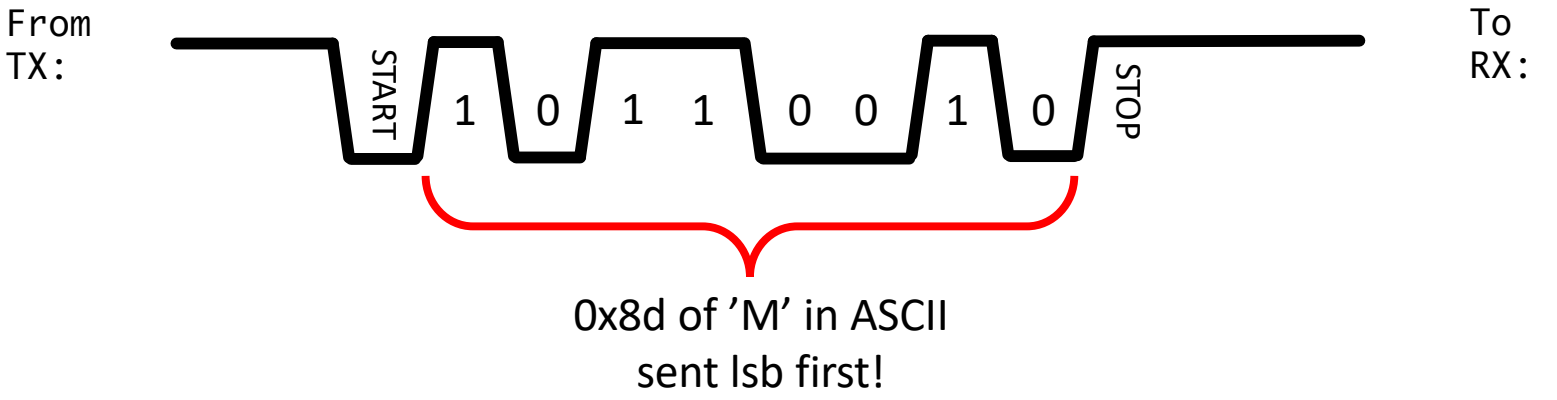
- Stands for Universal Asynchronous Receiver Transmitter
- Requires ***agreement ahead-of-time*** between devices regarding things like clock rate (BAUD), etc…
- Two wire communication for bi-directional (or one if you only want to talk and not listen like a bad relationship partner)
- Cannot really share
  - (every pair of devices needs own pair of lines so wires scales as $2n$ where n is the number of devices)
- Data rate generally < 1Mbps (though can maybe push a little bit)
- Data sent least significant bit (lsb) first

The Naming on UART is Perpetually a Mess with the TX/RX confusion

- When working with UART take care to pay attention to the TX and RX pins.
- They are complementary...one device's TX talks to another devices RX.
- But boards and datasheets will sometimes label things backwards

UART

- Line High at rest (“high” an “low” depend on system specs...5V/0V...3.3V/0V, -12V/+12V...)
- Drops Low to indicate start
- 8 (or 9 bits follows) sent least significant bit first
- Goes high (stop bit)
- Can have optional parity bit for simple error correction



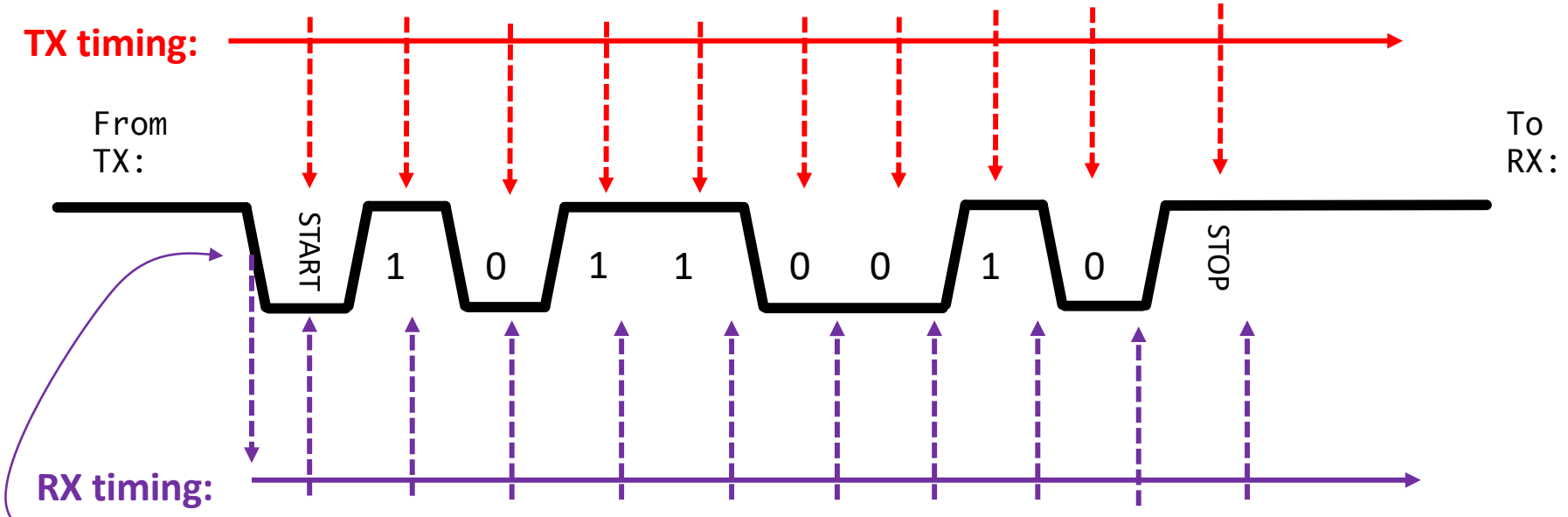
In UART, messages *must* be short (one byte)

- Both parties must agree ahead of time to a bit rate.
- A bit rate is bits per second
- Does everyone know what a second is?
- Does everyone *actually* know what a second is?
- What is a second?
- What are we even doing here?
- What are the implications of imperfect synchronization?

Timing Differences

- Atomic Clocks can range from $1500 to $200,000 depending on how good you want them.
- If we want commodity electronics to be cheap, $200,000 makes that hard to do.
- They must use “good-enough” local clocks and we build up communication protocols to accommodate for that.
- You must Synchronize your data transmission and reception

Synchronization

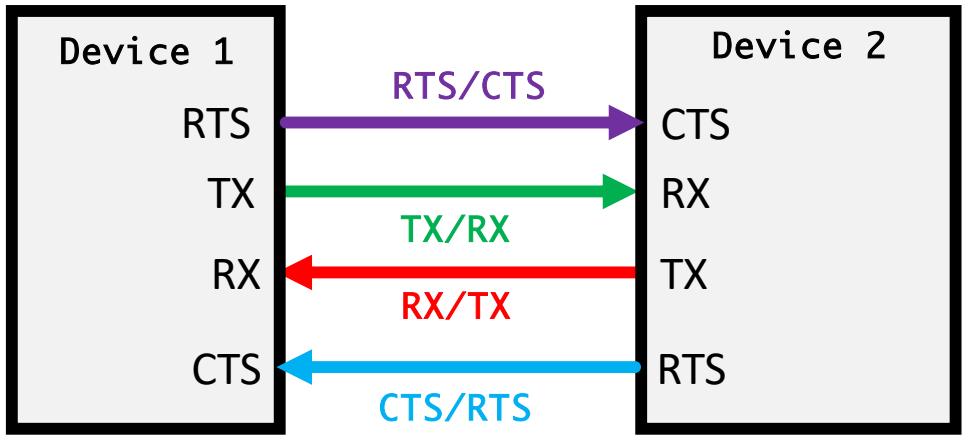


Receiver sees the high signal and waits for it to fall. From that edge it starts its timing

- Even if the timing of the RX and TX sides differ slightly, by keeping the messages short, the chance of getting too far out of sync is very, very low.
- Every new byte forces a resynchronization so errors never get a chance to accumulate too far!

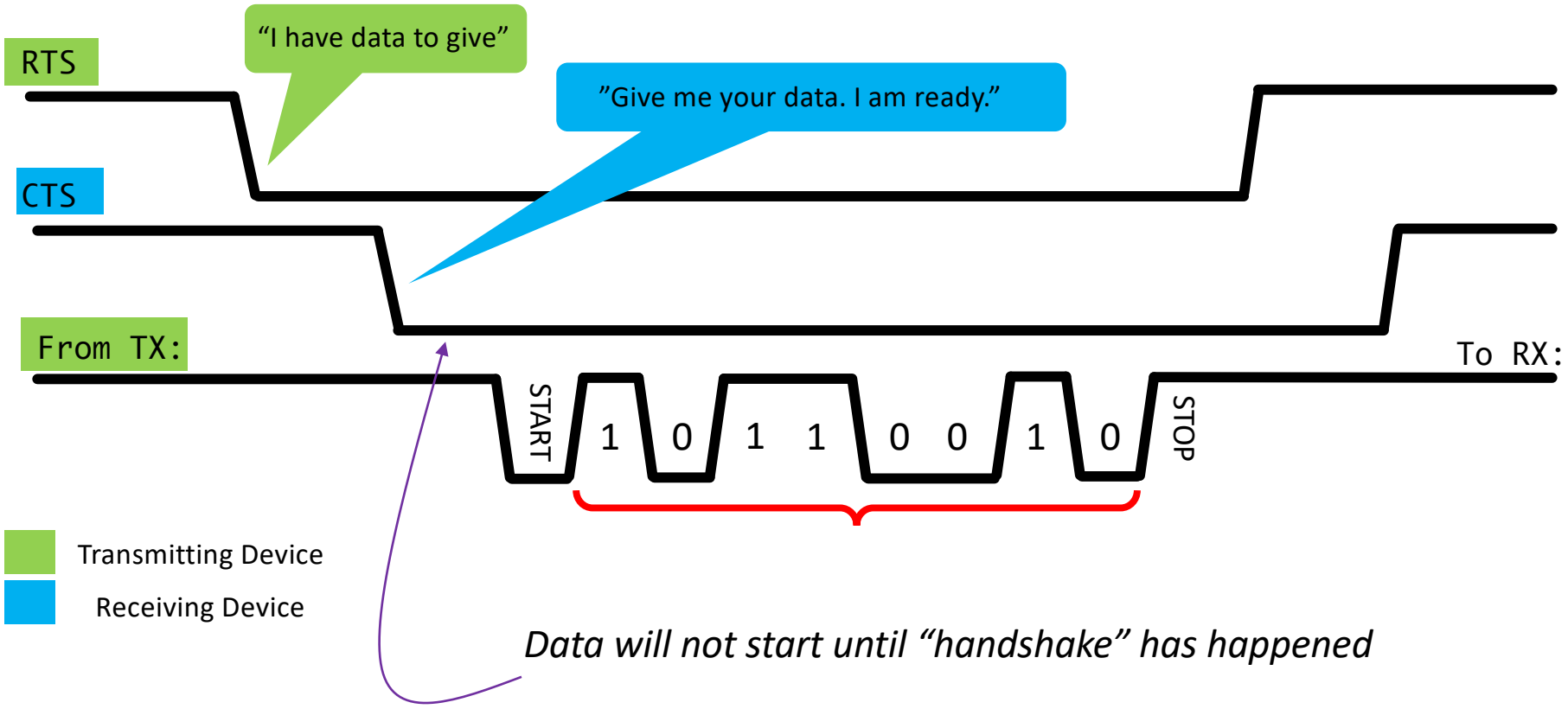
UART and RX/TX and RTS/CTS

- UART will also sometimes come with
  - “Ready to Send” signals (RTS)
  - “Clear to Send” signals (CTS)
- These are Flow-Control Signals that allow the two parties to tell each other if they have data to send if they are ready to receive data



UART Transmission

- RTS and CTS sit high. Each device in charge of setting the RTS and listening to the CTS
- Device pulls RTS low. Other device sees that and then pulls its CTS low in response



UART Thoughts? Goods? Bads?

- Everything contained within one wire for the most part?

- Not super fast

Data Synchronization

- In UART, small data bursts with periodic resynchronizations are needed to make sure both parties produce and read data at the same time.

- How else to do this?

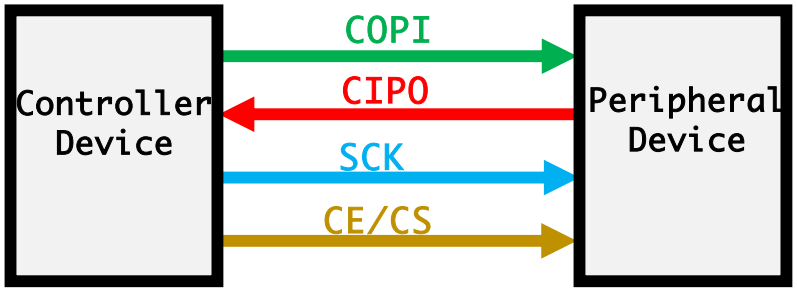
Common Chip-to-Chip Communication Protocols (not exhaustive)

- **Parallel** (not so much anymore)…mostly memory and things that need to send data at **very** high rates such as a camera, high-speed ADCs, etc…
- **UART "Serial"** (still common in random devices, reliable and easy to implement)
- **SPI (Serial Peripheral Interface)** very common
- **I2C (Inter-Integrated Circuit Communication)** very common
- **I2S (Inter-Integrated Circuit Sound Bus**) very common in audio-specific applications

Note on Terminology

- Master/Slave terminology is heavily used in SPI and I2C...Master controls a bus, Slave listens.
- Acknowledge the issues with it, but also because many datasheets/vendors still use it, it is hard for us to separate from it completely.
- Changing slowly
- Maybe use ”Main”/”Secondary” to keep the letters the same or ”Controller” and “Peripheral”
- Also seeing SDO/SDI for “Serial Data Out/In” with respect to controlling device more recently
- Or Controller/Peripheral in some other vendors

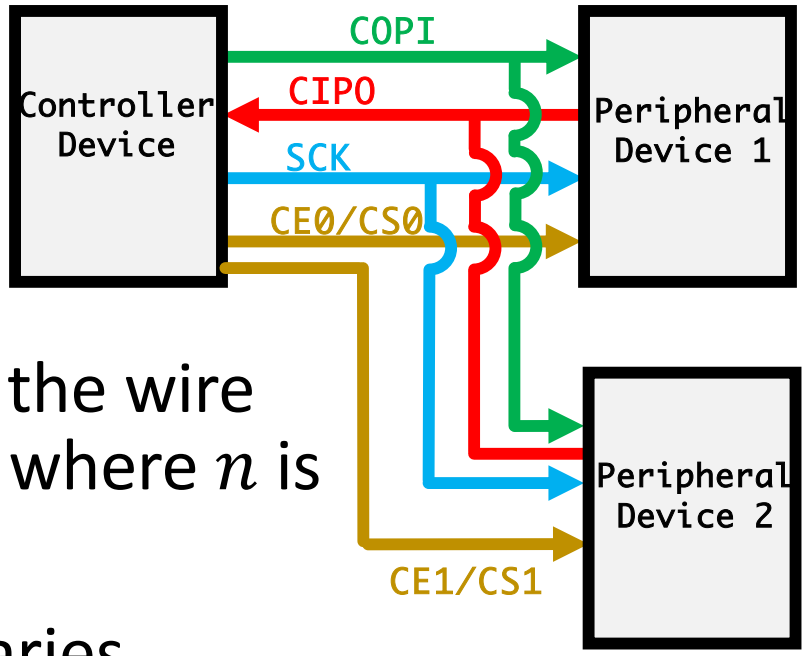
SPI



MOSI also = SDO "serial data out"
MISO also SDI "serial data in"
Also seeing now:
COPI = Controller Out Peripheral In
CIPO = Controller In Peripheral Out

- Stands for Serial-Peripheral Interface
- Four Wires:
  - COPI: Controller-Out-Peripheral-In…
  - CIPO: Controller-In-Peripheral-Out…
  - SCK: Serial Clock
  - CE/CS (Chip Enable or Chip Select)
- SCK removes need to agree ahead of time on data rate (from UART)…makes data interpretation much easier!
- High Data Rates: (1MHz up to ~70 MHz clock (bits))
- Data msb or lsb first…up to devices/spec

SPI Expansion

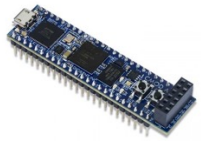


- Can share COPI/CIPO Bus so the wire requirement scales as $3 + n$ where n is the number of devices
- Addition of multiple secondaries requires additional select wires
- Hardware/firmware for SPI is pretty easy to implement:
  - Wires are uni-directional
  - Classic “duh” sort of approach to digital communication, but very robust.

SPI Example

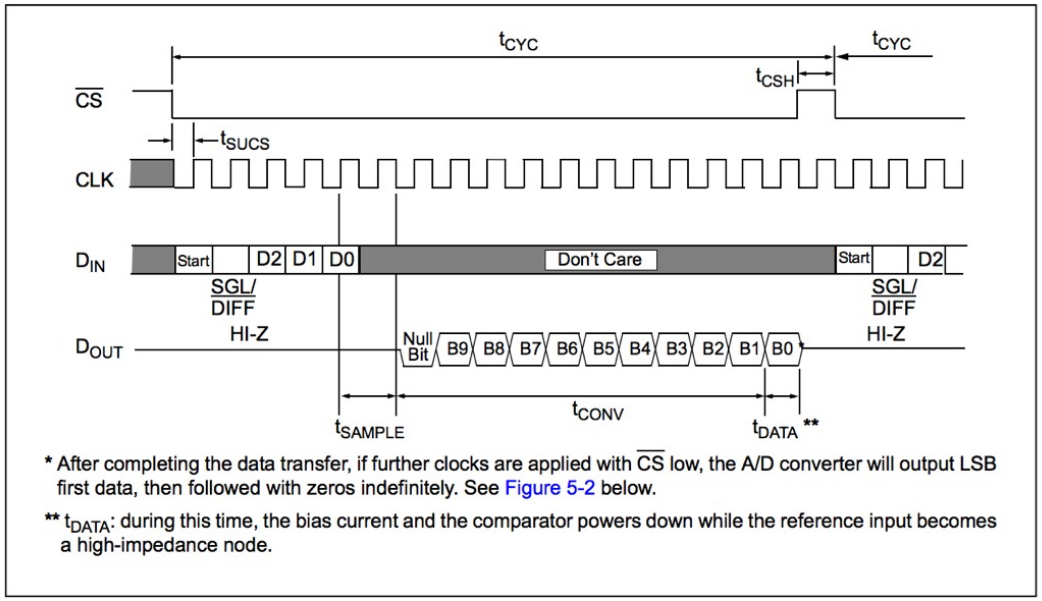
MCP3008 is a 8-channel 10 bit Analog to Digital Converter from Microchip Semi that communicates over SPI

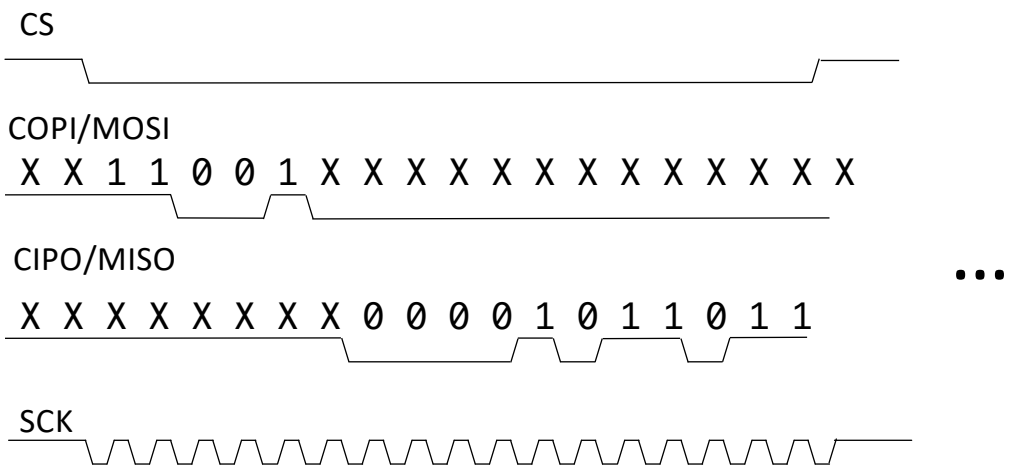
CMOD-A7-35T



MCP3008

From MCP3008 Datasheet





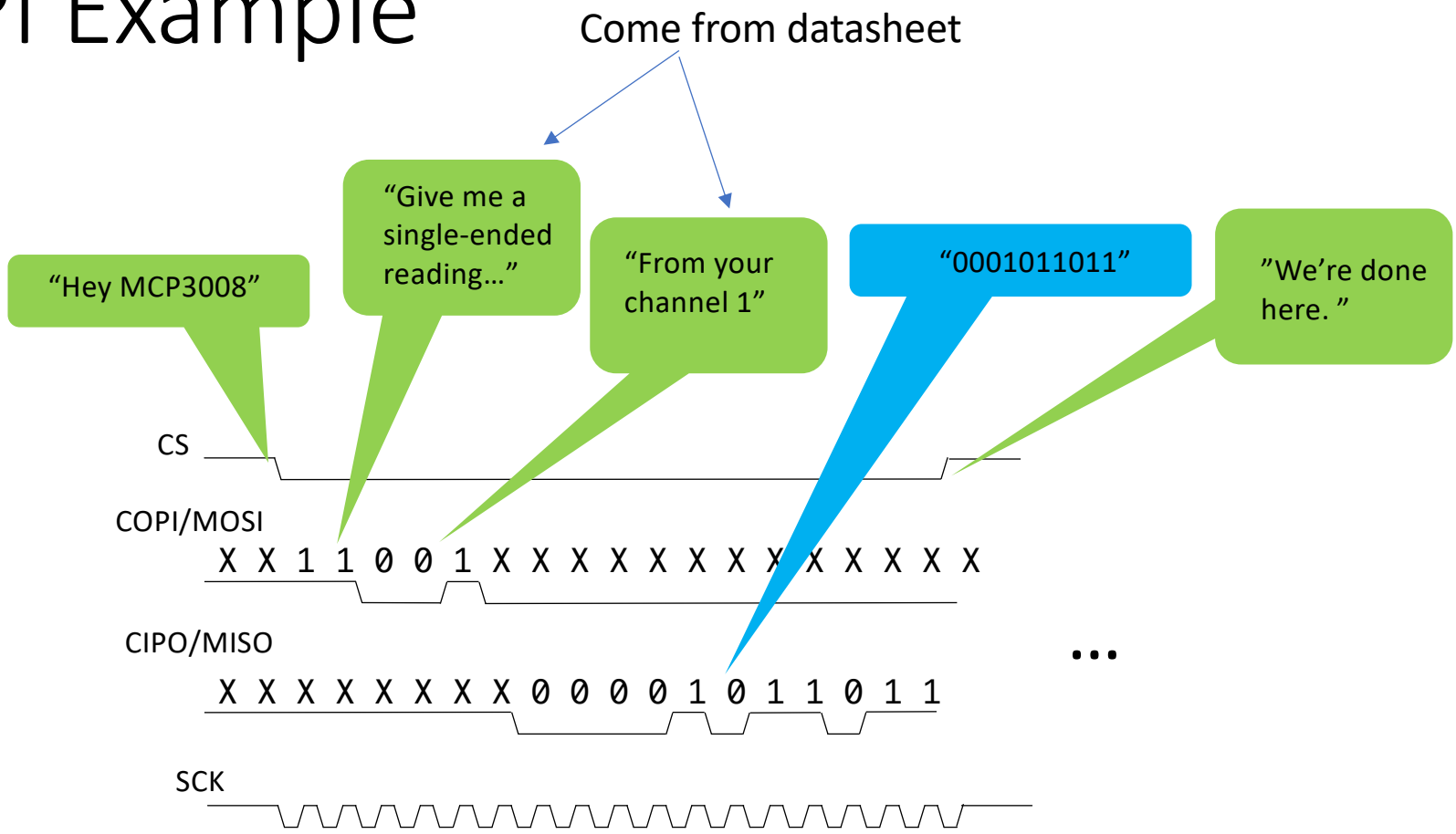
Here I am talking to a MCP3008 10 bit ADC

...

Sends its data **msb** first

Not all devices do this (must check datasheet)

SPI Example



(Controller/Main Device) Dialog

MCP3008 (Peripheral/Secondary Device) Dialog

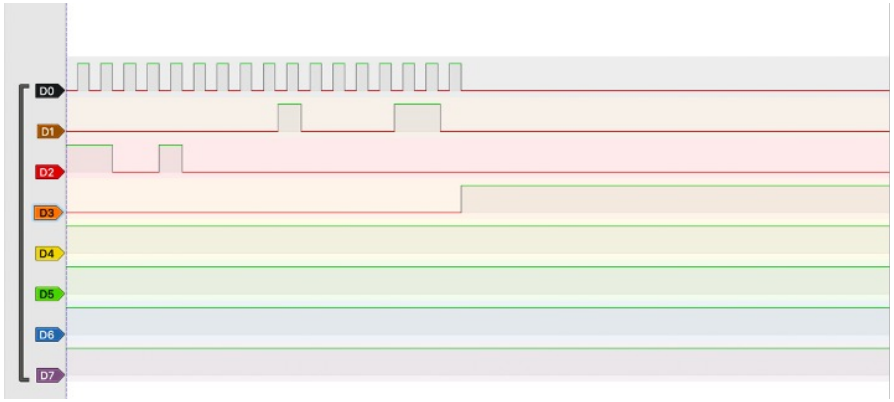
X means don't care as in could be 1 or 0

SPI In Real Life (You've Seen this before)

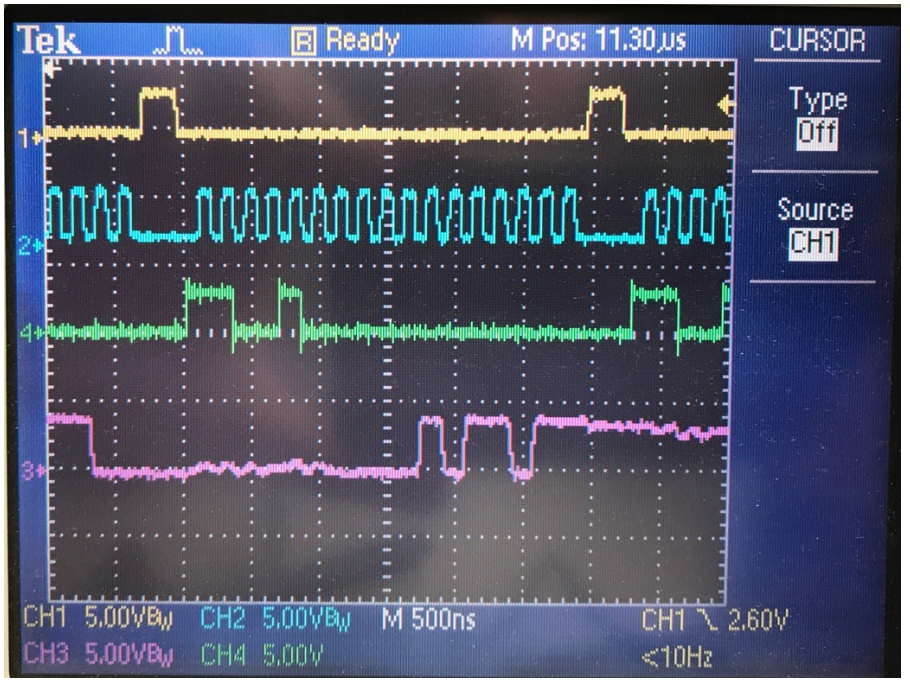
- Here I am talking to the same chip I was daydreaming about talking to on the previous slide.
- Dreams do come true
- I'm saying, "give me your measurement on Channel 1," and it is responding with "10'b0001011011" mapped to 3.3V or 0.293 V

REMINDER: Digital In Analog Life vs Digital in Digital Life

- *What noise? I don't care about noise (within reason)*

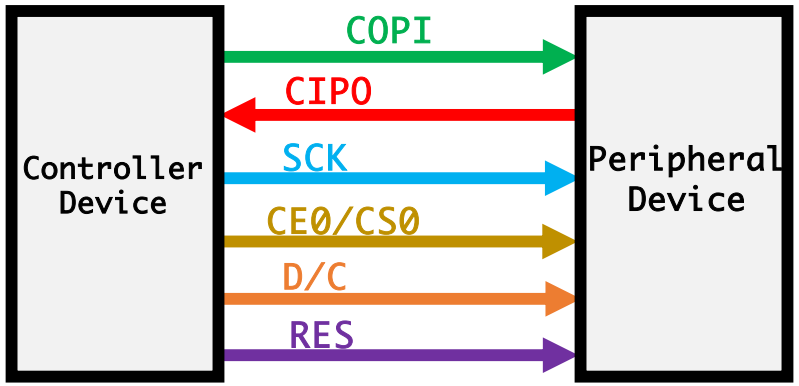


Logic Analyzer Capture of SPI transaction



Oscilloscope Analog Capture of different SPI transaction

SPI Variations

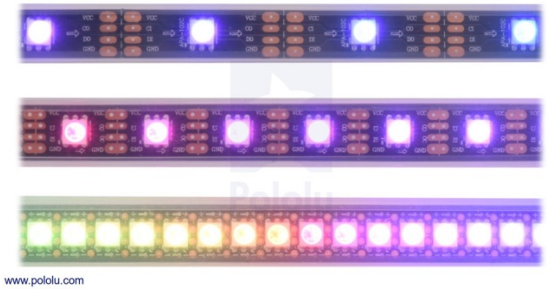


- Six Wires:
  - COPI: Controller-Out-Peripheral-In
  - CIPO: Controller-In-Peripheral-Out
  - SCK: Clock
  - CE/CS (Chip Enable or Chip Select)
  - RES: Reset Device
  - D/C: Data/Command (often seen in devices where you need to write tons of data (i.e. a display)

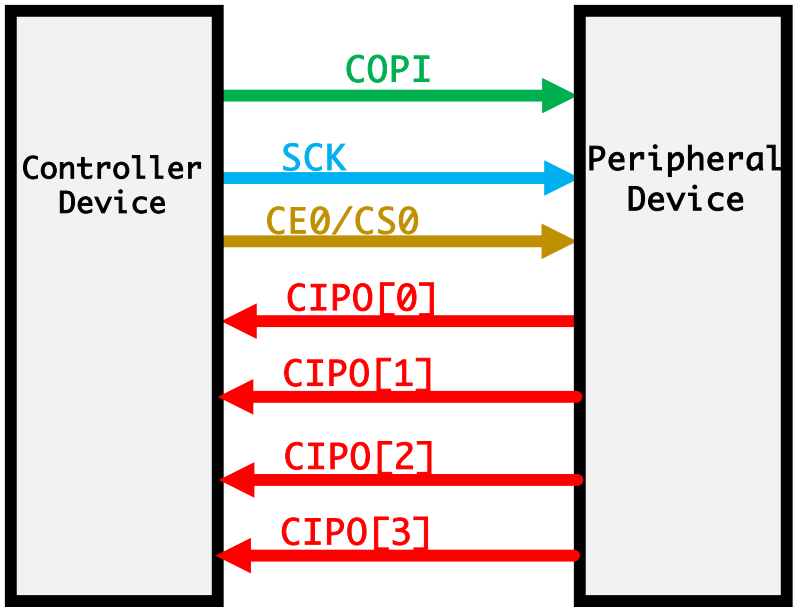
- Three/Two Wires:
  - If a device has nothing to say, drop CIPO:
  - If you assume only one device on bus drop CE/CS, so only have SCK and COPI, sometimes just called “DO” (for data out) in this situation

LCD Display:

Twitch Streamer LEDs:

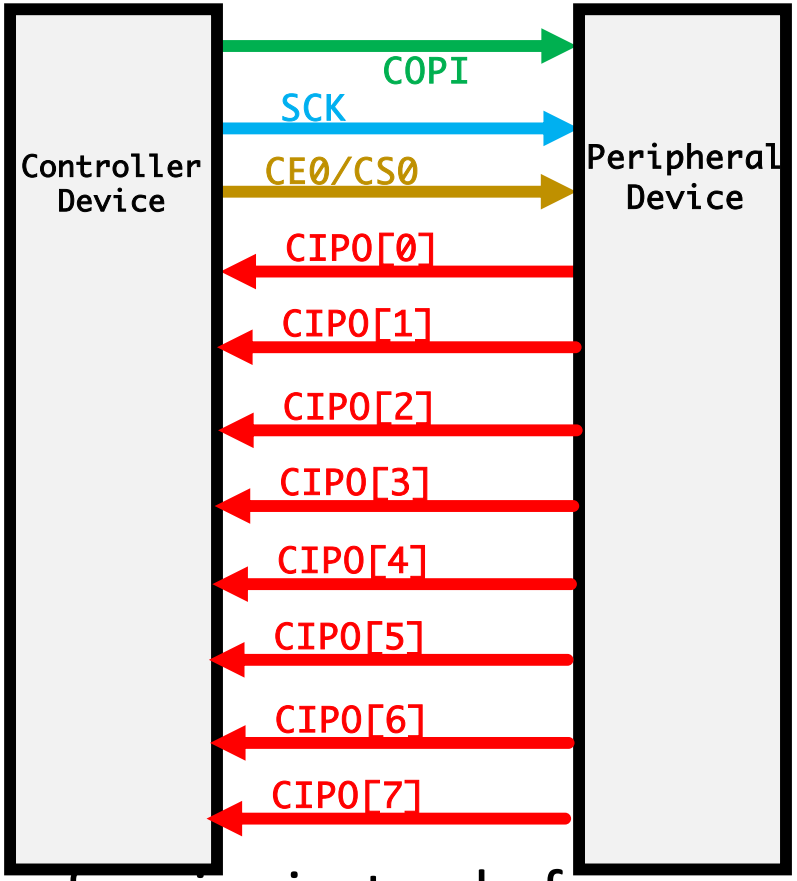


Other SPI Variations



- QSPI: “Quad SPI”
- This is basically SPI...
- But there will be four data transfer pins instead of one
- See in a lot of flash memory chips
- This really isn’t “Serial” in the way God meant it though...the bits are usually sampled together so it is a parallel data transfer just poorly named

Other SPI Variations



- 8SPI: “Octal SPI”
- This is basically SPI...
- ...
- But there will be eight data transfer pins instead of one
- See in a lot of weird hybrid RAM chips
- If this isn’t parallel data transmission, I don’t know what is...but it is called SPI...sorta like that one relative who just can’t admit they’re wrong so they keep redefining things to keep their worldview going

SPI Conclusions

- It is a very simple and very robust ”idea” of a protocol.
- The simplicity comes at the expense of any wires
- And often...some of the complexity is deferred to the ones deploying...There are many variations and dialects of it, so you should always always always read the datasheet for these things.

Be careful! Read Datasheets

- A big screw-up point is mixing up clock polarity...sometimes data is sampled on rising edge...others on falling edge

Table 1. SPI Modes with CPOL and CPHA

SPI Mode	CPOL	CPHA	Clock Polarity in Idle State	Clock Phase Used to Sample and/or Shift the Data
0	0	0	Logic low	Data sampled on rising edge and shifted out on the falling edge
1	0	1	Logic low	Data sampled on the falling edge and shifted out on the rising edge
2	1	0	Logic high	Data sampled on the falling edge and shifted out on the rising edge
3	1	1	Logic high	Data sampled on the rising edge and shifted out on the falling edge

- Clock level at idle can also matter sometimes

SPI Upsides?

- Simple to Implement
- Capable of Very High Speeds (50 MHz is not rare for some displays)

SPI Downsides?

- A lot of wires...which might seem like nbd, but in reality pincount is a huge cost factor in chip manufacture...tons of economic pressure to minimize this.

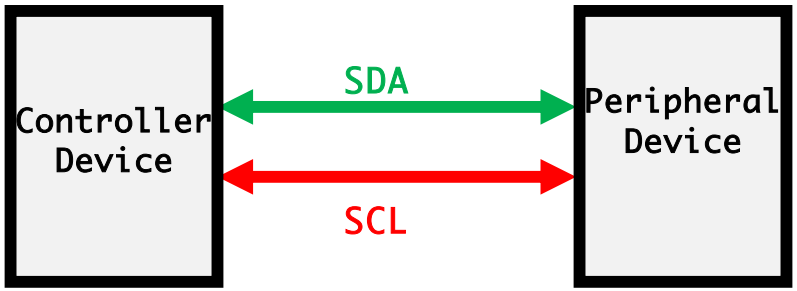
- At very high speeds it actually gets really noisy especially QSPI or OSPI

Common Chip-to-Chip Communication Protocols (not exhaustive)

- **Parallel** (not so much anymore)…mostly memory and things that need to send data at **very** high rates such as a camera, high-speed ADCs, etc…
- **UART "Serial"** (still common in random devices, reliable and easy to implement)
- **SPI (Serial Peripheral Interface)** very common
- **I2C (Inter-Integrated Circuit Communication)** very common
- **I2S (Inter-Integrated Circuit Sound Bus**) very common in audio-specific applications

I2C

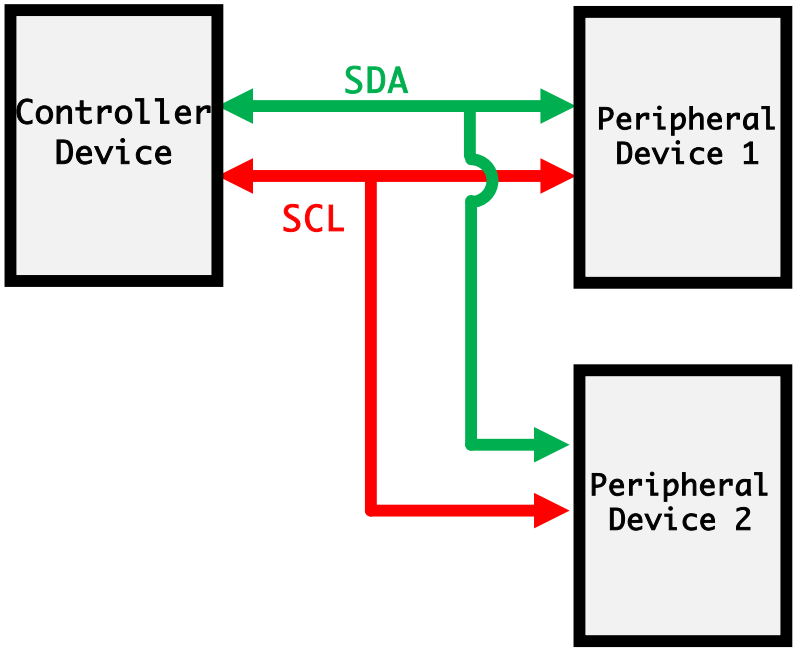
- Stands for Inter-Integrated Circuit communication
- Invented in 1980s
- Two Wire, One for Clock, one for data...Both wires are technically bidirectional, meaning each side can use them
- Usually 100kHz or 400 kHz clock (newer versions go to 3.4 MHz)



On i2C Multiple Devices Require Same # of Wires

- Devices come with their own ID numbers (originally a 7 bit value but more modern ones have 10 bits)…allows potentially up to 2^7 devices or 2^{10} on a bus (theoretically anyways)

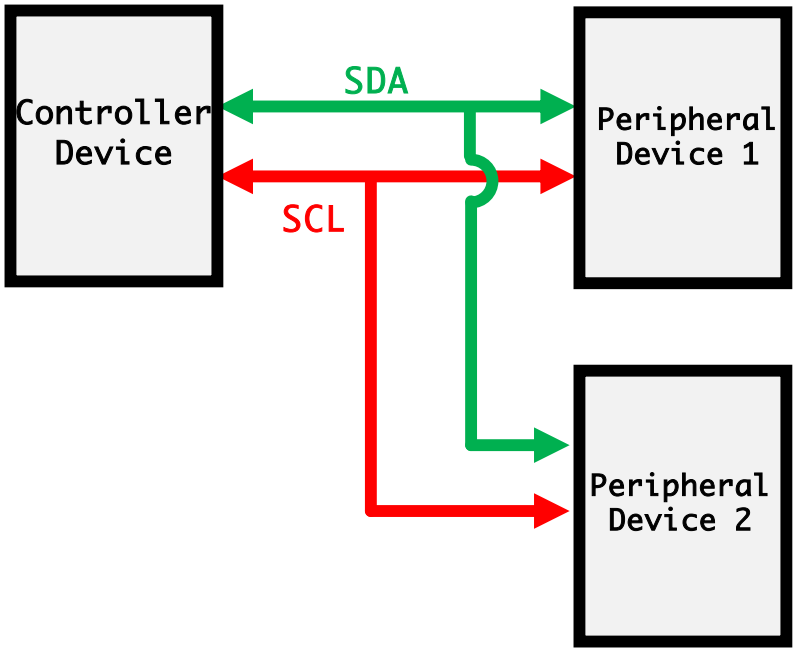
- ID's are specified at the factory*, usually several to choose from when you implement and you select them by pulling external pins HI or LOW



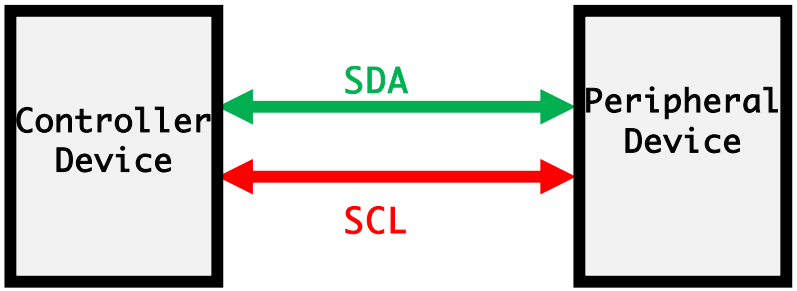
*sometimes programmable

I2C

- Only two wires…one used for synchronizing data and one used for conveying data in both directions:
  - Controller → Peripheral
  - Peripheral → Controller
- And also you need to let multiple devices possibly speak and listen…
- There's a lot here…
- It needs more complicated:
  - Hardware
  - Communication Protocols



Bi-Directional Communication

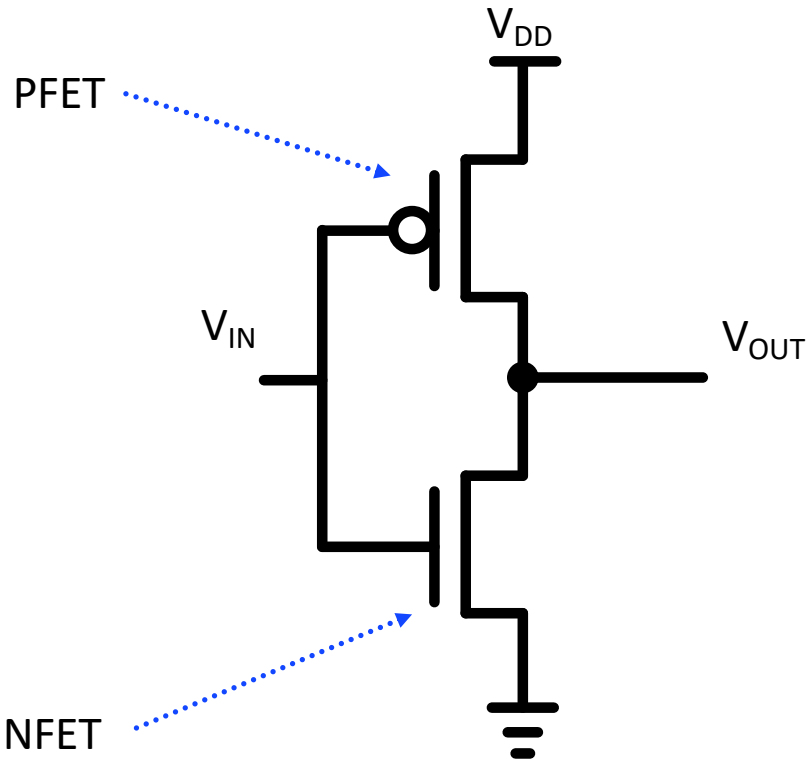


- Hey you thought you brush past those arrows going both ways...I caught that. I'm too quick for you. I go to M.I.T. How does that work?

- We should address that.

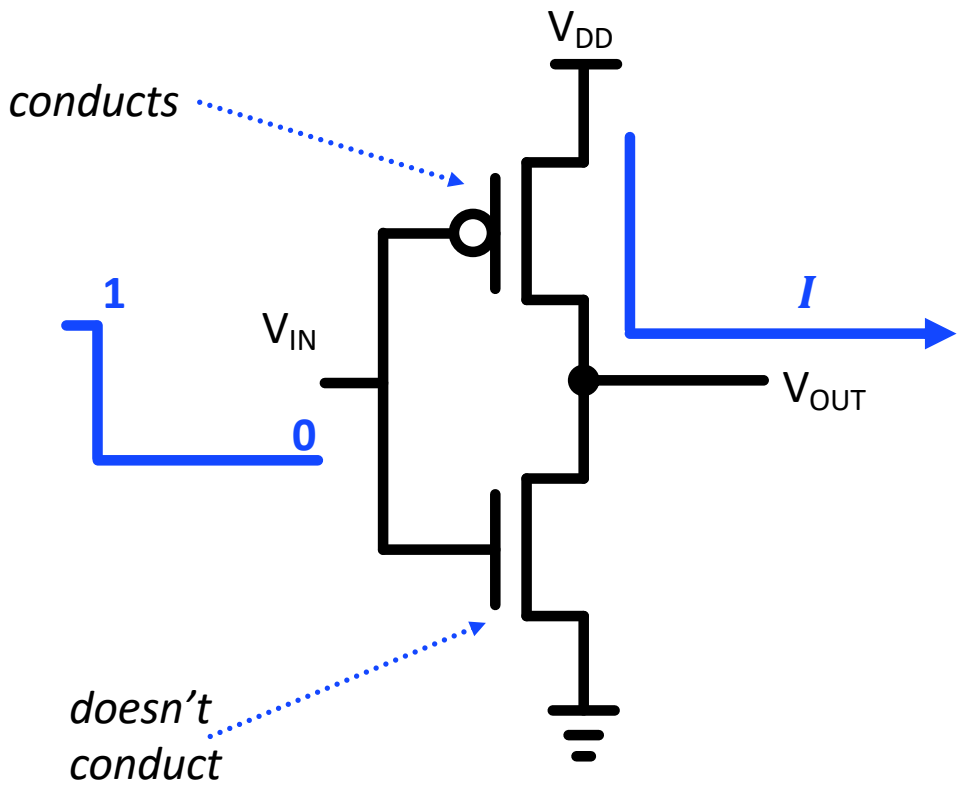
How Do Digital Electronics Set Voltages on a Line (Generally)?

- We use CMOS Logic
- A pair of complementary transistors that can alternately connect and isolate from VDD and Ground



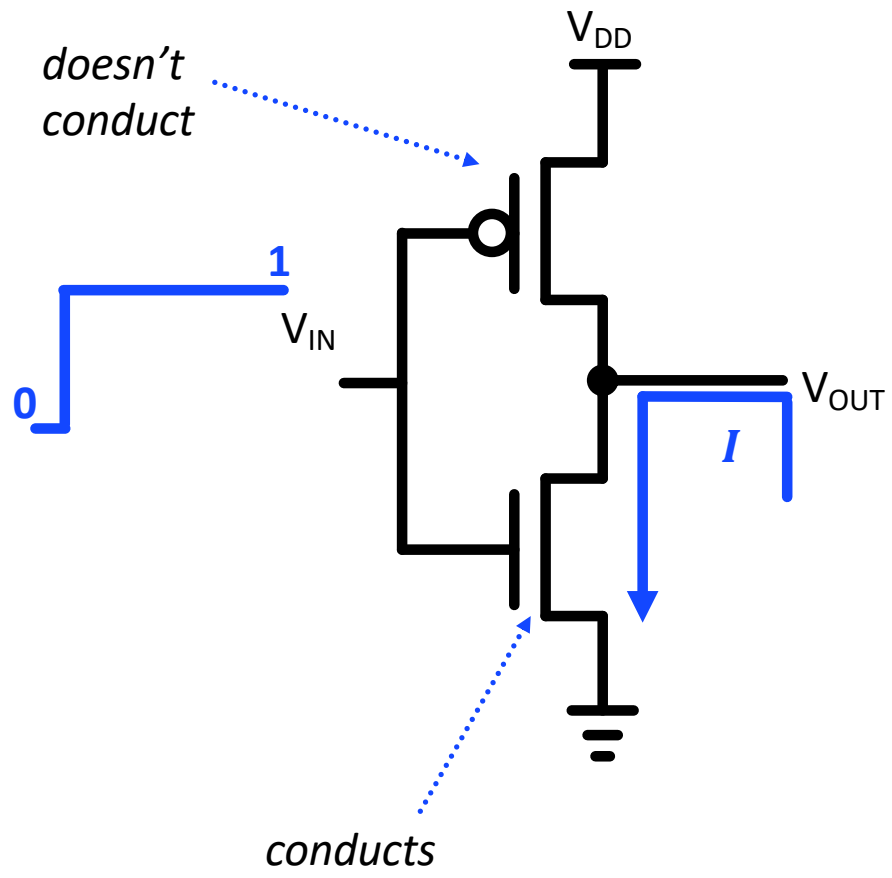
Put 1 on output?

- Low input
- PMOS conducts
- NMOS no conduct



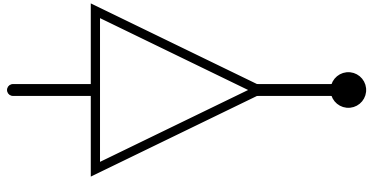
Put 0 on output?

- High input
- PMOS no conducts
- NMOS conduct



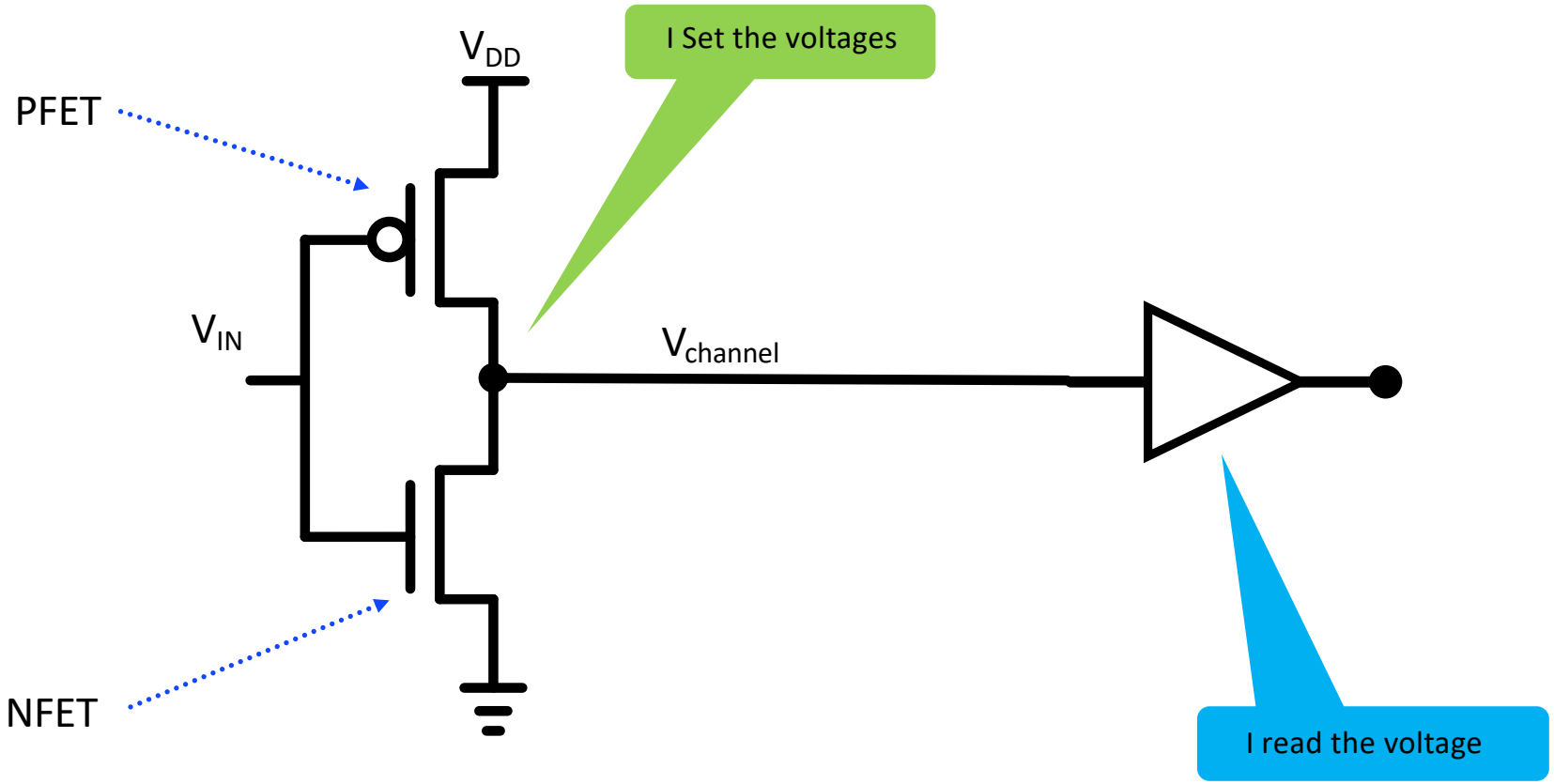
How Do Digital Electronics Listen to Voltages on a Line?

- Some sort of buffer
- Its input takes very little power/current from the line



- A quiet, ideal observer

So in Unidirectional Communication Schemes… UART, SPI wires, etc…

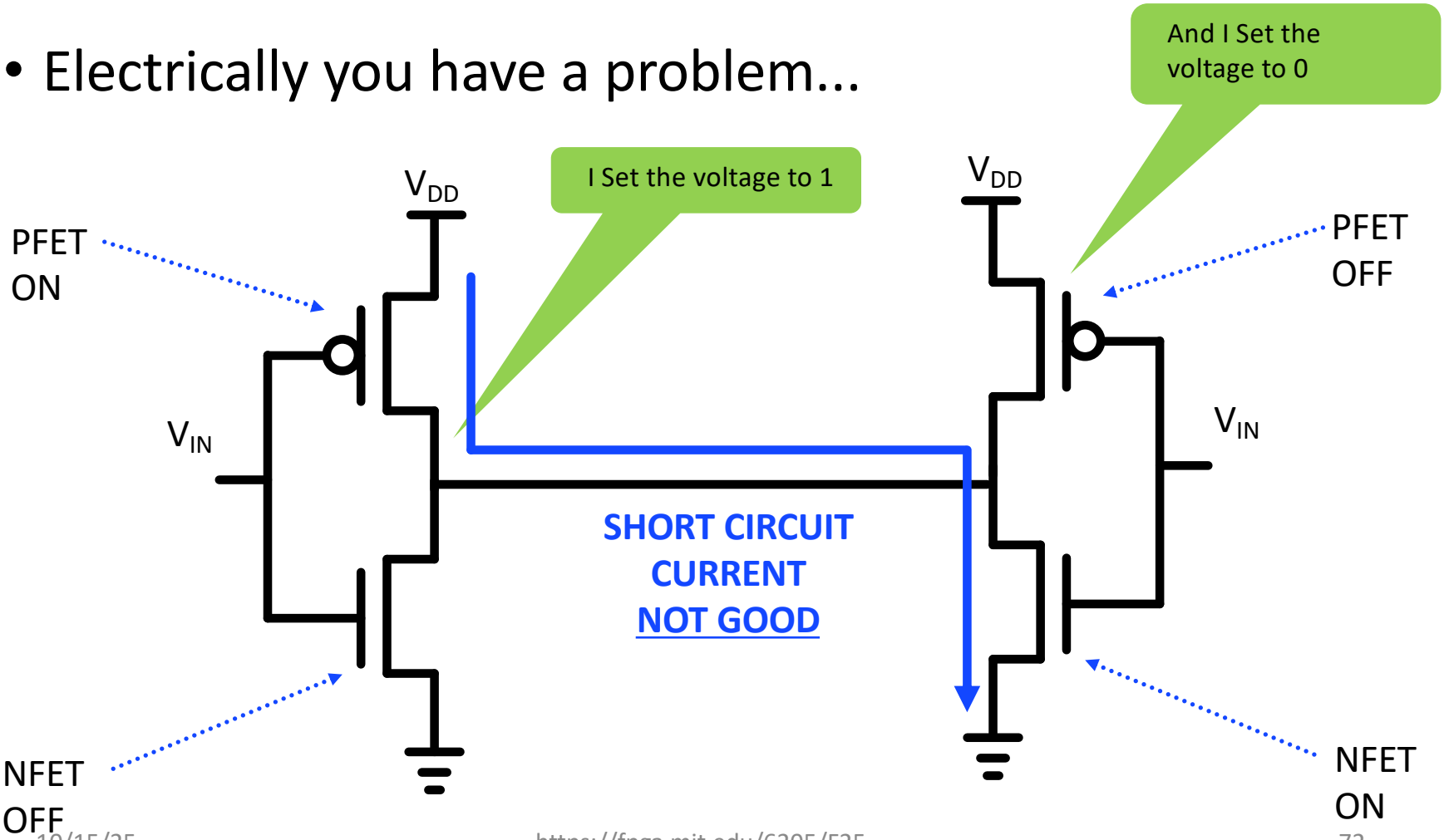


What about if two or more devices want to use one common wire to communicate?

- Sounds like socialism...can’t have that...it’ll be a mess and none of the transistors will ever want to work...
- jkjk
- But seriously electrically you have a problem...

What about if two or more devices want to use one common wire to communicate?

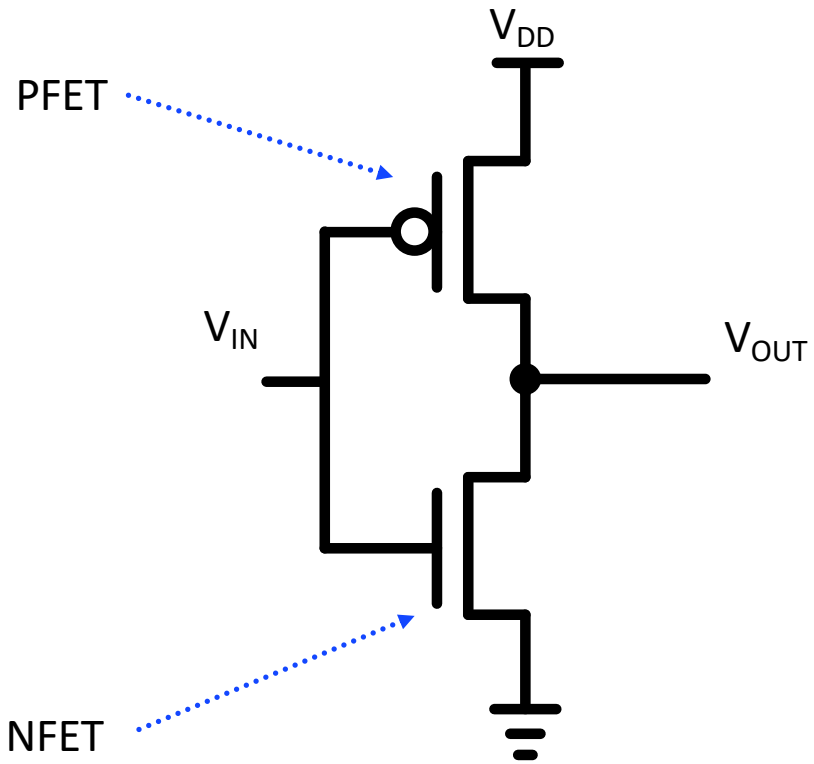
- Electrically you have a problem...



What Do You Do in Times of Conflict?

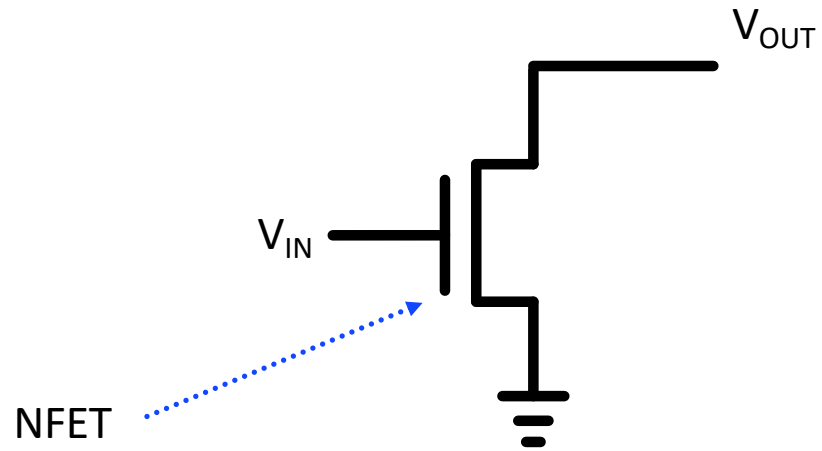
- You dig your heels in and waste tons of energy. Screw the other transistor. The correct answer is 1, not 0...everyone on reddit r/1 agrees with me and the 1News comment section backs me up.
- jkjk
- You come up with some compromises...everyone gives a little bit...everyone gets a little bit
- Each side gives up a transistor

Before We Had This...

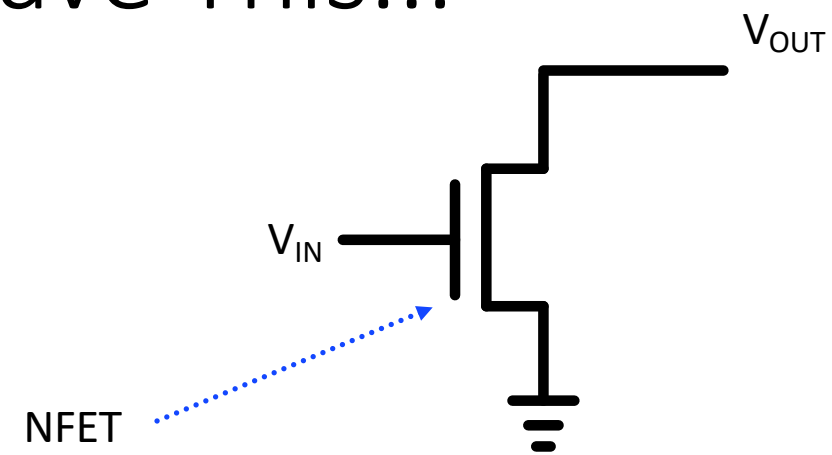


Now We Have This…

Call this "Open Drain" since the drain terminal of the transistor connects nowhere



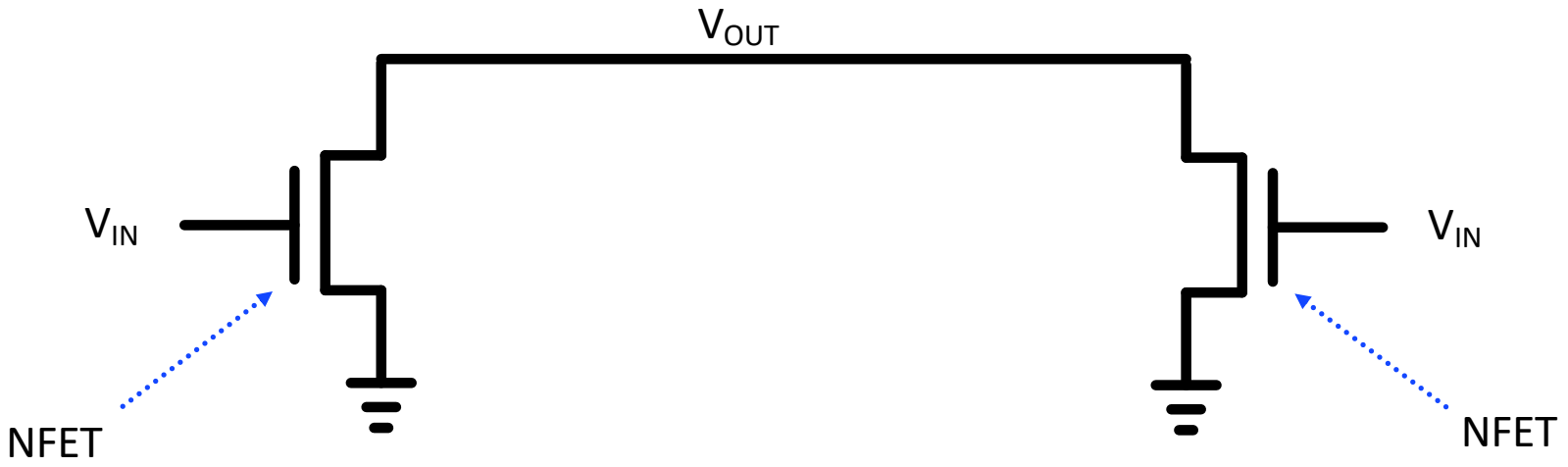
Now We Have This…



- Vout is either?????
- 0 (when transistor conducts)
- “HiZ”...basically electrically undefined (when transistor does not conduct)

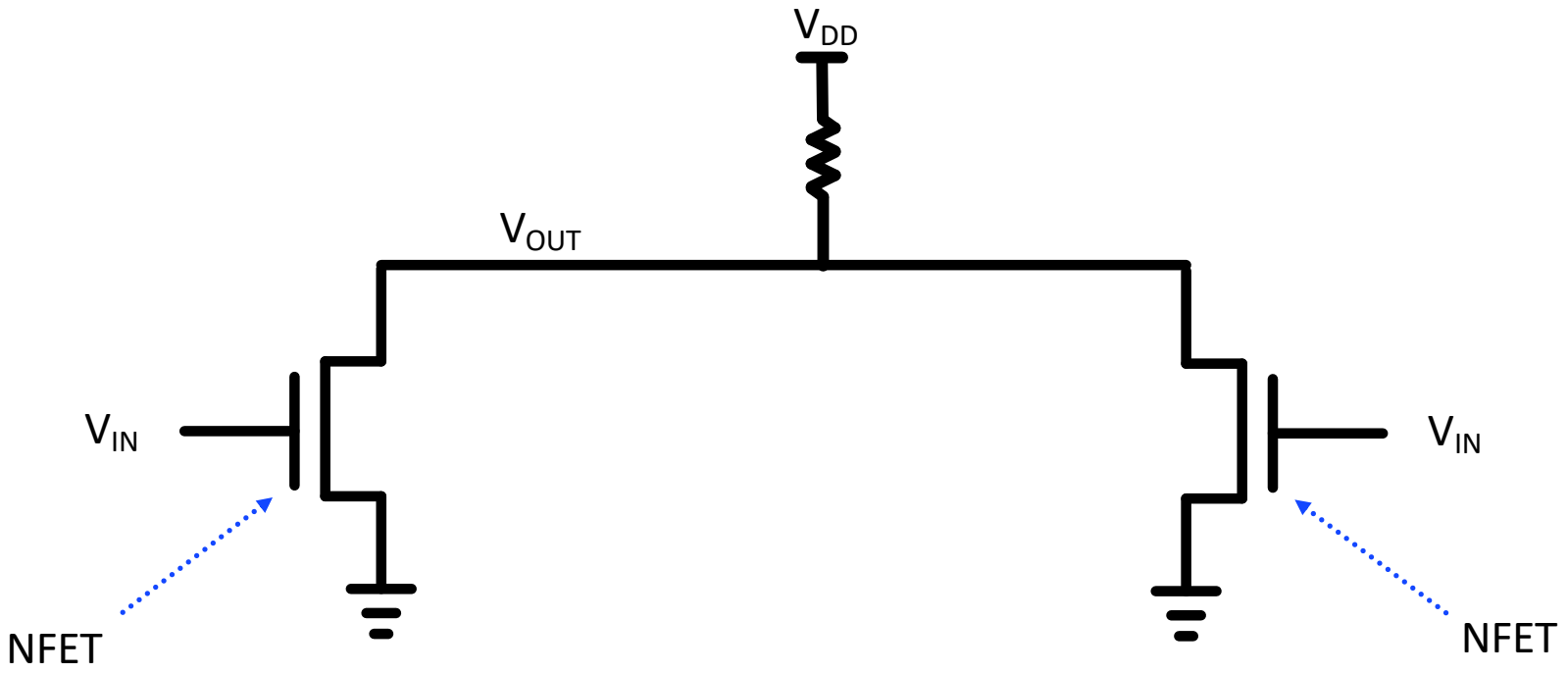
Now Connect up two of these circuits…

- Yeah…what does this give us?
- Each side can make a 0 by activating its transistor
- Can each side make a 1? ***NO***



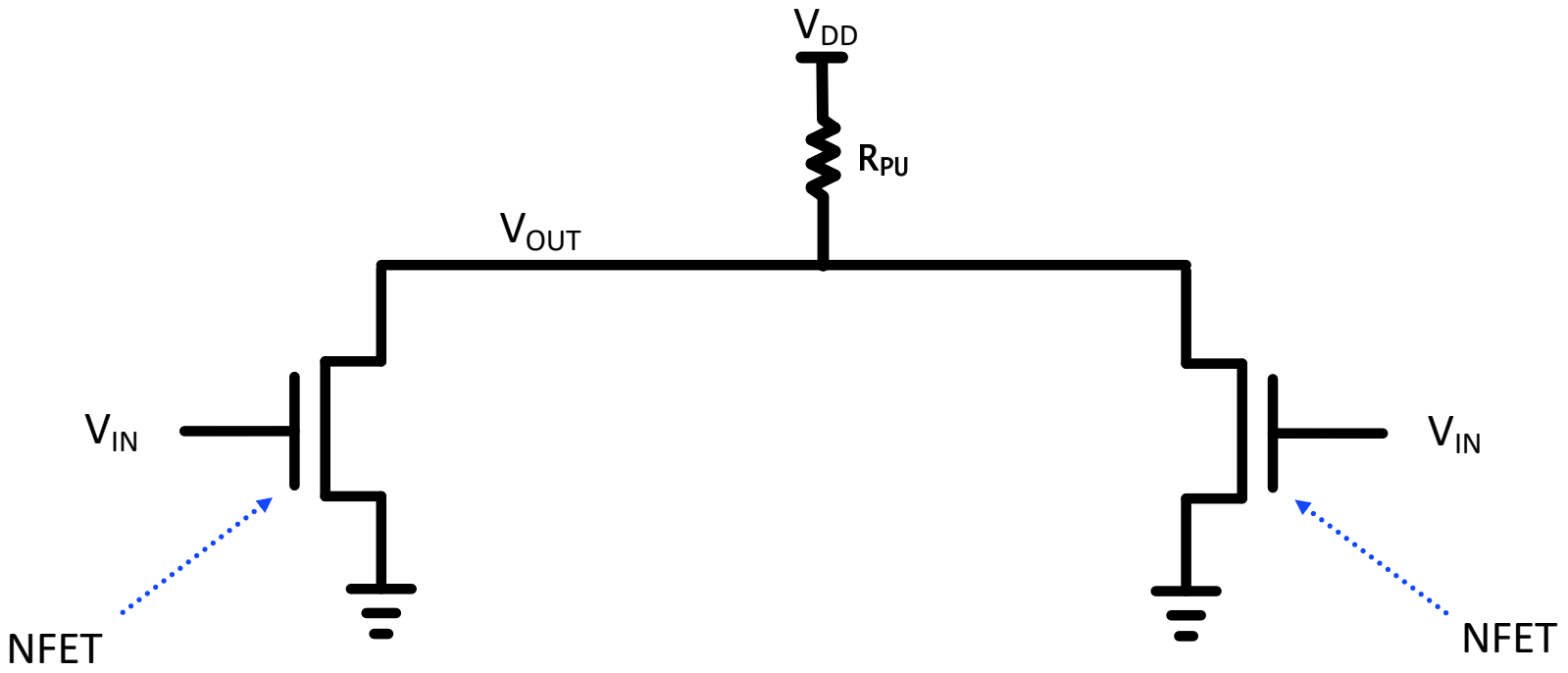
Bring in an Ombuds Component

- Each side can make a 0 by activating its transistor
- We use a neutral third-party component, trained in conflict mediation to give us 1.

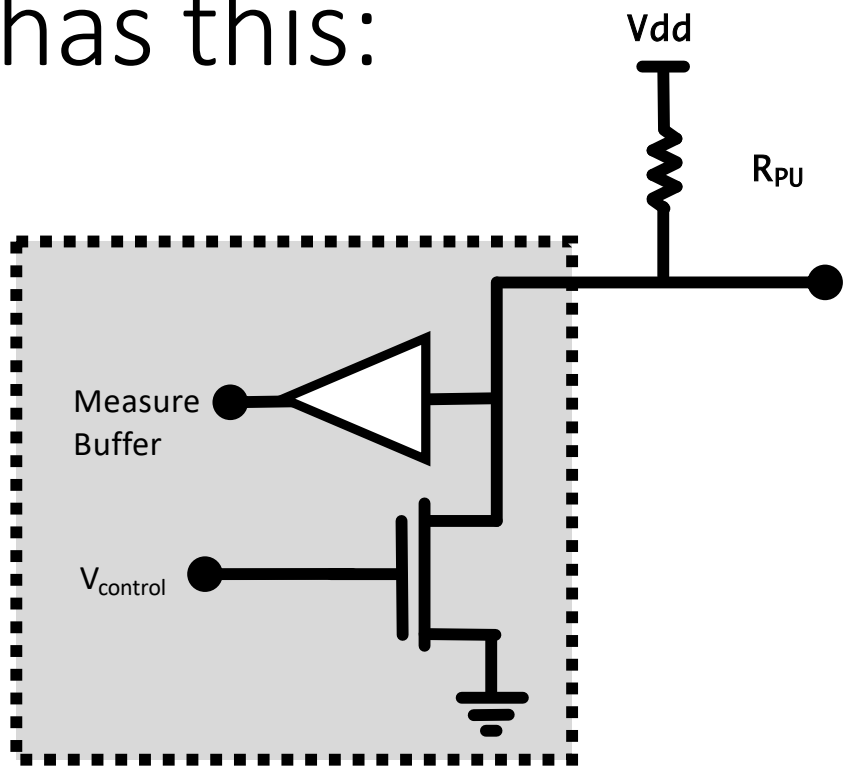


Common Pull-Up Resistor

- Prevents the Possibility of Short Circuits Always must go through this resistor (choose size to limit current)
- End up choosing several Kohms usually to keep current below 1mA

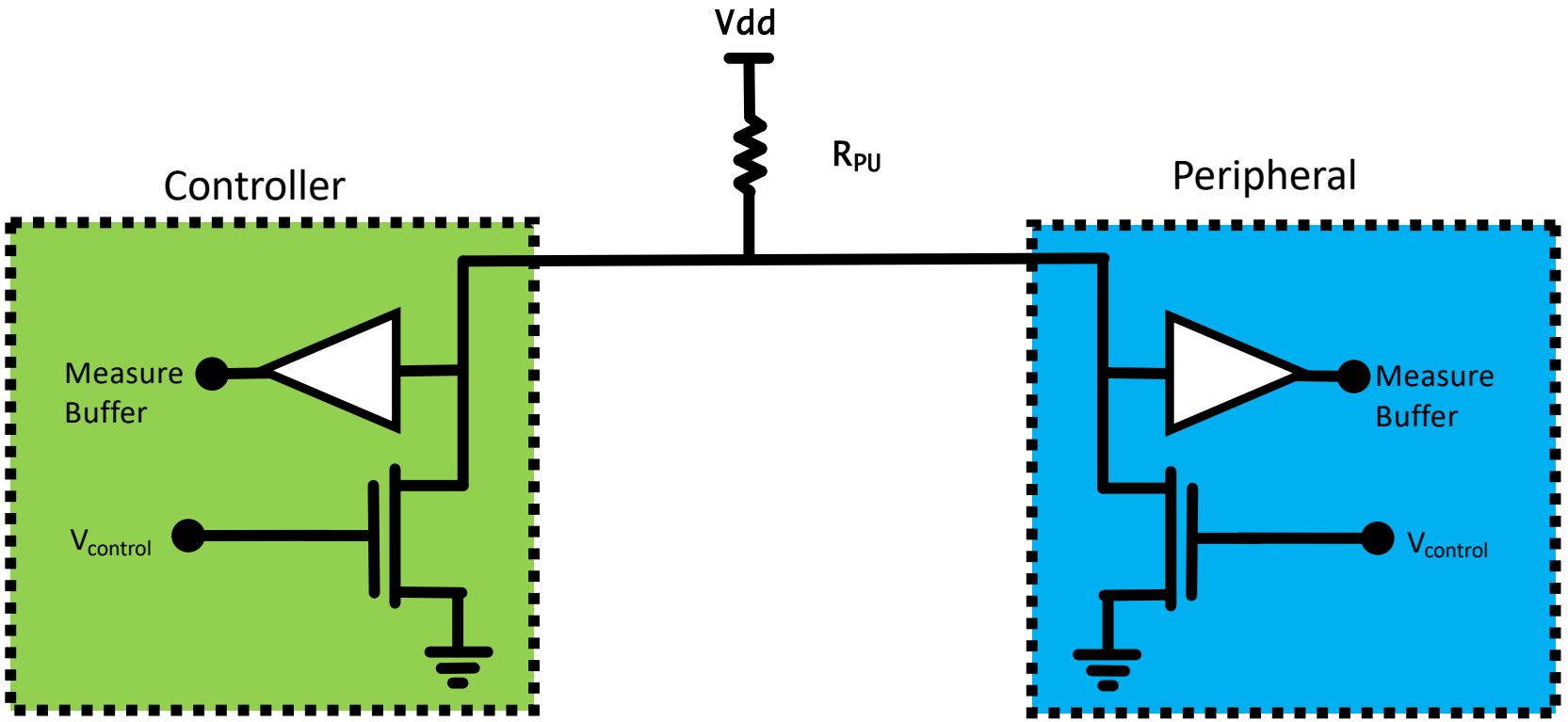


Result each side has this:



- If you want to say ”0”, you activate your transistor
- If you want to say ”1”, you inactivate your transistor and let resistor pull you up
- If you also want to listen you inactivate your transistor and monitor the line voltage

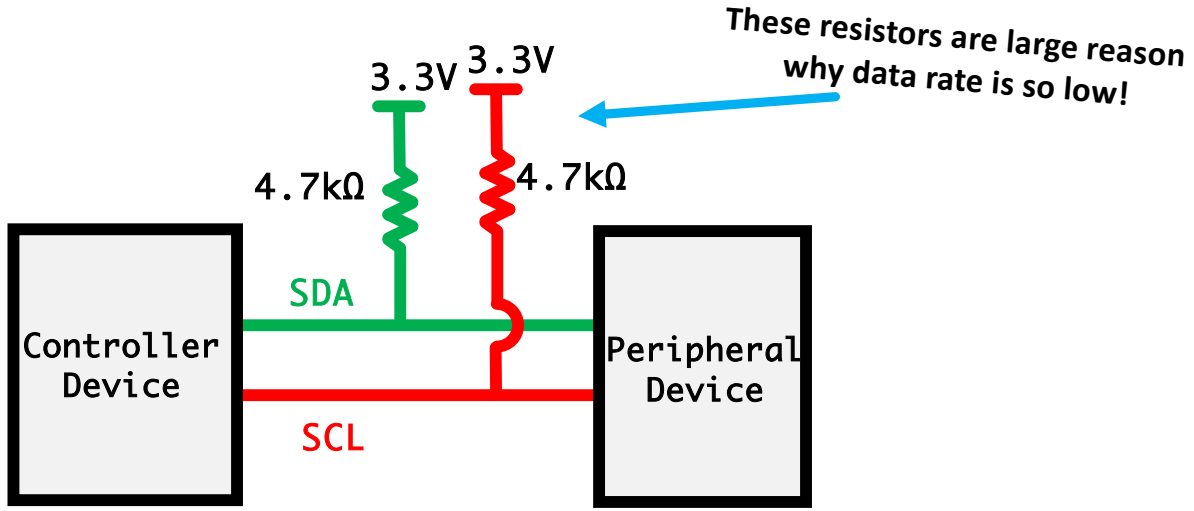
As a result:



Mode	Controller	Peripheral
Controller Transmit	HiZ (HI) or LOW	HiZ (listening)
Peripheral Transmit	HiZ (listening)	HiZ (HI) or LOW

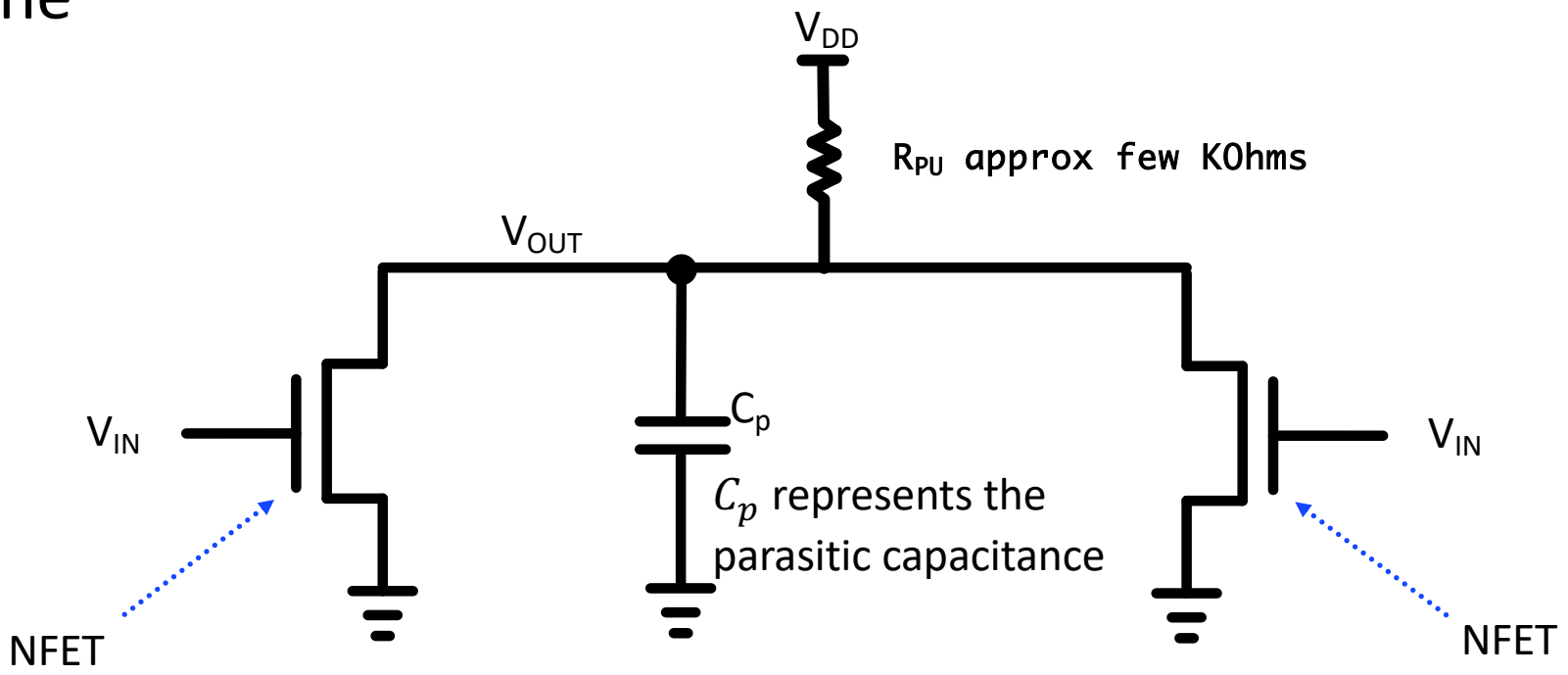
So in Deployment…

- i2C uses an open drain
- Meaning both Controller and Peripheral Device are either:
  - LOW
  - “High-Impedance”
- Need external pull-up resistors on both parts of I2C to make it work



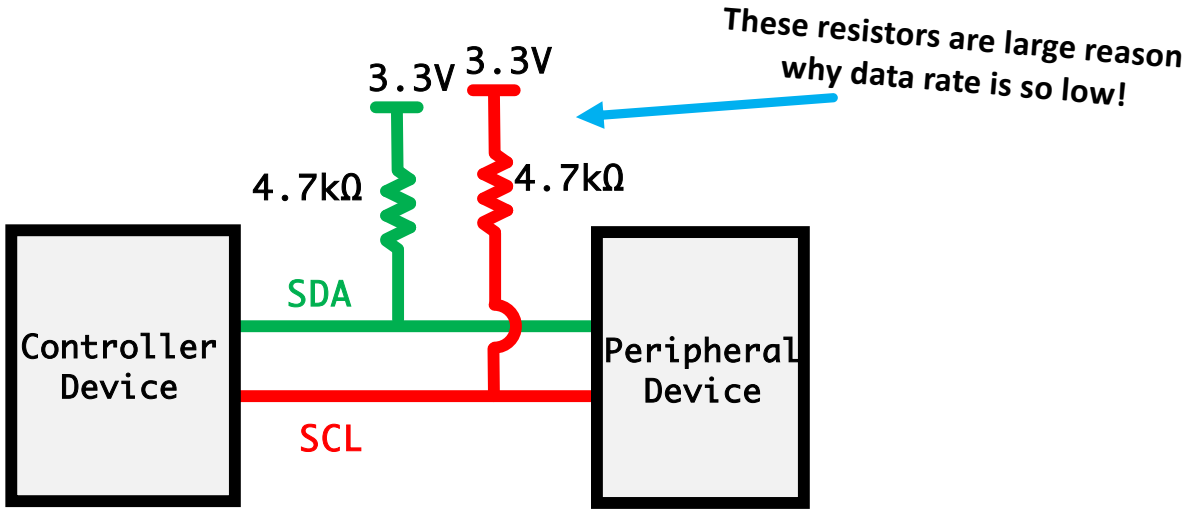
Common Pull-Up Resistor

- We choose the pull up resistors to be in the K range usually to keep current/power down.
- This has the downside that parasitic capacitances lead to relatively large time constants in charging/discharging the line



So in Deployment…

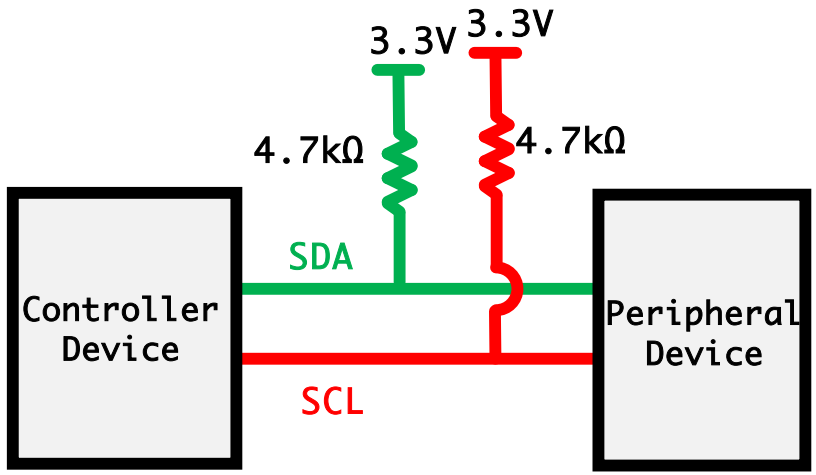
- So with all this together…we can see that there needs to be a lot more order in how to use the I2C wires…things pull double-duty depending on context.



Nobody ever seems to remember this or thinks I'm making this up.
You need to add these resistors.

So in Deployment…

- Because the data lines are “shared” it means devices need more structure when it comes to deciding what to say and whose turn it is to speak or listen.

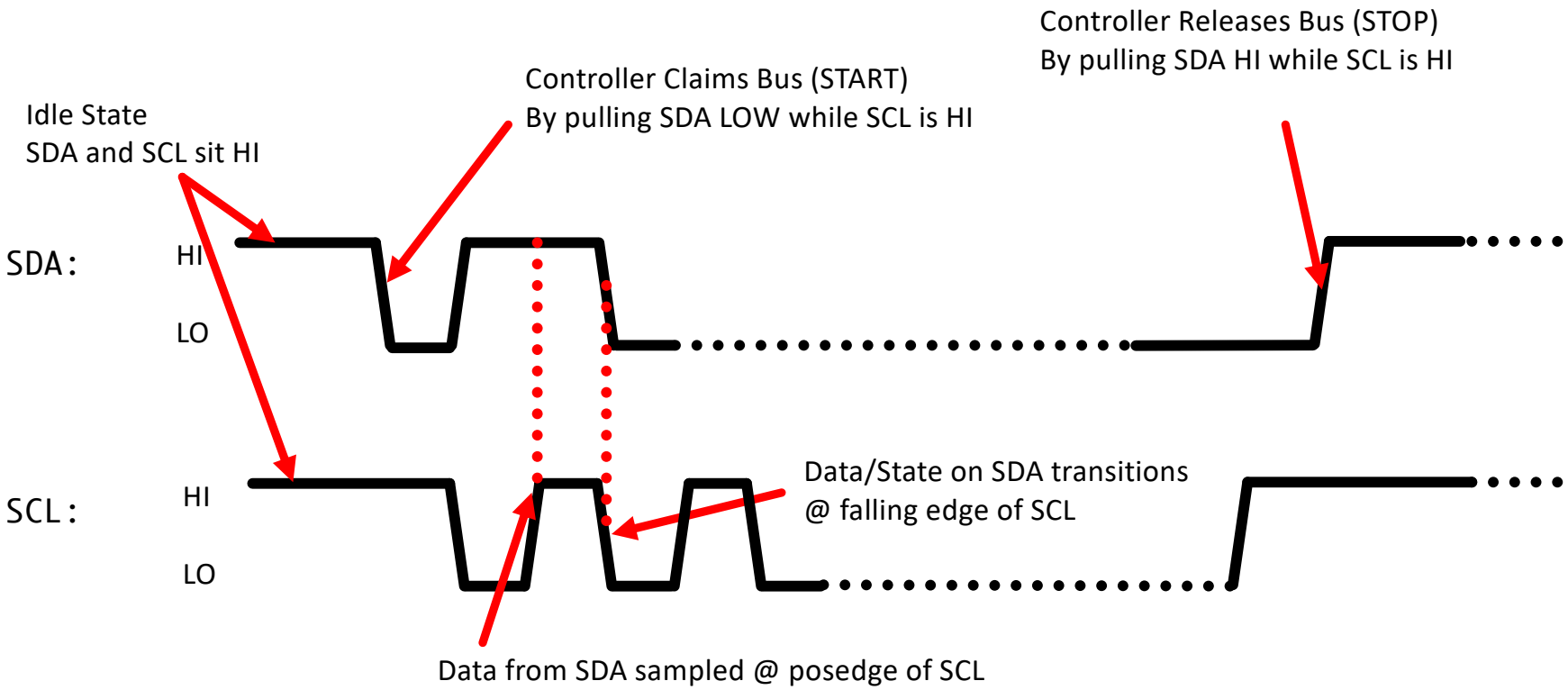


i2C Operation

- Data is conveyed on SDA (Either from Main or Secondary depending on point during communication)
- SCL is a 50% duty cycle clock
- SDA generally changes on falling edge of SCL (isn't required, but is a convenient marker for targeting transitions)
- SDA sampled at rising edge of SCL
- Main/Controller is in charge of setting SCL frequency and driving it
- Data is sent **msb** first

Notice how much more rigid this is compared to SPI

Meanings I: (Start, Stop, Sampling)



Meanings II Address

- First thing sent by Controller is 7 bit address (10 bit in more modern i2C…don’t worry about that)
- If a device on the bus possesses that address, it acknowledges (ACK=0/NACK=1) and it becomes the secondary for the time being.
- All other devices (other than Controller/Peripheral Devices) will ignore until STOP signal appears later on.

Meanings III (Read/Write Bit)

- After sending address, a Read/Write Bit is specified by Controller on SDA:
  - If Write (0) is specified, the next byte will be a register to write to, and following bytes will be information to write into that register
  - If Read (1) is specified, the Peripheral Device will start sending data out, with the Controller Device acknowledging after every byte (until it wants data to not be sent anymore)

Meanings IV (ACK/NACK)

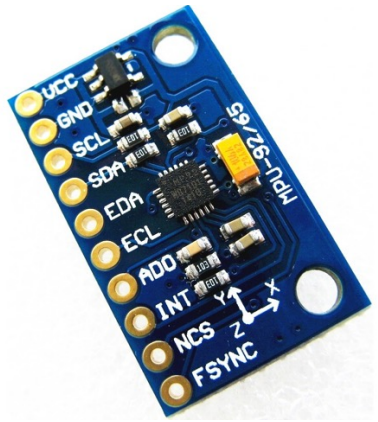
- After every 8 bits, it is the *listener's* job to acknowledge or not acknowledge the data just sent (called an ACK/NACK)
- Transmitter pulls SDA HI and listens for next reading the next time SCL transitions high:
  - If LOW, then receiver acknowledges data
  - If remains HI, no acknowledgement
- Transmitter/Receiver act accordingly

Meanings V (keeps going...)

- For Controller Device to **write** to Peripheral Device:
  - START
  - Send Device Address (with Write bit)
  - Send register you want to write to
  - Send data…until you're satisfied, doing ACK/NACKs along the way
  - STOP
- For Controller Device to **read** from Peripheral Device a common (though not universal procedure) is:
  - START
  - Send Device Address (with Write bit)
  - Send register you want to read from (*think of this like setting a cursor in the register map)*
  - ***ReSTART*** communication
  - Send Device Address (With Read bit)
  - Read the bits (*it'll start from where the cursor was left pointing at)*
  - After every 8 bits, it is Controller's job to ACK/NACK Peripheral…continued acknowledgement leads to continued data out by Peripheral.
  - Not-Acknowledge says "no more data from Peripheral"
  - STOP leads to Controller ceasing all communication

MPU-9250

Board: $5.00 from Ebay
Chip: $1.00 in bulk



- **3-axis Accelerometer (16-bit readings)**
- **3-axis Gyroscope (16-bit readings)**
- **3-axis Magnetic Hall Effect Sensor (Compass) (16 bit readings)**
- SPI or I2C communication (!)…no analog out
- On-chip Filters (programmable)
- On-chip programmable offsets
- On-chip programmable scale!
- On-chip sensor fusion possible (with quaternion output)!
- Interrupt-out (for low-power applications!)
- On-chip sensor fusion and other calculations (can do orientation math on-chip or pedometry even)
- So cheap they usually aren't even counterfeited! ☺
- **Communicates using either I2C or SPI**

I2C in Verilog...Tri-State

- inout is an “input-output”…needs some special handling...you can both write to them (**only using combinational logic**) and read from them...the usual way to work with them is the following:

In verilog…

```
inout sda;

logic sda_val;

assign sda = sda_val? 1'bz: 1'b0;

//if desired:
always_ff @(posedge clk)begin
  sda_val <= 1; //do a non-blocking assign to sda_val if desired
  //this indirectly affects sda then
end
```

Telling it to go to high-impedance (open Drain)

As a result:

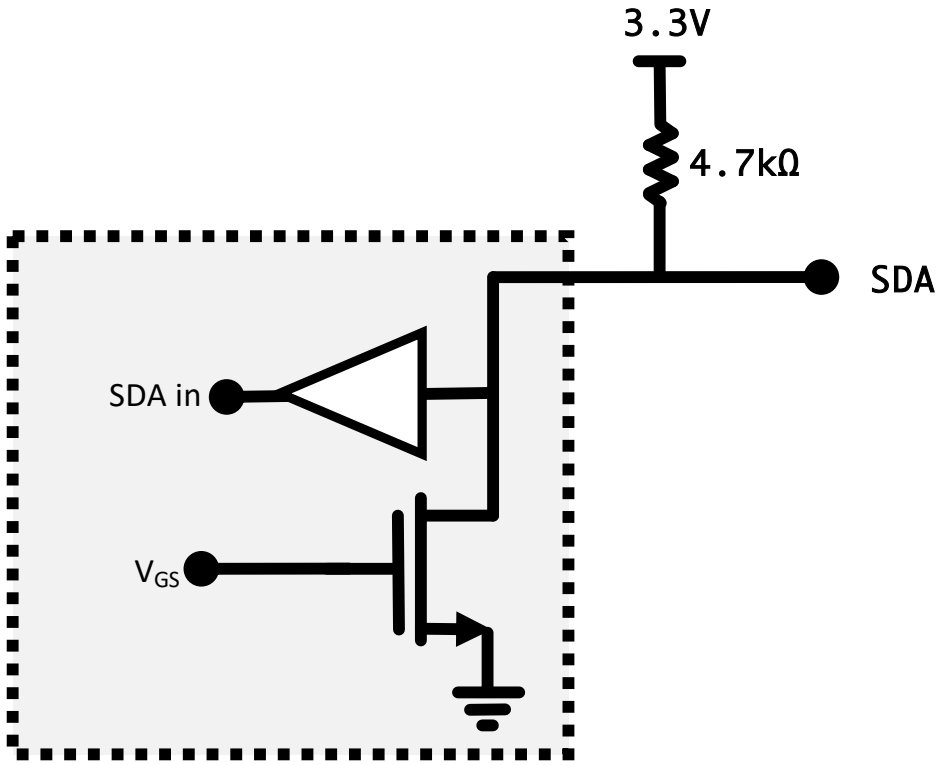
```
inout sda;
logic sda_val;
assign sda = sda_val? 1'bz: 1'b0;
```

Wanna write to SDA?

```
sda_val <= 0; //or 1 if desired
```

Wanna read to SDA?

```
sda_val <= 1;
//wait clock cycle…
some_reg <= sda; //read from input
```



Mode	Main	Secondary
Controller Transmit	HiZ (HI) or LOW	HiZ (listening)
Peripheral ACK/NACK	HiZ (listening)	HiZ (HI) or LOW
Peripheral Transmit	HiZ (listening)	HiZ (HI) or LOW
Controller ACK/NACK	HiZ (HI) or LOW	HiZ (listening)

Implementing i2C on FPGA with MPU9250:

- Made Controller i2C module in Verilog
- Used MPU9250 Data sheet: 42 pages (basic functionality, timing requirements, etc…)
- MPU9250 Register Map: 55 pages

Addr (Hex)	Addr (Dec.)	Register Name	Serial I/F
35	53	I2C_SLV4_DI	R
36	54	I2C_MST_STATUS	R
37	55	INT_PIN_CFG	R/W
38	56	INT_ENABLE	R/W
3A	58	INT_STATUS	R
3B	59	ACCEL_XOUT_H	R
3C	60	ACCEL_XOUT_L	R
3D	61	ACCEL_YOUT_H	R
3E	62	ACCEL_YOUT_L	R
3F	63	ACCEL_ZOUT_H	R
40	64	ACCEL_ZOUT_L	R
41	65	TEMP_OUT_H	R
42	66	TEMP_OUT_L	R
43	67	GYRO_XOUT_H	R
44	68	GYRO_XOUT_L	R
45	69	GYRO_YOUT_H	R
46	70	GYRO_YOUT_L	R
47	71	GYRO_ZOUT_H	R
48	72	GYRO_ZOUT_L	R

State-Machine Implementation of i2C Main/Controller

- Continuously reads 2 bytes starting at the 0x3B register (X accelerometer data)
- Print out value in hex in LEDs
- 34 States
- Clocked at 200kHz, and creates 100 kHz SCL
- Change SDA on falling edge of SCL
- Sample SDA on rising edge of SCL

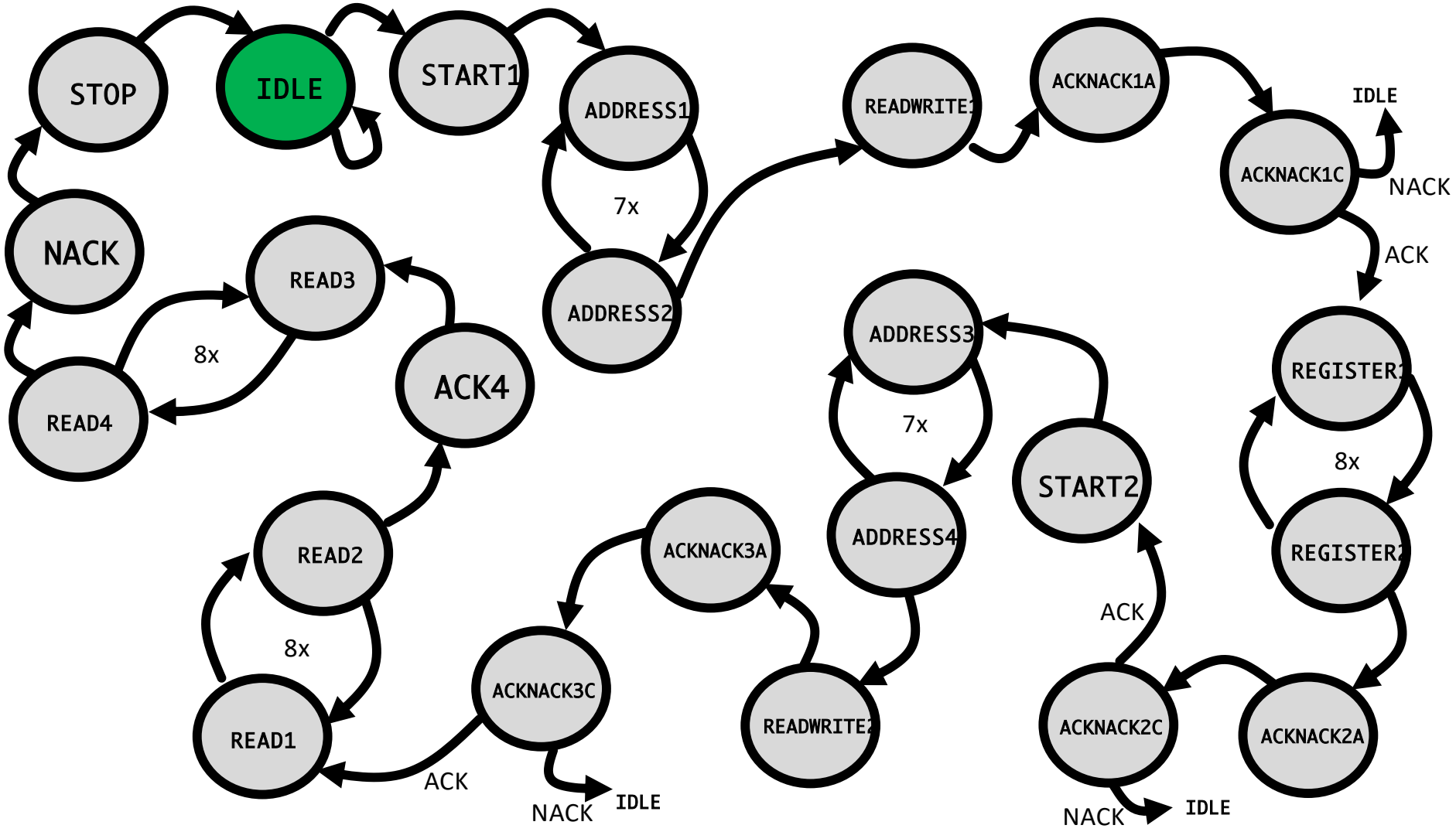
```
module i2c_master(input clock,
    input reset,
    output reg [15:0] reading,
    inout sda,
    inout scl,
    output [4:0] state_out,
    output  sys_clock);

    localparam IDLE = 6'd0; //Idle/initial state (SDA= 1, SCL=1)
    localparam START1 = 6'd1; //FPGA claims bus by pulling SDA LOW while SCL is HI
    localparam ADDRESS1A = 6'd2; //send 7 bits of device address (7'h68)
    localparam ADDRESS1B = 6'd3; //send 7 bits of device address
    localparam READWRITE1A = 6'd4; //set read/write bit (write here) (a 0)
    localparam READWRITE1B = 6'd5; //set read/write bit (write here)
    localparam ACKNACK1A = 6'd6; //pull SDA HI while SCL ->LOW
    localparam ACKNACK1B = 6'd7; //pull SCL back HI
    localparam ACKNACK1C = 6'd8; //Is SDA LOW (slave Acknowledge)? if so, move on, else go back to IDLE
    localparam REGISTER1A = 6'd9; //write MPU9250 register we want to read from (8'h3b)
    localparam REGISTER1B = 6'd10; //write MPU9250 register we want to read from
    localparam ACKNACK2A = 6'd11; //pull SDA HI while SCL -> LOW
    localparam ACKNACK2B = 6'd12; //pull SCL back HI
    localparam ACKNACK2C = 6'd13; //Is SDA LOW (slave Ack?) If so move one, else go to idle
    localparam START2A = 6'd14; //SCL -> HI
    localparam START2B = 6'd15; //SDA -> HI
    localparam START2C = 6'd16; //SDA -> LOW (restarts)
    localparam ADDRESS2A = 6'd17; //Address again (7'h68)
    localparam ADDRESS2B = 6'd18; //Address again
    localparam READWRITE2A = 6'd19; //readwrite bit...this time read (1)
    localparam READWRITE2B = 6'd20; //readwrite bit...this time read (1)
    localparam ACKNACK3A = 6'd21; //like other acknacks...wait for MPU to respond
    localparam ACKNACK3B = 6'd22; //else go back to IDLE
    localparam ACKNACK3C = 6'd23; //""""""
    localparam READ1A = 6'd24; //start reading in data from device
    localparam READ1B = 6'd25; //this data is 8MSB of x accelerometer reading
    localparam ACKNACK4A = 6'd26; //Master (FPGA) assets acknowledgement to Slave
    localparam ACKNACK4B = 6'd27; //Effectively asking for more data
    localparam READ2A = 6'd28; //start reading next 8 bits (8LSB)
    localparam READ2B = 6'd29; //assign to lower half of 16 bit register
    localparam NACK = 6'd30; //Fail to acknowledge Slave this time (way to say "I'm done so slave doesn't
    localparam STOP1A = 6'd31; //Stop/Release line
    localparam STOP1B = 6'd32; //FPGA master does this by pulling SCL HI while SDA LOW
    localparam STOP1C = 6'd33; //Then pulling SDA HI while SCL remains H
```

State-Machine Implementation of i2C Main/Controller

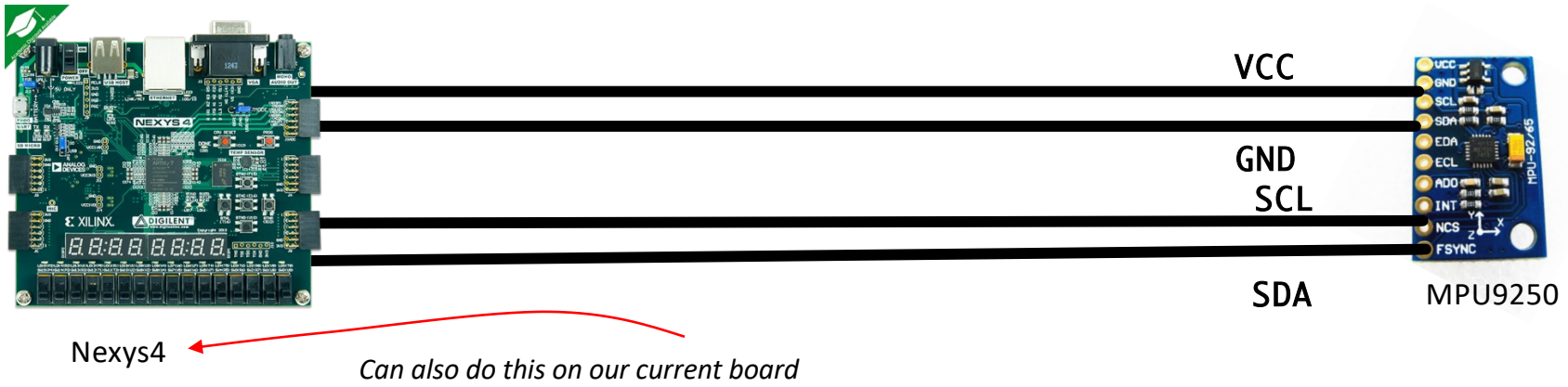
- Redundant states (repeated READ/WRITE, ADDRESS, ACK/NACK, etc…)
- ARM manual describes ~20 state FSM for **full I2C…this is just a toy implementation of specific I2C operation**
- Included code on site for reference/starting point
- Diagram: on next page for reference

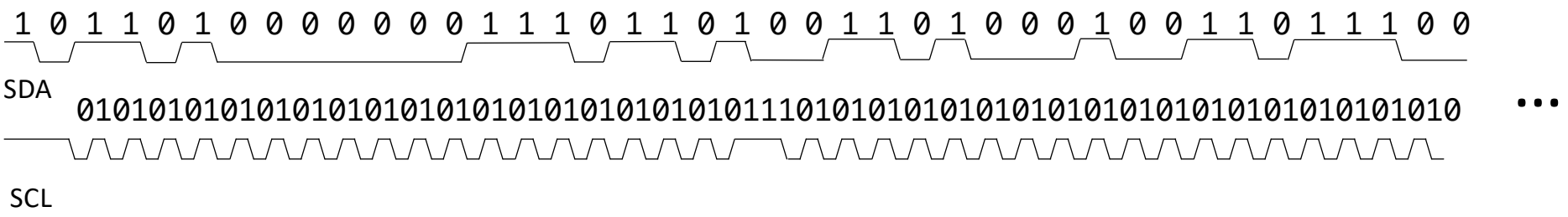
```
always @(posedge clock_for_sys)begin //update only on ri
    if (reset &&(state !=IDLE))begin
        state <= IDLE;
        count <=0;
    end else begin
        case (state)
            IDLE: begin
                if (reset) state <= IDLE;
                else if (count == 60)begin
                    state <= START1;
                    count <=0;
                end
                count <= count +1;
                sda_val <=1;
                scl_val <=1;

            end
            START1: begin
                sda_val <= 0; //pull SDA low
                scl_val <=1;
                state <=ADDRESS1A;
                count <= 6;
            end
            ADDRESS1A: begin
                scl_val<=0;
                sda_val <= device_address[count];
                state <= ADDRESS1B;
            end
            ADDRESS1B: begin
                scl_val <=1;
                if (count >= 1) begin
                    count <= count -1;
                    state <= ADDRESS1A;
                end else begin
                    state <= READWRITE1A;
                end
            end
            READWRITE1A: begin
                scl_val <=0;
                sda_val <=0;//write address
                state <= READWRITE1B;
```

…200 more lines

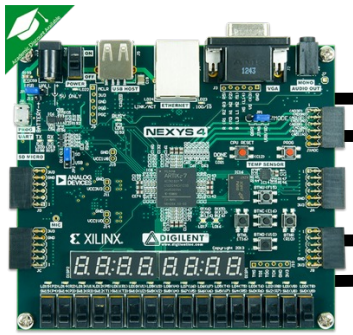

STOP
IDLE
START1
ADDRESS1
READWRITE1
ACKNACK1A
IDLE
ACKNACK1C
NACK
ACK
7x
NACK
READ3
ADDRESS2
8x
READ4
ACK4
ADDRESS3
REGISTER1
7x
8x
START2
ADDRESS4
REGISTER2
READ2
ACKNACK3A
ACK
8x
ACKNACK3C
READ1
READWRITE2
ACKNACK2C
ACKNACK2A
ACK
NACK
IDLE
NACK
IDLE

Communication Part





Communication Part



VCC

GND

SCL

Nexys4

Can also do

Needs a dialogue

1 0 1 1 0 1 0 0 0 0 0 0 0 1

SDA

0101010101010101010101010

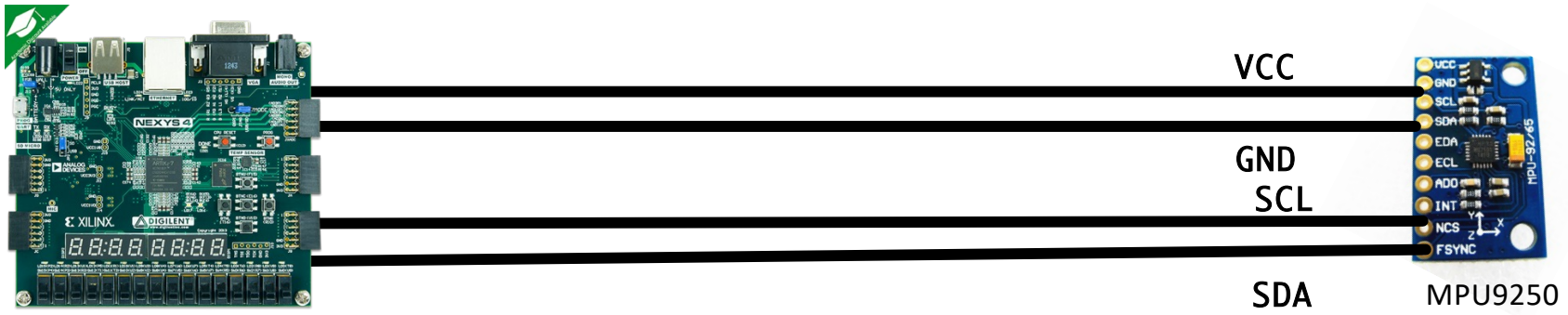
SCL

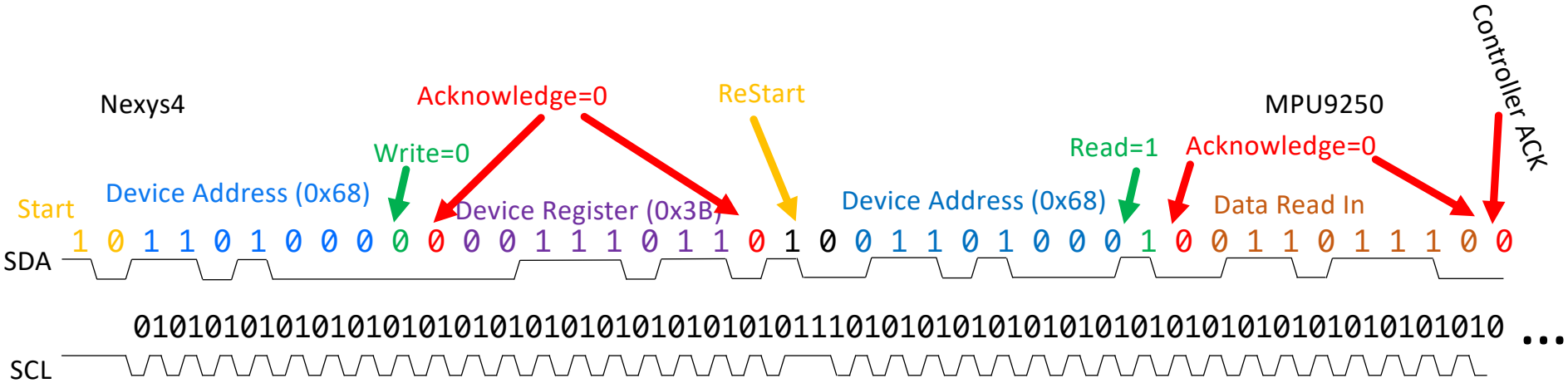
ACT I

A lab late at night, right before a final project is due.

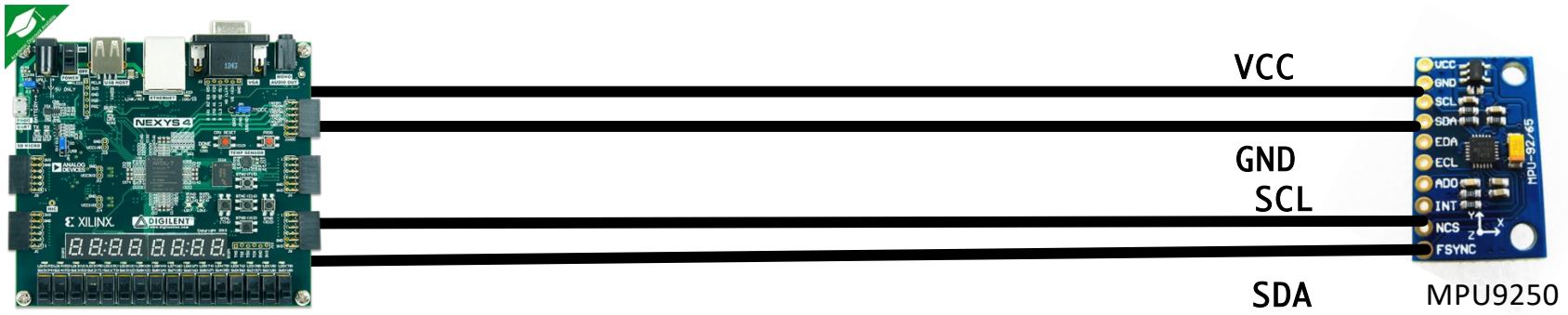
FPGA: 10. 1101000. 0.
MPU9250: 0.
FPGA: 0111011.
MPU9250: 0.
FPGA: 1.
(MPU9250 looks taken aback by the sudden change in conversation. FPGA appears not to notice.)
FPGA: 10. 1101000.....1.
MPU9250: 0.
(MPU9250's concern transforms into a knowing smile).
MPU9250: 01101110.
FPGA: 0.
(The two talk deep into the night as the curtain falls)
End of Act I

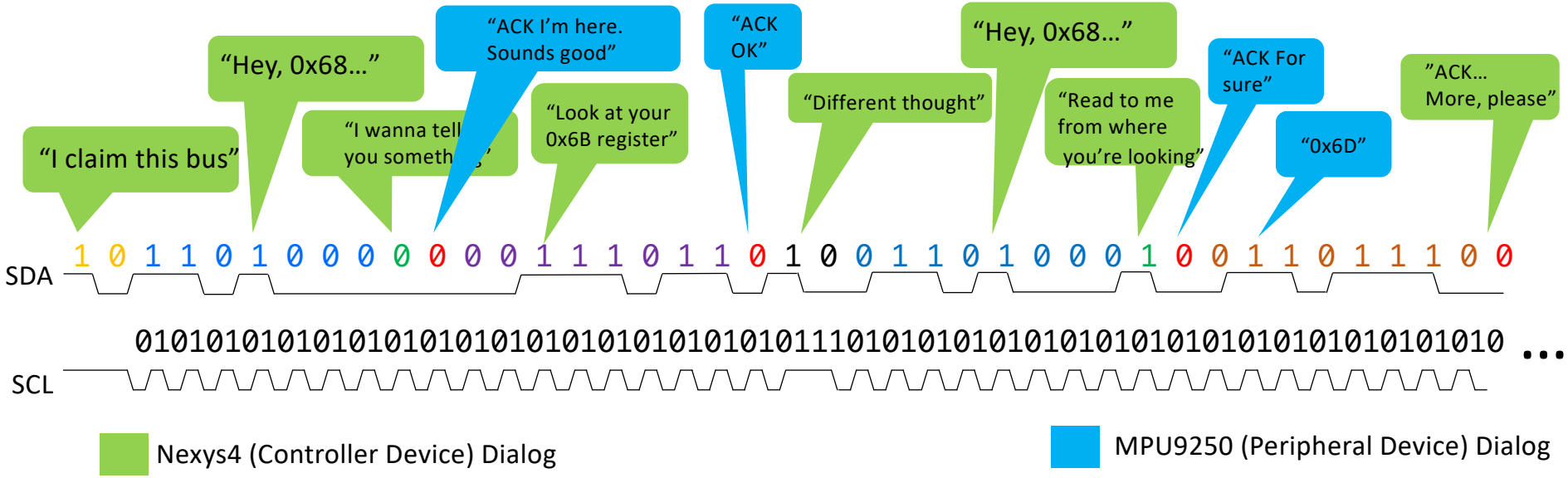
Communication Part





Communication Part

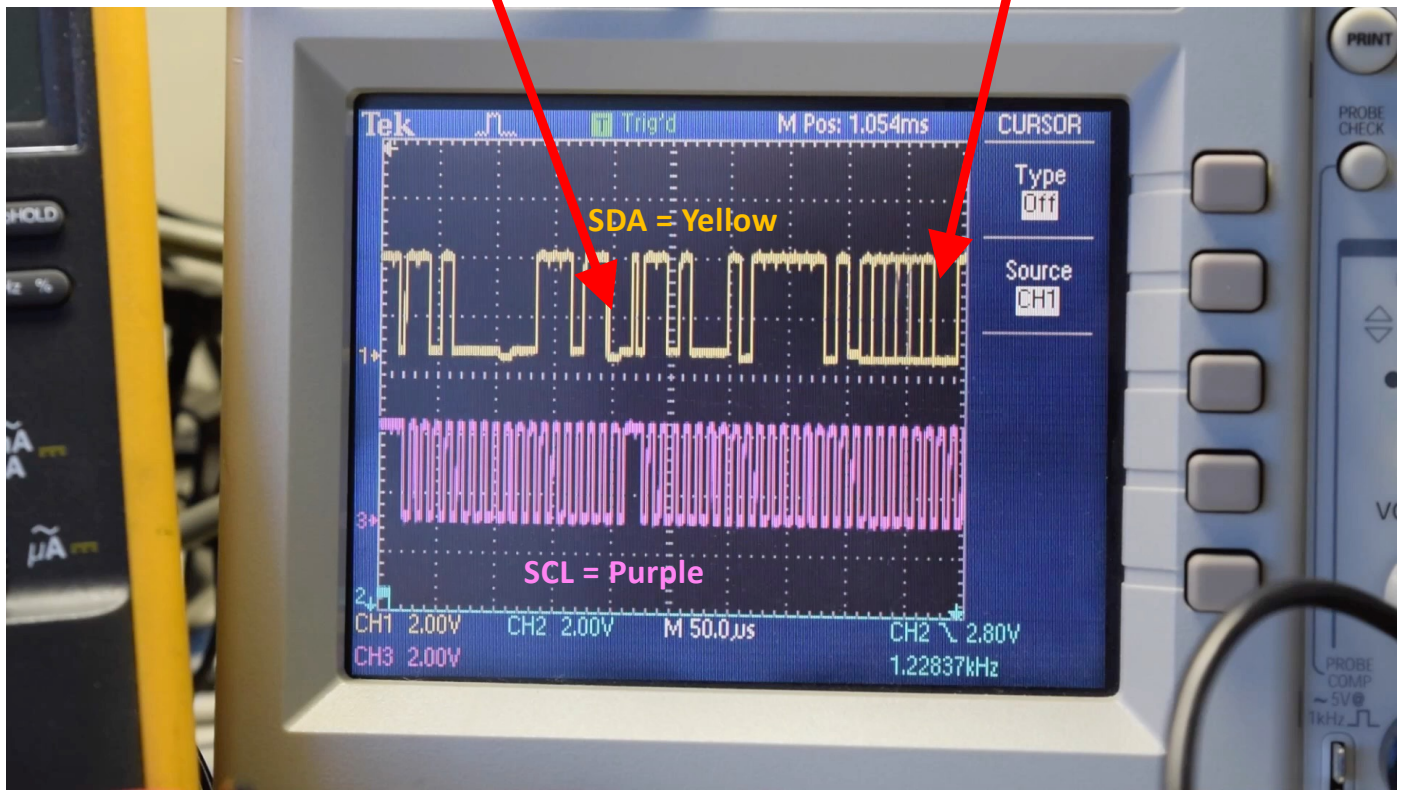




Communication in Real-Life:

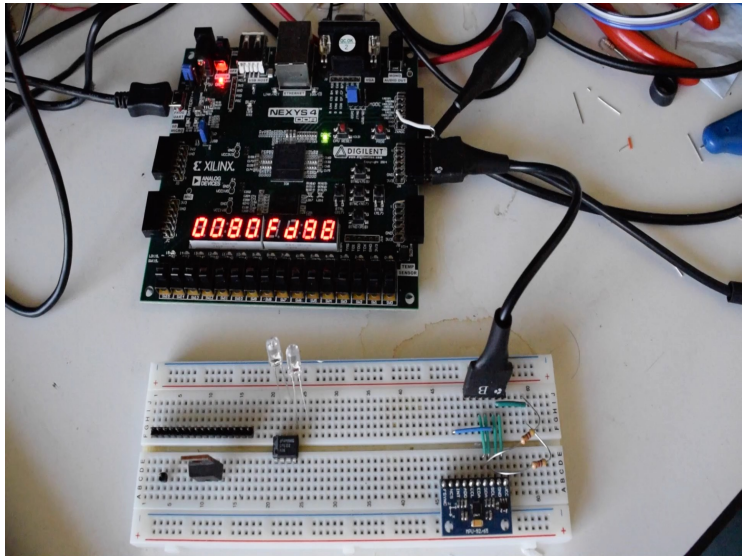
Data being sent **to** MPU9250

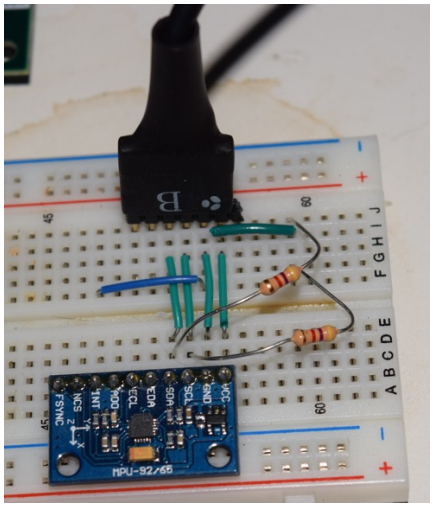
Data being sent **from** MPU9250



Triggered on leaving IDLE state

Running and reading X acceleration:





HOOKUP

Horizontal:

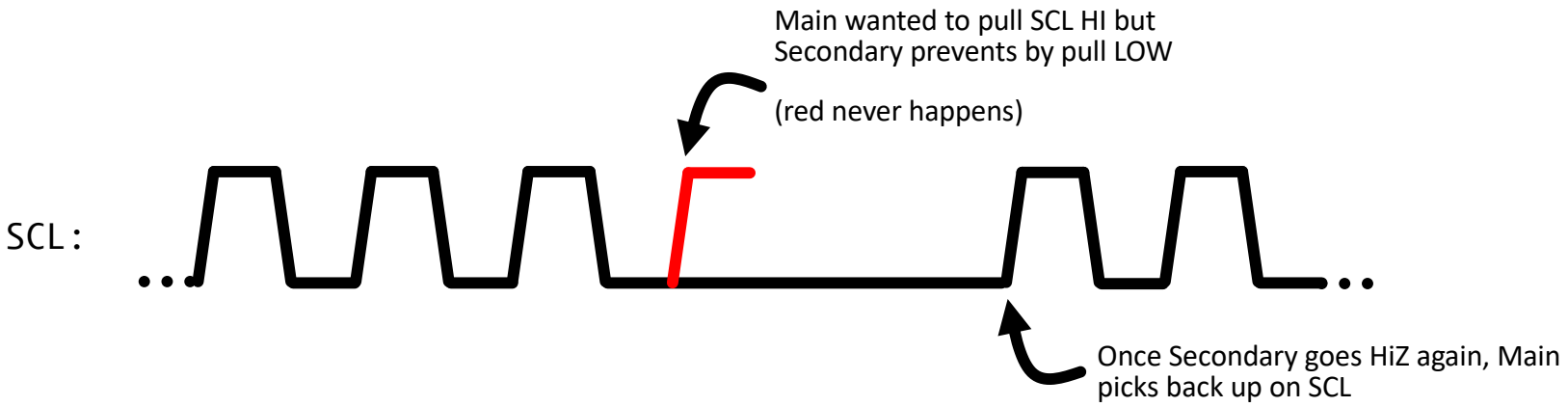
16'hFD88 = 16'b1111_1101_1000_1000 (2's complement)
Flip bits to get magnitude: 16'b0000_0010_0111_0111
=-315
Full-scale (default +/- 2g)
-315/(2**15)*2g = -0.02g ☺ makes sense

Vertical:

16'h4088 = 16'b0100_0000_1000_1000 (2's complement)
Leave bits to get magnitude: 16'b0100_0000_1000_1000
=+16520
Full-scale (default +/- 2g)
-16520/(2**15)*2 = +1.01g ☺ makes sense!

Clock-Stretching (Cool part of i2C!!!)

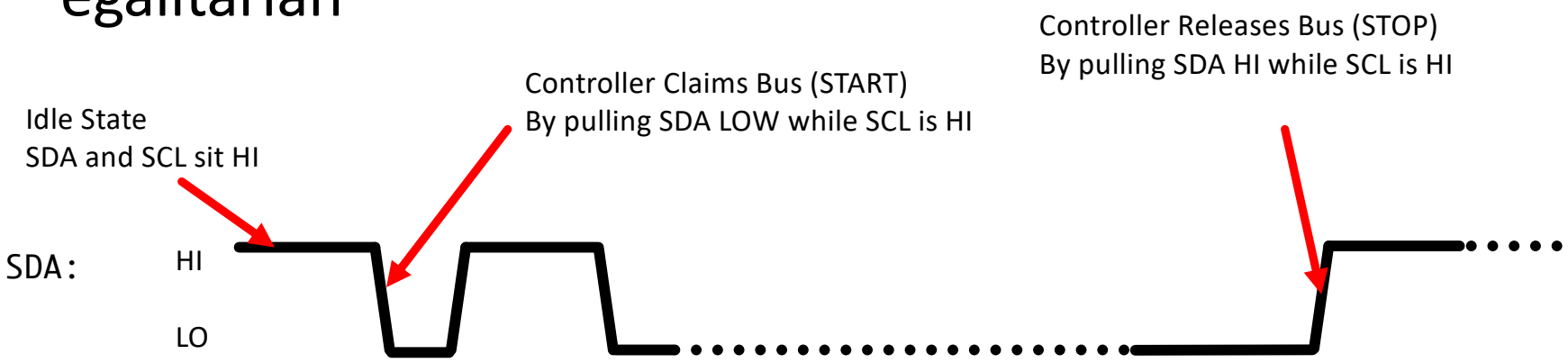
- Normally Controller drives SCL, but since Controller drives SCL high by going hiZ, it leaves the option open for Peripheral to step in and prevent SCL from going high by pulling SCL LOW



  - Allows Peripheral a way to buy time/slow down things (if it requires multiple clock cycles to process incoming data and/or generate output)

I2C Can Also Be a "Multi-Controller" Bus

- In SPI, there is a pre-determined device in charge of the system. I2C is potentially much more egalitarian



- Devices can be design to yield based on who claims a bus first…but you have to be careful…what if two devices claim a bus at the same time…potential problems? Can get bus contention so need to be careful

Goods and Bads of I2C?

- Lower clock frequency
- Far more complicated communication protocol

- Only need two wires (very scalable)...this is a bigger deal in modern systems than you may think at first. Wiring and pinout is by far one of the most expensive parts of making a chip. Compute is relatively cheap at this point so the “complexity” of I2C isn’t necessarily that much of a downside.

Have we used I2C so far in 6.205?

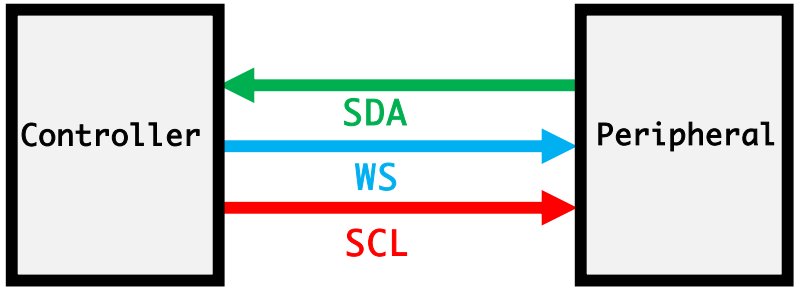
- Yes...the camera actually gets spoken to over I2C first before transmitting data. That is how it gets configured.

- HDMI actually uses I2C to do its original handshakes and monitor resolution communication (and copyright stuff) before the actual data starts getting sent over TMDS/high rate

Common Chip-to-Chip Communication Protocols (not exhaustive)

- **Parallel** (not so much anymore)…mostly memory and things that need to send data at **very** high rates such as a camera, high-speed ADCs, etc…
- **UART "Serial"** (still common in random devices, reliable and easy to implement)
- **SPI (Serial Peripheral Interface)** very common
- **I2C (Inter-Integrated Circuit Communication)** very common
- **I2S (Inter-Integrated Circuit Sound Bus**) very common in audio-specific applications

I2S (Inter-IC Sound Bus)

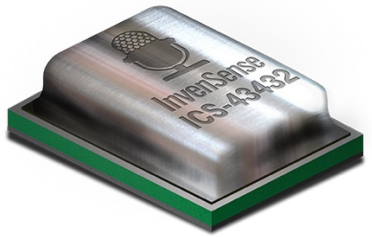


- *Not related to i2C at all*
- Intended for Digitized Stereo Data
- Three Wires:
  - SDA: Serial Data (The actual music)
  - WS: Word Select (Left/Right Channel)
  - SCL: Serial Clock (For Synchronization)
- Push-Pull Driving (like SPI…no need for pull-up resistors)
- Data sent msb first
- Clock-rate dictated by sample rate (44.1kHz @16 bits per channel /w 2 channels = ~1.4 MHz for example

Actually almost like a very constrained form of SPI in many senses

I2S

Two identical microphones share all three lines



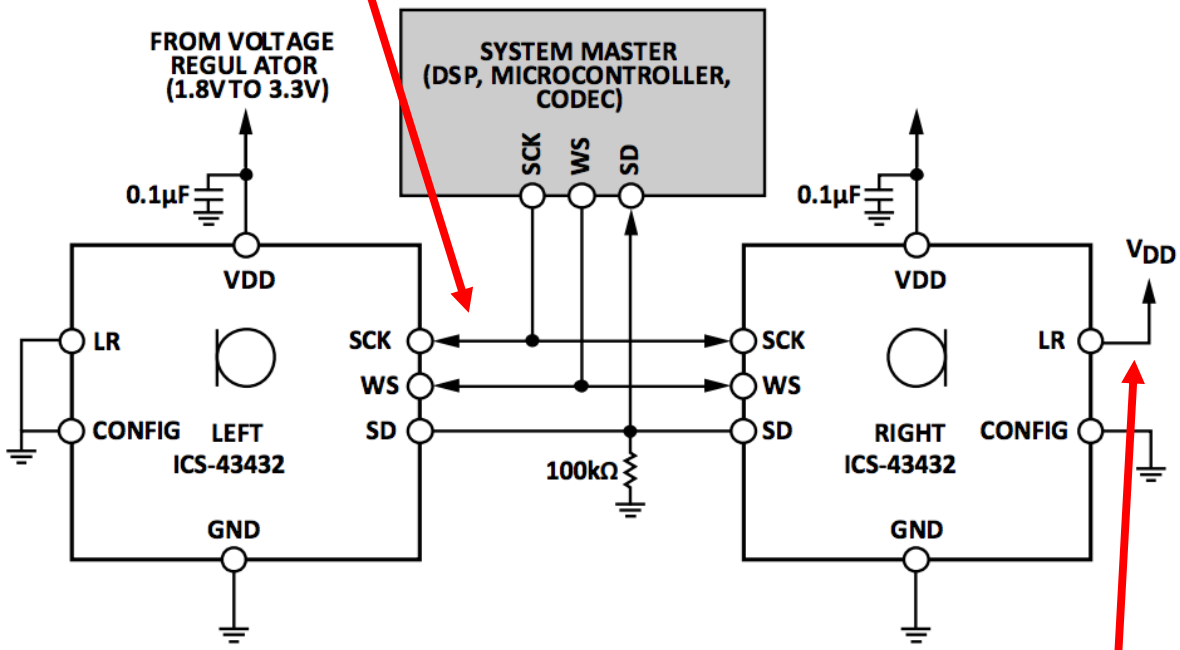
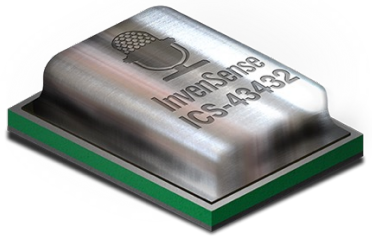


Figure 10. System Block Diagram

Microphone told to be the “left” microphone by hardwiring LR to ground

Microphone told to be the “right” microphone by hardwiring LR to VDD

i2S



WS specifies whose turn it is to speak (left or right channel)

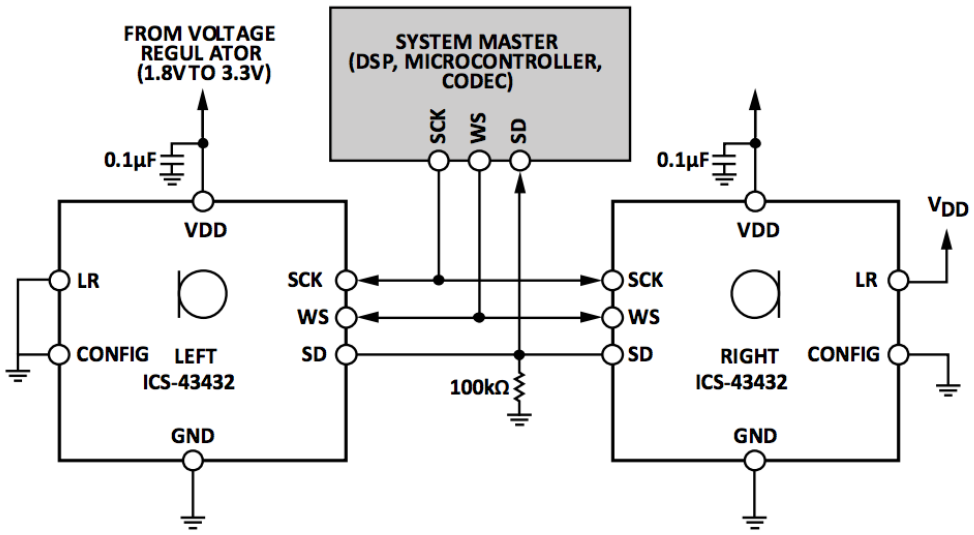


Figure 10. System Block Diagram

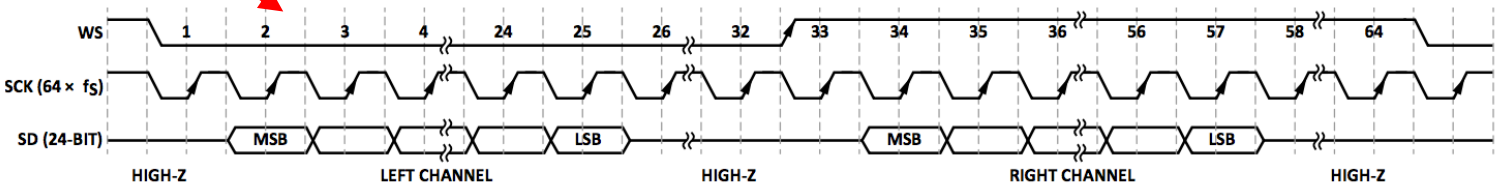


Figure 11. Stereo Output I²S Format

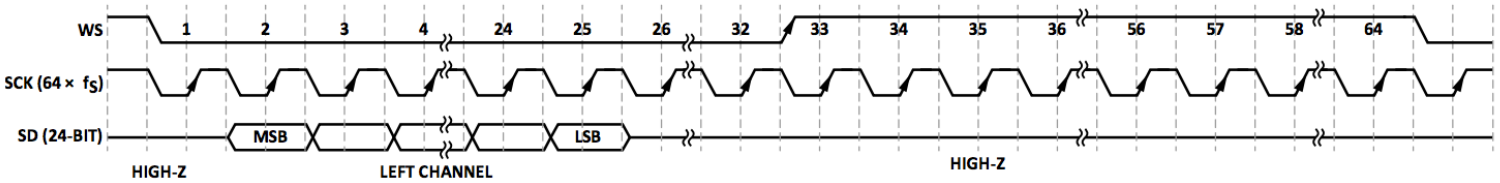


Figure 12. Mono Output I²S Format Left Channel (LR = 0)

Compare and Contrast All of Them?

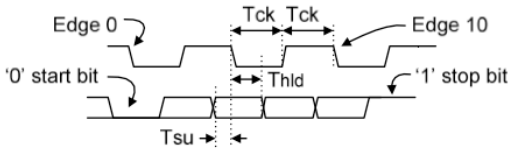
- Generally the fewer the wires the more rigid the protocol (i2C and to a certain extent UART)
- SPI can be very flexible and high speed (have only 10 bits to send? No problem…send 10!...can't do that do that with i2C…need to zero-pad up to the next full byte (16 bits)
- In terms of implementation, generally with communication protocols, the more wires, the easier the protocol/less overhead

Other protocols!

If Time...else we'll do in future class.

PS/2 Keyboard/Mouse Interface

- 2-wire interface (CLK, DATA), bidirectional transmission of serial data at 10-16kHz
- Format
  - Device generates CLK, but host can request-to-send by holding CLK low for 100us
  - DATA and CLK idle at “1”, CLK starts when there’s a transmission. DATA changes on CLK, sampled on CLK
  - 11-bit packets: one start bit of “0”, 8 data bits (LSB first), odd parity bit, one stop bit of “1”.
  - Keyboards send scan codes (not ASCII!) for each press, 8’hF0 followed by scan code for each release
  - Mice send button status, Δx and Δy of movement since last transmission

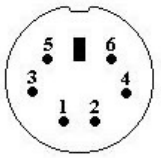


Symbol	Parameter	Min	Max
T_{CK}	Clock time	30us	50us
T_{SU}	Data-to-clock setup time	5us	25us
T_{HLD}	Clock-to-data hold time	5us	25us

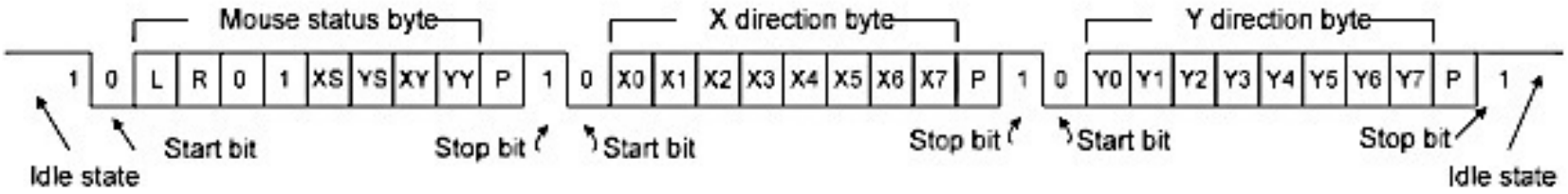
Figures from digilentinc.com

PS/2 Keyboard/Mouse Interface

- 2 signal wire interface (CLK, DATA), bidirectional transmission of serial data at 10-16kHz



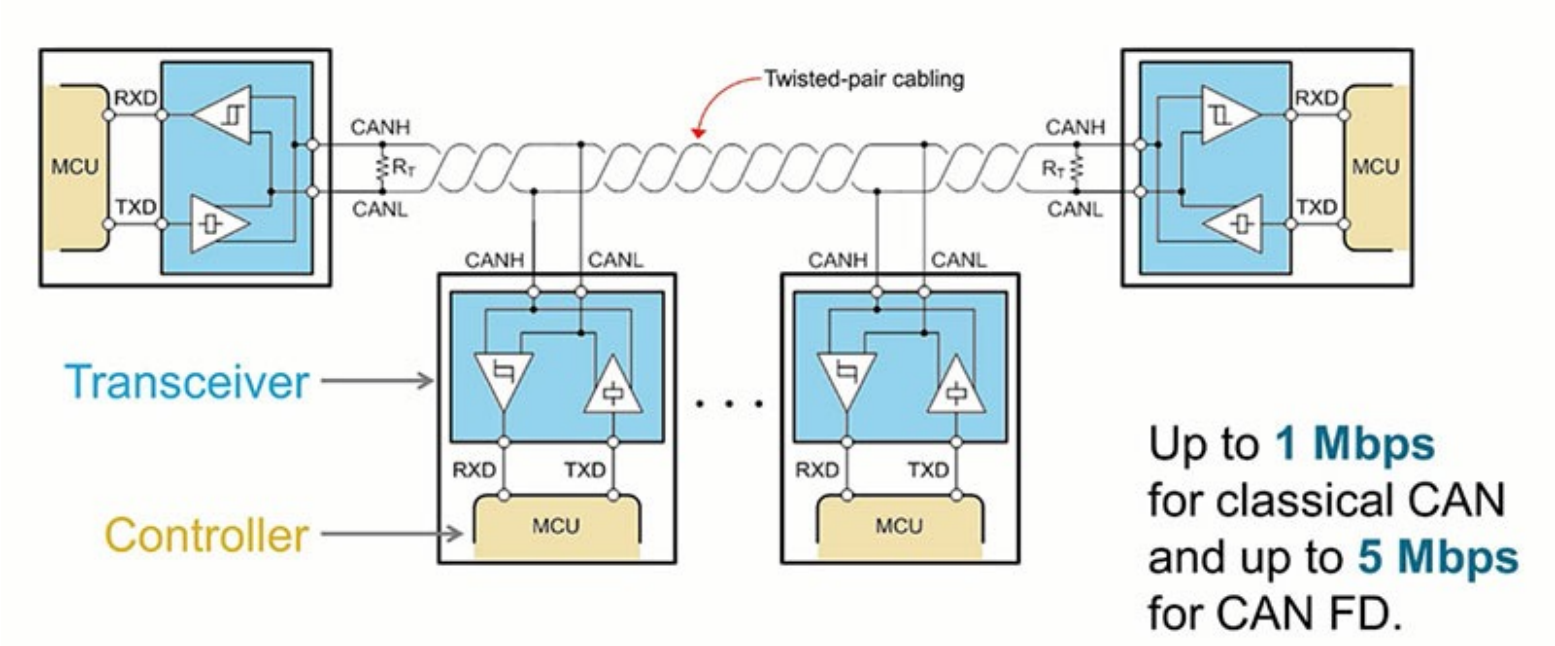
Pin	Signal	In/Out
1	Data	Out
2	N/C	
3	Ground	
4	+5V	
5	Clock	Out
6	N/C	



Figures from digilentinc.com

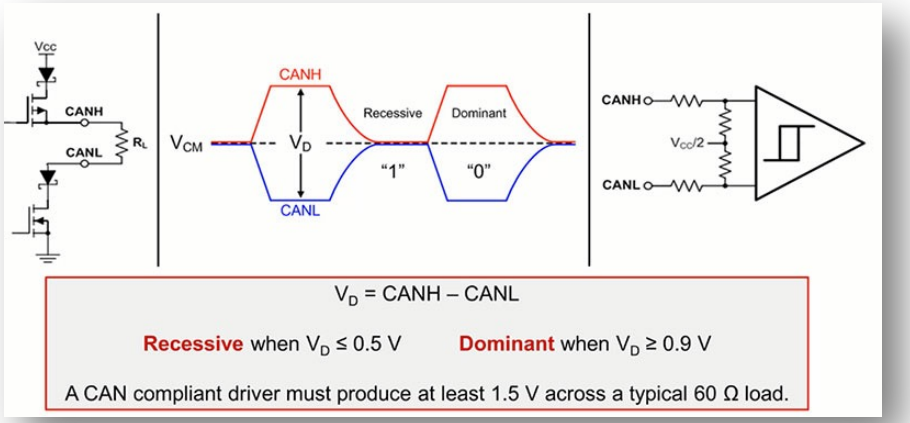
Controller Area Network (CAN) Bus

- Common bus protocol found in cars and other systems



https://www.digikey.com/en/blog/how-to-simplify-the-test-of-can-bus-networks

CAN Bus



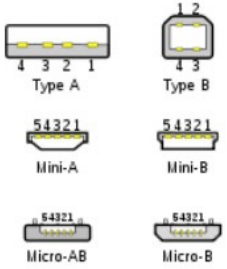
- Modules all share one common twisted wire channel
- Signaling is differential rather than single-ended (like HDMI)
  - Allows cables to be run long distances with good noise suppression
- Devices claim bus and listen with addressing scheme kinda similar to I2C

https://www.digikey.com/en/blog/how-to-simplify-the-test-of-can-bus-networks

USB: Universal Serial Bus

- USB 1.0 (12 Mbit/s) introduced in 1996
- USB 2.0 (480 Mbit/s) in 2000
- USB 3.0 (5 Gbit/s) in 2012
- USB-C 2016.
- USB 3.2 (30 Gbit/s) in July 20, 2017
- USB 4.0 (40 Gbit/s) 2019
- USB 4.0 2.0 (120 Gbits/s) 2022
- Created by Compaq, Digital, IBM, Intel, Northern Telecom and Microsoft.
- Uses differential bi-direction serial communications

Type A USB 2.0 – 4 pins

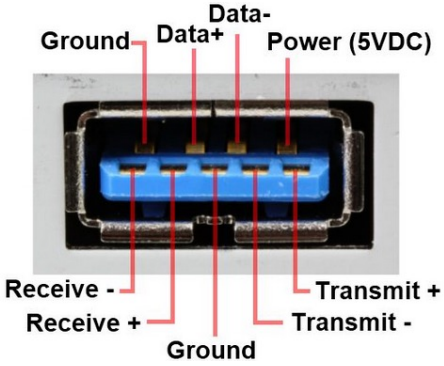


Pin	Name	Cable color	Description
1	VCC	Red	+5 V
2	D−	White	Data −
3	D+	Green	Data +
4	GND	Black	Ground

Type A & B Pinout

Pin	Name	Color	Description
1	VCC	Red	+5 V
2	D−	White	Data −
3	D+	Green	Data +
4	ID	none	permits distinction of Micro-A- and Micro-B-Plug Type A: connected to Ground Type B: not connected
5	GND	Black	Signal Ground

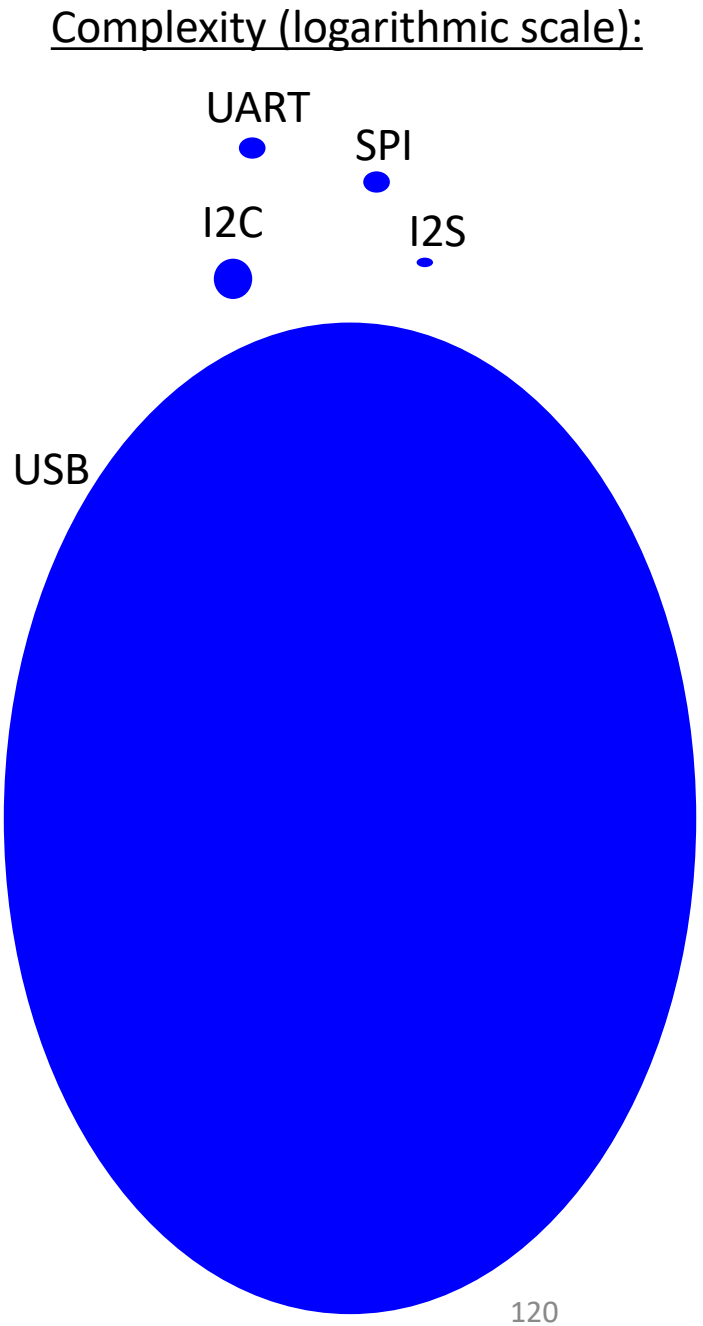
Mini/Micro Pinout



USB 3.0

USB: Universal Serial Bus

- Far, far more defined layers than your other things we've seen
- The 2000 version of USB spec was 570 pages long
- USB 3.2 (2017) Approximately 900 pages long at this point +supplemental stuff
- USB 4.0 (2019)…similar and so on



How is Data Transmitted in USB (High Level):

- Communication uses handshakes to establish capable/expected data rates
- Host device (computer for example), assigns connected devices temporary IDs on shared bus.
- Packets of information, including headers, payloads, and error checks (CRC5, CRC16, and CRC32 are used) are sent between host and client devices

How is Data Transmitted in USB (Bit Level):

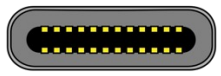
- USB uses twisted wire pairs and there is no CLOCK wire
- All data is transmitted using Non-Return-Zero-Inverted (NRZI) encoding:
  - A 0 is encoded as a value change
  - A 1 is encoded by no change
- After initial synchronization byte, the receiver extracts the clock from the on-average probability of 0's in the data (which give transitions) using local oscillator and Phase-Locked Loops
- Avoid long stretches of 1's by bit-stuffing (shoving 0's in to avoid periods of time where no transitions happen)…similar to ether protocols

USB - C

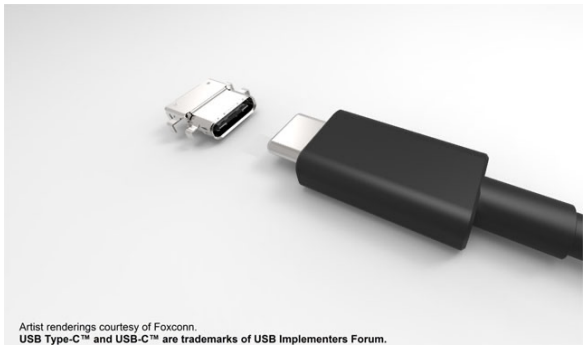
- New connector brought in with USB 3 standard
- Universal connector for power and data – first product MacBook Air – one and only port!
- Symmetrical – no “correct” orientation (Good for 10,000 insert/withdrawals…10 kiloinserts)
- Supports DisplayPort, HDMI, power, USB, and VGA. Uses differential bi-direction serial communications
- Supplies up to 100W power (5V @ up to 2A, 12V @ up to 5A, and 20V @ up to 5A)
- Voltage dictated by software handshake, etc..

Figure 2-1 USB Type-C Receptacle Interface (Front View)

A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12
GND	TX1+	TX1−	VBUS	CC1	D+	D−	SBU1	VBUS	RX2−	RX2+	GND
GND	RX1+	RX1−	VBUS	SBU2	D−	D+	CC2	VBUS	TX2−	TX2+	GND
B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1



Type-C



USB 4

- 2019 saw introduction of USB4
- Partially motivated by Intel/Apples donation of Thunderbolt spec to USB consortium in ~2017
- *Requires* use of USB-C-type cable
- Data rates up to 40 Gbps (1 full HD movie per second)

USB 4 2.0

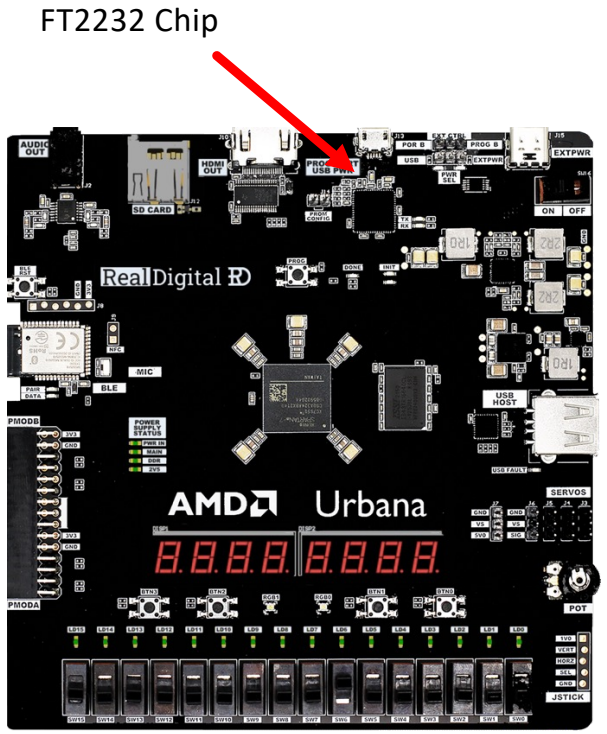
- 2022 and 2023 saw introduction of USB4 2.0
- *Requires* use of USB-C-type cable
- Data rates up to 120 Gbps (3 full HD movie per second because society needed that rather than UBI or universal healthcare)

FTDI Chipsets

- Future Technology Devices International Ltd (FTDI) is a Scottish Electronics firm that makes USB interfaces
- They produce devices that convert between USB and:
  - UART
  - SPI
  - I2C
  - Parallel Out
  - Etc…
- Extremely common (we use a few on our FPGA)

Lies!

- The UART you wrote in Lab 3 wasn't actually to the computer.
- It was to an FT2232 chip by FTDI
- Takes UART and converts back and forth to USB for you automatically



The Great FTDI Bricking of 2014

- From the beginning of USB to only recently, most USB devices used FTDI-based chip sets to interface (source of those annoying FTDXX.h library issues you'd always see in Windows)
  - Your optical mouse would have some circuit and it would communicate internally with UART…then the FTDI chip would convert to USB
- Dozens of "clones" were built to work with that software, these clones often times selling for a small fraction of the cost of the original FTDI chips
- In 2014 FTDI they released a software update, included in most Windows Service Packs that bricked all "non-genuine" devices
- Turned out a lot of "legit" products were using counterfeits/clones....lost them a lot of good will.
- Did it again later on too.

Conclusions

- Tons of protocols (just skimming the surface here)
- Great way to add complexity to a project!
- But! Plan ahead if talking to devices in final projects.
  - If interfacing to FPGA directly, interfacing anything above the most simple devices can take time!
    - That Virtual Reality headset team from 2019 probably spent 40% of their time writing a driver to control the screens over SPI (at 70 MHz)