

Lecture 11

Wired Communication:

SPI and I2C

CE346 – Microprocessor System Design

Branden Ghena – Spring 2025

Slides adapted from:

Branden Ghena (Northwestern), Josiah Hester (Georgia Tech), Prabal Dutta (UC Berkeley), Sparkfun

Project Hardware

- I have a bunch of stuff to hand to groups today!
- If you haven't placed your first set of parts orders yet, do so ASAP
 - Due by end-of-day Saturday
 - I will place next round of orders on Sunday
- Warnings about receiving hardware
 - Check what you received, and ask questions!
 1. I might have substituted something
 2. I may have a different number of items
 3. I may have forgotten some adapter you need

Today's Goals

- Discuss additional wired communication protocols: SPI and I2C
- Understand tradeoffs in design
 - UART, SPI, and I2C are each useful for different scenarios
- Explore real-world usage of SPI and I2C

Outline

- **Review**
- SPI
- I2C
- Using SPI and I2C

Tradeoffs in Wired Communication

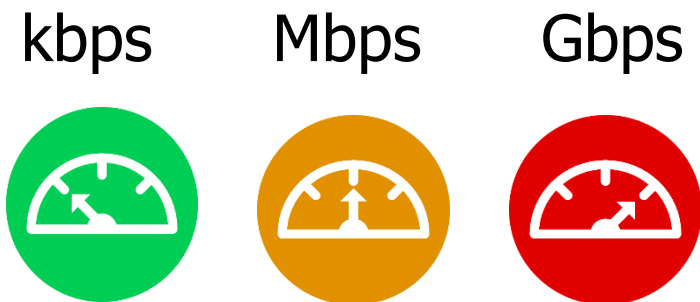
- Number of Signals



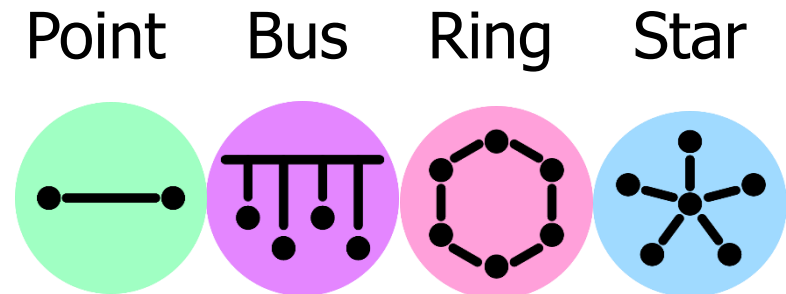
- Controlling Timing



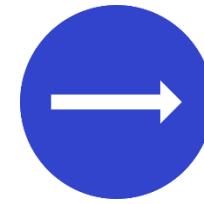
- Communication Speed



- Network topology



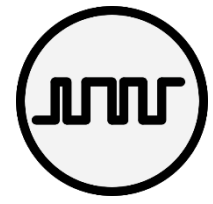
UART



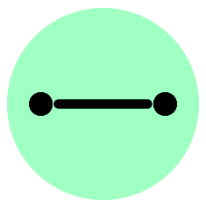
Serial



kbps



Async



Point

- Number of Signals: Two (plus optional flow control)
- Communication Speed: Usually 1 to 100 kbps
- Controlling Timing: Asynchronous
- Network Topology: Point-to-Point

UART Pros and Cons

- Pros

- Only uses two wires
- No clock signal is necessary
- Can do error detection with parity bit

- Cons

- Data frame is limited to 8 bits (20% signaling overhead)
- Doesn't support multiple device interactions (point-to-point only)
- Relatively slow to ensure proper reception

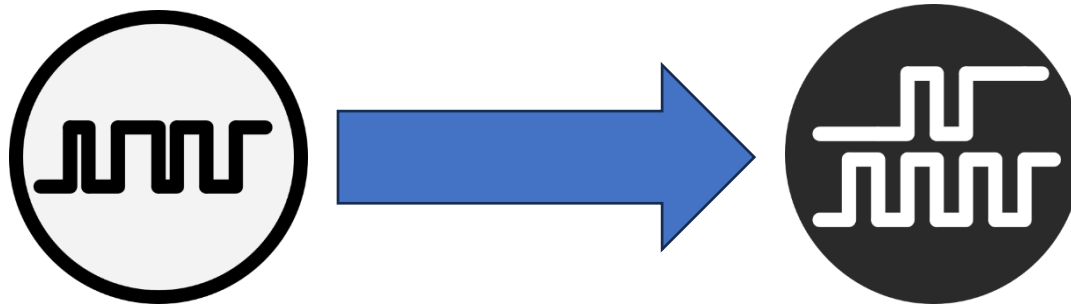
- Let's get rid of all the cons (by sacrificing on all the pros)

Outline

- Review
- **SPI**
- I2C
- Using SPI and I2C

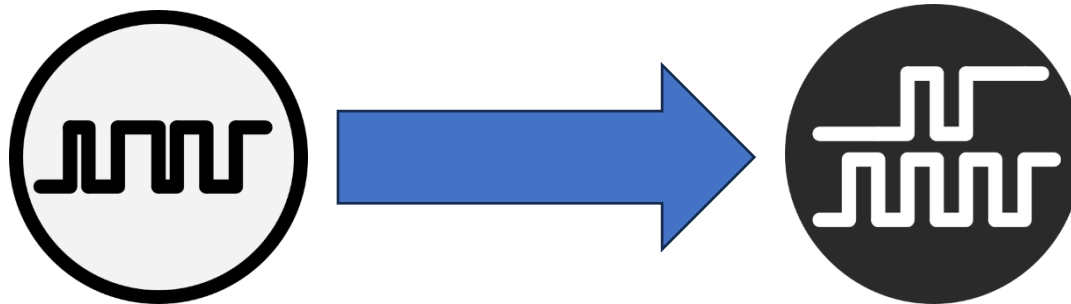
Let's start with timing control

- How do we move from asynchronous to synchronous communication?



Let's start with timing control

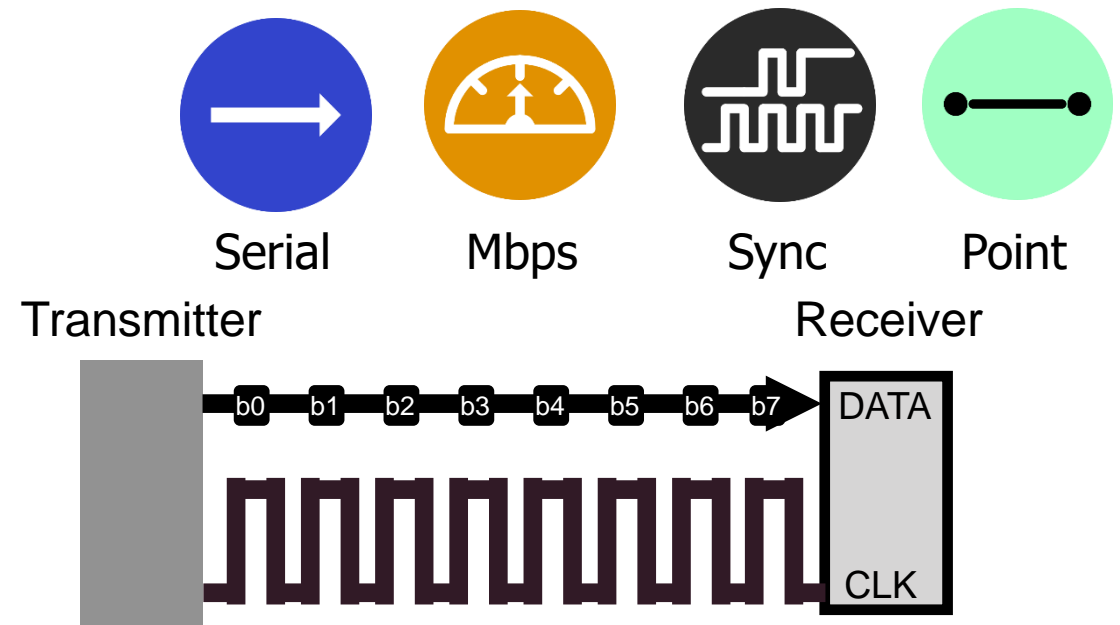
- How do we move from asynchronous to synchronous communication?



- Just add a clock signal!

Synchronous UART

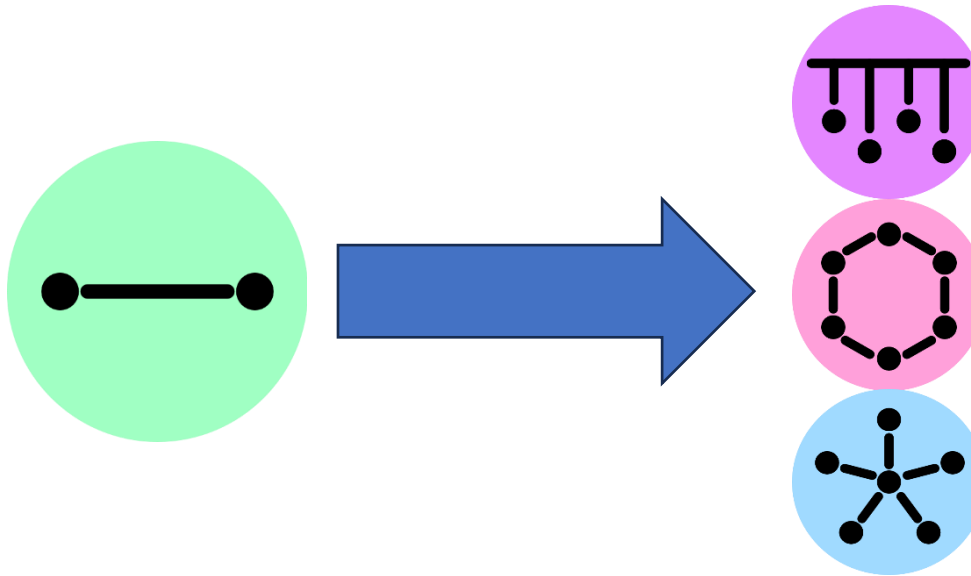
- USART
 - Synchronous/Asynchronous
 - Just add a clock line



- Common peripheral in many microcontrollers to allow adaptable communication
 - Could build various protocols (like SPI or UART) on top of it
- Still point-to-point limited in this form

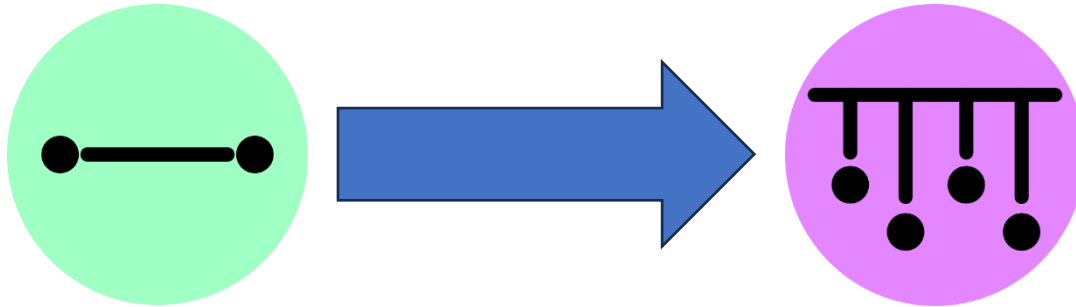
Now, let's change the network topology

- How do we allow communication with more than one device?
 - Start by choosing a network topology.



Now, let's change the network topology

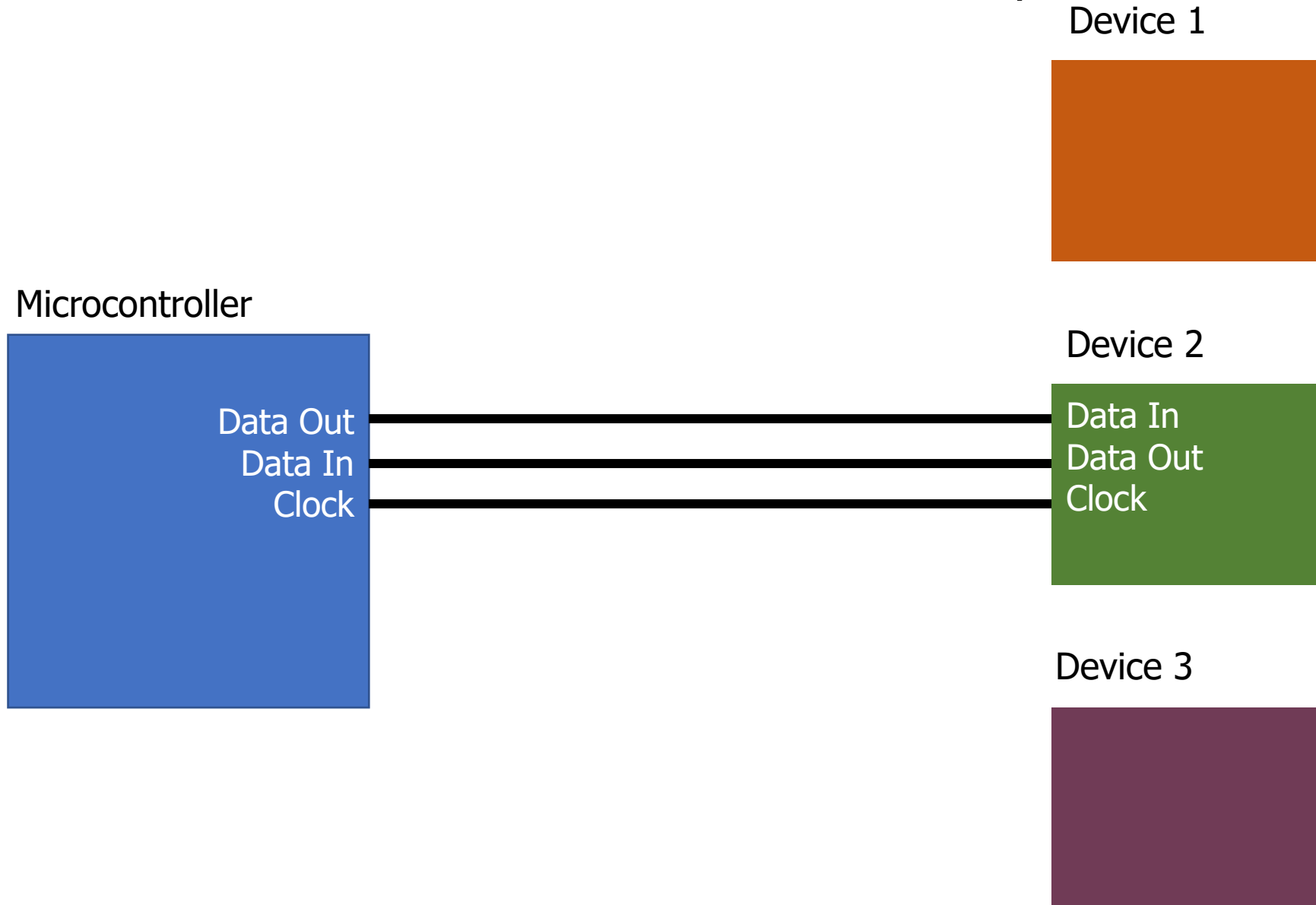
- How do we allow communication with more than one device?
 - Start by choosing a network topology.
 - Bus = (relatively) simple to implement



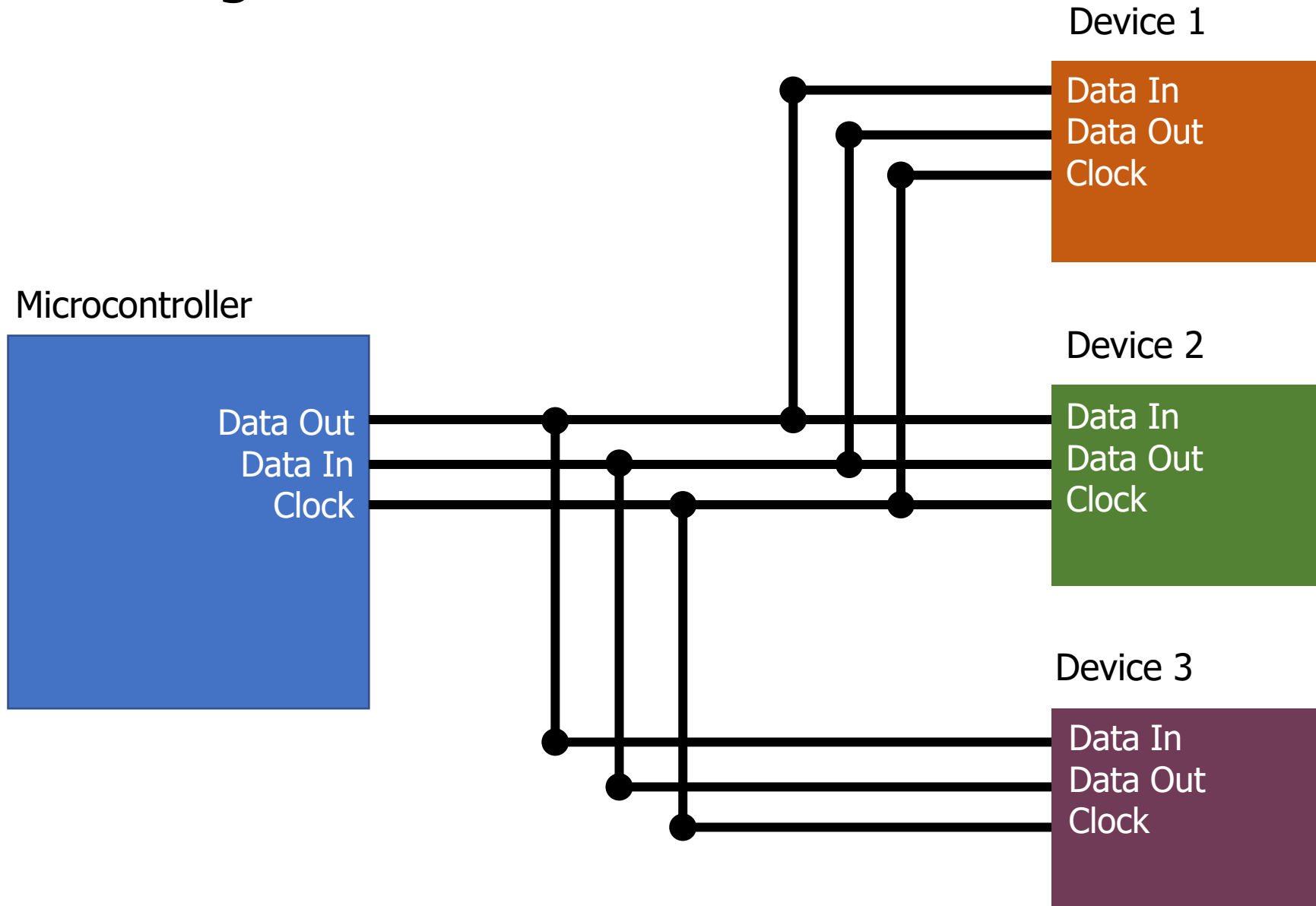
Synchronous serial communication with a single device



Want bi-directional communication, so three wires



Wire signals to all devices to form a bus



Communicating on a bus



How do you distinguish which device you are talking to?

Communicating on a bus



How do you distinguish which device you are talking to?

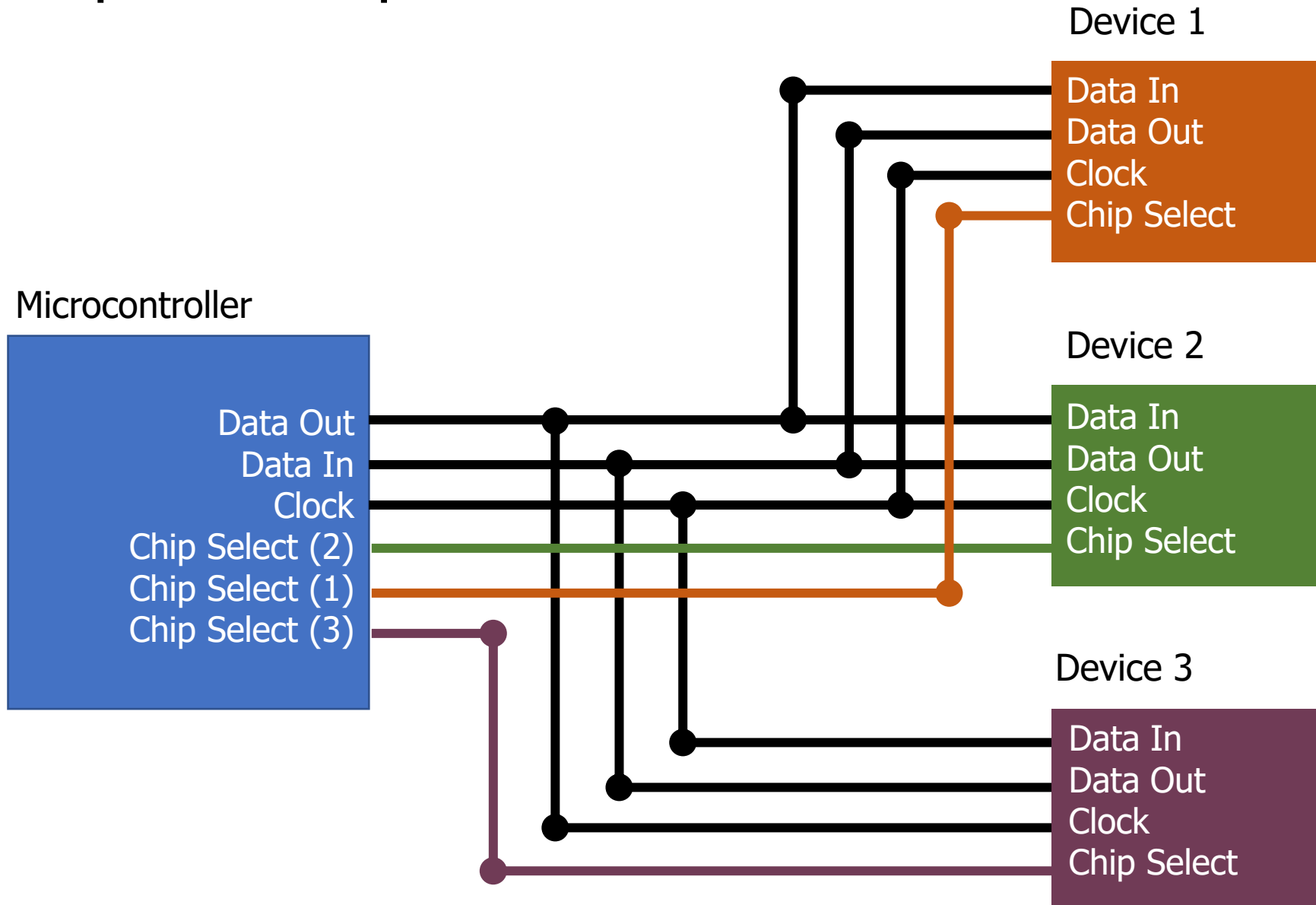
1. Address for each device

- Devices must always listen and then discard messages that aren't for them
- Need to define packet format so it's clear where the address is
- Need a method for addressing devices

2. GPIO pin for each device

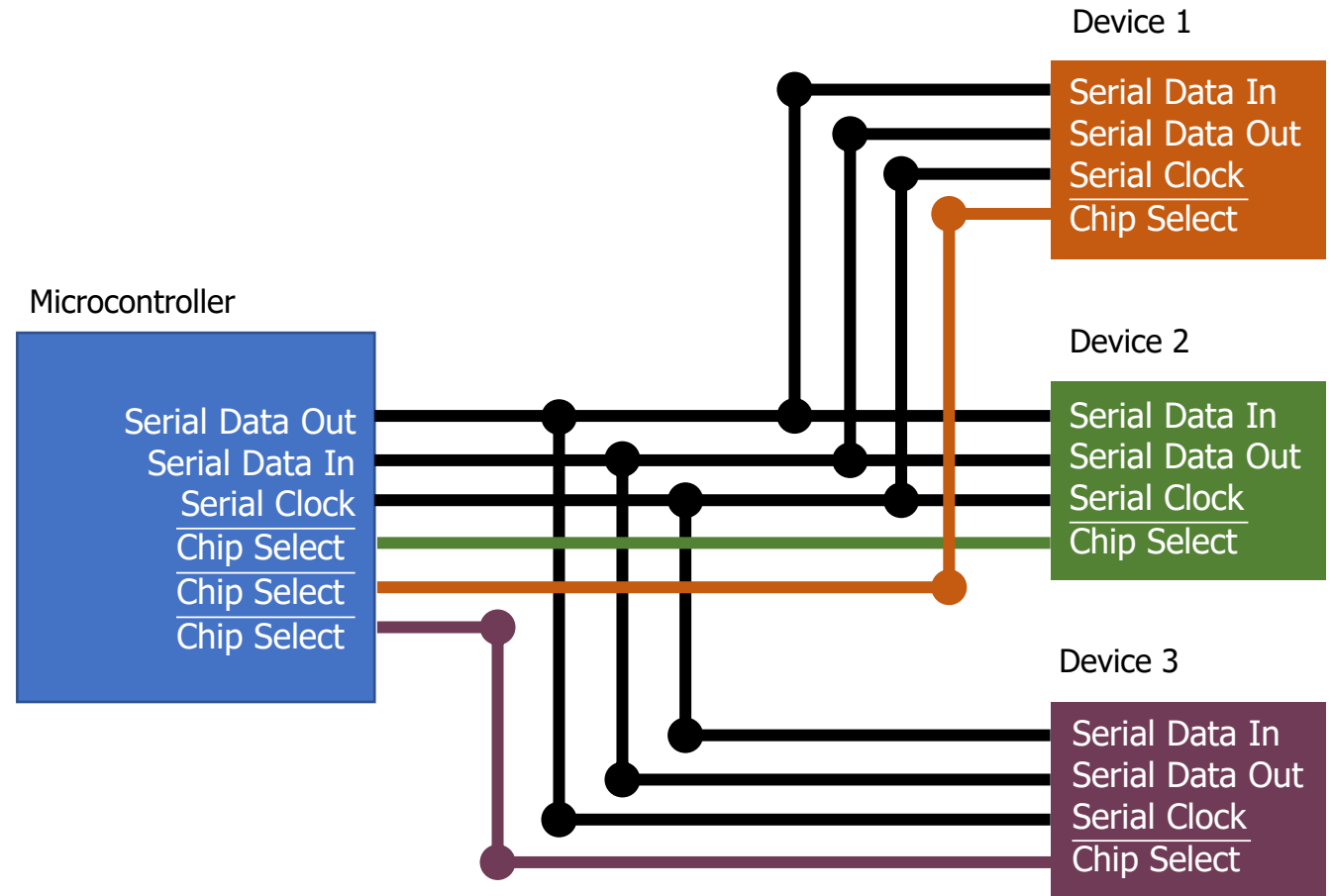
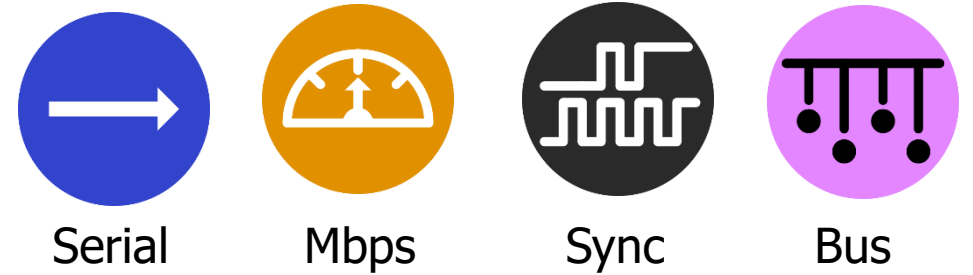
- Signal which device is being communicated with
- Only activates communication on transition of "select" line
- Needs a separate pin for each device

Separate chip select line for each device



Serial Peripheral Interface (SPI)

- Serial, synchronous, bus communication protocol
- Single controller with multiple peripherals
 - Within a circuit board
- High-speed communication
 - Multiple Mbps



A note on outdated notation

- Master/Slave paradigm
 - Master is the “Computer” and is in charge of interaction
 - Slave is the “Device” and has little control over interaction parameters
 - Really common notation in EE side of the world.
 - Not intended to be harmful, but also literally inconsiderate.
- Field is changing for the better. It’s going to take some time.
 - **Controller/Peripheral**
 - Central/Peripheral
 - Device/Peripheral
 - Master/Minion
 - Primary/Secondary

SPI naming schemes

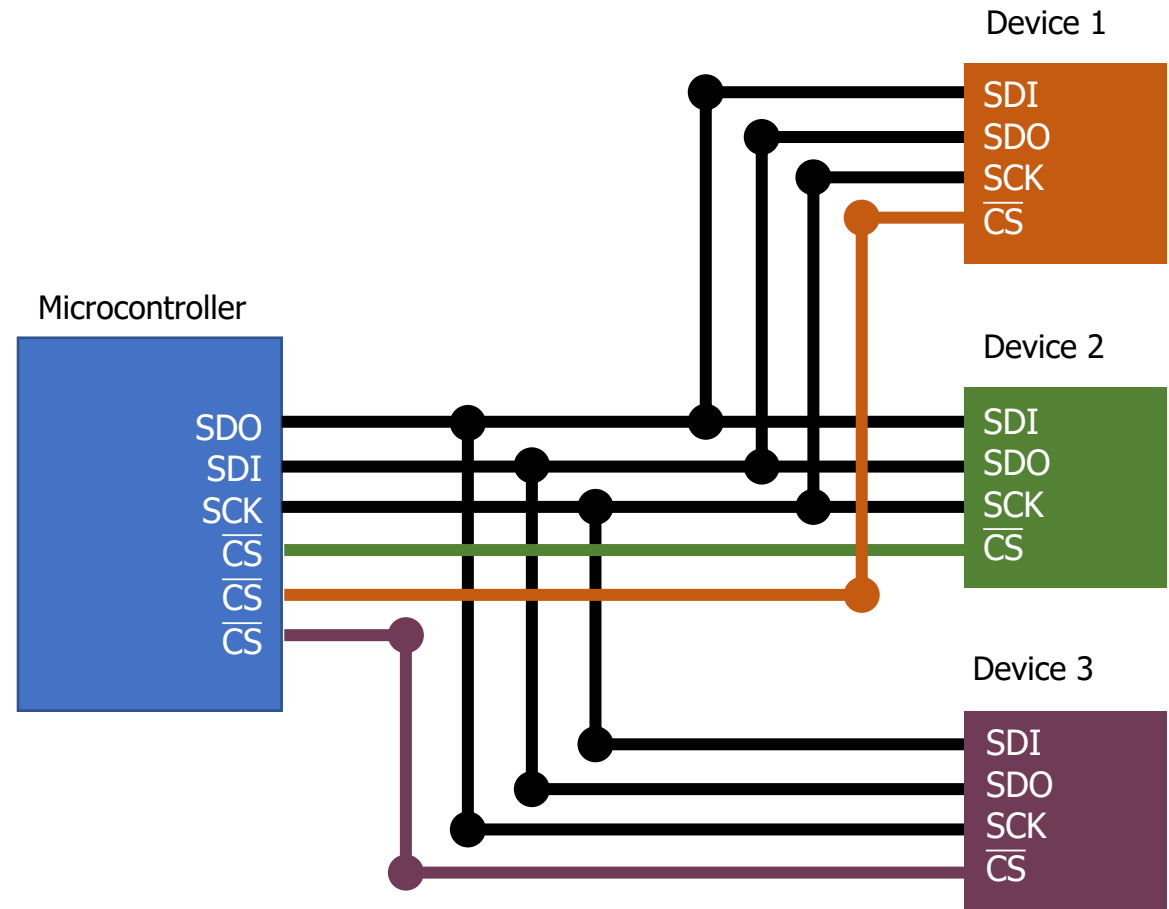
- Historical SPI Naming
 - MISO – Master In Slave Out
 - MOSI – Master Out Slave In
 - SS – Slave Select
- Revised SPI Naming
 - SDI – Serial Data In -> also known as CIP0 (Controller In, Peripheral Out)
 - SDO – Serial Data Out -> also known as COP1 (Controller Out, Peripheral In)
 - CS – Chip Select

<https://www.oshwa.org/a-resolution-to-redefine-spi-signal-names>

https://www.sparkfun.com/spi_signal_names

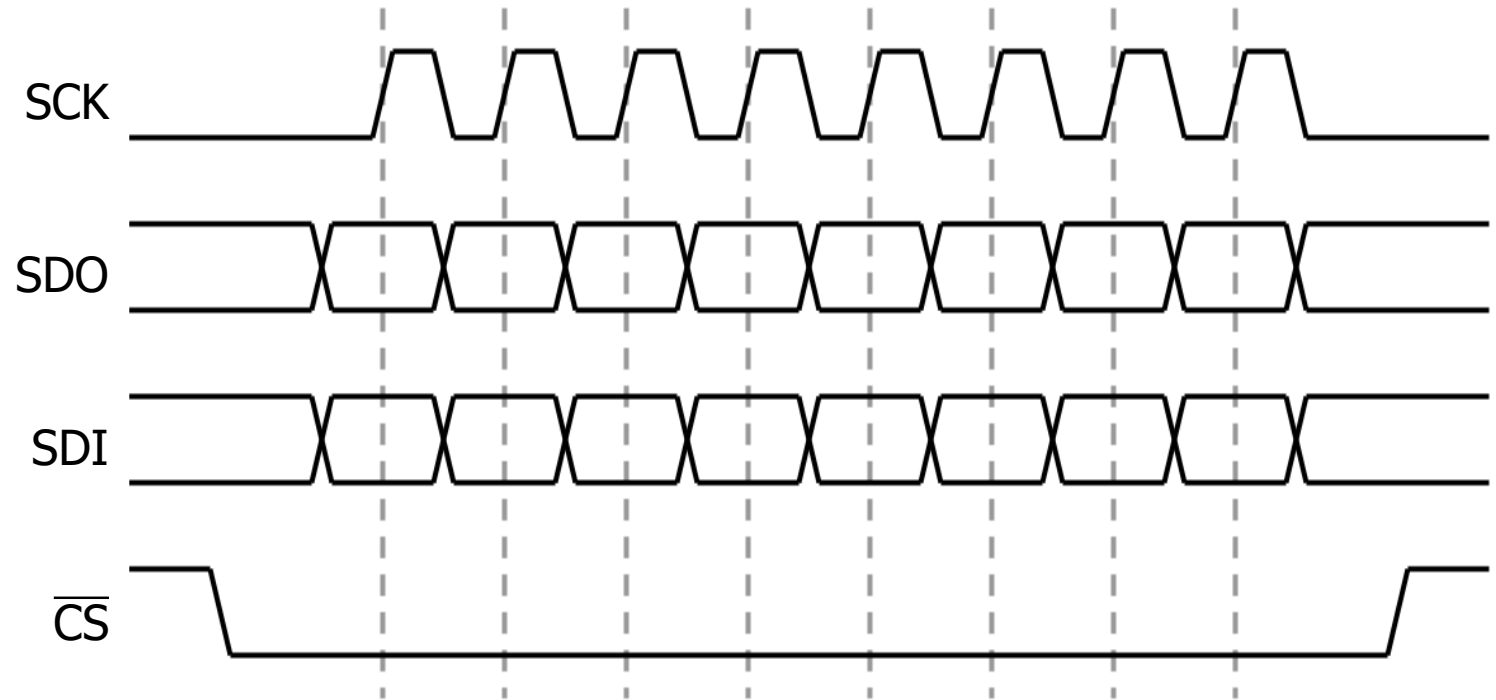
SPI wiring

- $3 + N$ wires for N peripherals
- SDI – input to the chip
- SDO – output from the chip
- SCK – Serial Clock
- CS – $\overline{\text{Chip Select}}$
 - Active low signal
- Names are always relative to this particular chip
 - SDO connects to SDI
 - SDI connects to SDO



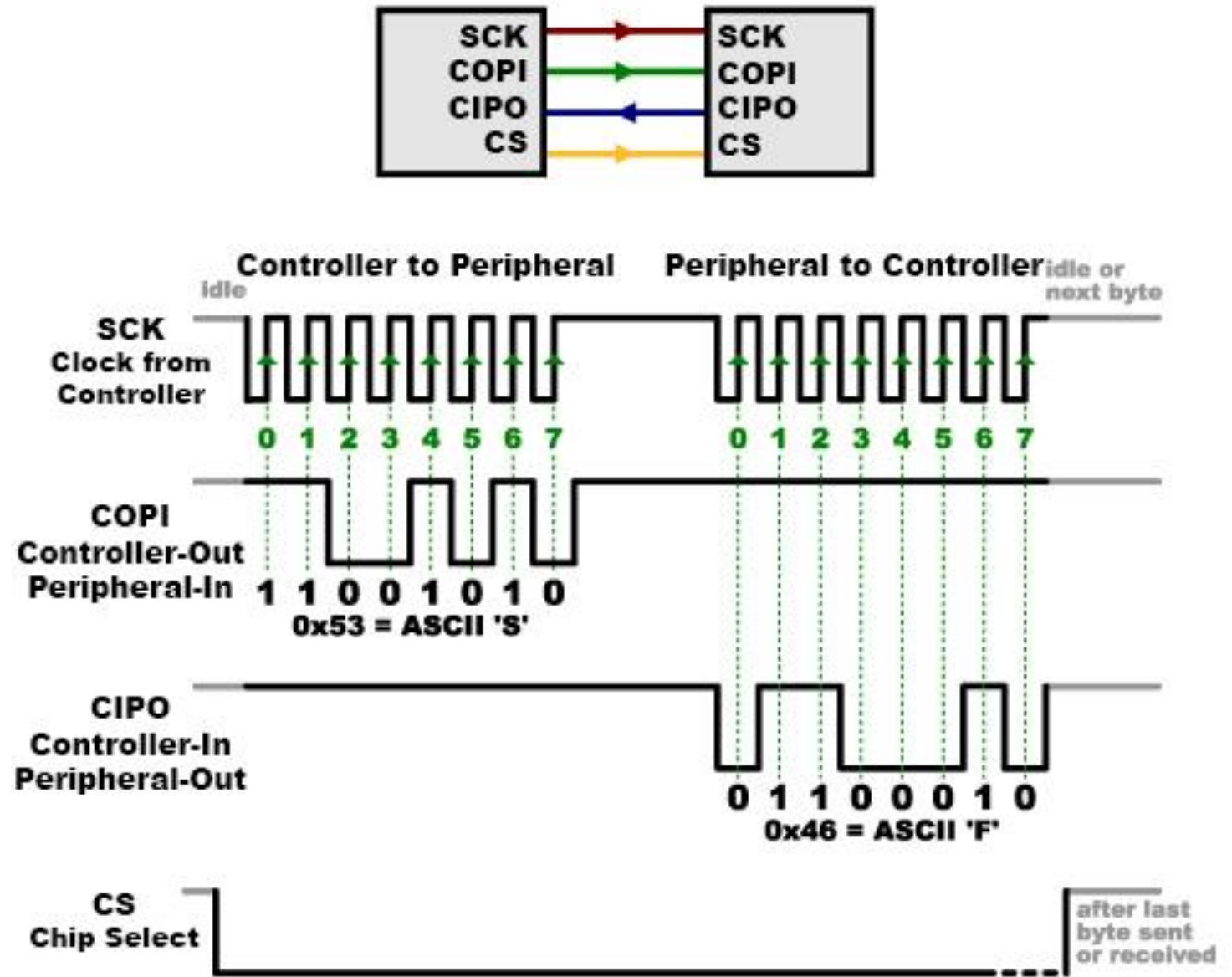
SPI timing diagram

- CS goes low to start transaction and high to end
- Data is sent synchronously with clock signals
- Capable of full-duplex transfers
 - Both directions at the same time

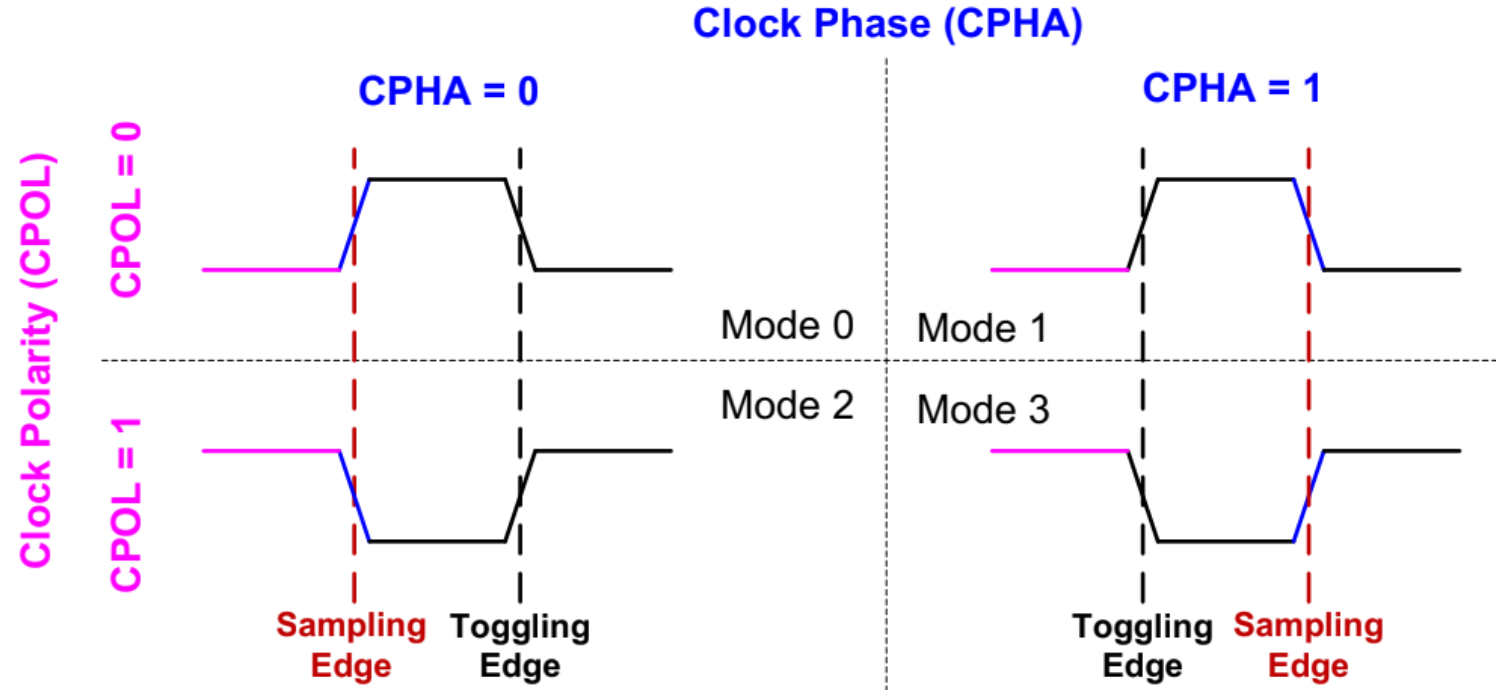


SPI communication

- Transactions usually in multiples of bytes (as many as needed)
- Either bit endianness is possible
 - nRF can do LSb first OR MSb first
- No need for framing bits (start/stop)
 - CS handles that



SPI configurations



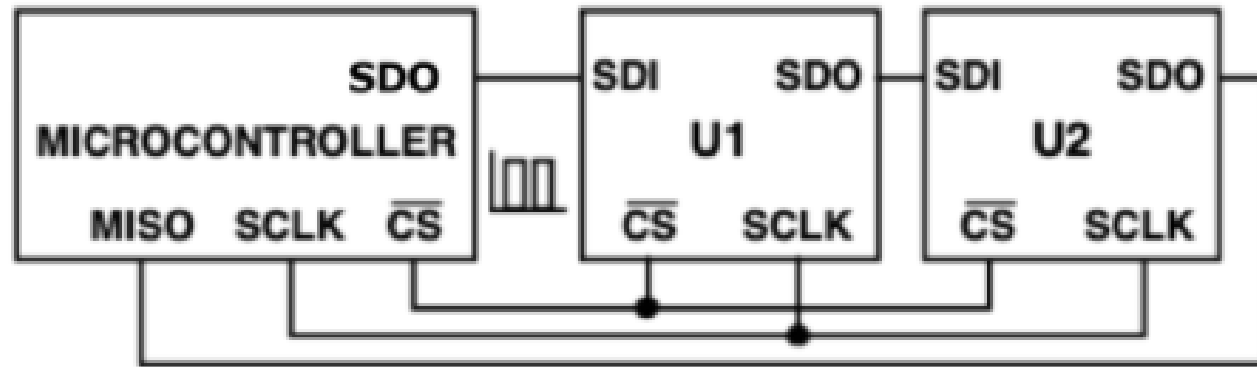
- CPOL – is the clock default low or default high
- CPHA – is data read on first edge or second edge
- Peripherals tell you what their configuration is (usually 0 and 0)

SPI data rate

- No particular requirements
 - Speed can go as fast as your clock and line capacitance can handle
- Datasheet for devices will specify their speeds
 - Sort-of standard rates (less so than UART, for example)
 - 700 kbps
 - 3.4 Mbps
 - 10 Mbps

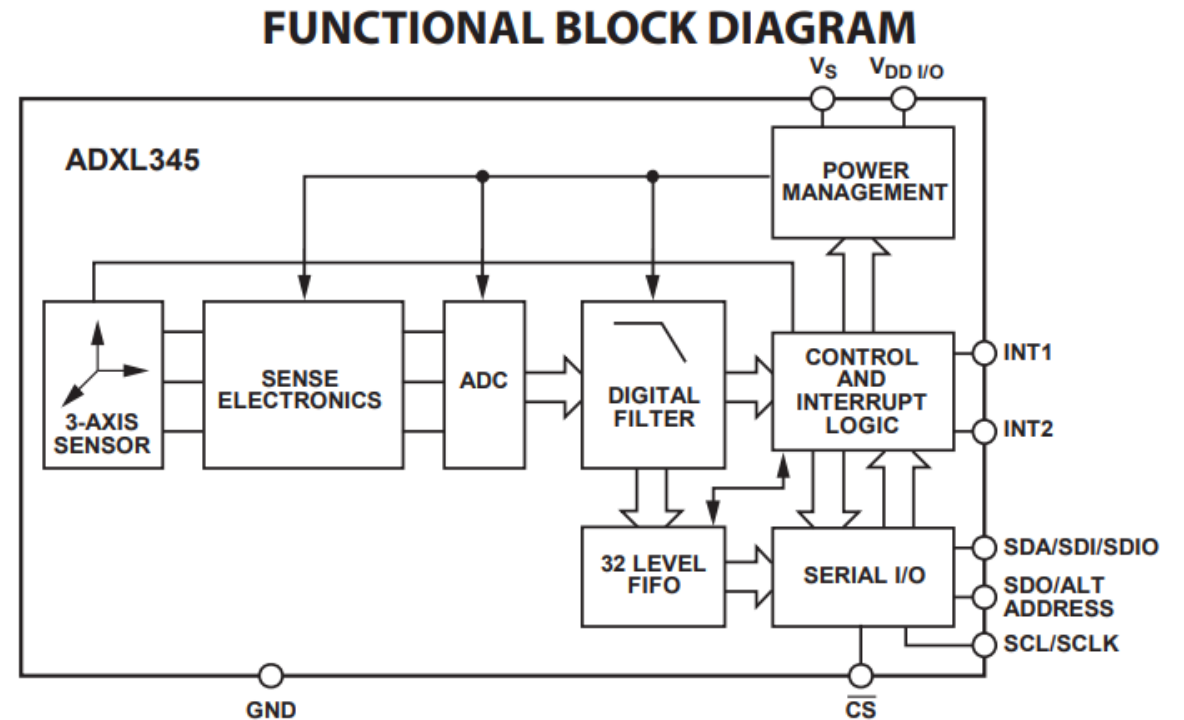
Daisy-chaining SPI

- SPI can also be formed into a ring bus
- Doesn't save on pins, but does reduce wires...
 - At the cost of reliability and speed
- Fairly rare in practice



How do we determine when peripheral has information?

- Controller starts/stops SPI transfers
 - Could ask peripheral periodically
- Peripherals often add interrupt outputs to signal controller that an event has occurred
 - More pins, yay!



Use Cases

- High-speed peripherals
 - Microphone, External ADC, Displays!
- External memory
 - Memory chips
 - SD cards
 - All SD cards support a SPI communication mode
- QSPI – Quad SPI (four SDO lines for more throughput)
 - Often used for communication with external memory

SPI Pros and Cons

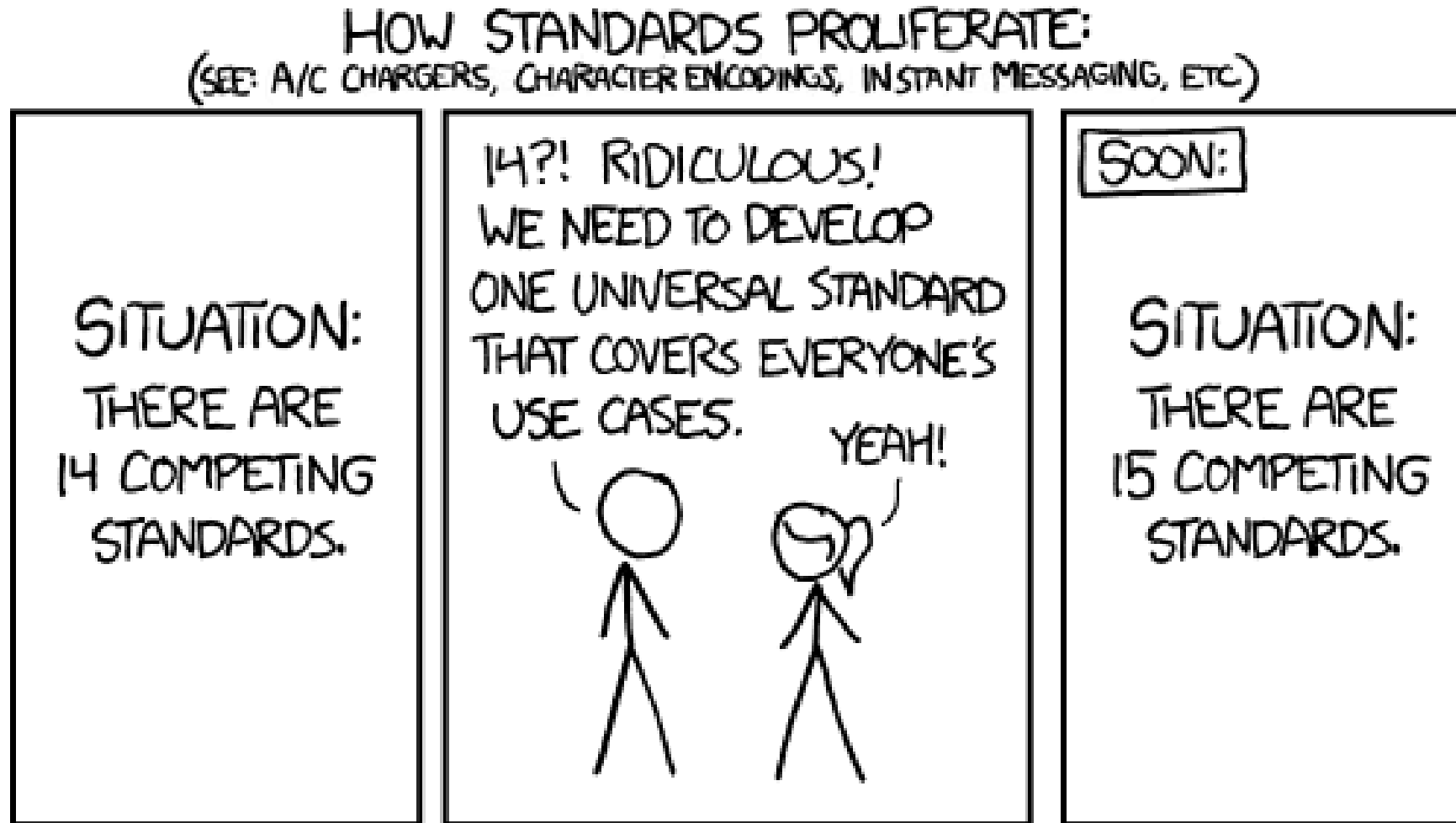
- Pros

- Faster throughput (and no overhead)
- No restrictions on data frame
 - No addressing requirements or word size assumptions
- Full duplex transfers

- Cons

- Many pins: $3 + N$ (for N peripherals)
 - CS line scales linearly (other signals are a bus)
- Controller must initiate all transfers
 - Not designed for multi-controller scenarios

Relevant xkcd



Outline

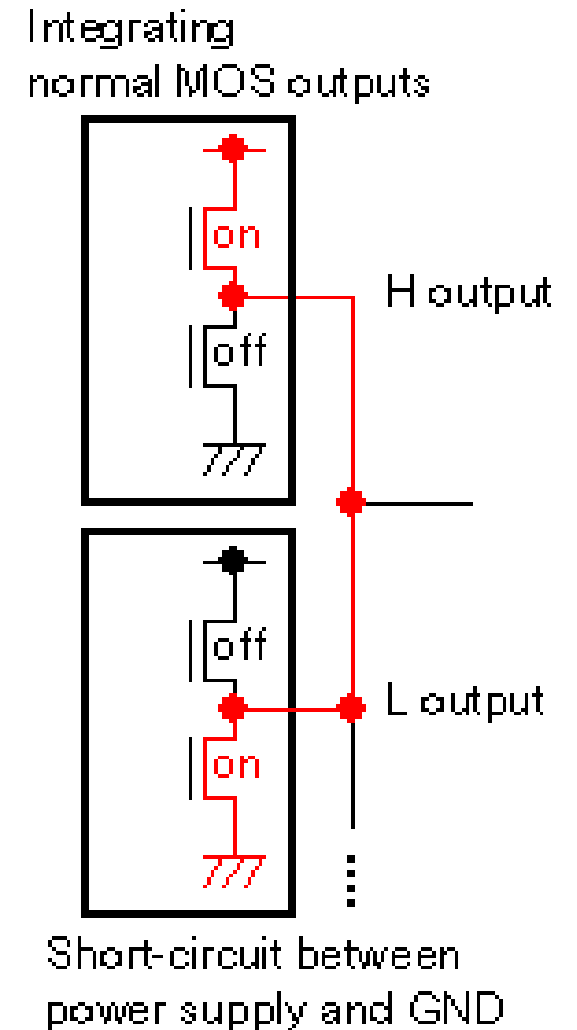
- Review
- SPI
- **I2C**
- Using SPI and I2C

Choosing different tradeoffs from other wired communication

- Things we like from SPI
 - Communication over a bus
 - Synchronous communication
- Things we want from new protocol
 - Fewer I/O pins
 - Use a single data line for bi-directional communication
 - Needs addressing and more specified data frame
 - Multiple controllers sharing the bus
 - Needs a bus contention solution

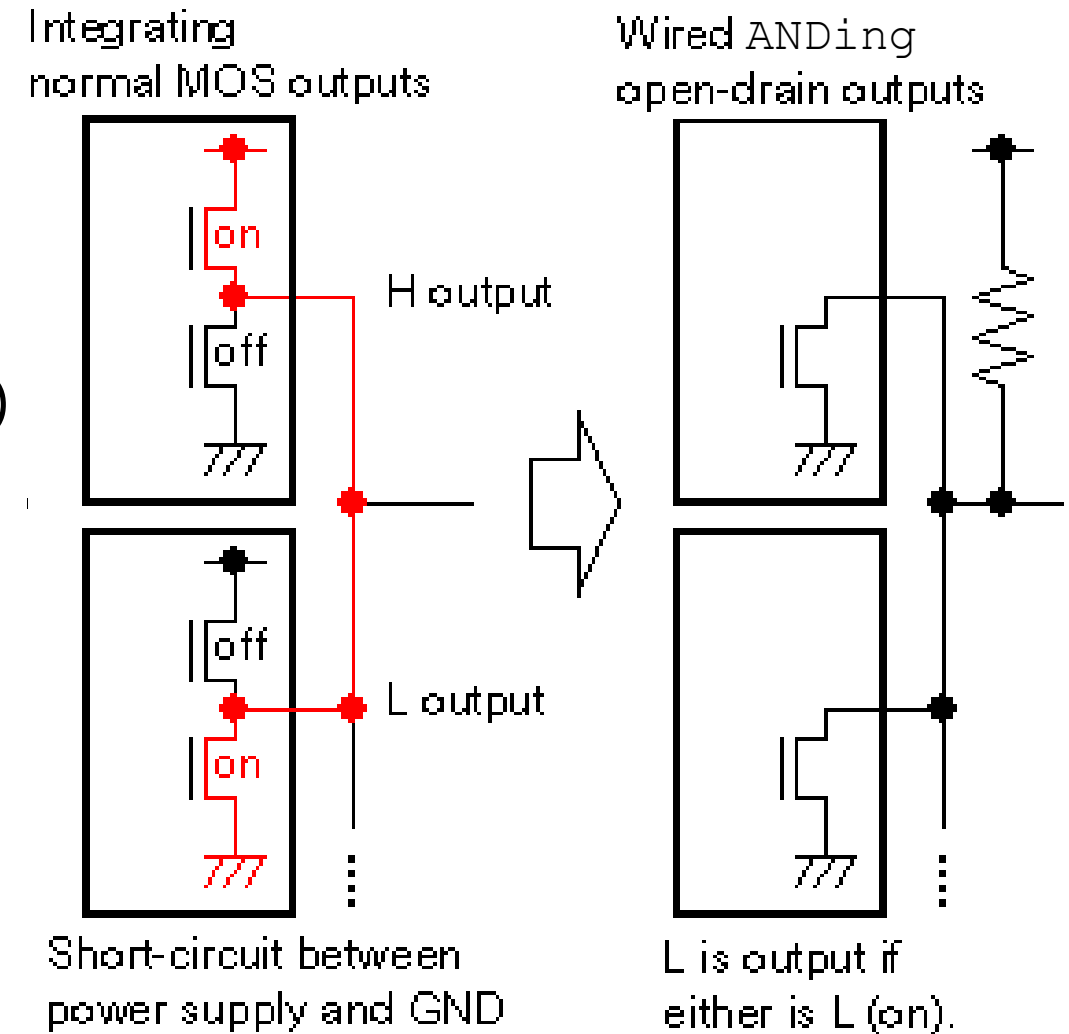
Bus contention could short a shared bus

- Want to enable multiple controllers
- Problem
 - What if they each try to transmit different data?
 - At some point, there will be a short-circuit

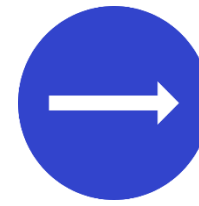


Disconnected I/O pins enable shared communication

- I/O pins often have three states
 - High
 - Low
 - Disconnected
(also known as High-Impedance/High-Z)
- We can use this third state to enable communication over a shared line
 - Low or Disconnected
 - Wired-AND
 - 1 if they are all disconnected
 - 0 if any are low



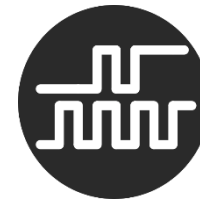
Inter-Integrated Circuit (I²C)



Serial



kbps

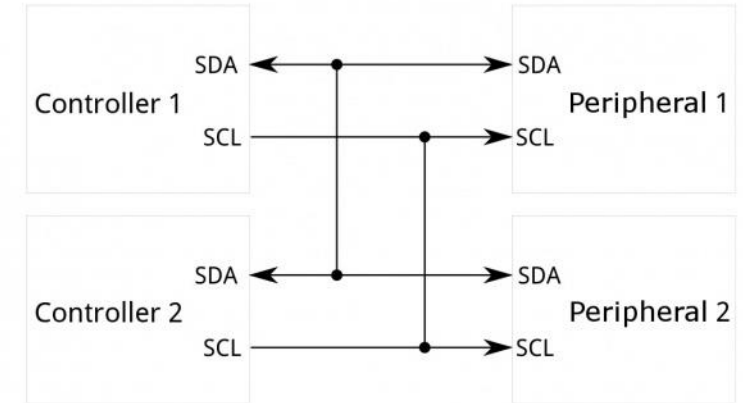


Sync



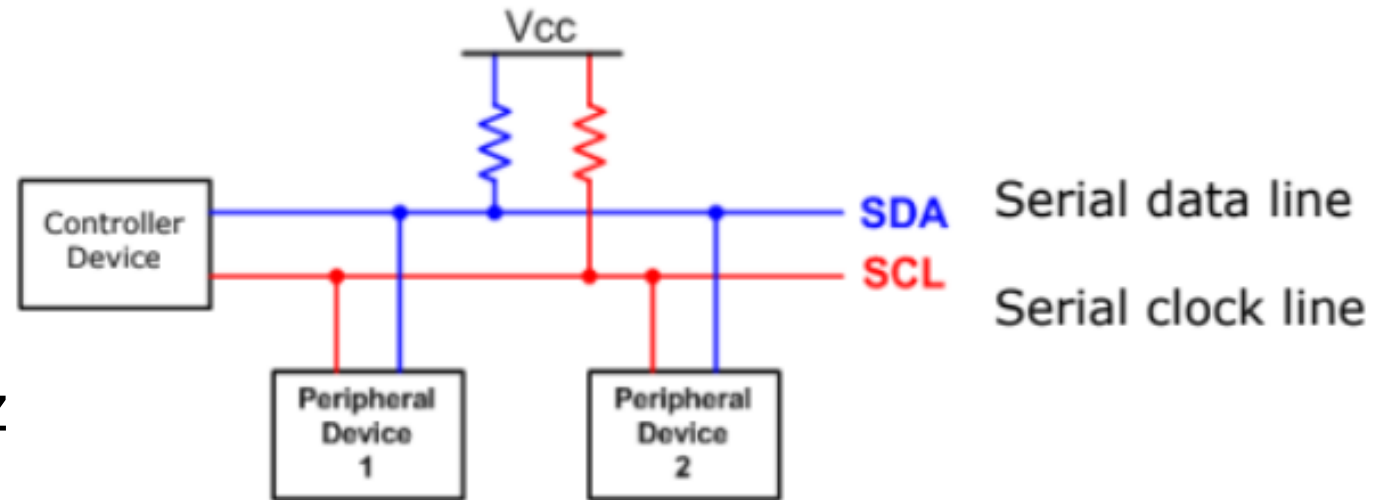
Bus

- Two-wire, synchronous, bus communication
 - Ubiquitous in the embedded world
 - De-facto standard for sensors
- Invented and patented by Phillips (now NXP)
 - Patent expired in 2004
- Also known as Two-Wire Interface (TWI)
 - Occasionally as System Management Bus (SMBus or SMB) but that's actually a related but separate thing

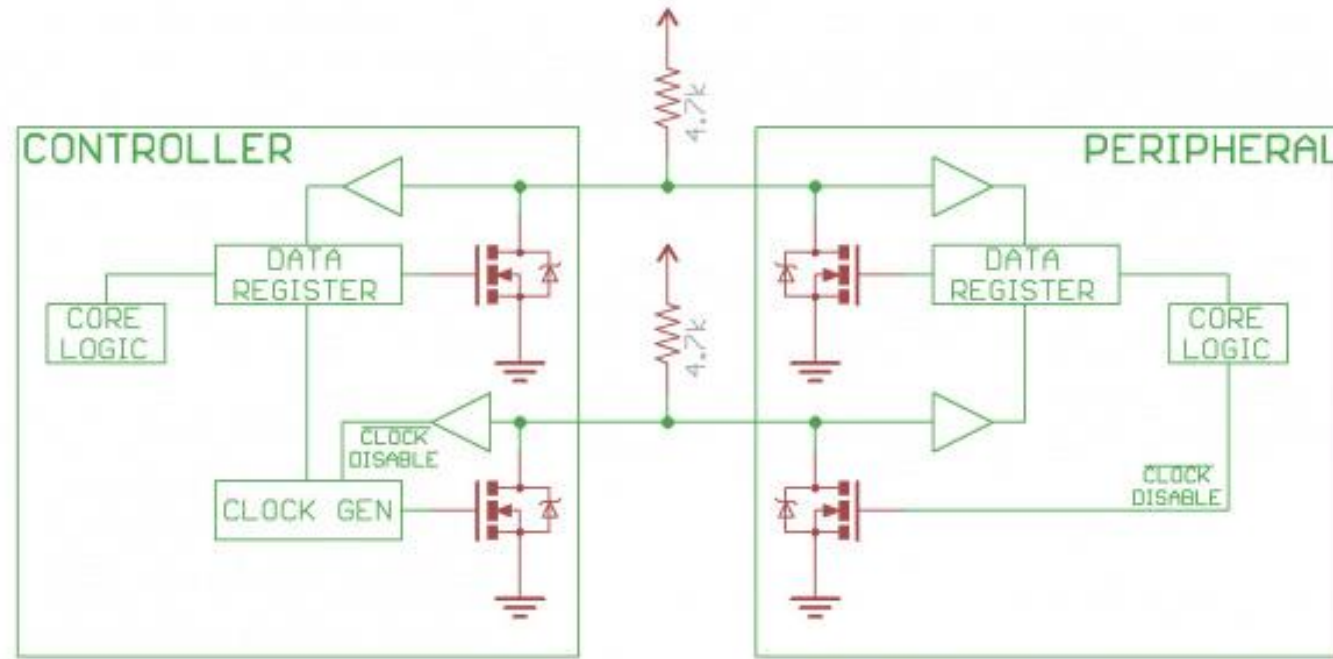


I2C overview

- SDA – Serial Data
- SCL – Serial Clock
 - Usually 100 kHz or 400 kHz
- Communication is a shared bus between all controller(s) and peripheral(s)
- Pull-up resistors for open-drain communication



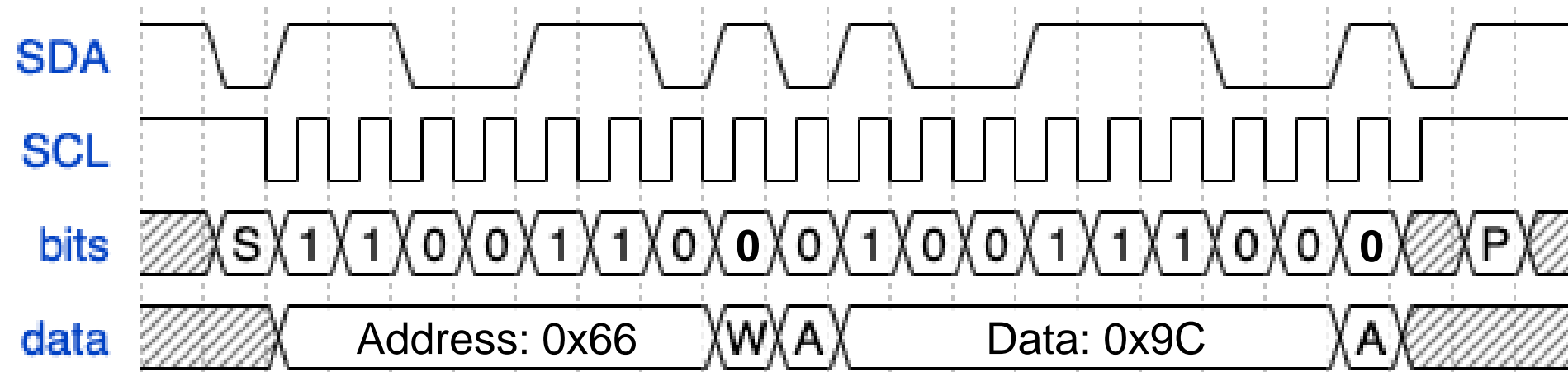
Open drain bus communication



- SDA and SCL are open-drain
 - 1 – high-impedance, let line float high
 - 0 – active drive, pull line low

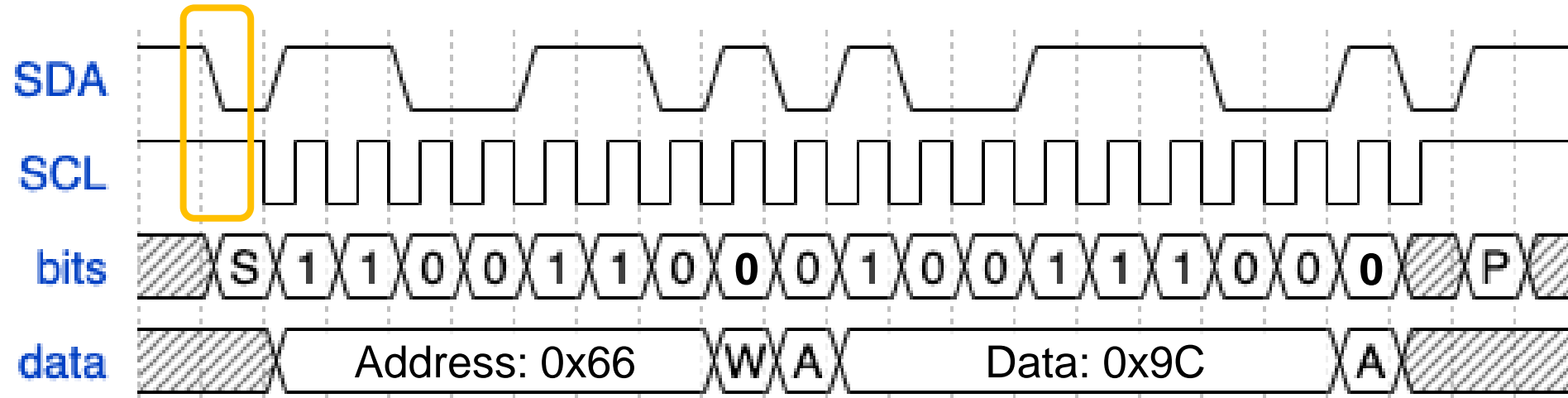
- Pull-up resistor to provide high signal
 - Low enough resistance that current can flow in a reasonable amount of time
 - Common value: 4.7 kΩ

I2C transaction walkthrough



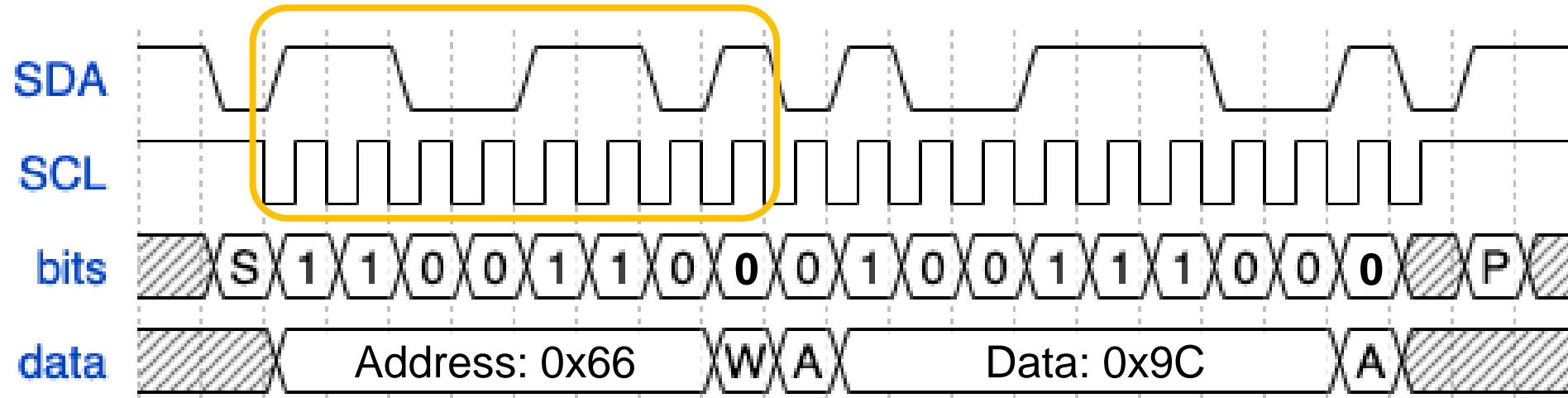
- Here's a one-byte I2C write to a device
 - Let's walk through all the parts of how this works

I2C transaction walkthrough



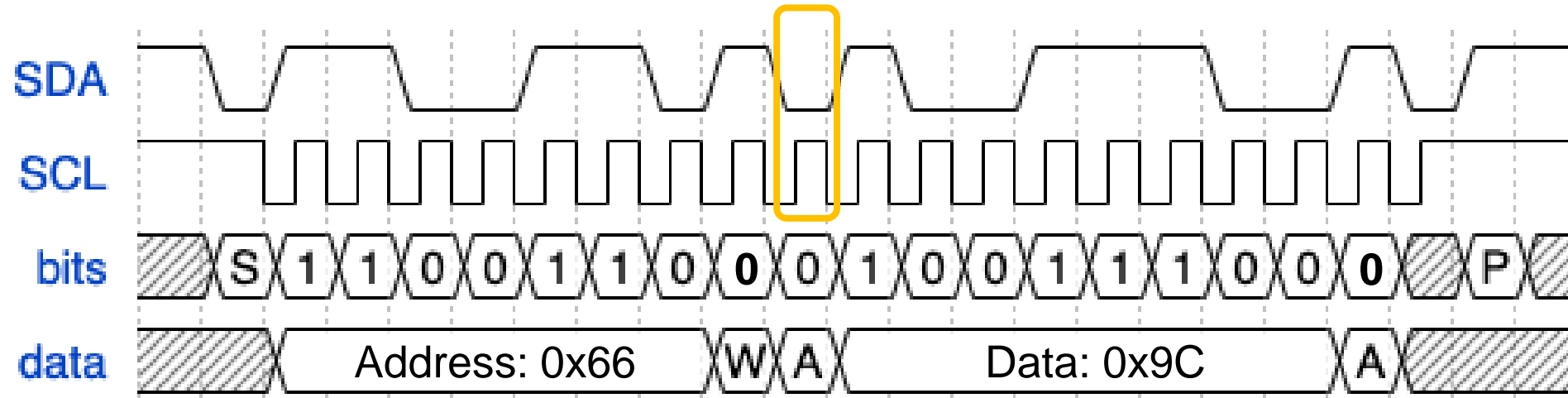
- Default state for SDA and SCL is high
- A transaction starts when data goes low while clock is still high
 - Usually, data changes when clock is low
 - Known as a “Start Condition”

I2C transaction walkthrough



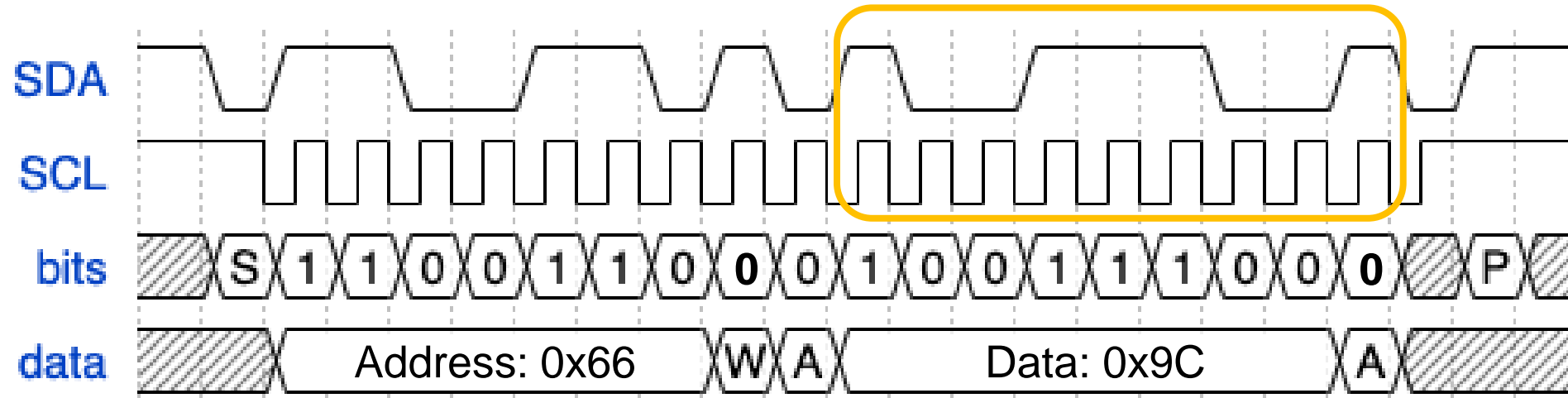
- First byte sent has two parts
 - 7-bit chip address
 - 1-bit read/write indication (1 for read, 0 for write)
- Values are read MSb first (reverse of other protocols 🤪)

I2C transaction walkthrough



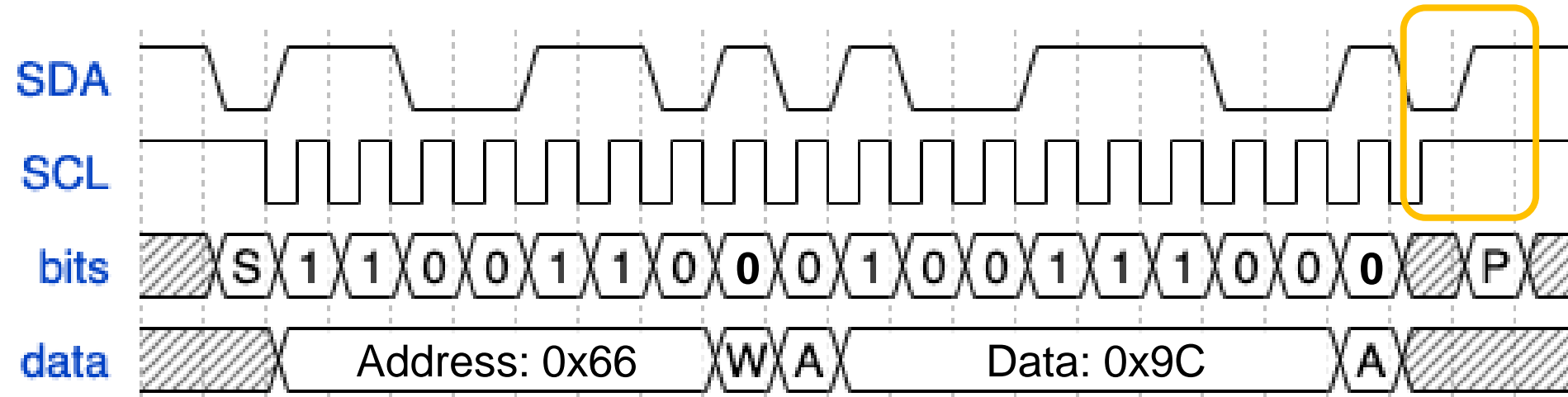
- Acknowledgement is sent by the **other** device
 - Sender briefly changes to an input (high-z)
 - 1 is the default state of the bus, which means no device responded
 - 0 means some device responded, which is an ack

I2C transaction walkthrough



- Data byte(s) are sent after the address
 - For a write, bytes are sent by the controller
 - For a read, bytes are sent by the peripheral
- Other device acknowledges each byte along the way

I2C transaction walkthrough



- Stop condition
 - Let data go high while clock is still high
 - Returns bus to default state

Bus arbitration

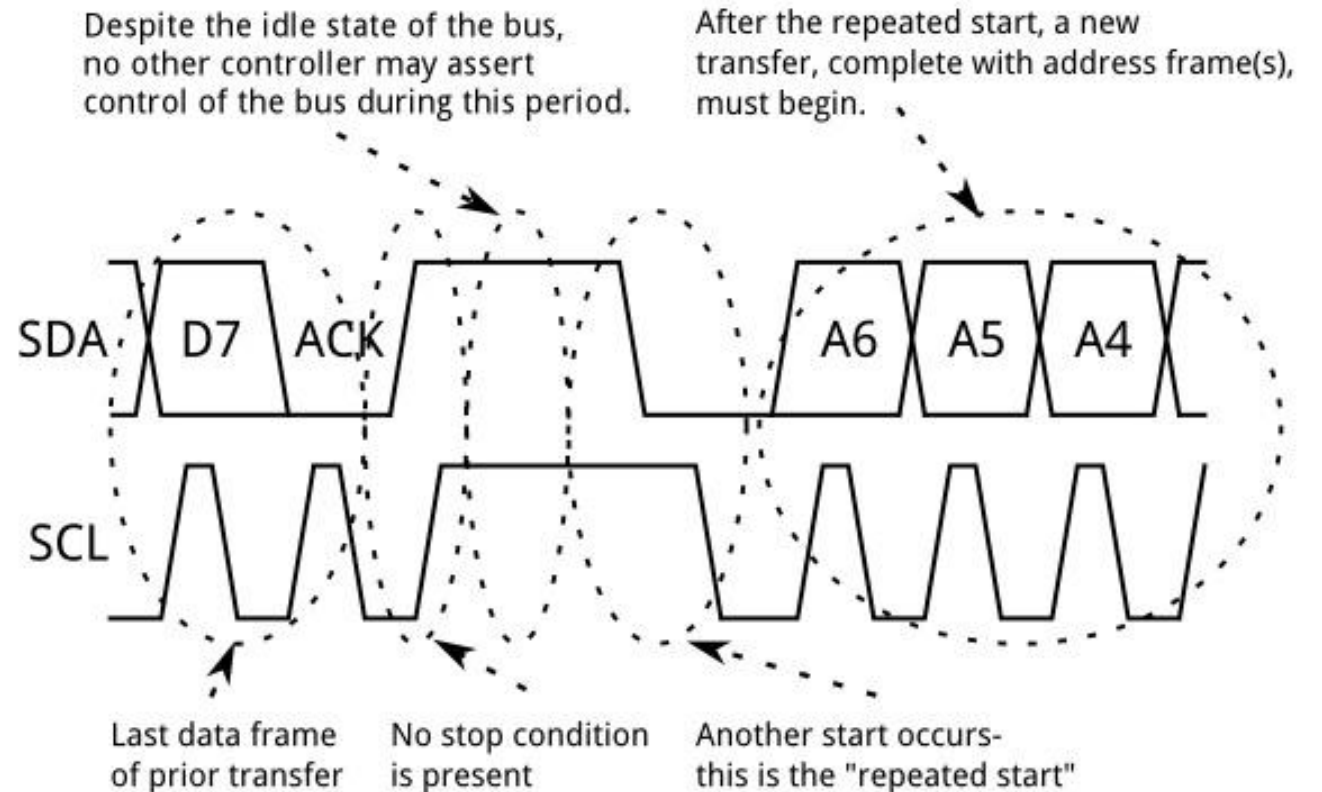
- Arbitration decides which controller gets to proceed if multiple try to communicate simultaneously
- **What happens in I2C if one controller wants a low bit and the other wants a high bit?**

Bus arbitration

- Arbitration decides which controller gets to proceed if multiple try to communicate simultaneously
- **What happens in I2C if one controller wants a low bit and the other wants a high bit?**
 - Low bit wins! (so smaller address or data)
- Each controller constantly checks whether SDA matches the voltage level it expects
 - Stops attempting to transmit if it ever does not
 - (Only actually needs to check high signals)

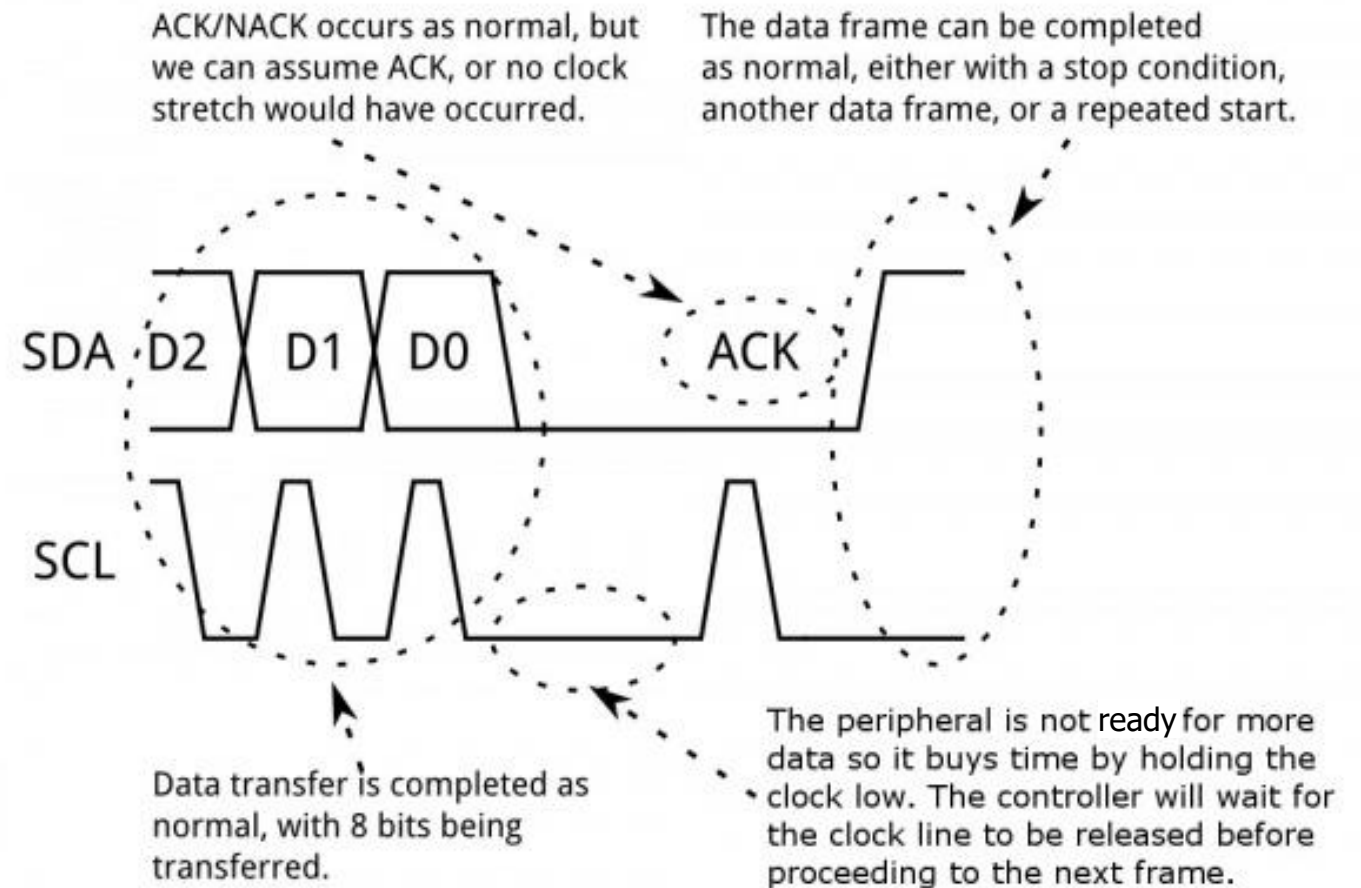
Repeated start conditions

- Repeated start conditions allow the bus to be used again while arbitration was won
- Trigger another Start condition without triggering Stop condition
 - Send address again
- Frequently used for write then read pattern
 - Write which value you want
 - Then repeated start and read

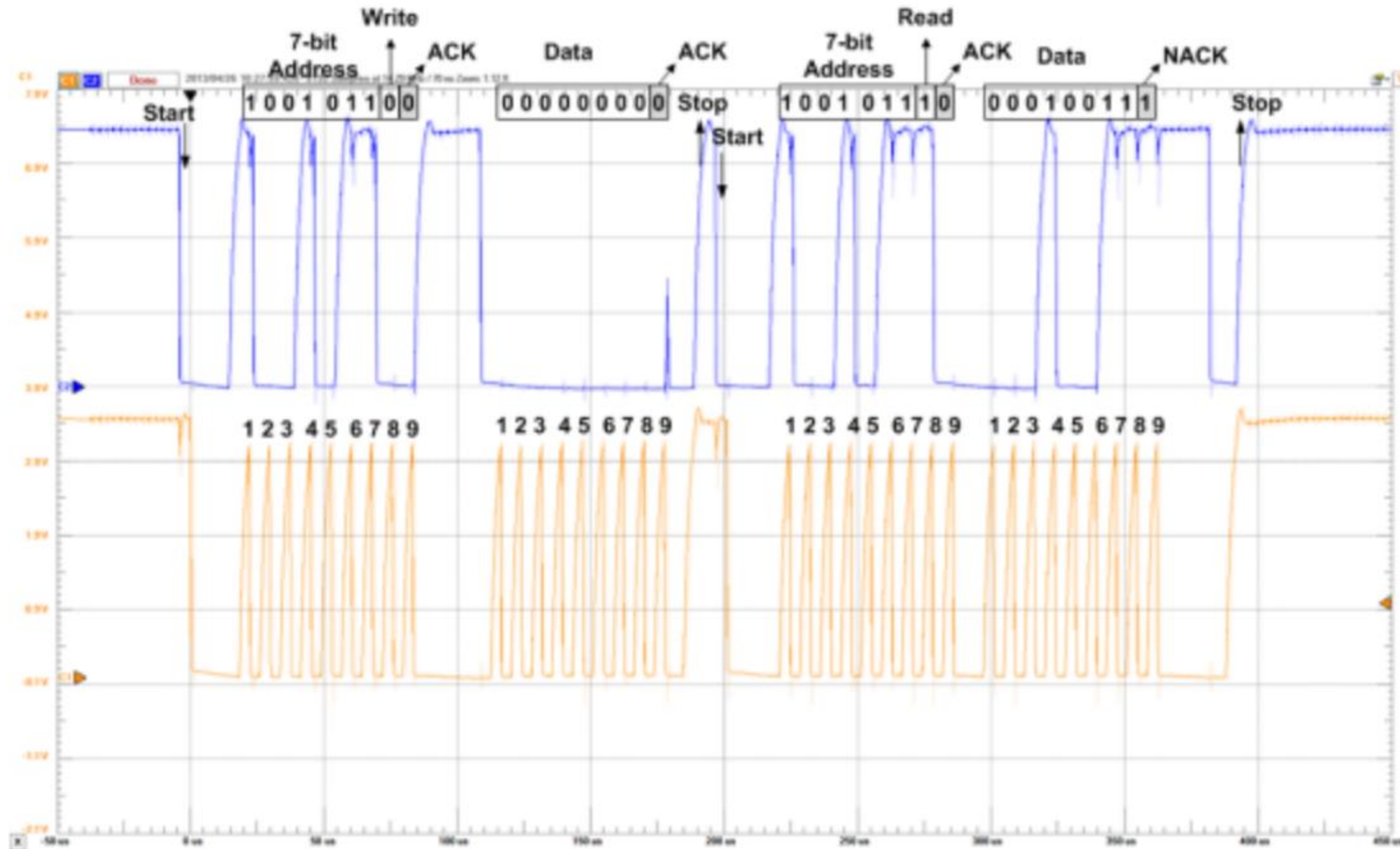


Clock stretching

- Clock is an open-drain line too
 - Either device could keep it low
- Transaction can be briefly paused by holding SCL low

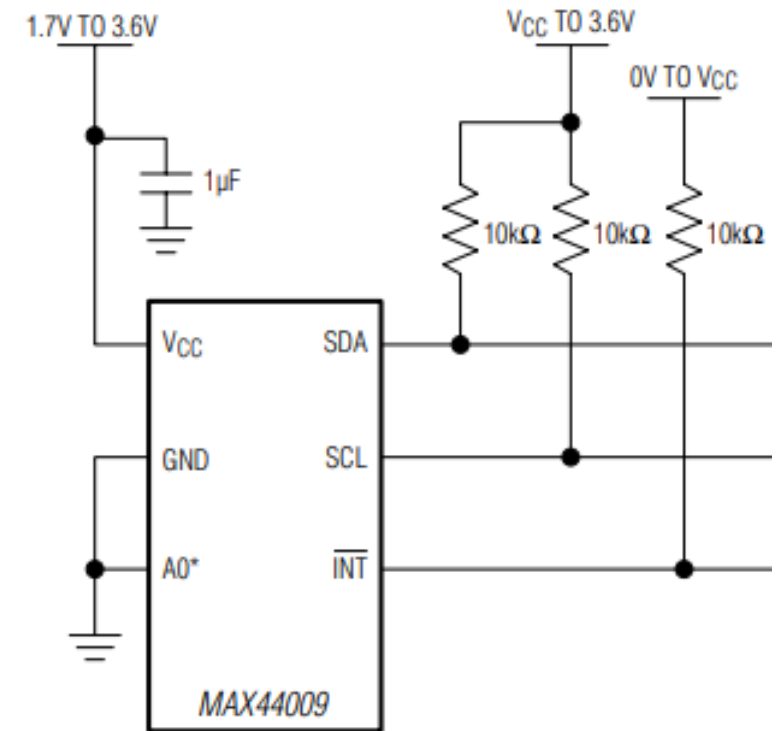


Real-world I2C transactions



Each I2C device on a bus must have a different address

- Shared addresses would cause both to respond
- ICs often have one or more address pin(s) used to select bit(s) of address
 - 0 pins: only one may be on bus
 - 1 pin: two may be on bus
 - 2 pins: four may be on bus
- If no address pins (or not enough), need an I2C address translator chip
 - Translates addresses for one or more peripheral chips

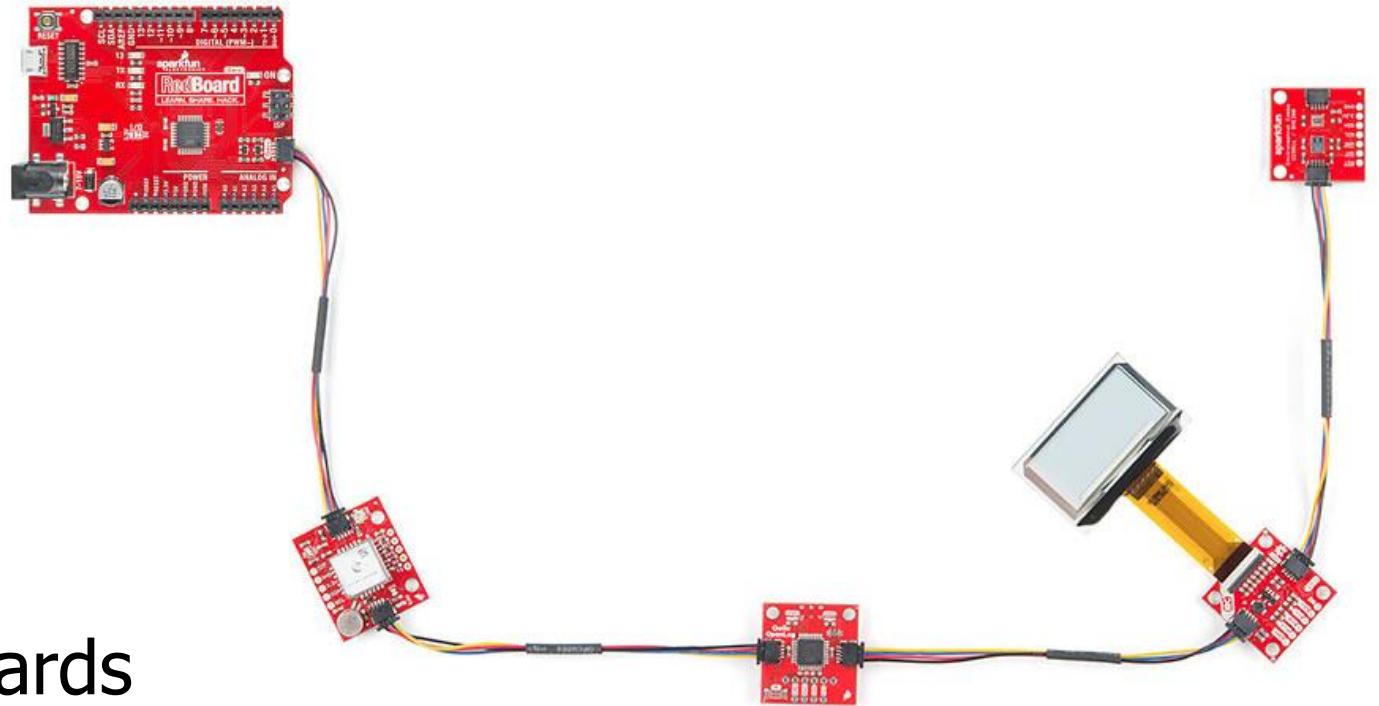


A0 is low: address 1001010x

A0 is high: address 1001011x

Sparkfun Qwiic connect system

- System for wiring multiple prototyping boards together
- Four-pin connector
 - VCC (3.3 volts)
 - Ground
 - SDA
 - SCL
- Daisy-chains through boards
 - Actually connects to chips in parallel as a bus



<https://www.sparkfun.com/qwiic>

System Management Bus (SMBus)

- Related communication specification
 - A little more strict in places, but generally interoperable
- Adds ability to broadcast or unicast messages
 - Generic addresses for Controller and various peripherals (Battery)
- Adds an open-drain shared interrupt signal
 - High-impedance or pull low, just like SDA and SCL
 - Allows any device to alert a controller
 - Controller has to probe bus to determine which device wants attention

I2C use cases

- Various sensors
 - Usually low to medium speed
 - Even relatively high speed stuff often has I2C for convenience
 - Accelerometers and microphones
 - Often with intelligent filtering built in
- Communication between microcontrollers
 - Either can act as the Controller when necessary
- Commonly exists internally within smartphones and laptops too
 - Light sensors, Temperature sensors, etc.

I2C Pros and Cons

- Pros

- Wiring is simple
- Only uses two pins
- Very widely supported

- Cons

- Relatively slow communication rate
- Speed versus power use tradeoff (due to pull-down resistor)
- Open collector makes debugging difficult

Open Question

- Why are SPI and I2C common internally in embedded systems, but not common externally? (like USB, Ethernet, HDMI, etc.)

Open Question

- Why are SPI and I2C common internally in embedded systems, but not common externally? (like USB, Ethernet, HDMI, etc.)
 - Too slow:
 - Especially I2C (100 Kbps compared to 12 Mbps for slowest USB)
 - Not robust:
 - No effort put into the electrical encoding of data or error checking
 - Long external cables lead to additional errors
 - Overall: they're too simple

Outline

- Review
- SPI
- I2C
- **Using SPI and I2C**

Common sensor interaction pattern

- First write one byte to the device
 - This selects what data you want to interact with, called a “register address”
- Second read/write one (or more) bytes
 - This is the actual data
- SPI and I2C devices both work this way
 - Datasheet will have a list of registers you can read/write
 - Each register will have some address: that’s the first byte you write

Example: Microbit accelerometer

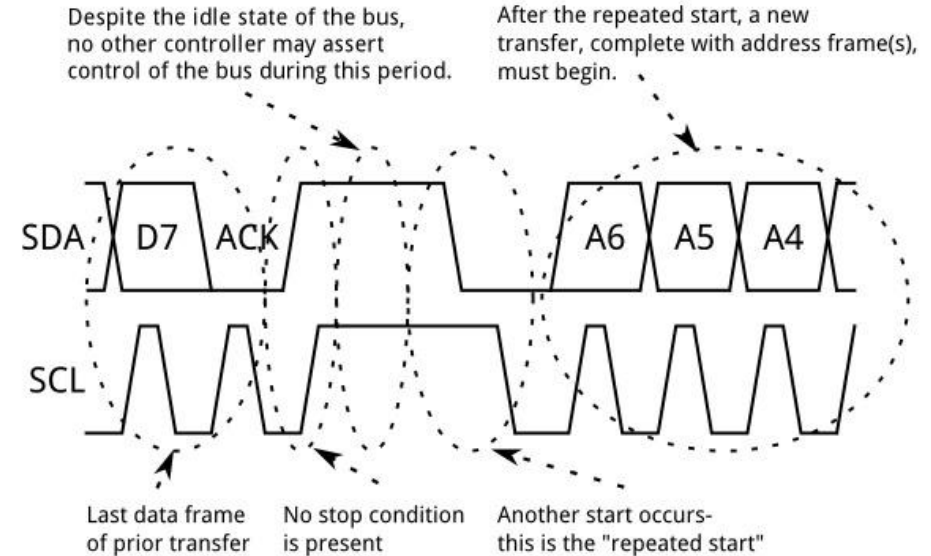
Table 26. Register address map

Name	Type ⁽¹⁾	Register address		Default	Comment
		Hex	Binary		
Reserved		00 - 06			Reserved
STATUS_REG_AUX_A	R	07	000 0111		
Reserved	R	08-0B			Reserved
OUT_TEMP_L_A	R	0C	000 1100	Output	Output registers
OUT_TEMP_H_A	R	0D	000 1101	Output	
INT_COUNTER_REG_A	R	0E	000 1110		
WHO_AM_I_A	R	0F	000 1111	00110011	Dummy register
Reserved		10 - 1E			Reserved
TEMP_CFG_REG_A	R/W	1F	001 1111	00000000	
CTRL_REG1_A	R/W	20	010 0000	00000111	Accelerometer control registers
CTRL_REG2_A	R/W	21	010 0001	00000000	
CTRL_REG3_A	R/W	22	010 0010	00000000	
CTRL_REG4_A	R/W	23	010 0011	00000000	
CTRL_REG5_A	R/W	24	010 0100	00000000	
CTRL_REG6_A	R/W	25	010 0101	00000000	
REFERENCE_DATA_CAPTURE_A	R/W	26	010 0110	00000000	

- Details of each register on later pages show you the structure of the data read or written

Register/data pattern in I2C

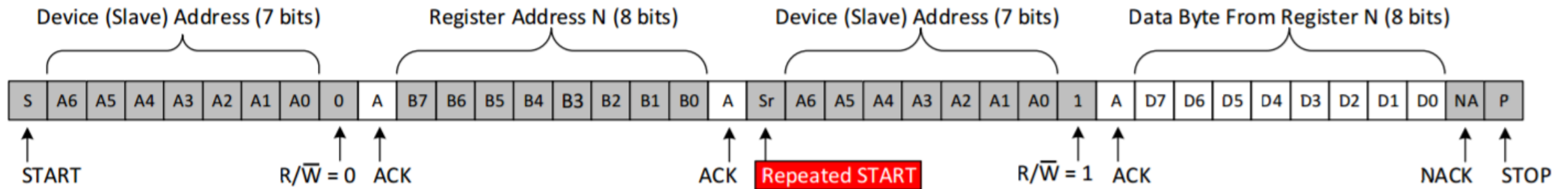
- I2C is the more difficult of these
 - Need some way to tell the device "this transaction is still going", but switch from writing to reading
- This is the use of the "repeated start" option
 - Continues the "transaction"



I2C Read Transaction



- Controller Controls SDA Line
- Peripheral Controls SDA Line

Read From One Register in a Device

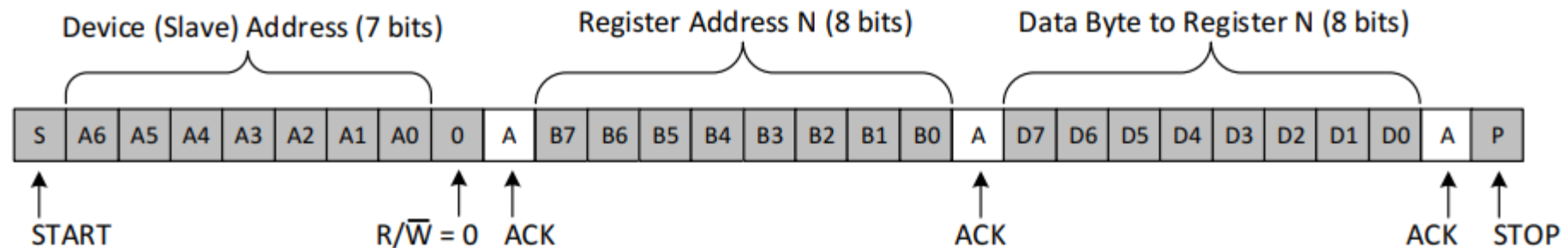


- First, write the address of the register you want
- Then, repeated start
- Finally, read the data from the device

I2C Write Transaction

-  Controller Controls SDA Line
-  Peripheral Controls SDA Line

Write to One Register in a Device



- Just write the data. No need to change modes in the middle
- Some devices also allow “repeated start” in the middle of write transactions
 - But it’s not necessary

nRF I2C Implementation

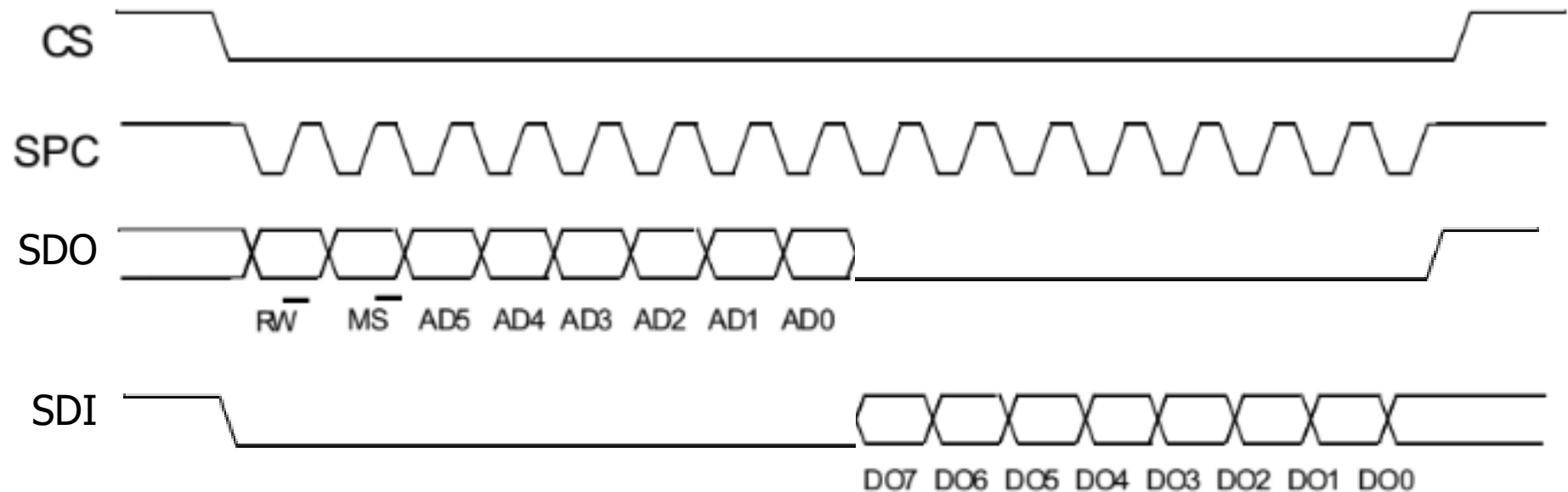
- `nrf_twi_mngr` driver: I2C (Two-Wire Interface) manager
 - Expects transactions to occur and is set up to run those
- Takes in an array of “transfer” operations as an argument
- Each operation is either a read or a write
 - Includes a device address, includes a pointer to data and length
 - Includes flags like `NRF_TWI_MNGR_NO_STOP` which does not execute a stop bit after one operation
- Your job is to set up the array of transfer operations
 - Then the driver will make it happen

Register/data pattern in SPI

- SPI is easier to implement transactions for
 - No indication of reading/writing by default
 - You can just hold Chip Select low and stop clocking if you want to pause
- Need some way to indicate to the peripheral whether you're reading or writing though
 - Possibly different register addresses for read versus write
 - Possibly 7-bit addresses, with a bit leftover for read/write specification
 - Some devices just work uni-directionally
 - Can't read from a screen

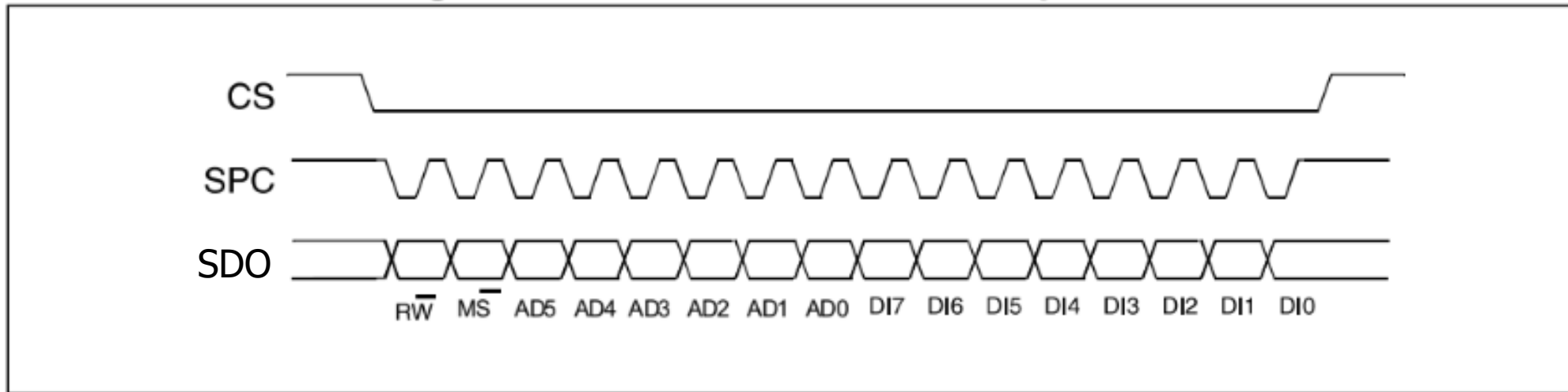
SPI Read Transaction

- Chip select goes low to select the device
- First byte is the register address and read selection
- Next bytes are from the peripheral (read data)



SPI Write Transaction

- Chip select goes low to select the device
- First byte is the register address and write selection
- Next bytes are the data to write



nRF SPI Implementation

- `nrfx_spim` driver: nRF SPI Master (Controller)
- Expects data in "XFER" (transfer) operations
 - Can either be read, write, or read AND write (both simultaneously)
- Flags control whether CS pin goes high afterwards or if it stays low
 - Or you could just manually control the CS pin with GPIO which is usually easier...

Example: ADXL345 accelerometer

- 3-axis accelerometer
 - Pretty high accuracy: 13-bit resolution, +/- 16 g
- Can use either SPI or I2C

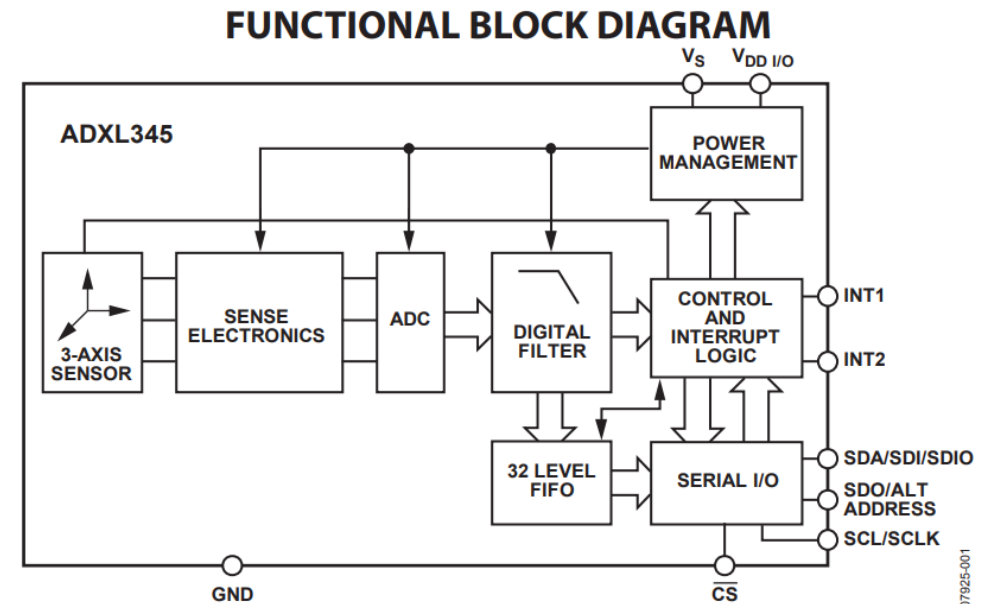
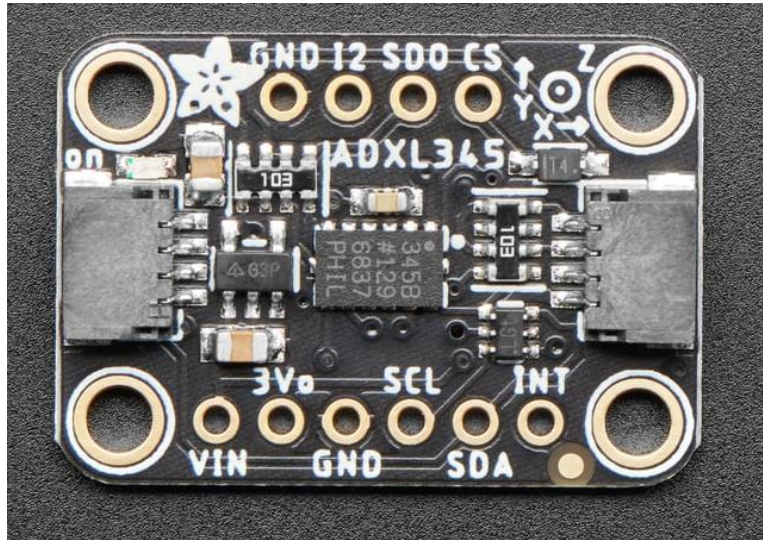


Figure 1.

07925-001

What fields can we access?

- Same “registers” for SPI and I2C
- Transactions:
 - Register address sent first
 - Register value sent second
- Some are read/write while others are read-only

REGISTER MAP

Table 19.

Address		Name	Type	Reset Value	Description
Hex	Dec				
0x00	0	DEVID	R	11100101	Device ID
0x01 to 0x1C	1 to 28	Reserved			Reserved; do not access
0x1D	29	THRESH_TAP	R/W	00000000	Tap threshold
0x1E	30	OFSX	R/W	00000000	X-axis offset
0x1F	31	OFSY	R/W	00000000	Y-axis offset
0x20	32	OFSZ	R/W	00000000	Z-axis offset
0x21	33	DUR	R/W	00000000	Tap duration
0x22	34	Latent	R/W	00000000	Tap latency
0x23	35	Window	R/W	00000000	Tap window
0x24	36	THRESH_ACT	R/W	00000000	Activity threshold
0x25	37	THRESH_INACT	R/W	00000000	Inactivity threshold
0x26	38	TIME_INACT	R/W	00000000	Inactivity time
0x27	39	ACT_INACT_CTL	R/W	00000000	Axis enable control for activity and inactivity detection
0x28	40	THRESH_FF	R/W	00000000	Free-fall threshold
0x29	41	TIME_FF	R/W	00000000	Free-fall time
0x2A	42	TAP_AXES	R/W	00000000	Axis control for single tap/double tap
0x2B	43	ACT_TAP_STATUS	R	00000000	Source of single tap/double tap
0x2C	44	BW_RATE	R/W	00001010	Data rate and power mode control
0x2D	45	POWER_CTL	R/W	00000000	Power-saving features control
0x2E	46	INT_ENABLE	R/W	00000000	Interrupt enable control
0x2F	47	INT_MAP	R/W	00000000	Interrupt mapping control
0x30	48	INT_SOURCE	R	00000010	Source of interrupts
0x31	49	DATA_FORMAT	R/W	00000000	Data format control
0x32	50	DATA0	R	00000000	X-Axis Data 0
0x33	51	DATA1	R	00000000	X-Axis Data 1
0x34	52	DATAY0	R	00000000	Y-Axis Data 0
0x35	53	DATAY1	R	00000000	Y-Axis Data 1
0x36	54	DATAZ0	R	00000000	Z-Axis Data 0
0x37	55	DATAZ1	R	00000000	Z-Axis Data 1
0x38	56	FIFO_CTL	R/W	00000000	FIFO control
0x39	57	FIFO_STATUS	R	00000000	FIFO status

ADXL345 SPI configuration

- Normally uses 4-wire SPI
- Can also use a 3-wire SPI version
 - One wire for in and out
- SPI Config
 - 5 MHz (or slower)
 - CPOL=1 CPHA=1 (ugh, not the default)

SPI bit (bit D6) in the DATA_FORMAT register (Address 0X31) selects 4-wire mode, whereas setting the SPI bit selects 3-wire mode. The maximum SPI clock speed is 5 MHz with 100 pF maximum loading, and the timing scheme follows clock polarity (CPOL) = 1 and clock phase (CPHA) = 1. If power is applied to the ADXL345 before the clock polarity and phase of the host

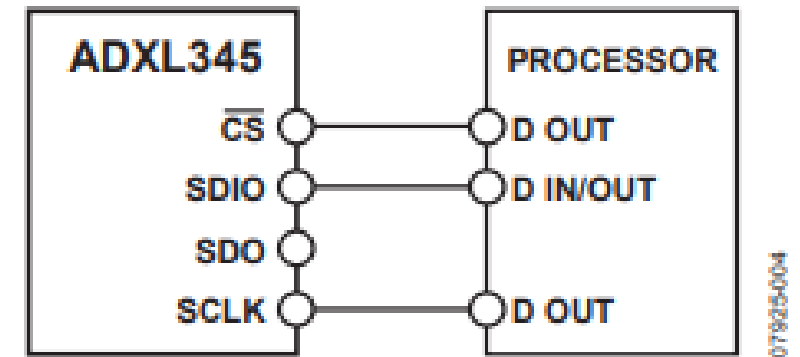


Figure 34. 3-Wire SPI Connection Diagram

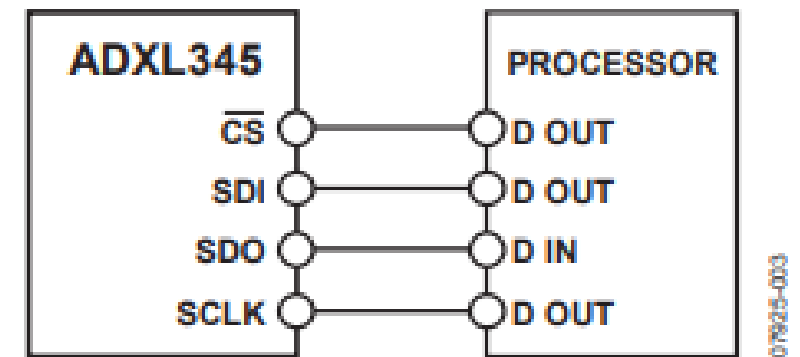


Figure 35. 4-Wire SPI Connection Diagram

ADXL345 SPI transactions

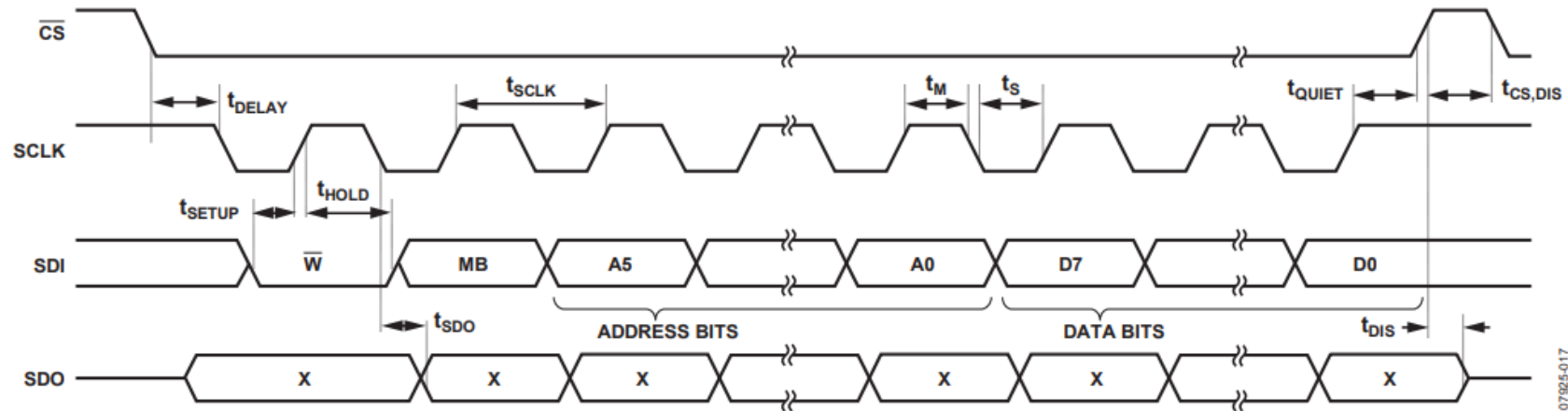


Figure 37. SPI 4-Wire Write

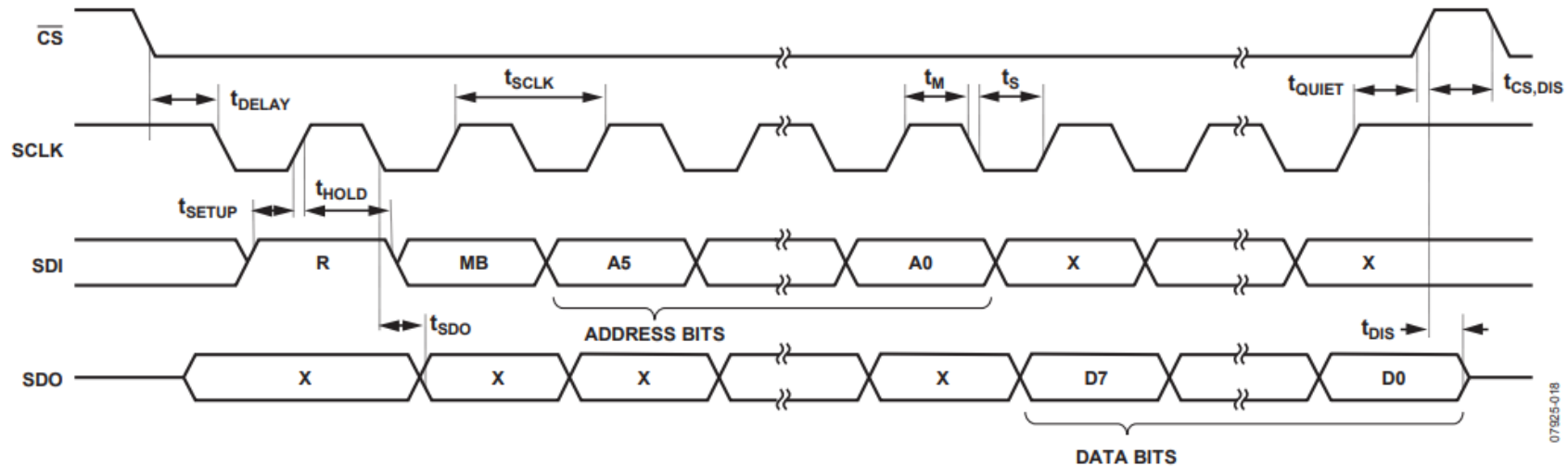


Figure 38. SPI 4-Wire Read

ADXL345 I2C configuration

- Connection of CS pin determines mode
 - Connecting to power puts it in I2C mode
- Supports 100 kHz or 400 kHz mode
- Address: selectable with pin
 - Pin low: 0b1010011 (0x53)
 - Pin high: 0b0011101 (0x1D)

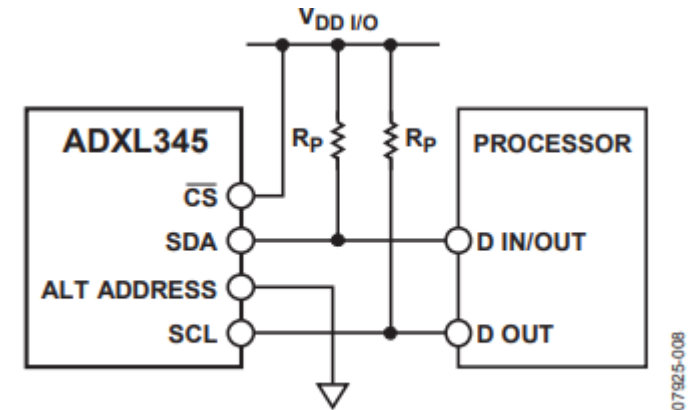


Figure 40. I²C Connection Diagram (Address 0x53)

ADXL345 I2C transactions

SINGLE-BYTE WRITE									
MASTER	START	SLAVE ADDRESS + WRITE		REGISTER ADDRESS		DATA		STOP	
SLAVE			ACK		ACK		ACK		

MULTIPLE-BYTE WRITE									
MASTER	START	SLAVE ADDRESS + WRITE		REGISTER ADDRESS		DATA		DATA	STOP
SLAVE			ACK		ACK		ACK		ACK

SINGLE-BYTE READ									
MASTER	START	SLAVE ADDRESS + WRITE		REGISTER ADDRESS		START ¹	SLAVE ADDRESS + READ		
SLAVE			ACK		ACK			ACK	DATA

MULTIPLE-BYTE READ									
MASTER	START	SLAVE ADDRESS + WRITE		REGISTER ADDRESS		START ¹	SLAVE ADDRESS + READ		
SLAVE			ACK		ACK			ACK	DATA

NOTES

1. THIS START IS EITHER A RESTART OR A STOP FOLLOWED BY A START.
2. THE SHADED AREAS REPRESENT WHEN THE DEVICE IS LISTENING.

Outline

- Review
- SPI
- I2C
- Using SPI and I2C