

Lecture 14

Other Protocols

CE346 – Microprocessor System Design
Branden Ghena – Fall 2023

Some slides borrowed from:
Josiah Hester (Northwestern), Prabal Dutta (UC Berkeley)

Administrivia

- Place hardware orders ASAP!
 - 12 out of 23 groups have done so
 - Can't start your project if you don't have any hardware...

- No lab this Friday! Everybody enjoy your extra time!
 - And use it to work on projects!

- We'll hold open office hours on future Fridays

Administrivia

- Lecture schedule for the rest of the quarter
 - Thursday (11/09) – Wireless Communication
 - Tuesday (11/14) – Nonvolatile Memory & Energy Management
 - Also the final quiz
 - Thursday (11/16) – Microprocessors + Wrapup
 - Tuesday (11/21) – Embedded Systems Research
 - Tuesday before Thanksgiving
 - Tuesday (11/28) & Thursday (11/30) – Project Office Hours

Today's Goals

- Discuss more advanced wired communication protocols
 - With a little less detail
 - Just give a taste of what they are like
- Think about higher-layer concerns like data routing, interpretation, and reliability

Outline

- **USB**
- CAN
- Bit-banging
 - 1-wire
 - Neopixel LED strips
- Video Protocols
 - VGA
 - HDMI

USB references

- USB in a NutShell
 - <https://www.beyondlogic.org/usbnutshell>
- Other stuff I found useful
 - <https://www.usbmadesimple.co.uk/>
 - http://kofa.mmta.arizona.edu/stm32all/blue_pill/usb/an57294.pdf
 - <https://en.wikipedia.org/wiki/USB>

Universal Serial Bus (USB)

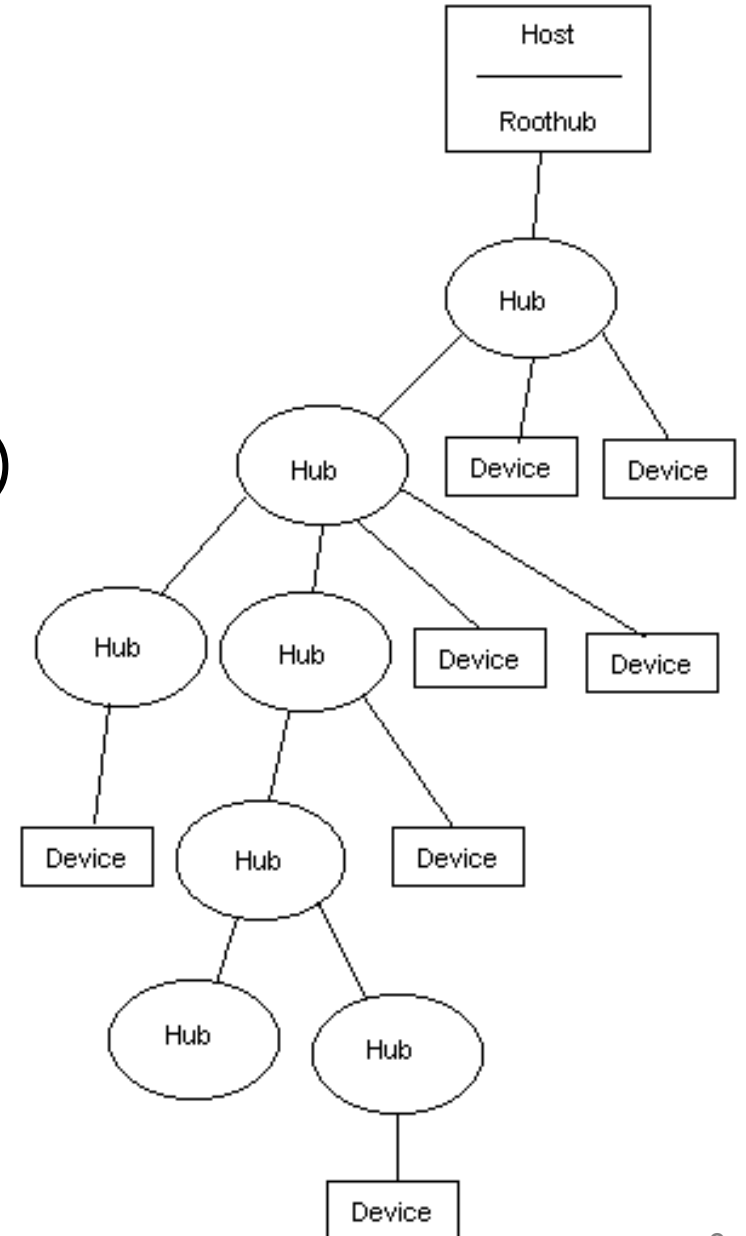
- Pervasive wired communication protocol
 - Universal accurately applies!
 - Targets predominantly external devices over a plug/cable
- Good combination of simple and capable
 - Base version for simple devices does not require too much in terms of pins or resources
 - More complex versions can transfer a significant amount of data
 - These grew organically over time though
- Great support for interoperability
 - Generic device profiles that allowed for plug-and-play
 - Supported by OS initiatives to include driver software

USB is a layered protocol

- USB protocol describes how to:
 - Electrically send bits
 - Send frames of multiple bytes
 - Communicate data between two devices
 - Communicate specific application data (through device classes)
- Much more complicated, compared to others
 - SPI: only how to electrically send bits
 - UART and I2C: how to send frames of bytes

Roles and topology

- Hosts and Devices
 - USB On-The-Go allows host negotiation
 - Added later. Support devices like smartphones
- Host is in charge of communication (“Upstream”)
- Devices provide various capabilities Host can interact with (“Downstream”)
- Tiered star topology
 - Host connects to hubs, which connect to devices
 - Up to 127 devices per hub. Up to 5 layers of hubs



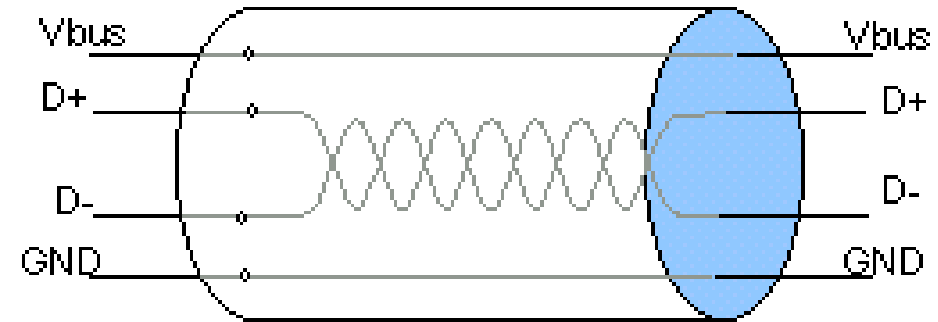
USB Outline

- **Low-layer details**
 - **How are bits sent**
 - **How are packets (collections of bits) sent**
- Higher-layer details
 - How do we interact with devices
 - How do we determine what devices are and how they work

USB signals

- Four signals

- Vbus (5 volts, can power devices)
- D+
- D-
- Ground

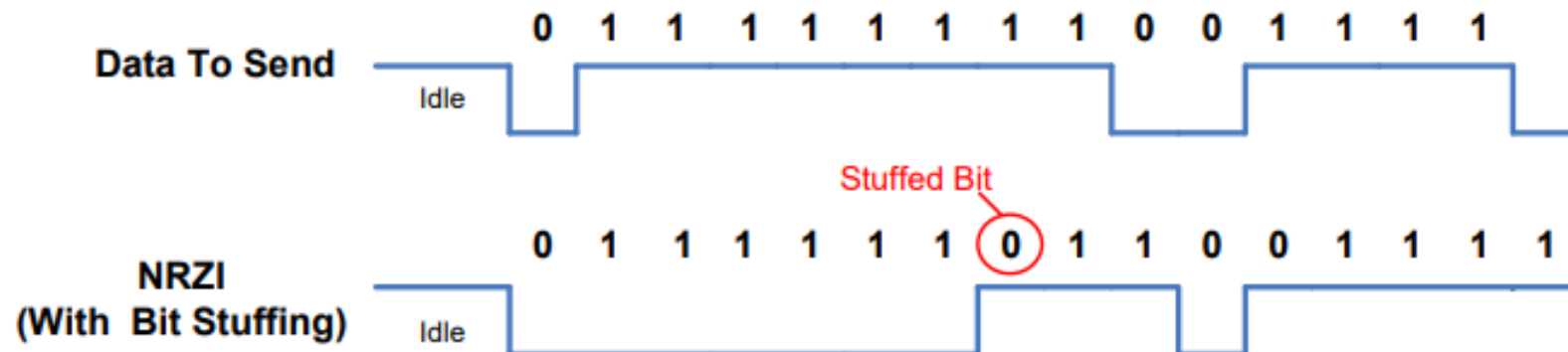


- D+/D- are a *differential pair*

- Signals are inverses of each other
 - Usually, occasionally act separately to signal special conditions
 - Increases voltage difference between states ($5 - -5 = 10$ volts)
- Wires are twisted to avoid interference

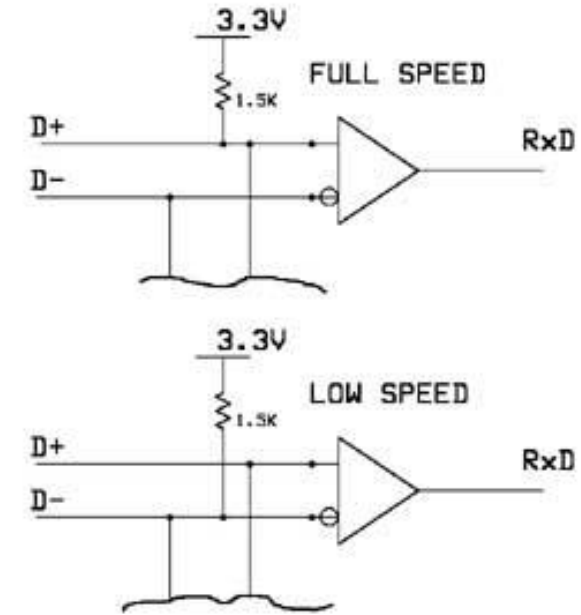
Synchronizing data

- No clock signal!! How is USB so fast?
 - Partially EE magics: better receivers, matched wire impedance
 - Partially easier to distinguish signal states
 - Also guaranteed transitions, which allow resynchronization
- Transitions are used to denote data (non-return-to-zero inverted)
 - With guaranteed transition in within every 8 bits (bit stuffing)
 - Allows clocks on the two devices to synchronize



USB speeds

- USB 1.0
 - Low Speed: 1.5 Mbps
 - Not clear if this is used anymore
 - Full Speed: 12 Mbps
 - Microcontrollers tend to support Full Speed
 - We're focusing on details from it
- USB 2.0
 - High Speed: 480 Mbps
- USB 3.0+
 - Super Speed: 5-20 Gbps
 - Adds multiple parallel data connections



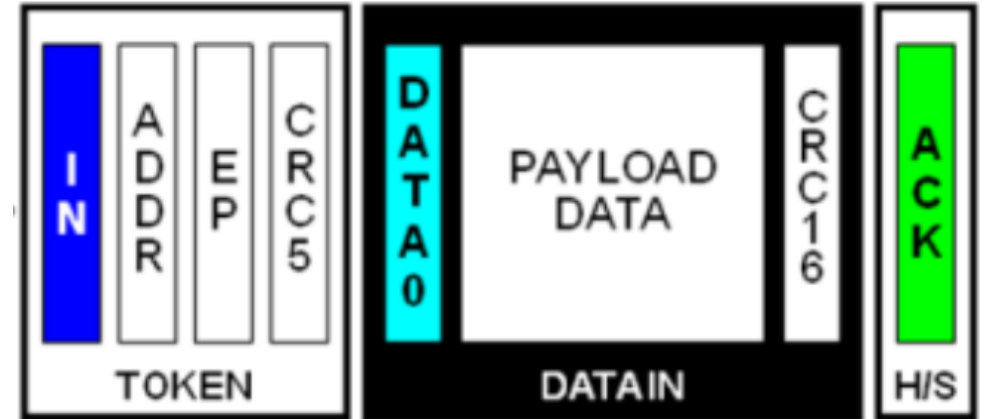
- Pull-up resistors allow for detection of a plugged device
- Also identify speed

USB interactions

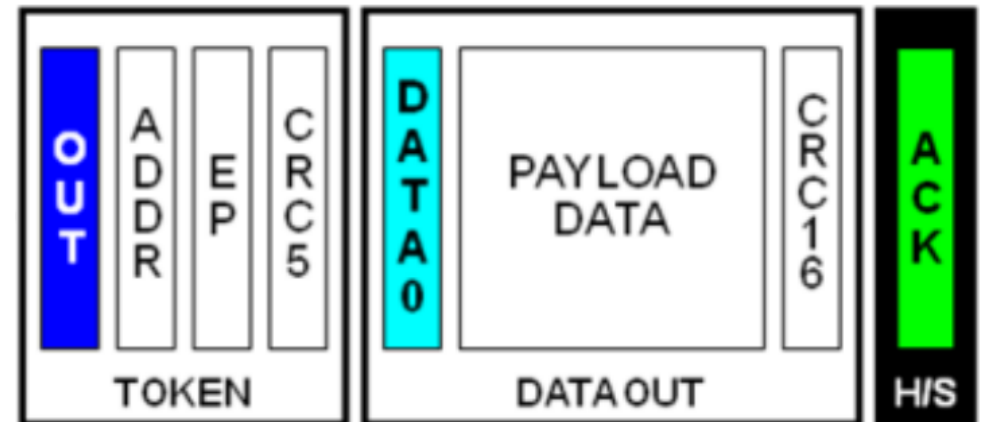
- General transaction format
 1. Host sends a Token packet: identifies transfer direction and device
 2. Host or Device send data depending on direction
 3. Other side acknowledges receipt of data

- Like a maxed-out version of the I2C transaction pattern
 - Host *always* initiates communication

Reading data from Device



Writing data to Device



USB token packets

- Packet fields
 - Sync field, allows transmitter and receiver clocks to synchronize
 - Packet ID, determines what type of packet is being sent
 - Token type: Setup device, Read from device, or Write to device
 - Address+Endpoint to identify Device
 - CRC, (Cyclical Redundancy Check) to detect bit errors
 - 5-bit CRC

USB data packets

- Packet fields
 - Sync field, allows transmitter and receiver clocks to synchronize
 - Packet ID, determines what type of packet is being sent
 - Data: application data
 - Data, up to 1023 bytes (full speed, often capped at 64 for microcontrollers)
 - CRC, (Cyclical Redundancy Check) to detect bit errors
 - 16-bit CRC

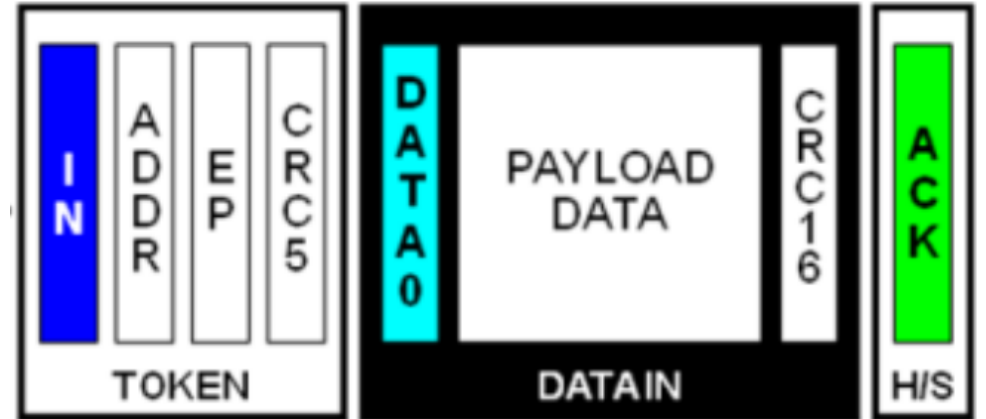
Cyclic Redundancy Check (CRC)

- Determines if the data received matches the data sent
 - CRC value is calculated on original data and appended to message
 - CRC value is recalculated on the received data
 - Value appended to message and value recalculated MUST match
- Essentially some kind of hash operation
 - Turns many bits into some smaller number of bits that are unique-ish
- CRC algorithms are:
 - Particularly good at single bit errors AND contiguous bit errors
 - Relatively simple to calculate
 - Very widely used in communication

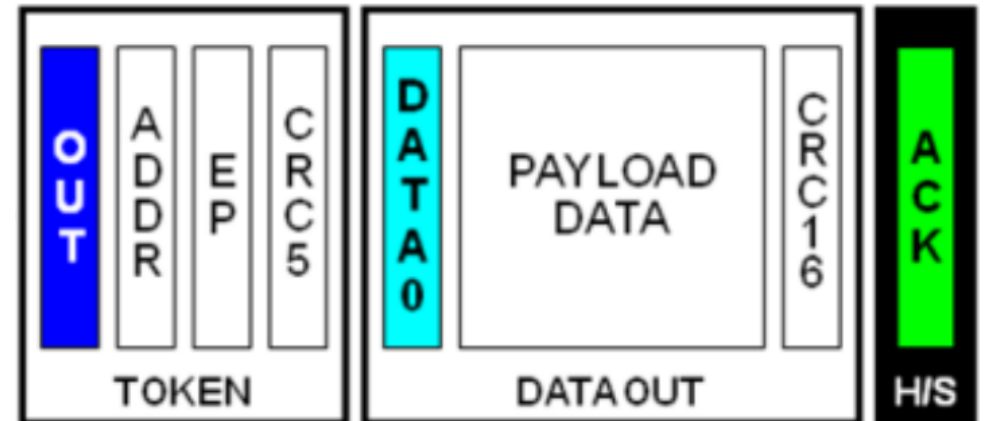
Break + Question

- Why have two CRC values?

Reading data from Device



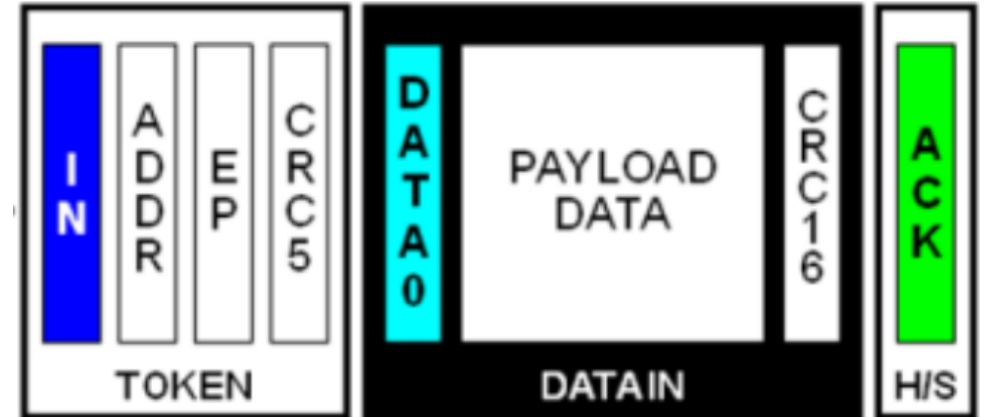
Writing data to Device



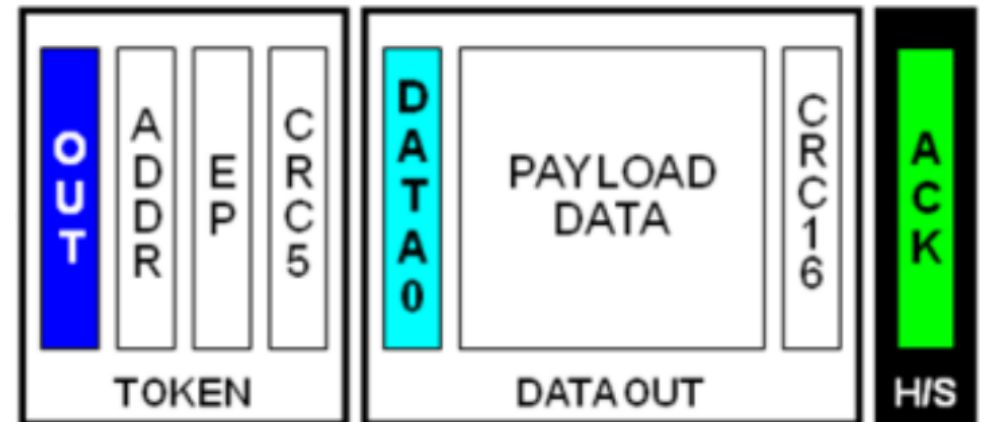
Break + Question

- Why have two CRC values?
 - Devices that aren't addressed want to be able to determine that right away without reading all of the data
 - For reading, data is sent by two different devices, so it needs two CRCs
 - Keep writes the same for symmetry

Reading data from Device



Writing data to Device

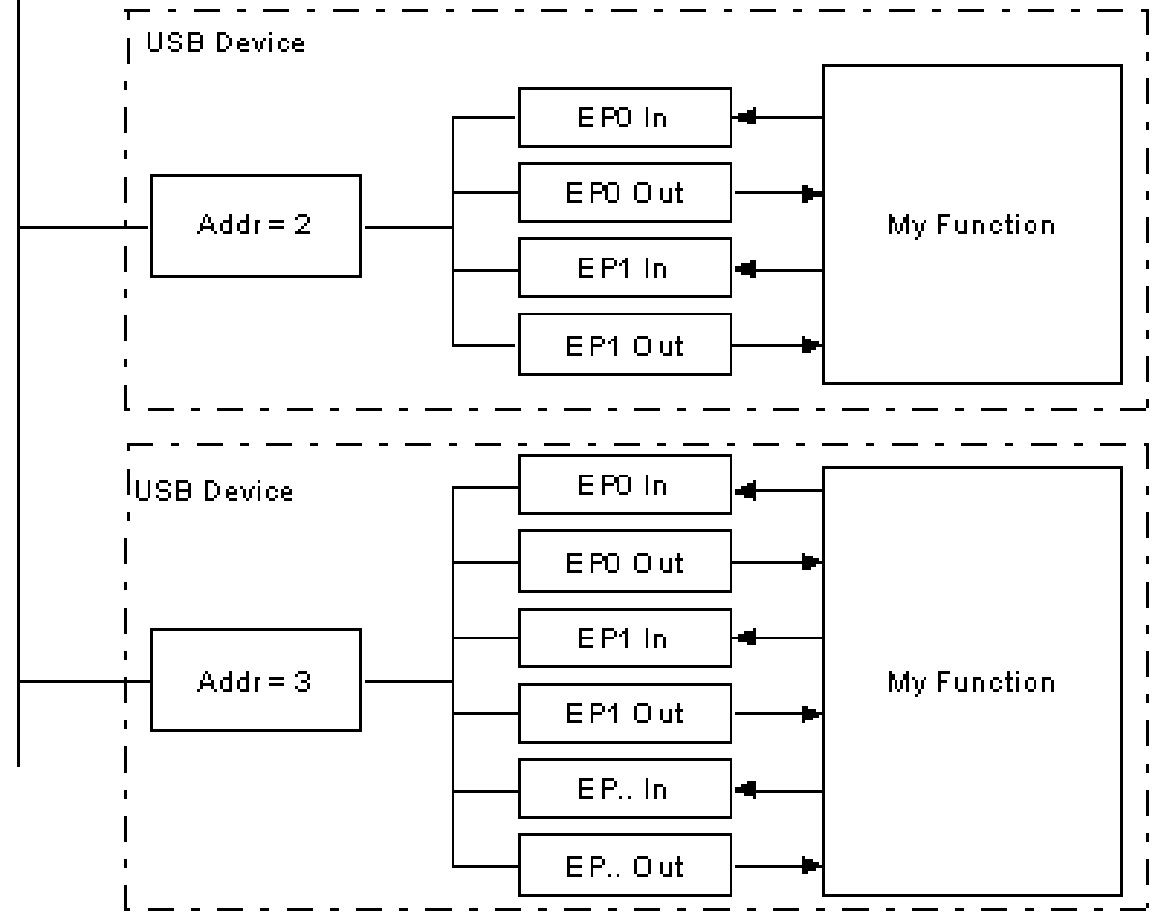
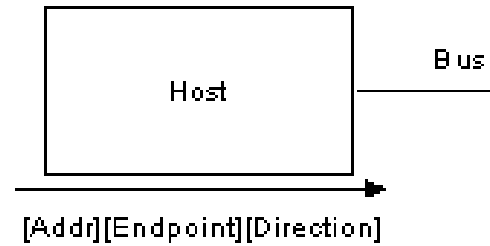


USB Outline

- Low-layer details
 - How are bits sent
 - How are packets (collections of bits) sent
- **Higher-layer details**
 - **How do we interact with devices**
 - **How do we determine what devices are and how they work**

Interacting with USB devices

- Each Device is given a separate address on the bus
- Each Device also has a number of Endpoints
 - Logical communication channels
 - Direct data and guide communication patterns

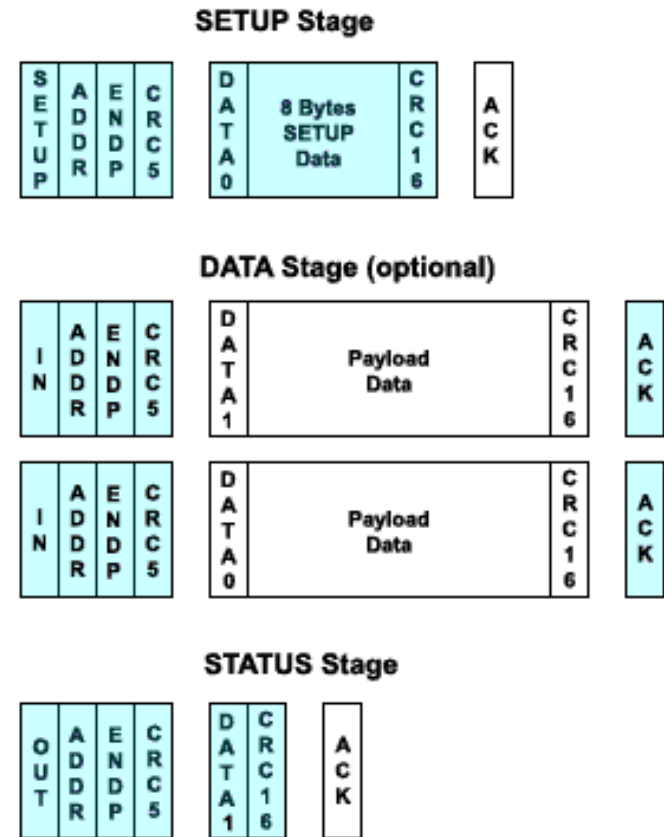


USB endpoint types

- Interrupt transfers
 - Guaranteed latency, small amounts of data
 - Important sensor data (mice and keyboards)
 - Polled frequently by Host
- Bulk transfers
 - Sporadic large transfers, reliable communication
 - General reading/writing of data (flash drives and USB serial)
 - Polled by Host whenever there is available bandwidth
- Isochronous transfers
 - Guaranteed data rate, unreliable communication
 - Continuous data streaming (audio and webcams)
 - Polled frequently by host

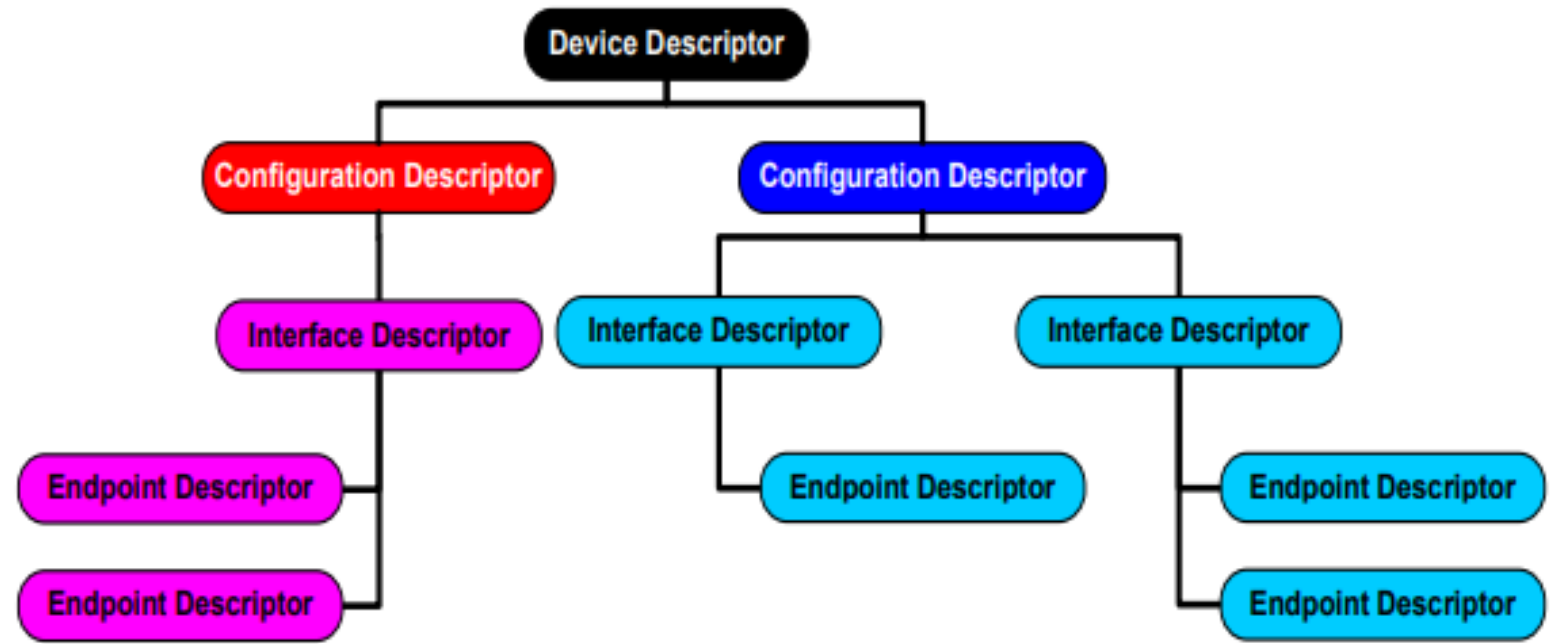
USB control endpoint

- Every USB Device has a special Control endpoint as well
- Used for setting up the USB Device driver on the Host
- Initializing a Device
 - Host sends SETUP transaction requesting device descriptor
 - Host performs IN transaction to read device descriptor
 - Host performs OUT transaction to write device status



USB device descriptors

- Packed version of tree structure describing the device
 - Interfaces it provides
 - Endpoints associated with each interface



Example Microbit


- Interface: Communications, Abstract (modem), CDC
 - Endpoint: 3, IN, Interrupt
- Interface: CDC Data, CDC DATA interface
 - Endpoint: 1, IN, Bulk
 - Endpoint: 2, OUT, Bulk
- Interface: Vendor Specific Class, Subclass, Protocol
 - Endpoint: 5, IN, Bulk
 - Endpoint: 4, OUT, Bulk
- Interface: Mass Storage, SCSI, MSD interface
 - Endpoint: 7, IN, Bulk
 - Endpoint: 6, OUT, Bulk



Virtual serial device



SEGGER JTAG interface



USB external filesystem

lsusb output

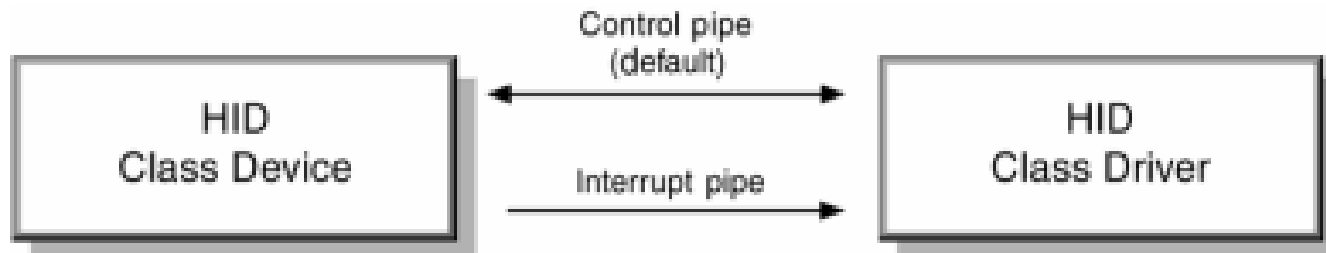
- `lsusb`
 - List USB devices
- Combine with `-s` flag to select a single device
- Combine with `-v` flag for verbose mode with more information

Minimal virtual serial USB Device

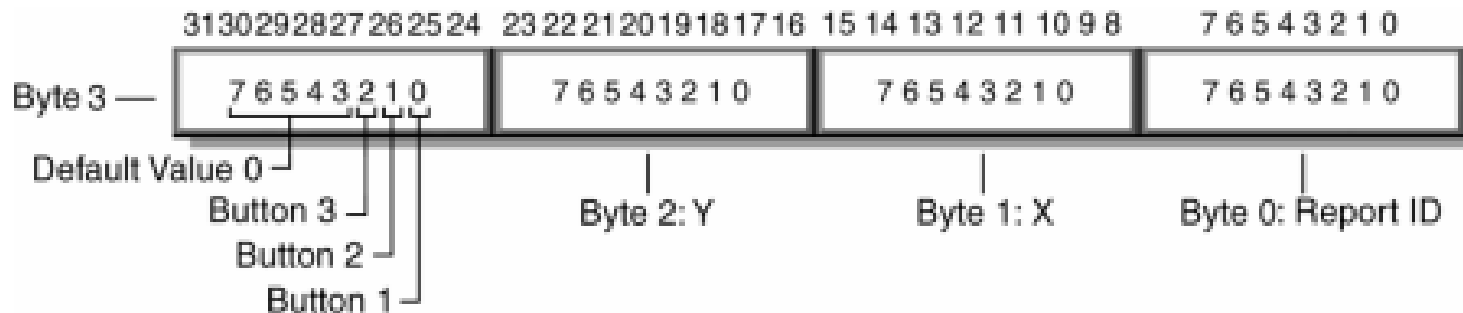
- Virtual Serial Device
 - Endpoint 0: Control, IN/OUT
 - Respond to IN requests by setting up OUT with a buffer of descriptor data of the correct size
 - Endpoint 1: Interrupt, IN
 - Needed for serial modem controls, just ignore it
 - Endpoint 2: Bulk, OUT
 - Connect to buffer from `_write()` (just takes raw characters)
 - Endpoint 3: Bulk, IN
 - Connect buffer to `_read()` (just provides raw characters)

HID USB Device (Human Interface Device)

- Used for human interaction devices, like keyboard/mouse



- “Report” structure is provided over Interrupt IN endpoint
 - Or on demand via Control IN endpoint



Example mouse with x,y and three buttons

USB summary

- Specification for fast data communication
- Specification for interacting with abstract device types
 - Connects correct driver to interpret and send data
- Pros
 - Very fast
 - Very interoperable
- Cons
 - Hardware and software are way more complex than simple protocols like UART, SPI, and I2C
 - Not very energy efficient

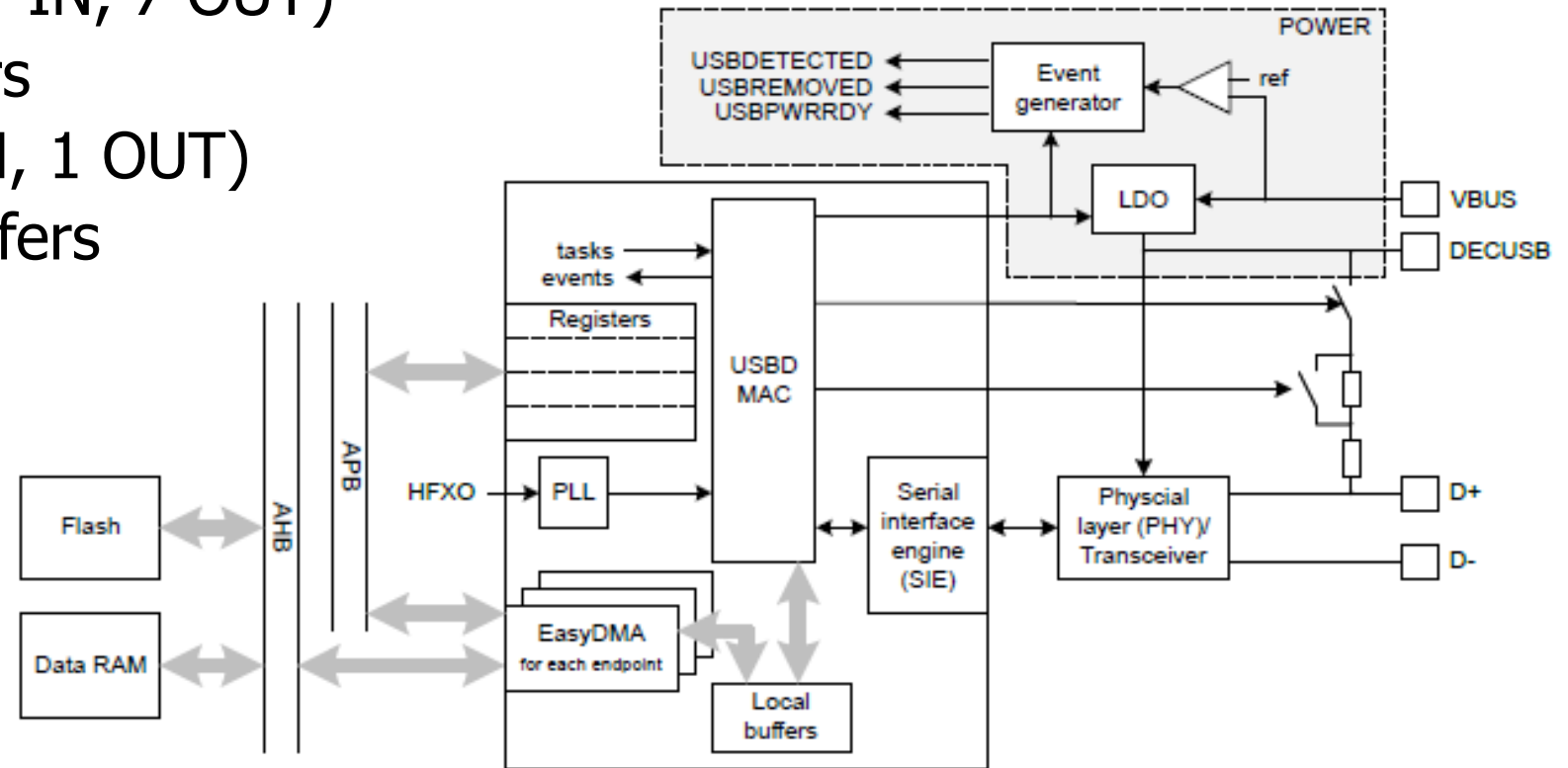
nRF52 USBD

- Implements USB Device (**not Host**)

- Control endpoint
- 14 bulk/interrupt (7 IN, 7 OUT)
 - 64-byte transfers
- 2 isochronous (1 IN, 1 OUT)
 - 1023-byte transfers

- Full-speed USB

- With 5 volt signals



Break + Question

- What are the ramifications of many USB devices sharing a bus?
 - Consider: throughput and latency

- What if I really had 127 USB mice on a single USB hub?
 - What if it was microphones instead?

Break + Question

- What are the ramifications of many USB devices sharing a bus?
 - Consider: throughput and latency
 - Devices need to share bandwidth
 - Also devices are “polled” one-at-a-time
- What if I really had 127 USB mice on a single USB hub?
 - What if it was microphones instead?
 - You might saturate the data throughput capabilities!

Outline

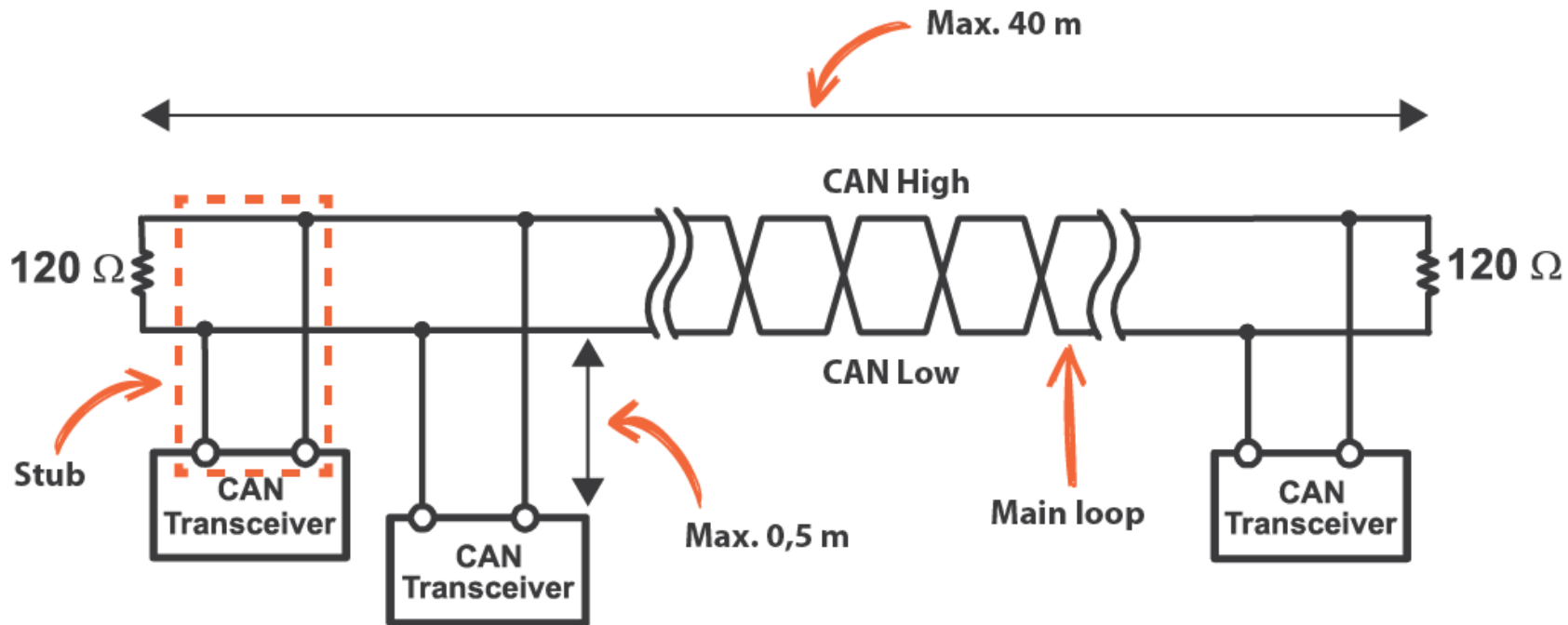
- USB
- **CAN**
- Bit-banging
 - 1-wire
 - Neopixel LED strips
- Video Protocols
 - VGA
 - HDMI

Controller Area Network (CAN bus)

- Designed for highly reliable interactions within a vehicle
- Multi-master with arbitration
 - Similar to I2C
- Mechanism for sending messages with “identifiers”
 - Identifies the data in the message, not the device its for
 - Lower value identifiers have high priority
 - All messages are received by all CAN nodes
 - Which can decide at higher levels which identifiers they care about

CAN physical connections

- Two differential, wired-AND signal lines
 - Transitions are used to transmit bits (non-return-to-zero) with bit-stuffing
 - Combines aspects of USB and I2C
 - 125 kHz – 5 Mbps speeds



CAN packet format



- 11-bit identifier
 - Check bits as they are sent to see if you win arbitration
- Up to 8 bytes (64 bits) of data
 - Very small messages!
- CRC for checking
- Acknowledgement
 - Like I2C, let the line float and see if another device responds
 - If not, explicitly retransmit!

CAN message types

- Data frame
 - Transmission of data for a certain identifier
- Remote frame
 - Requests data transmission of a certain identifier
- Error frame
 - Transmitted when an error is detected with the previous message
- Overload frame
 - Transmitted by a node that is too busy to respond right now

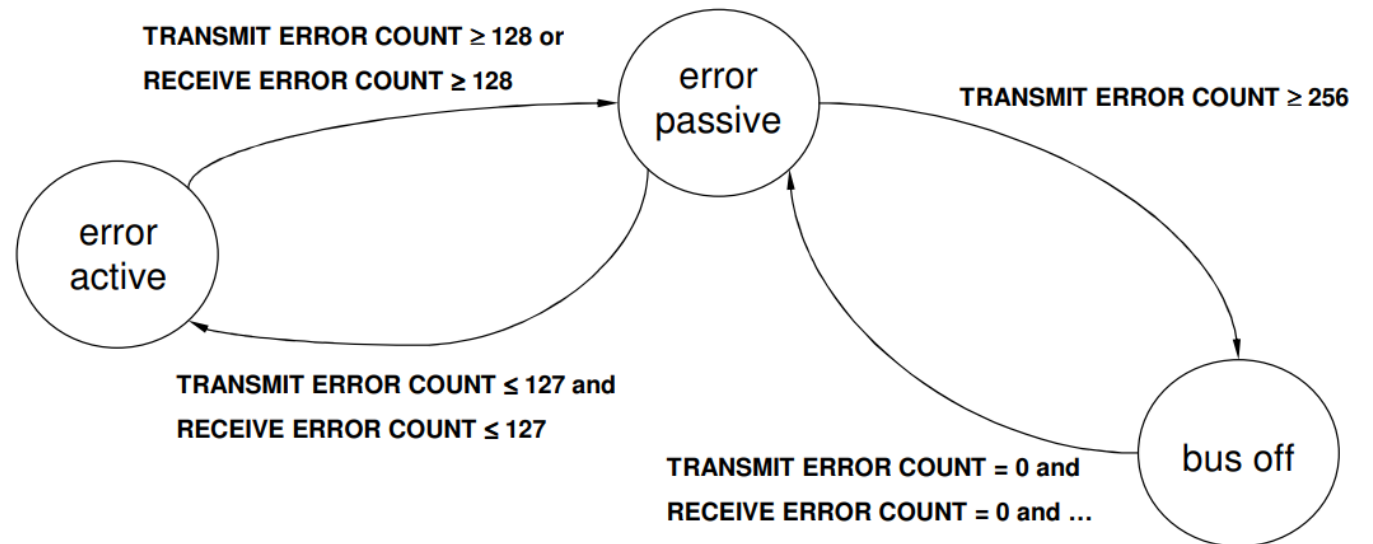
CAN reliability design – detecting errors

- Check for errors everywhere and appropriately handle
 - Bit error
 - If the value found on the bus differs from the one sent
 - Stuff error
 - If 6 consecutive bits of the same type are found
 - CRC error
 - If CRC does not match
 - Form error
 - Format field has unexpected values
 - Acknowledgement error
 - No ACK received
- Devices detecting an error broadcast a message signifying it!
 - Multiple devices sending the same message works without arbitration loss
 - Previous message is then retransmitted

CAN reliability design – handling errors

- Each node accepts the possibility that maybe it is the faulty one
- Track errors and successes and change device state
 - Passive: limited error signaling and transmissions
 - Bus off: does not transmit in any way

- Idea is that the CAN controller hardware can be faulty but still detect it in some cases



CAN summary

- Designed for reliable vehicular communication
- Multi-master bus with serial communication

- Pros
 - Highly reliable
 - Extensible to many devices
- Cons
 - Special-purpose design. Whole system has to agree on identifiers
 - Relatively slower throughput

Break + Open Question

- Why is CAN a good choice for vehicles (historically)?
- What problems is CAN facing today?

Break + Open Question

- Why is CAN a good choice for vehicles (historically)?
 - High reliability
 - A few low-data-rate sensors scattered throughout the car
- What problems is CAN facing today?
 - Camera data doesn't fit in 8-byte packets...

Outline

- USB
- CAN
- **Bit-banging**
 - **1-wire**
 - **Neopixel LED strips**
- Video Protocols
 - VGA
 - HDMI

Controlling a protocol without a peripheral

- If your microcontroller doesn't have a hardware peripheral to control a certain protocol, you could still emulate it
 - Known as bit-banging
- Example: use GPIO to create a UART
 - Set high/low values with certain timings to make start, data, and stop bits
 - Probably based on an interrupt if microcontroller is fast enough
 - Or maybe is hard-coded with delays
 - For reading the UART, instead read bits from the bus at a given rate
 - Expect start/stop bits, but ignore them

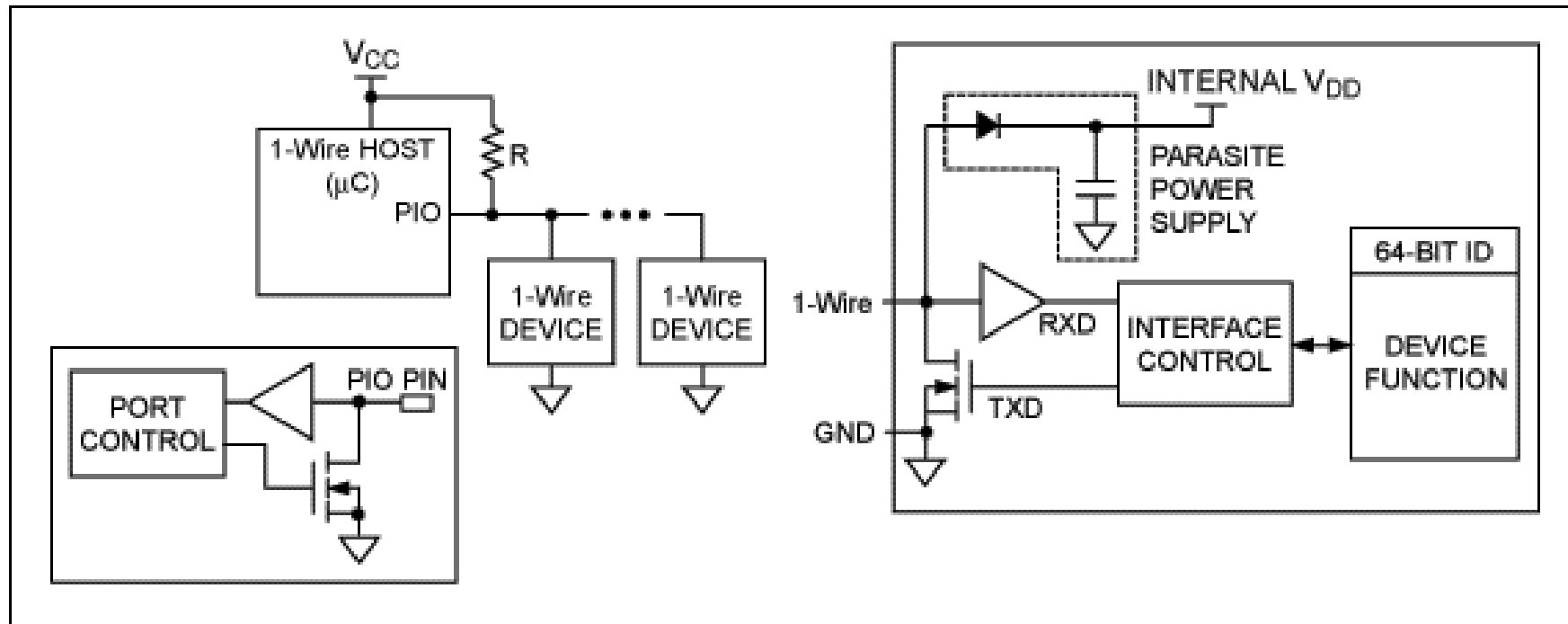
Two bit-banging examples

1. 1-Wire Protocol

2. Neopixel LED strip

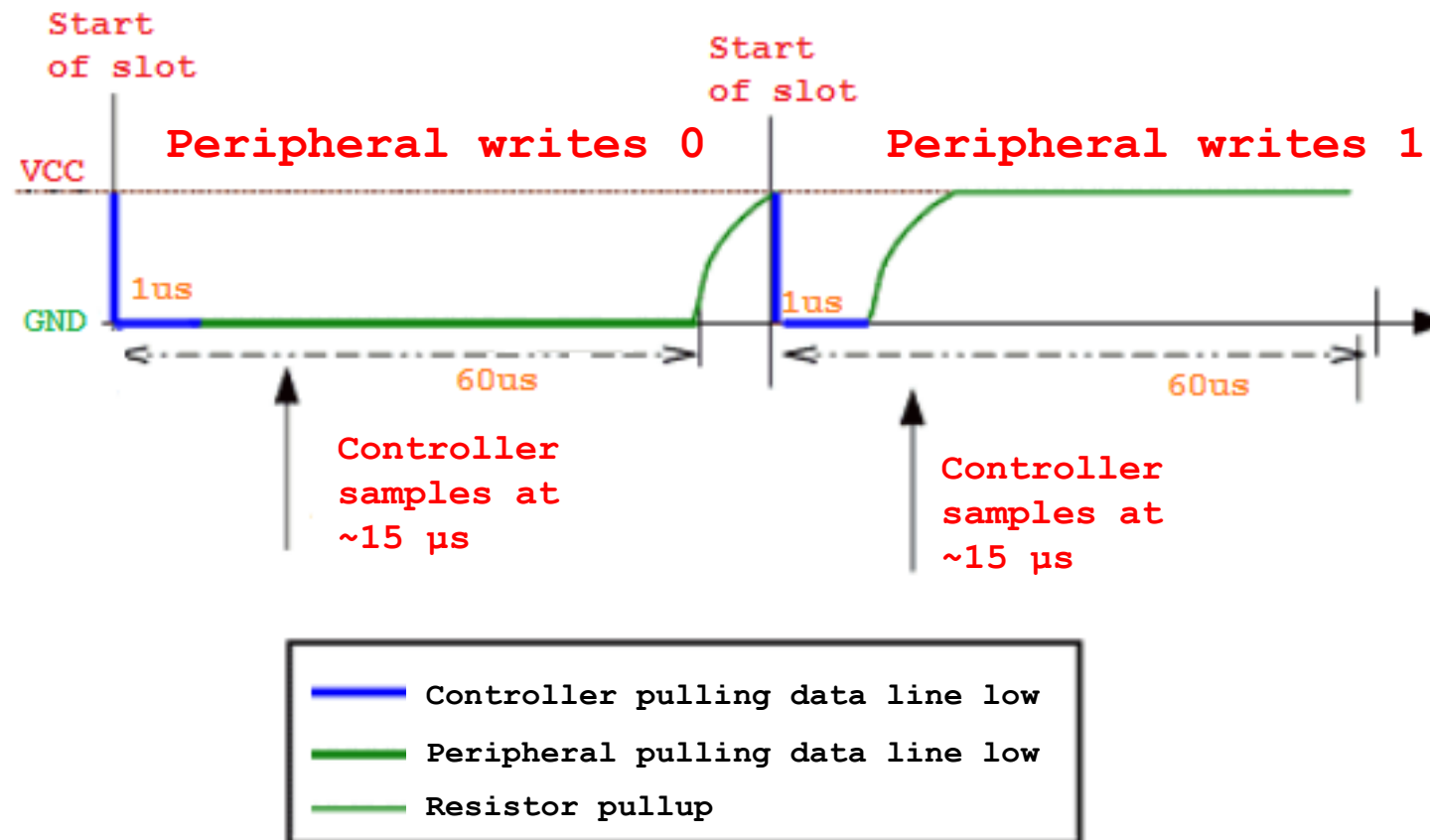
1-Wire Protocol

- Similar to I2C, but without a clock line
- Devices can optionally power themselves off the data line



1-wire read operations

- Controller drives line low, then releases samples 15 μ s later
 - Falling edge wakes up peripheral devices
 - Peripheral device either drives low for a 0 or also releases for a 1



Controlling 1-wire with a UART

- 1-wire is not supported by most microcontrollers
 - Instead, it must be emulated either with GPIO or with another protocol
- Example: UART plus an external transistor can emulate 1-wire
 - Run UART way faster than 1-wire: each UART byte is one 1-wire bit
 - Connect TX to a transistor which can connect to Ground or disconnect
 - Connect RX to the data line
- To read, write data 0xFF
 - Start bit pulls line low to trigger peripheral device
 - 8 data bits do not affect data line (disconnected)
 - RX line will match 0xFF for reading a 1, or will be anything else for a 0

Two bit-banging examples

1. 1-Wire Protocol

- 2. Neopixel LED strip**

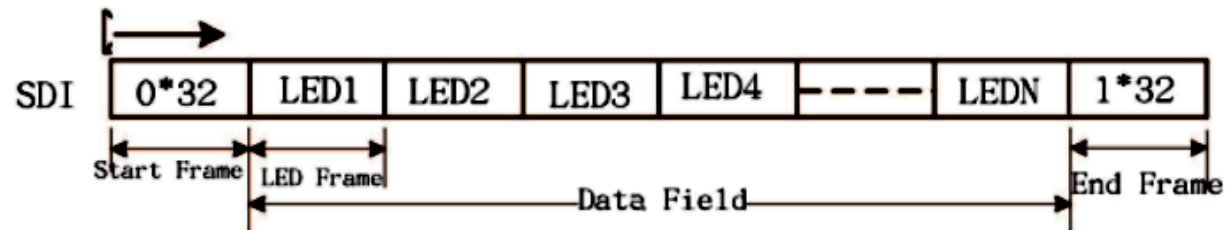
Neopixel LED strips

- LEDs on a long strip connected in series
 - Often capable of RGB control
- Custom protocol for driving them

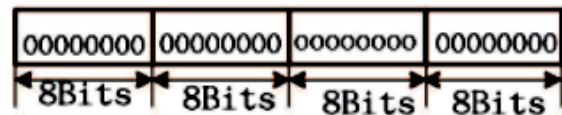


Neopixel protocol

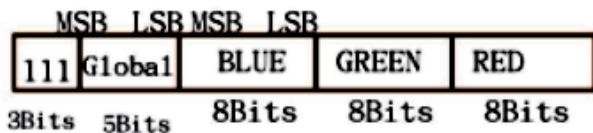
- 32 bits for each LED
 - After the first LED gets its data, all the next bits are forwarded down the chain



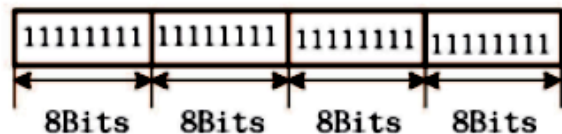
Start Frame 32 Bits



LED Frame 32 Bits

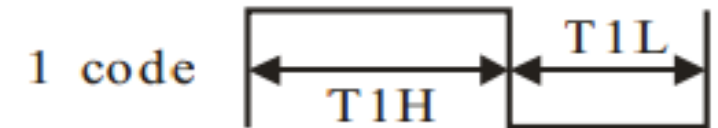
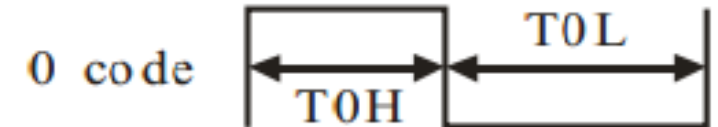


End Frame 32 Bits



Each bit is represented by an amount of time high and low

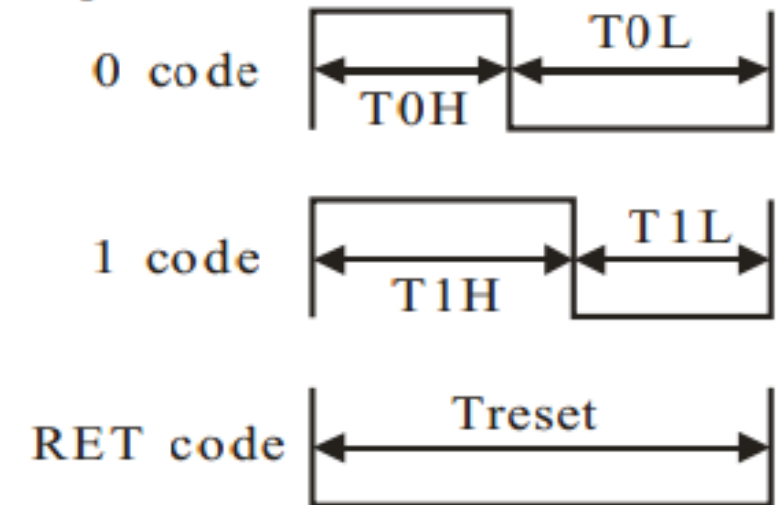
Sequence chart:



Emulating Neopixel protocol with SPI

- Idea: configure SPI speed so N bits line up with the period of the bit
 - For example, 8 bits per period
- To send a 0 bit: 0xE0 (0b11100000)
- To send a 1 bit: 0xF8 (0b11111000)
- From the Neopixel perspective, that data is just the square wave it wanted

Sequence chart:

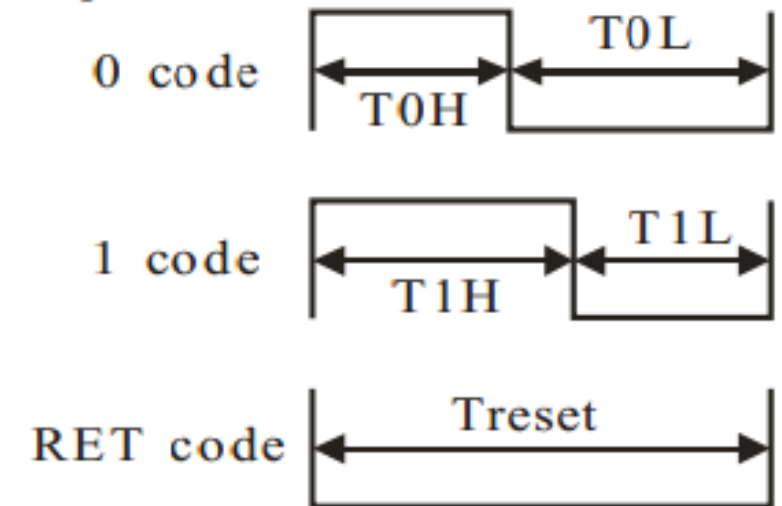


Warning: I don't promise the actual numbers here are correct. Just the overall idea. Recalculate the numbers yourself based on timing requirements

Emulating Neopixel protocol with PWM

- Idea: these are just differing duty-cycle square waves i.e., PWM
 - Set PWM period to equal the period for sending a bit
 - Needs to be a left-aligned PWM signal
- To send a 0 bit: 30% duty cycle
- To send a 1 bit: 70% duty cycle
- Driver translates colors into a byte array, then bits in the array into duty cycles, then just plays the duty cycles

Sequence chart:



Warning: I don't promise the actual numbers here are correct. Just the overall idea. Recalculate the numbers yourself based on timing requirements

Details on NeoPixels

- Datasheets:

- https://cdn-shop.adafruit.com/product-files/2757/p2757_SK6812RGBW_REV01.pdf
- <https://cdn-shop.adafruit.com/datasheets/WS2812B.pdf>

- Good writeups on using NeoPixels:

- <https://github.com/japarc/ws2812b/blob/master/firmware/README.md>
- <https://wp.josh.com/2014/05/13/ws2812-neopixels-are-not-so-finicky-once-you-get-to-know-them/>

Outline

- USB
- CAN
- Bit-banging
 - 1-wire
 - Neopixel LED strips
- **Video Protocols**
 - **VGA**
 - **HDMI**

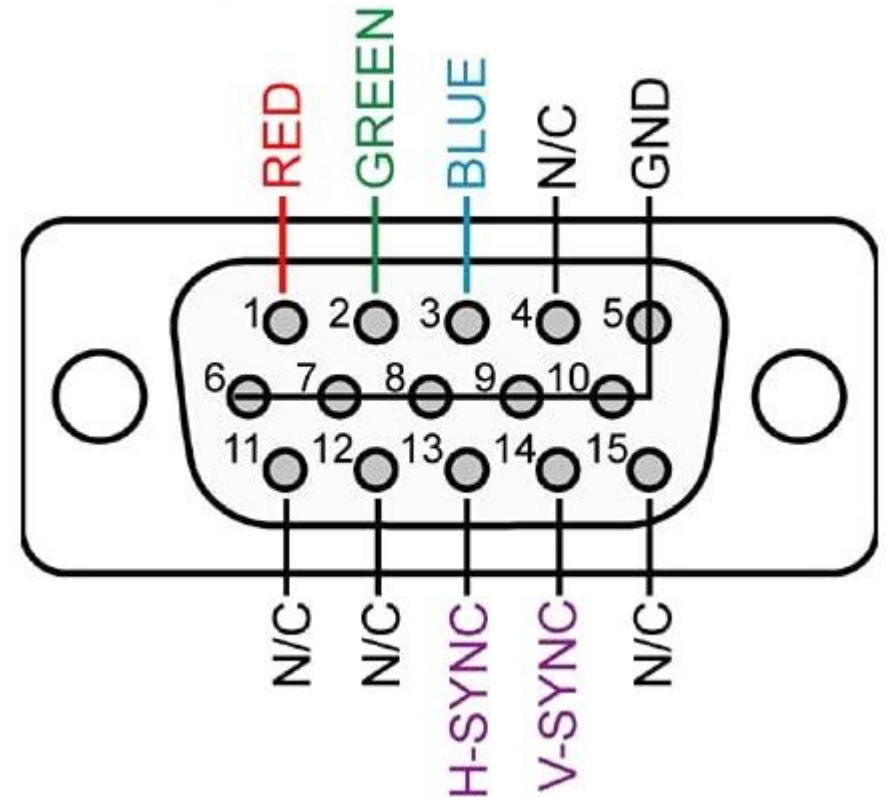
Controlling video output

- Not very common for embedded systems
 - Either directly connected to a screen over SPI/I2C, or not at all
- Very common for traditional computing though
 - And while we're talking about protocols, let's look at them!
- Reminder: memory is a huge embedded systems concern here
 - 640 x 480 pixels @ 18-bit color = 675 kB
 - nRF52833 RAM: 128 kB, Flash: 512 kB

VGA connector

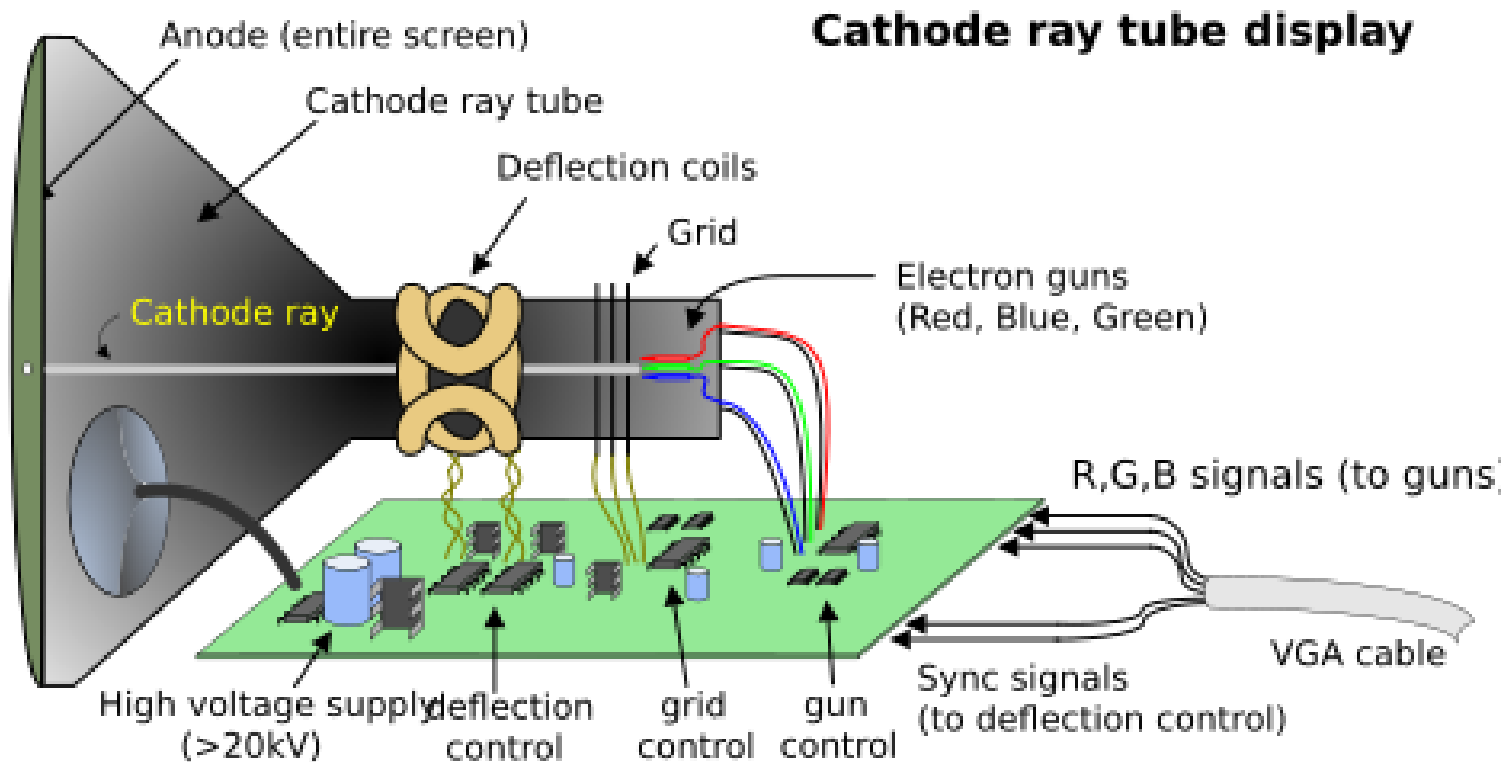
- All data goes to the display
- Horizontal and Vertical sync lines
 - Timing control
 - Horizontal: when each row restarts
 - Vertical: when the entire screen restarts
- Red/Green/Blue lines
 - Analog voltage for intensity: 0-0.7 volts
- Ground pins
 - To connect ground between devices

VGA port, view from Wire Side



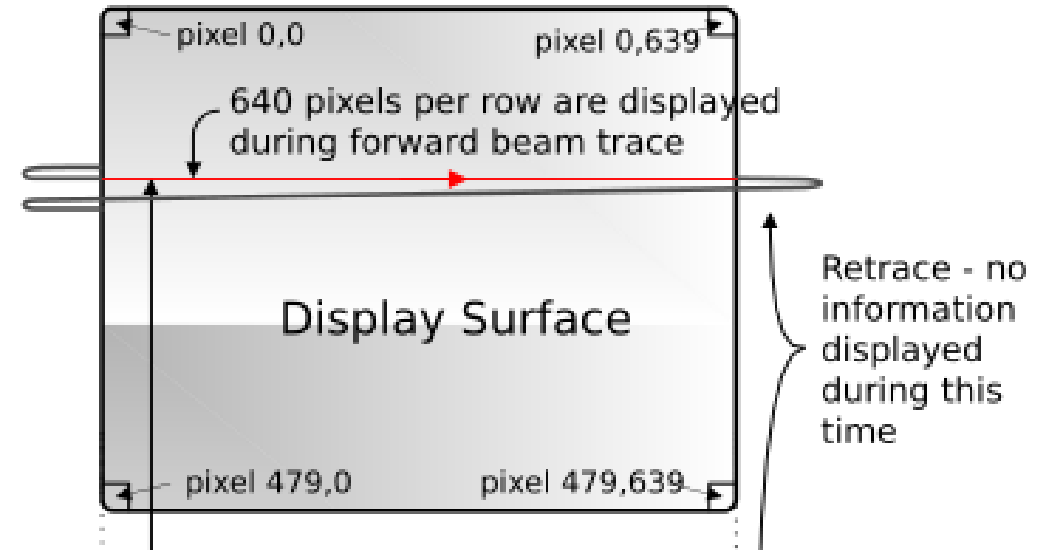
Old-school VGA display design

- Screen lights up when hit with electrons
 - Separate Red, Green, and Blue activation areas
- System directs electrons at each pixel of the screen on a loop



Example display: 640x480 at 60 Hz

- Send each pixel in a row at 20 MHz (40 ns per pixel)
 - 25 μ s for the entire 640 pixel row
- 6.35 μ s pause before next row
- After all 480 rows, 1.43 ms pause before next frame
- High resolution devices would have to go faster
 - Possibly MUCH faster



<https://digilent.com/reference/learn/programmable-logic/tutorials/vga-display-controller/start>

More modern VGA devices

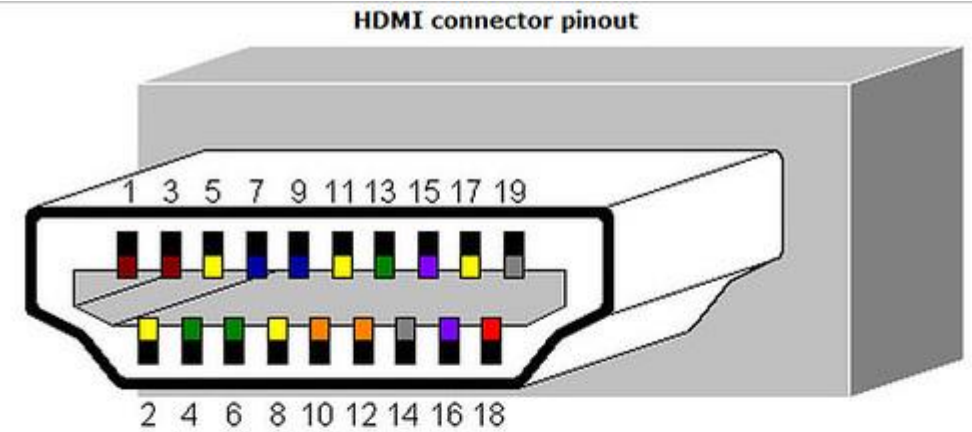
- Protocol doesn't really make any sense for a digital device
 - But it already existed and was very popular
 - So LCD monitors implemented it too
- ADC reads in analog voltages for Red, Green, and Blue into a memory
- Data from the memory is displayed on the screen

VGA Extension - Display Data Channel (DDC)

- Uses extra wires in the VGA connector
 - Originally “4-bit monitor ID”, but rarely used in practice
 - Instead uses two as I2C data and clock lines
- Monitor is always a peripheral device at address 0x50
- Provides a “Extended Display Identification Data” payload
 - 128-256 bytes of data
 - “Descriptor” for the display, like USB descriptors
 - Identifies manufacturer, resolution(s), refresh rate(s), timing requirements

HDMI

- Clock plus three data “channels”
 - Transition-Minimized Differential Signaling (TMDS)
 - Similar to USB with twisted pair wiring
 - Send data packets to display
- I2C data and clock for Display Data Channel (DDC)
 - Adapted from VGA
- Consumer Electronics Control (CEC) enables control of devices
 - Similar to the 1-wire protocol



Pin	HDMI 1.0	New in HDMI 1.4
1	TMDS channel 2 +	
2	TMDS channel 2 shield	
3	TMDS channel 2 -	
4	TMDS channel 1 +	
5	TMDS channel 1 shield	
6	TMDS channel 1 -	
7	TMDS channel 0 +	
8	TMDS channel 0 shield	
9	TMDS channel 0 -	
10	TMDS clock +	
11	TMDS clock shield	
12	TMDS clock -	
13	CEC	
14	<i>n/c</i>	HEC data -
15	DDC I ² C clock SCL	
16	DDC I ² C data SDA	
17	DDC/CEC/HEC shield	
18	+5 VDC power	
19	Hot Plug Detect	HEC data +

Outline

- USB
- CAN
- Bit-banging
 - 1-wire
 - Neopixel LED strips
- Video Protocols
 - VGA
 - HDMI