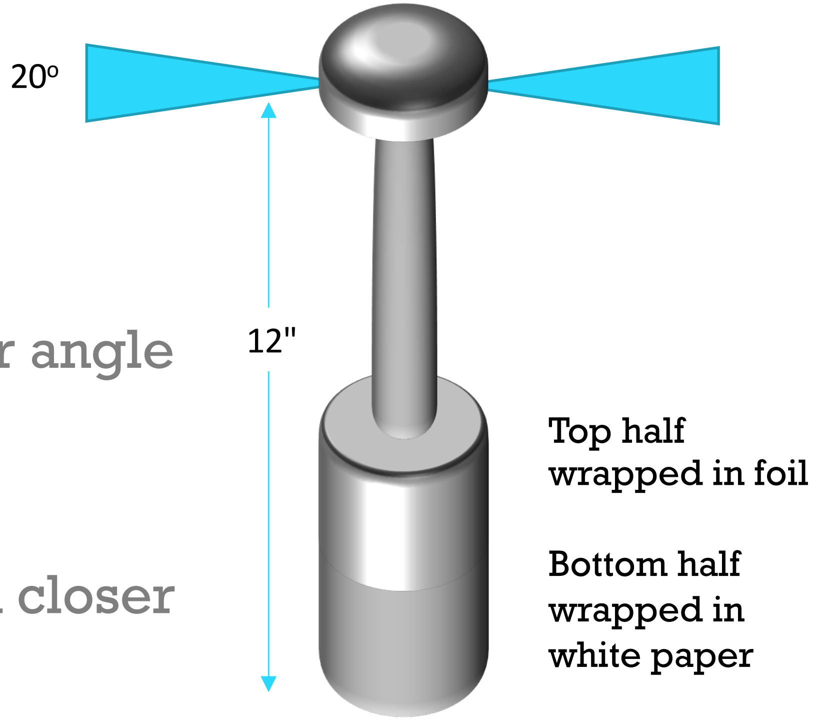
Lecture 24

Wired Comms pt 2 (UART, SPI) and ESP-Now

Today's Agenda

01. Arduino Library I2C example
02. UART (RS232, Logic Level)
03. SPI
04. Other protocols (I2S, CAN, USB) [Overview]
05. ESP-NOW

Rules updates



- Beacons 12" high with 20 Half Power angle
- Cans have half paper half foil.
- Vive Frame will be moved (lowered closer to field) in final game.
- Vive diode PD70 correct anode/cathode

Dual Motor Drivers often used in MEAM510

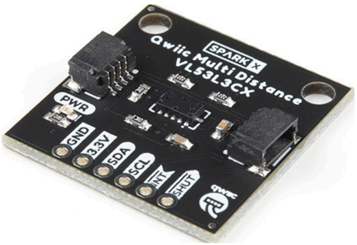
Motor driver	Max Amp cont (peak)	Vout loss @1A typ	Voltage range	Notes
SN754410	1A (2A)	2.6V	4.5 to 36	Vih = 2.0V
L293D*	1.2A (2A)	2.5V	4.5 to 36	Vih = 2.3V
FAN8100	0.8A (1.5A)	0.8V	1.8 to 9V	Mod DIP pkg
FAN8100 dbl	1.6A (3.0A)	0.8V	1.8 to 9V	Two ch parallel
LV8401V	1.2A (3.8A)	0.33V	4V ro 15V	SMD
LV8401V dbl	2.4A (7.6A)	0.17V	4V ro 15V	Two ch parallel

* Popular original version of SN754410. Available as Arduino shield.

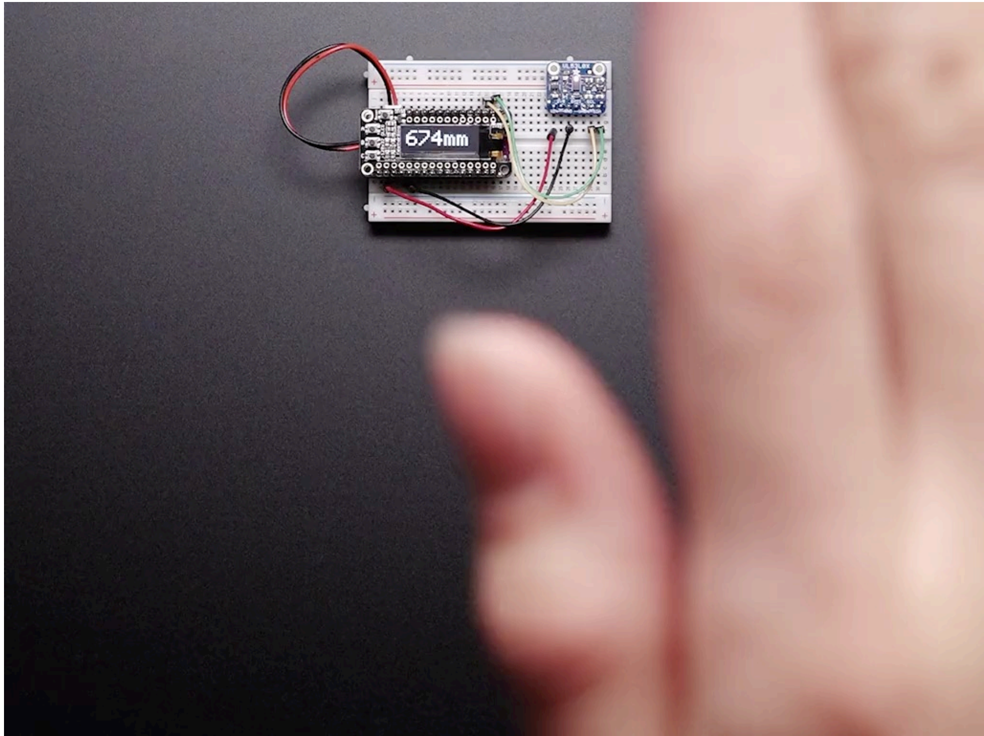
01

Arduino Library Example I2C with ToF sensor

TOF Range Finder (VL53L0X)



- Range: 5cm to 120cm
- Speed: ~30 samples/sec
- Beam width: ~25°
- $14.95 (maybe ~$5 on amazon)
- Interface with **I2C**,
- Adafruit has Arduino library.
- Very accurate and linear



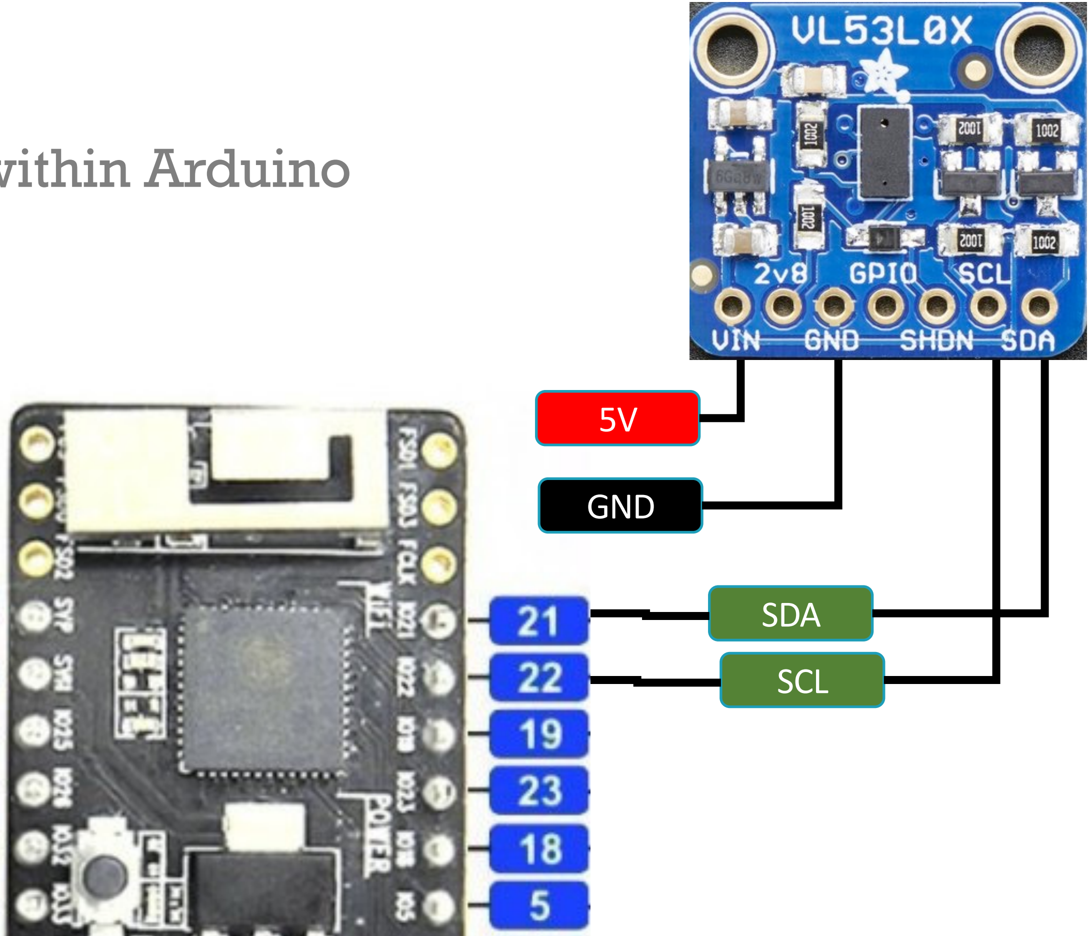
https://www.adafruit.com/product/3317

I2C example VL53L0X from Adafruit

Download libraries from within Arduino

Then hook up:

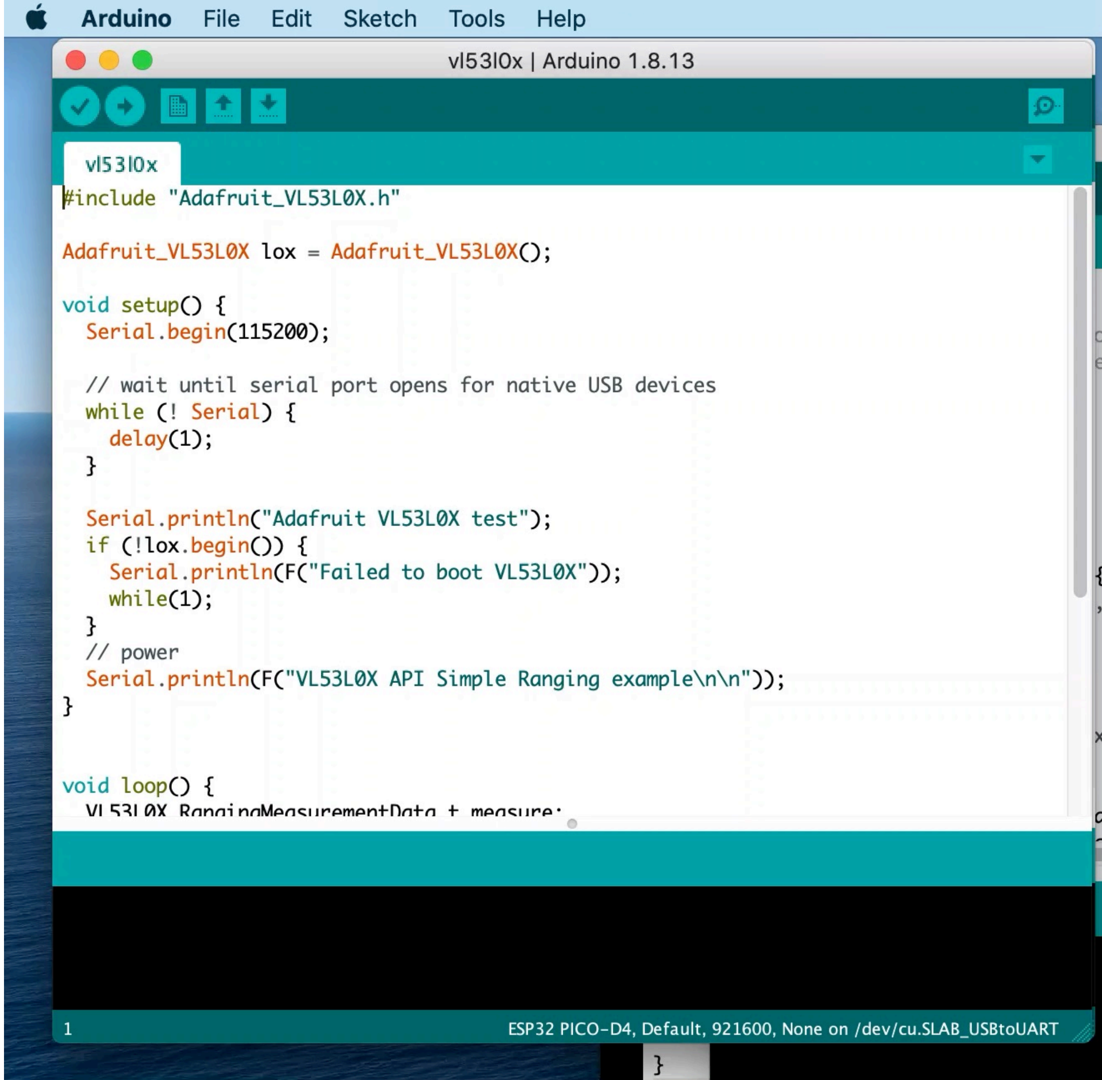
- ESP32 default SCL is GIPO21
- ESP32 default SDA is GPIO22
- Attach SDA to SDA
- Attach SCL to SCL



Arduino ToF Library

Built into Arduino IDE

- Sketch -> Include Library -> Manage Libaries
- Search for VL53L0X install all libraries
- File -> Examples -> Adafruit_VL53L0X -> vl53l0x



/dev/cu.SLAB_USBtoUART

Send

```
Reading a measurement...  out of range
Reading a measurement...  out of range
Reading a measurement...  out of range
Reading a measurement...  out of range
Reading a measurement...  out of range
Reading a measurement...  out of range
Reading a measurement...  out of range
Reading a measurement...  out of range
Reading a measurement...  out of range
Reading a measurement...  out of range
Reading a measurement...  out of range
Reading a measurement...  out of range
Reading a measurement...  out of range
Reading a measurement...  out of range
```

☑ Autoscroll ☐ Show timestamp | Carriage return | 115200 baud | Clear output

ToF ranger pros vs cons

PROS

- Easy to install library
- Range values independent of color
- Angle of reflection does not change the precision, if value is received.
- You know when you have good values or no value.

CONS

- Color does affect ability to get values at all
- Angle of reflection affects ability to get any range value.

02

UART

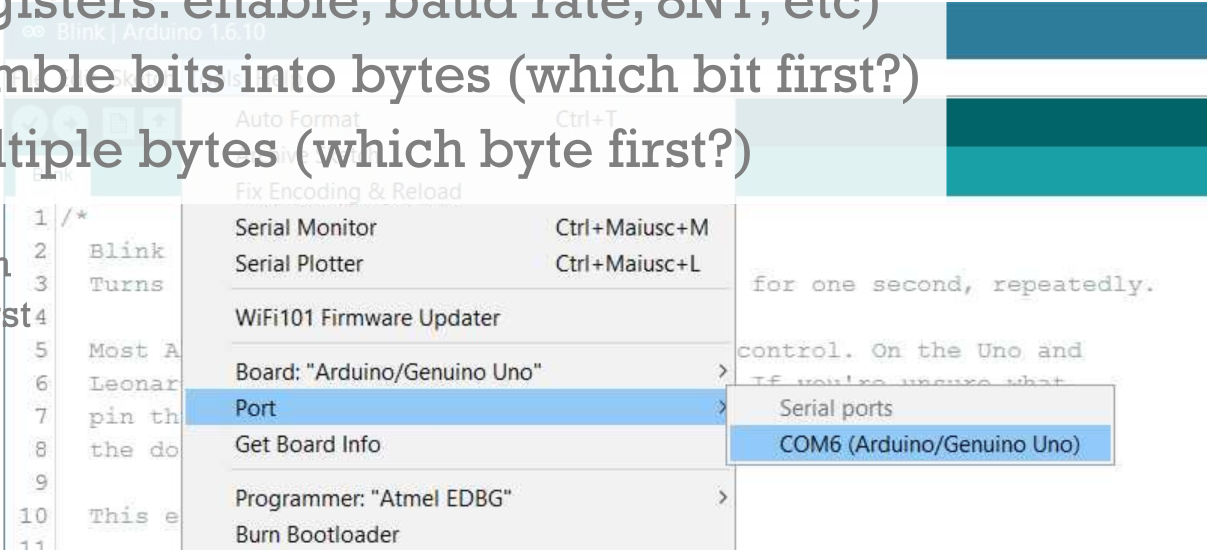
Universal Asynchronous Receiver and Transmitter

Quiz questions at end of Section 4

1. For asynchronous comm, you must set the same __________ on both ends
2. For synchronous, _________ sets the clock speed (mostly, though I2C has slightly different)
3. _______and _____ are the two most common *synchronous* protocols used between embedded processors
4. ______________ can be used for simple limited state changes

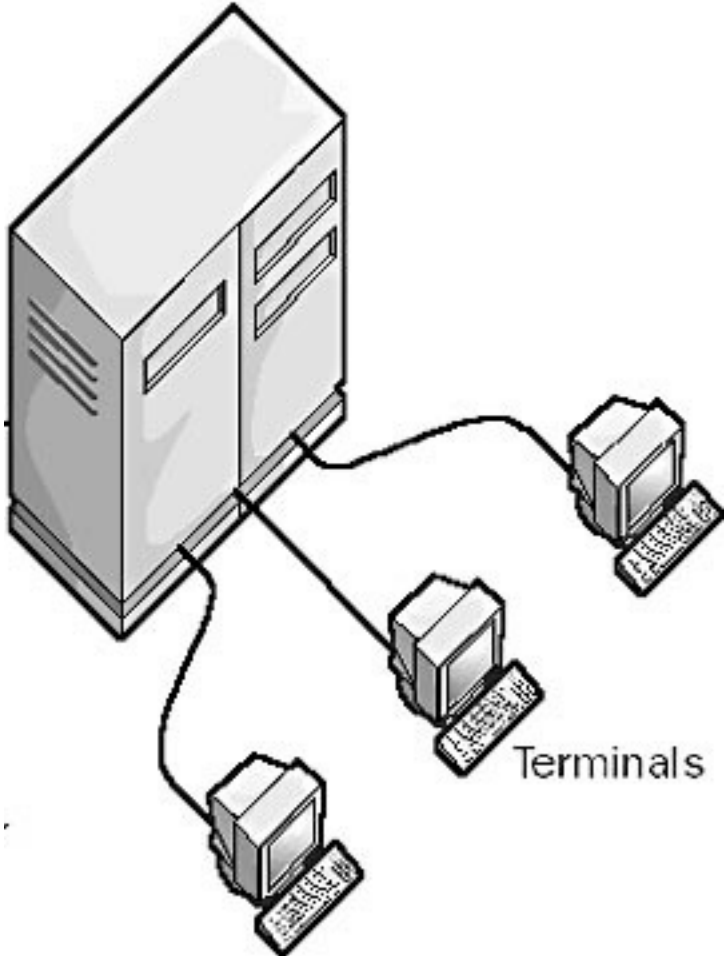
UART: Universal Asynchronous Receiver/Transmitter – aka "serial port" or "COM"

- UART is the most common MCU communications method.
  - Slight variant: USART (include synchronous)
- Can use logic levels or include interface layer, typ: RS232, RS422, R485.
- Initialization (setup registers: enable, baud rate, 8N1, etc)
- UART's will send/assemble bits into bytes (which bit first?)
- Integers/floats are multiple bytes (which byte first?)
  - Byte order
    - Big-endian, little-endian
    - LS byte first, MS byte first
  - Use Packets?
  - Do handshaking?



Serial port history (all async)

- 1970's Mainframe and terminals.
  - All connected with RS232 serial ports running 9600 baud
- 1980's IBM PC's and knock offs
  - All personal computers had RS232 ports (called COM ports)
  - All running at 9600 baud default, 115200 baud max.
- 1990's USB
  - Serial ports started to disappear on computers, but UARTs still on MCUs
  - Most USB still had UART converter still called COM ports on PC's



Asynchronous Framing
NRZ (non return to zero)

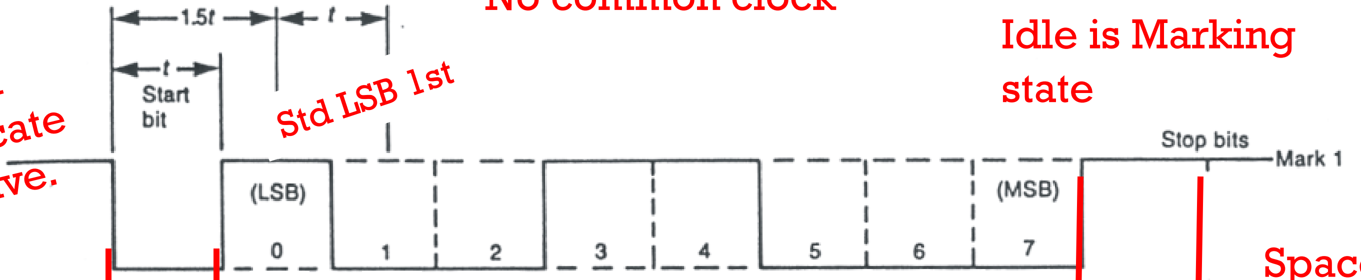
time

No common clock

Historically nominal high state to indicate device is alive.

Std LSB 1st

Idle is Marking state



1 0 0 1 1 0 0 0

Space state

1 bit time

Usually 8 bits, Sometimes 7, 9

At least one Stop bit

Signifies beginning of byte

8N1 : 8 Data, No parity, 1 stop bit
8N2 : 8 Data, No parity, 2 stop bit
7E1 : 7 Data, Even parity, 1 stop bit
7E2 : 7 Data, Even parity, 2 stop bits

Baudrates are the bits/second, not data which has overhead (10bits for every byte), Most common: 9600 and 115.2Kbaud.

Communications between MCUs and peripherals

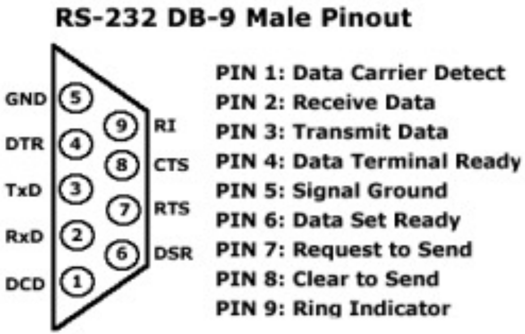
CAN	UART Logic	UART RS232	UART RS485	SPI	I2C		
x						2. Data Link	Packetizing
		o			x	1d Data rate	Flow control
x				x	x	1c Network	Addressing
x	x	x		x	x	1b Encoding	Byte frames
x		x	x	x	x	1a Interface	Voltage level

physical layer

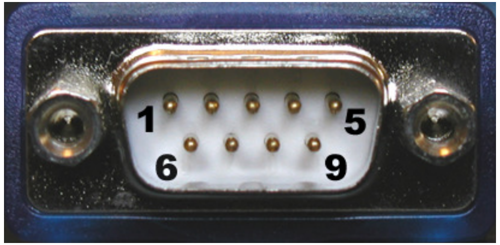
x: protocol specifies
o: protocol has this option (often omitted)

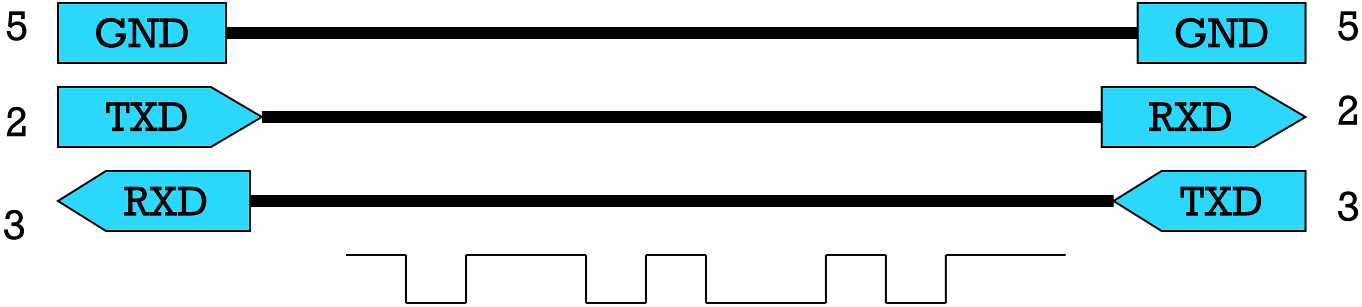
Asynch. Serial Communication– RS232

- RS232 used to be most popular form of inter-processor standard, now RS232 form factor is almost obsolete.
- Most common is essentially 3 wire communication



Female is reversed
Numbers start 1 on right side

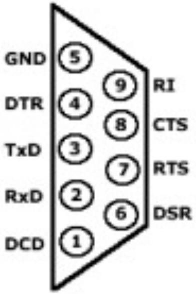




Asynch. Serial Communication

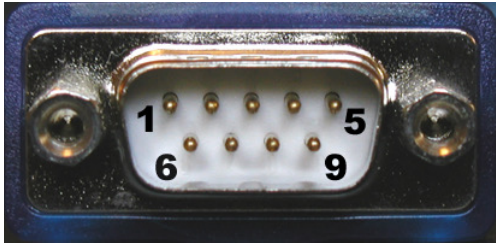
- RS232 used to be most popular form of inter-processor standard, now RS232 form factor is almost obsolete.
- Most common is essentially 3 wire communication

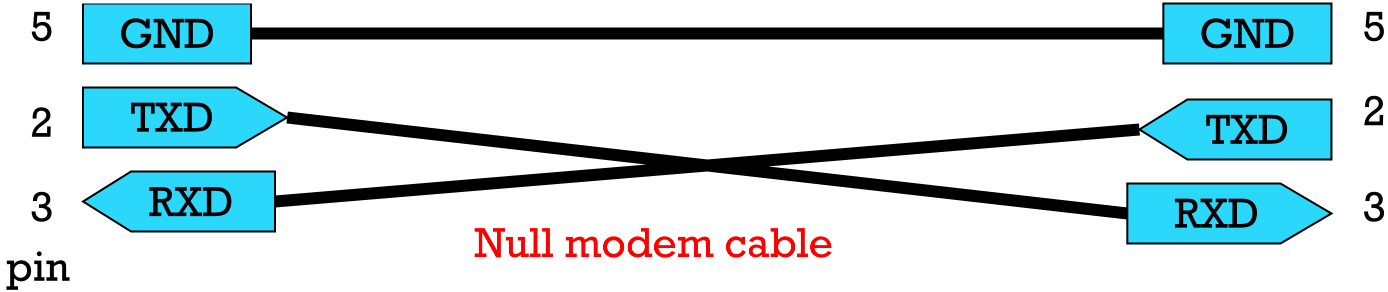
RS-232 DB-9 Male Pinout



PIN 1: Data Carrier Detect
PIN 2: Receive Data
PIN 3: Transmit Data
PIN 4: Data Terminal Ready
PIN 5: Signal Ground
PIN 6: Data Set Ready
PIN 7: Request to Send
PIN 8: Clear to Send
PIN 9: Ring Indicator

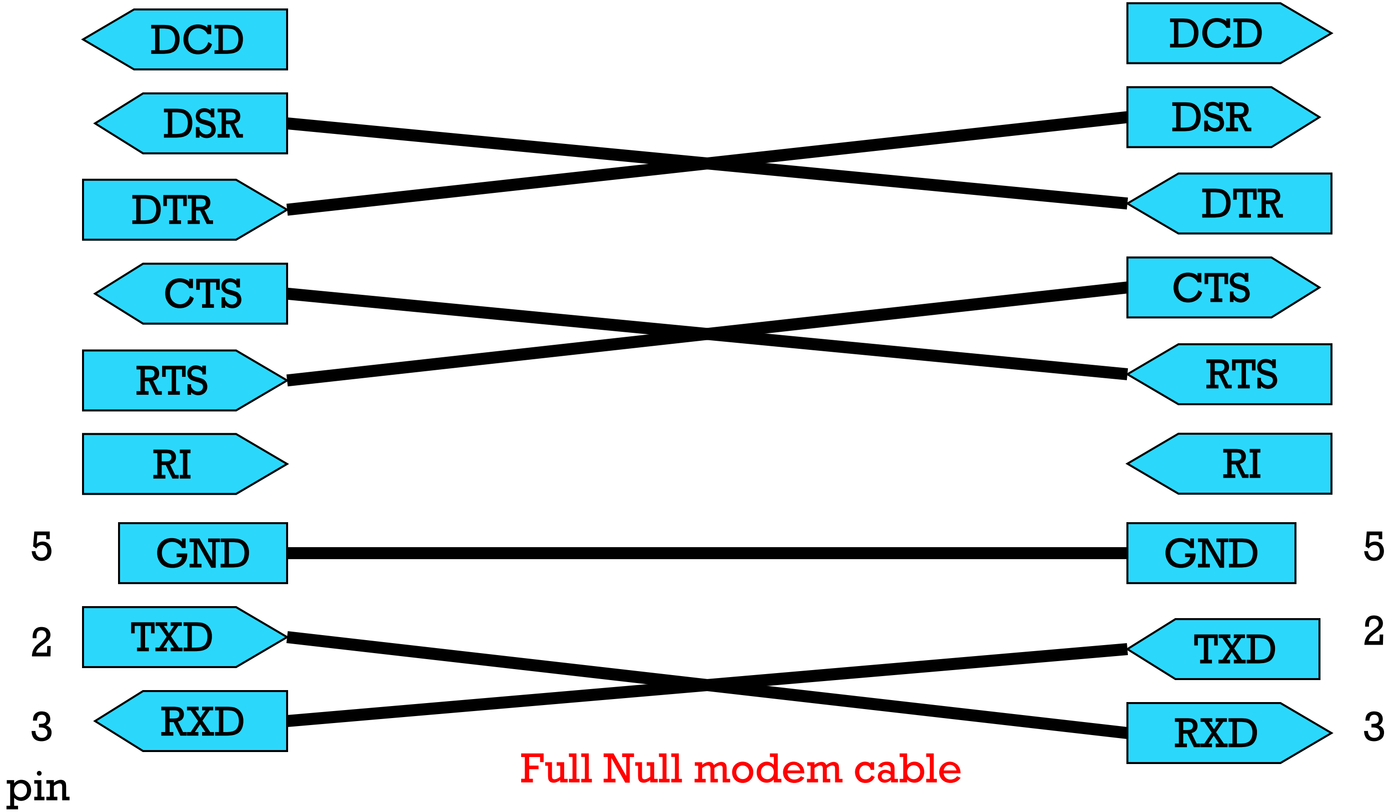
Female is reversed
Numbers start 1 on right side





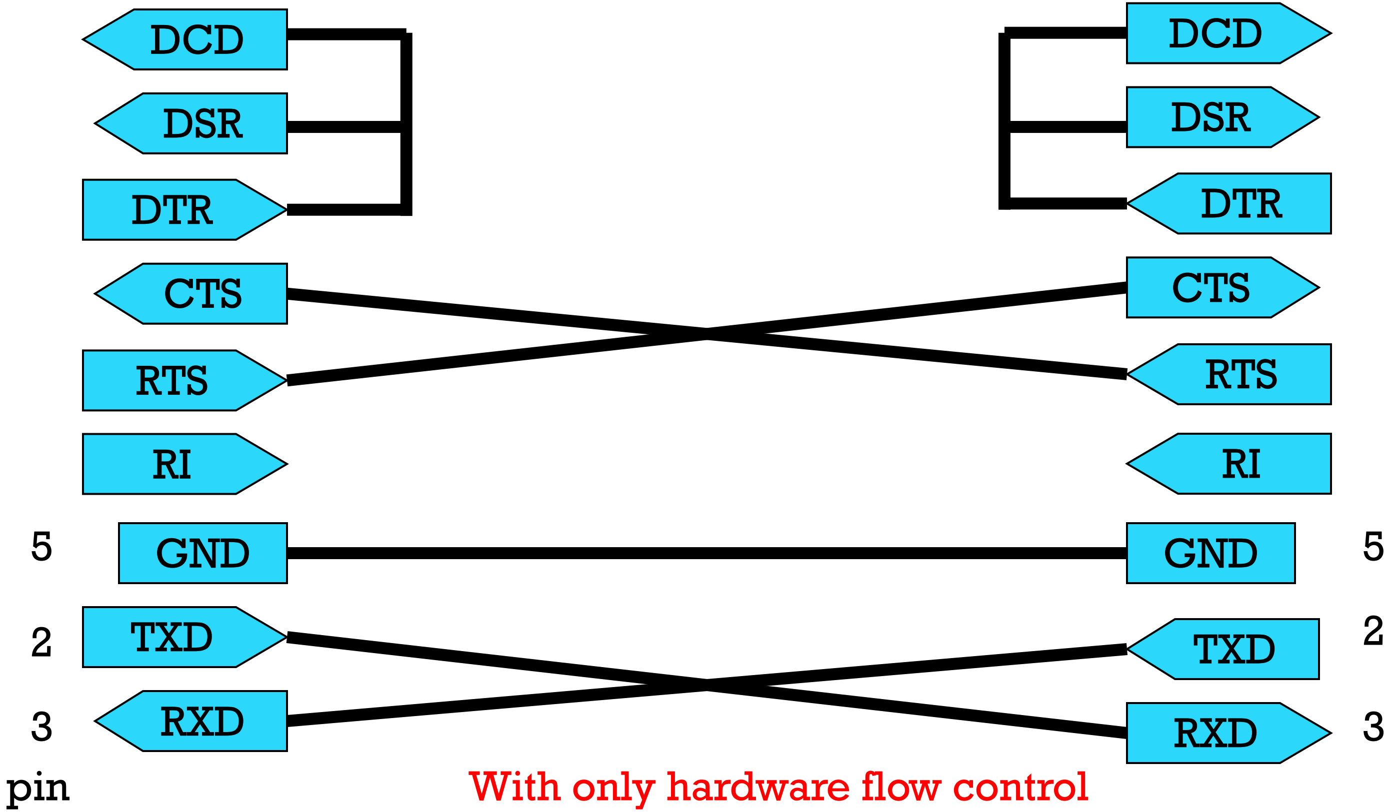
Null modem cable

Asynch. Serial Communication (RS232)



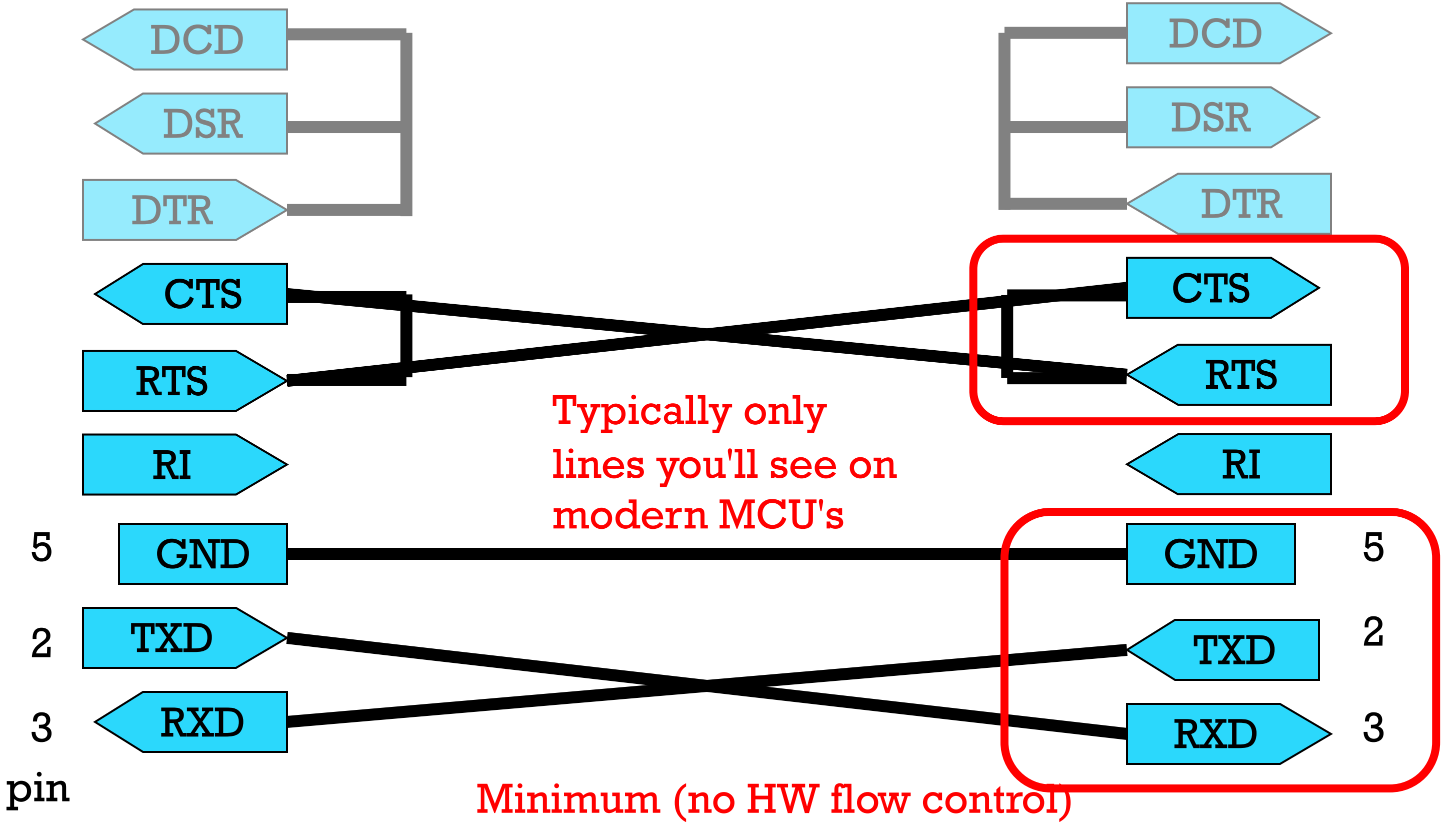
Full Null modem cable

Asynch. Serial Communication (RS232)

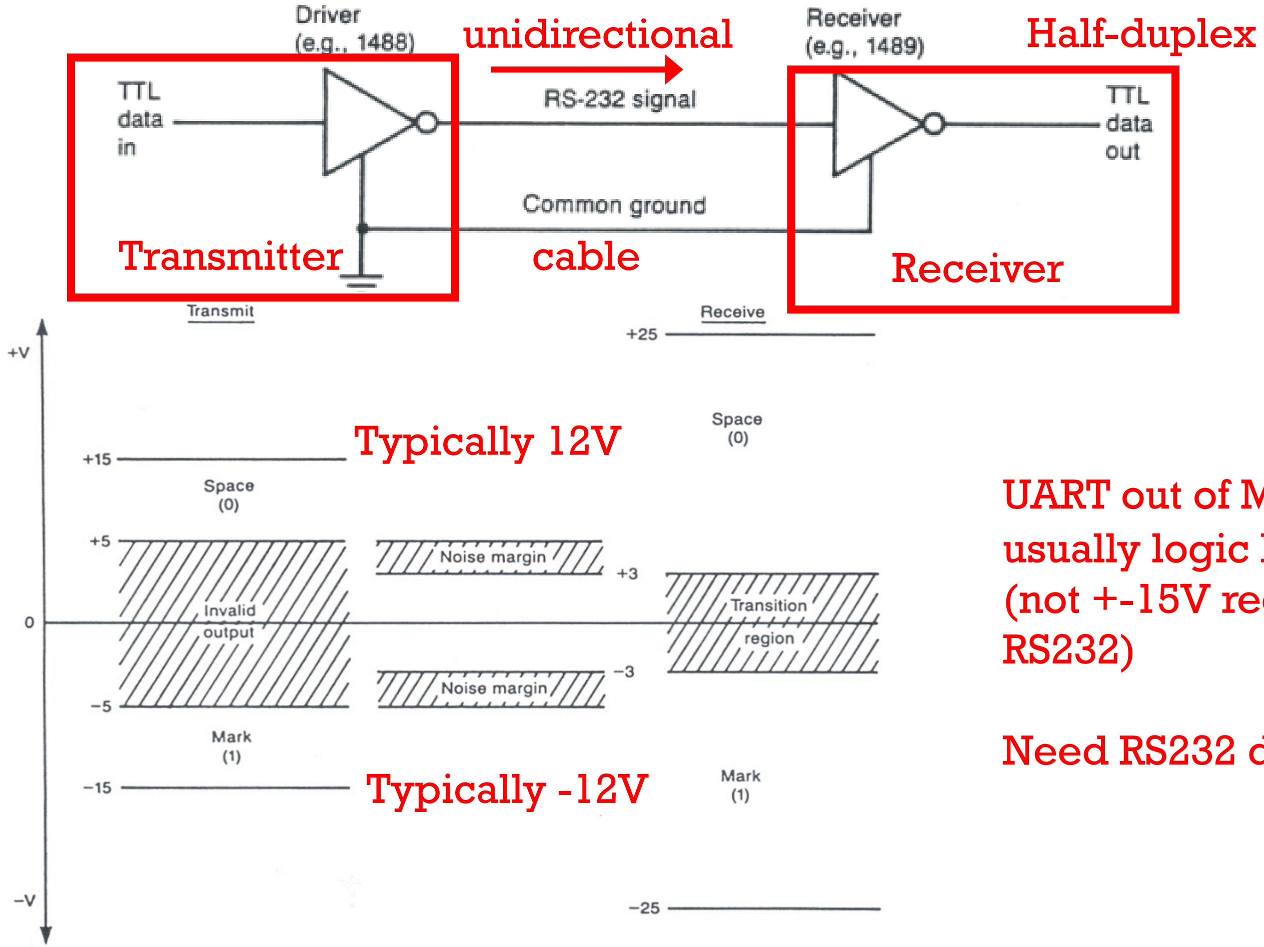


With only hardware flow control

Asynch. Serial Communication (RS232)



Physical Layer – RS232



UART out of MCUs are usually logic level (not +-15V required for RS232)

Need RS232 drivers.

Communications between MCUs and peripherals

CAN	UART Logic	UART RS232	UART RS485	SPI	I2C	Layer	Function
x						2. Data Link	Packetizing
		o			x	1d Data rate	Flow control
x				x	x	1c Network	Addressing
x	x	x		x	x	1b Encoding	Byte frames
x		x	x	x	x	1a Interface	Voltage level

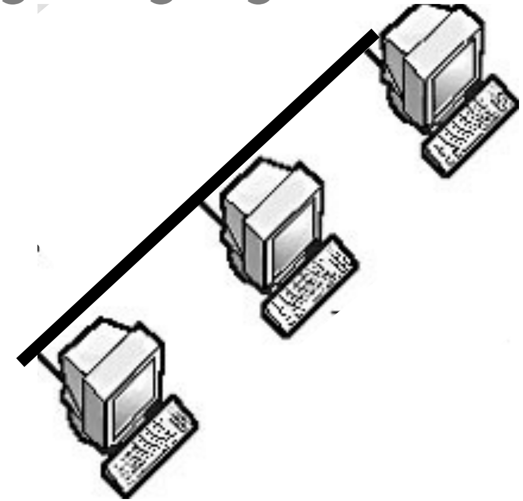
physical layer

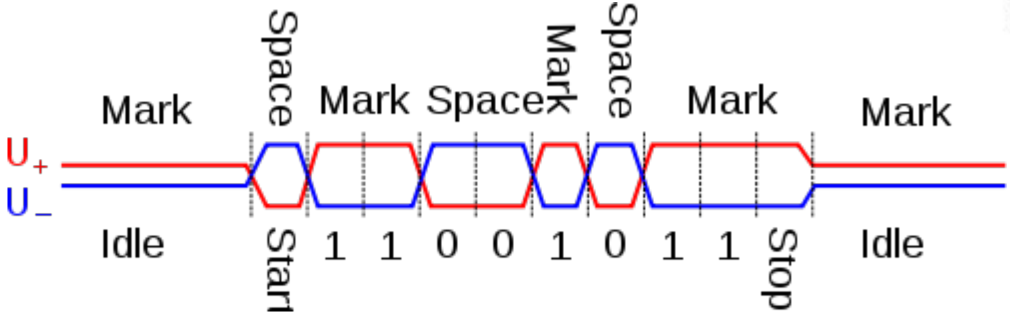
x: protocol specifies

o: protocol has this option (often omitted)

Physical layer – RS485 – ex: differential drive

- Differential drive is the more modern way of getting high speed digital transmission.
- Uses twisted pair for each signal
  - Robust to noise
  - Implicit gnd, 3 line half-duplex, 5 line full duplex
- Caveats
  - Termination resistors can help robustness

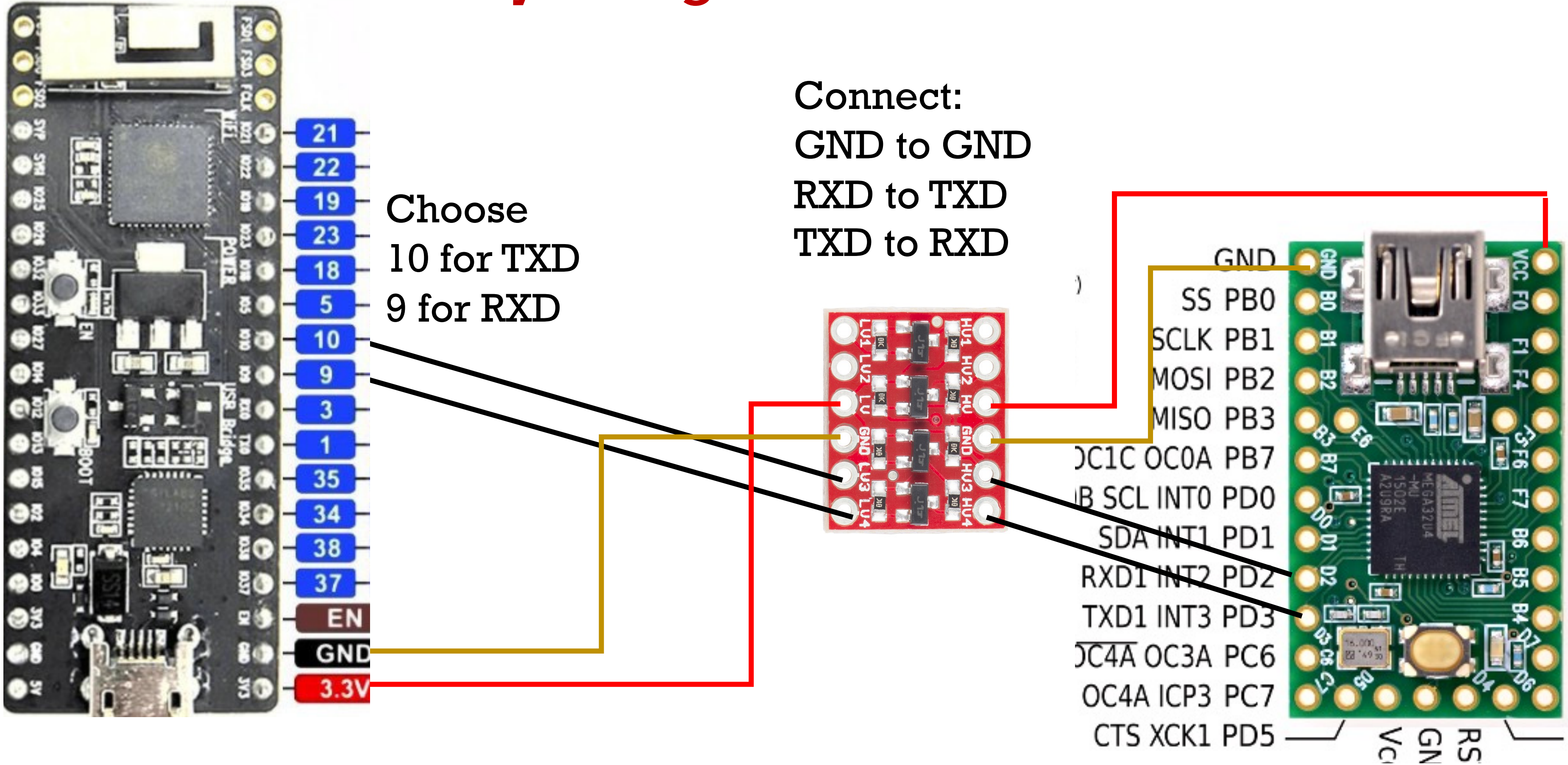




ESP32 to Teensy using UART

Connect:
GND to GND
RXD to TXD
TXD to RXD

Choose
10 for TXD
9 for RXD



Resources -> ESP32 Arduino Lecture Examples -> ESP32-UART.ino
Resources -> Teensy files -> Examples from Lecture -> Teensy_uart.c

UART ESP32 -> Teensy

- ESP32 Arduino code

```
#define RXD2 9
#define TXD2 10

void setup() {
  Serial.begin(115200);
  Serial2.begin(115200, SERIAL_8N1, RXD2, TXD2);
}

int i;
void loop() {
  while (Serial2.available()) { // if recv, print it
    Serial.print(char(Serial2.read()));
  }
  if (millis()%1000==1){
    Serial2.println(i++);  // send an increasing #
    Serial.printf("ESP32 write %d\n",i);
    delay(10);
  }
}
```

- Teensy - Needs uart.c uart.h

https://www.pjrc.com/teensy/uart.html

```
#include "teensy_general.h"
#include "uart.h"


int main(void) {
  uint8_t c;
  teensy_clockdivide(0);
  uart_init(115200);
  while (1) {
    if (uart_available()) {
      uart_putchar('.');     // put '.'

      c = uart_getchar(); // get a char
      uart_putchar(c);    // put a char
    }
  }
}
```

/dev/cu.SLAB_USBtoUART

Send

```
ESP32 write 9
.8..
ESP32 write 10
.9..
ESP32 write 11
.1.0..
ESP32 write 12
.1.1..
ESP32 write 13
.1.2..
ESP32 write 14
.1.3..
ESP32 write 15
.1.4..
```

Autoscroll Show timestamp Carriage return 115200 baud Clear output

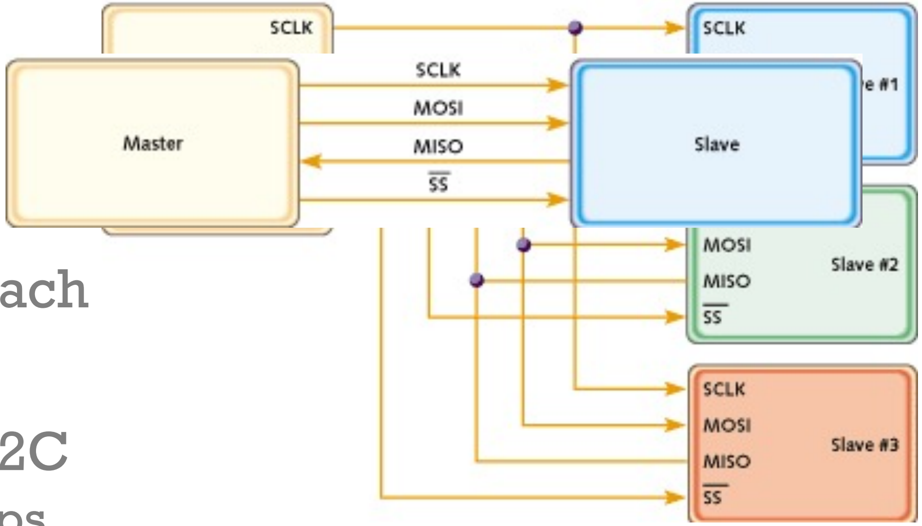
03

SPI

Serial Peripheral Interface

Serial Peripheral Interface (SPI)

- Synchronous comms started by Motorola.
- 4 wire:
  - clock: SCLK
  - master out slave in: MOSI
  - master in slave out: MISO
  - slave select: SS
- Multidrop: need one SS for each
- Generally faster than RS 232
- SPI is inherently faster than I2C
  - Driven hi/low instead of pullups...
  - Theoreticlaly 800K Byte/sec on teensy



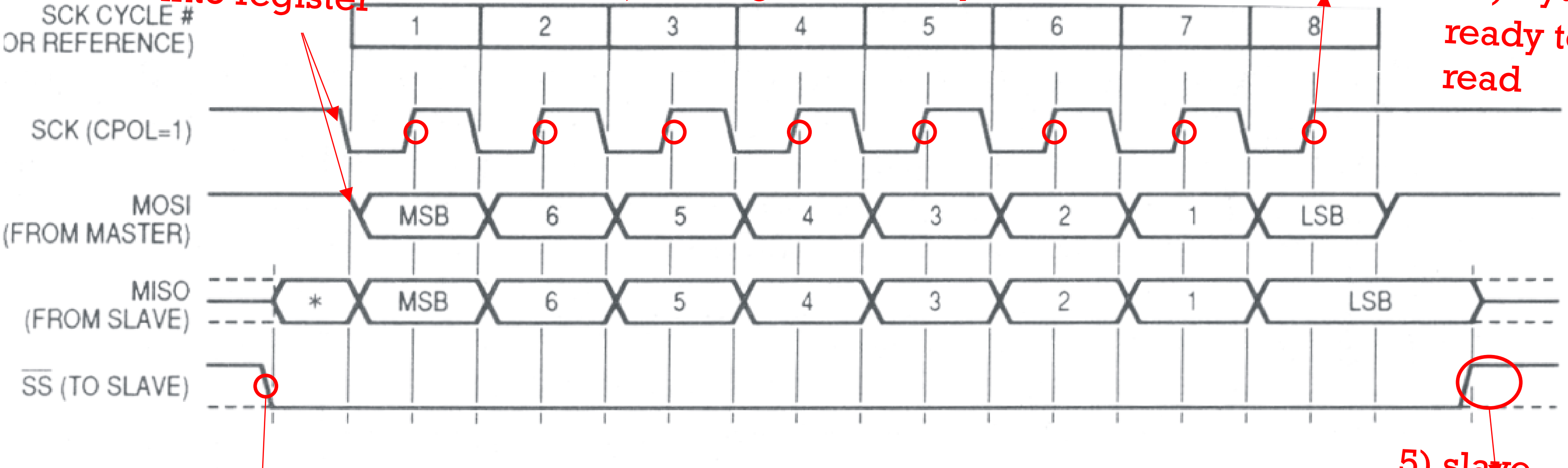
SPI Typical transfer sequence

2) Master writes one byte to send into register

3) storing bits as they come in

4a) interrupt is asserted on master and/or slave

4b) Byte is ready to be read



*Not defined but normally LSB of previously transmitted character.

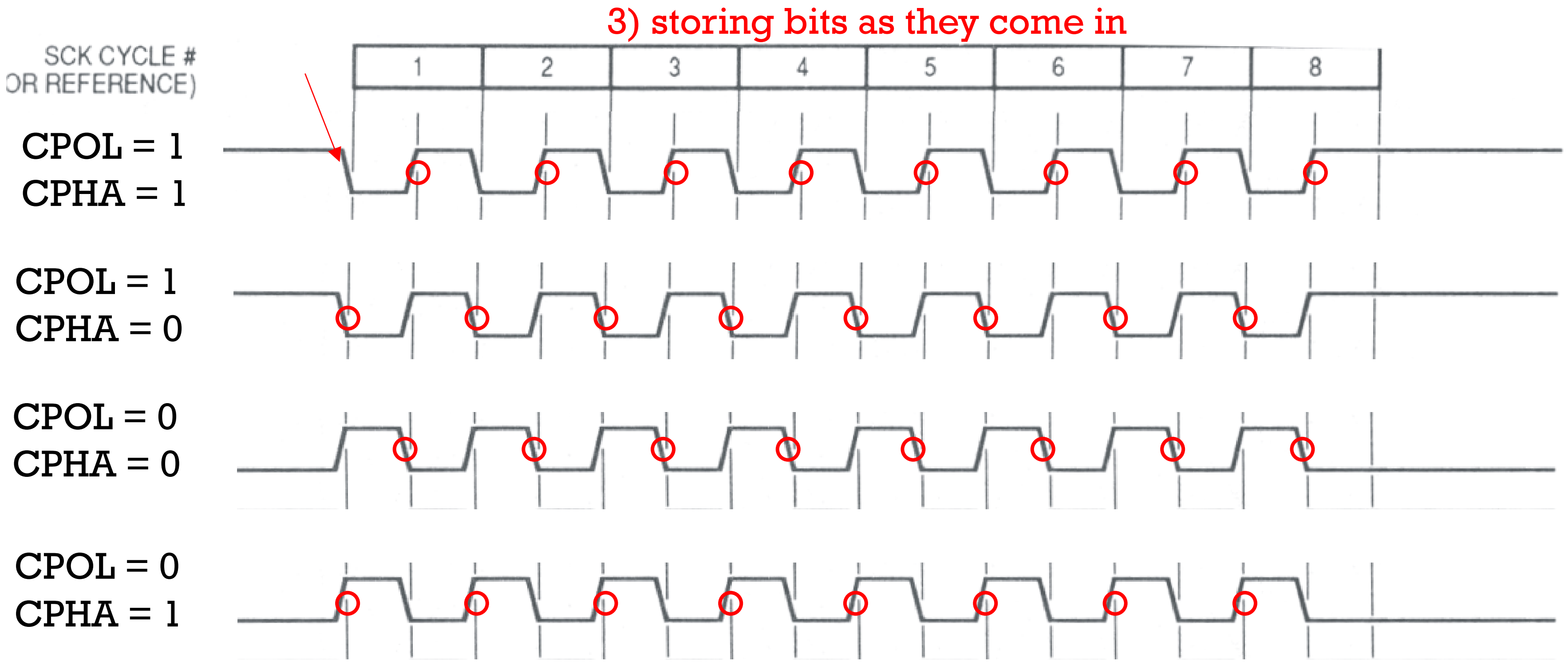
1) slave changes MISO to output

5) slave changes MISO to input

SS sometimes ignored (always asserted)

But transition insures proper byte alignment

SPI Clock Polarity (CPOL) and Clock Phase (CPHA)

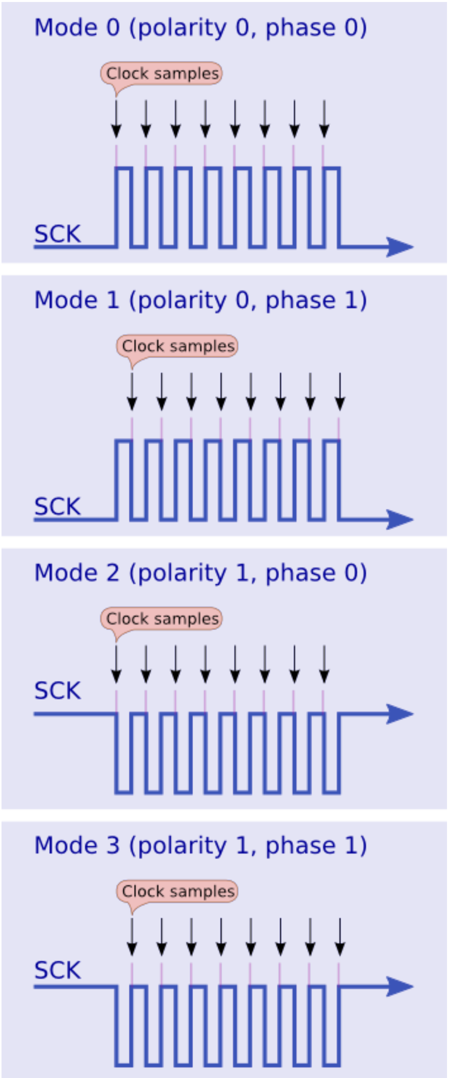


SPI options: clock polarity and bit order

- CPOL: Clock Polarity
- CPHA: Clock Phase, rising or falling edge triggered

- Commonly CPOL=1, CPHA=1 (AKA: mode 3)
- CPOL=0, CPHA=0 (Mode 0) is default on adruino.

- DORD: Data Order, most significant bit first or least significant bit first.



SPI on ATmega32U4 - registers

- CPOL = CPHA = mode 0 (default)
- Data Order DORD = 0 = MSB first (default)

Data Register

Master writes SPDR to initiate transfer

Master reads SPDR byte from slave after transfer

Bit	7	6					1	0	
	MSB							LSB	SPDR
Read/Write	R/W	R/W						R/W	
Initial Value	X	X	X	X	X	X	X	X	Undefined

interrupts | enable | bit order | master | polarity | phase | clock rate

Bit	7	6	5	4	3	2	1	0	
	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	SPCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0		0	0	0	0	0	0	

Control Register

flag

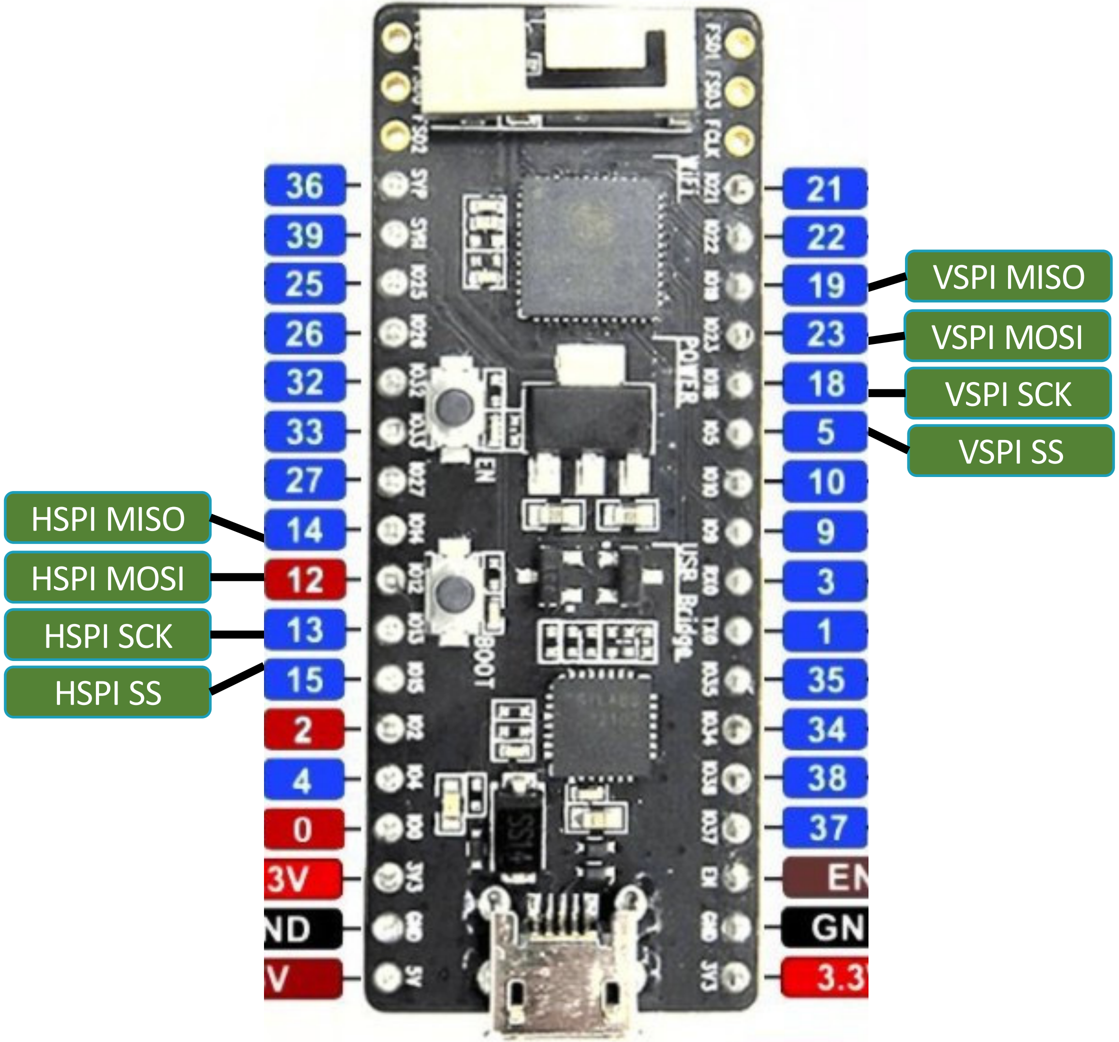
Bit	7	6	5	4	3	2	1	0	
	SPIF	WCOL	–	–	–	–	–	SPI2X	SPSR
Read/Write	R	R	R	R	R	R	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Status Register

SPI on ESP32

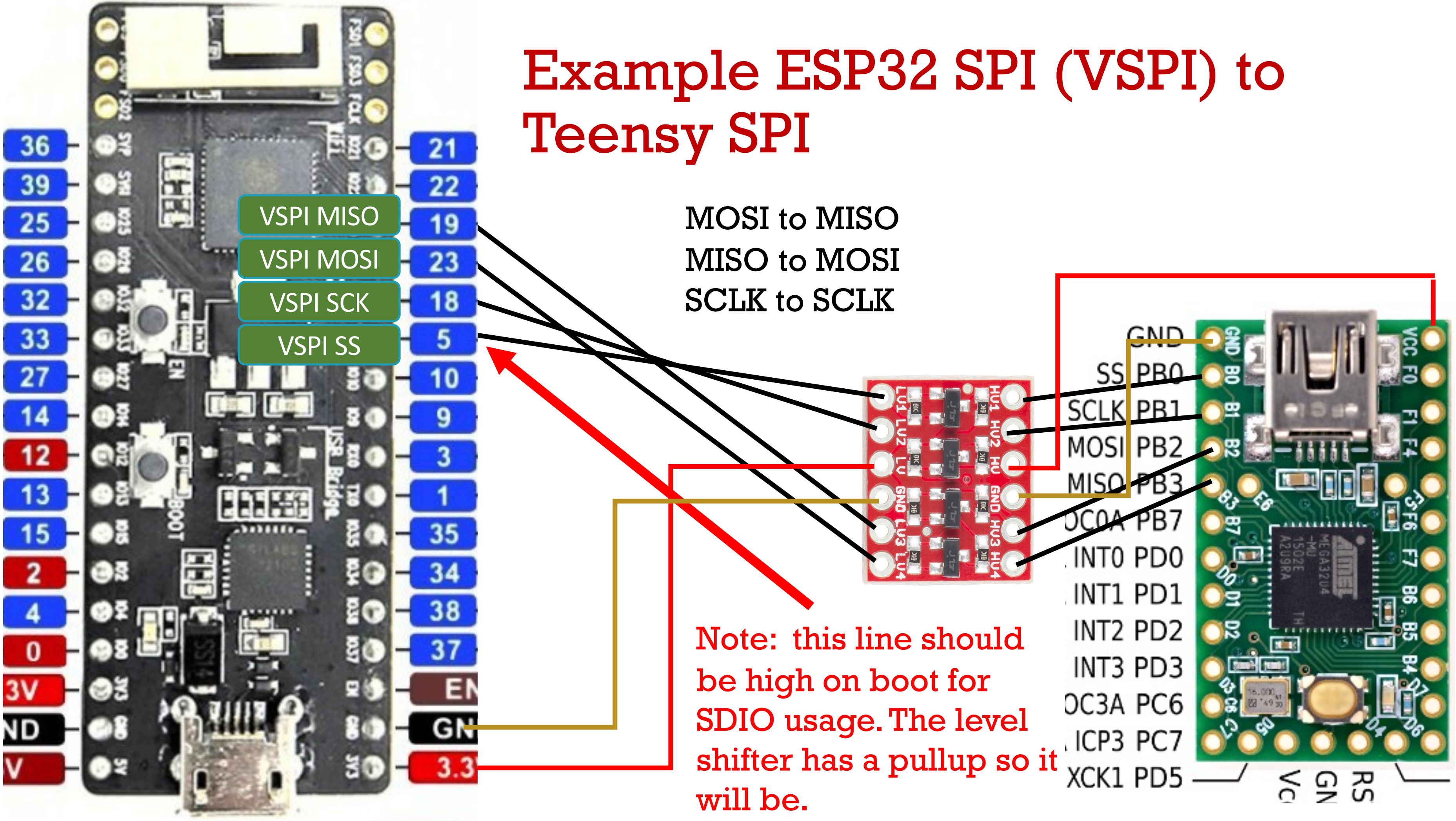
There are four SPI. Two are used for on board memory. Two are available (VSPI, HSPI).

Most functions on ESP32 can be remapped to other pins.



Example ESP32 SPI (VSPI) to Teensy SPI

MOSI to MISO
MISO to MOSI
SCLK to SCLK



Note: this line should be high on boot for SDIO usage. The level shifter has a pullup so it will be.

CANVAS files->Resources->Teensy Files->Teensy_SPI_slave.c

Sample C slave code (on Teensy)

[prints messages as they come in]

Q1) How would we get the Teensy as slave to write a byte to the master?

```
#include "teensy_general.h"
#include "t_usb.h"

char buf[26];
// volatile since interrupt may change value
volatile unsigned char pos;
volatile char process_packet;

ISR (SPI_STC_vect)
{ // grab byte from SPI Data Register
  unsigned char c = SPDR;
  if (pos < sizeof buf) { // add to buffer if room
    buf [pos++] = c;
    if (c == '\n')   // set flag if newline
      process_packet = 1;
  }
  else pos = 0;
}  // end of interrupt routine SPI_STC_vect
```

```
int main() {
  int i;
  m_usb_init();
  set(DDRB,3); // set  MISO B3 as output,
  set(SPCR, SPE);  // enable SPI
  set(SPCR, SPIE); // enable SPI interrupt
  sei();  // enable all interrupts

  while(1) {
    if (process_packet) {
      for (i=0;i<pos;i++)
        m_usb_tx_char(buf[i]);
      pos = 0;
      process_packet = 0;
    }  // end of process_packet
  }  // end of loop
}
```

CANVAS files->Resources->ESP32 Arduino Lecture Examples->ESP32-SPI-master.ino

Sample Arduino Master Code (on ESP32) [Sends byte slave]

Q2) How would we read the byte sent from the slave Teensy?

```
#include <SPI.h>
static const int spiClk = 1000000; // 1 MHz
SPIClass * vspi = NULL;

void setup() {
  vspi = new SPIClass(VSPI);
  vspi->begin(); //default pins: SCLK = 18, MISO = 19, MOSI = 23, SS = 5
                 //alternatively set  pins e.g. vspi->begin(0, 2, 4, 33);
  pinMode(5, OUTPUT);  // use 5 for SS
}

void loop (void) {
  byte data = 0b01010101;
// junk data to illustrate usage

  vspi->beginTransaction(SPISettings(spiClk, MSBFIRST, SPI_MODE0));
  digitalWrite(5, LOW); //pull SS slow to prep other end for transfer
  vspi->transfer(data);  //send one byte, also returns byte read from slave
  digitalWrite(5, HIGH); //pull ss high to signify end of data transfer
  vspi->endTransaction();delay(100);
}
```

Q3) What if you needed GPIO 5 for something else? What could you do for SS?

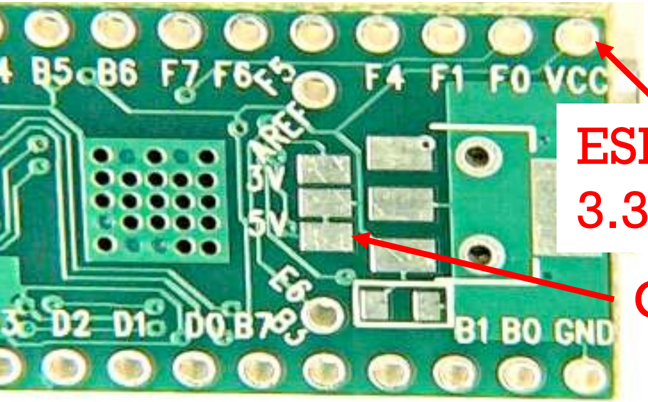
clock speed

Bit order

CPOL, CPHA mode

3.3V conversion

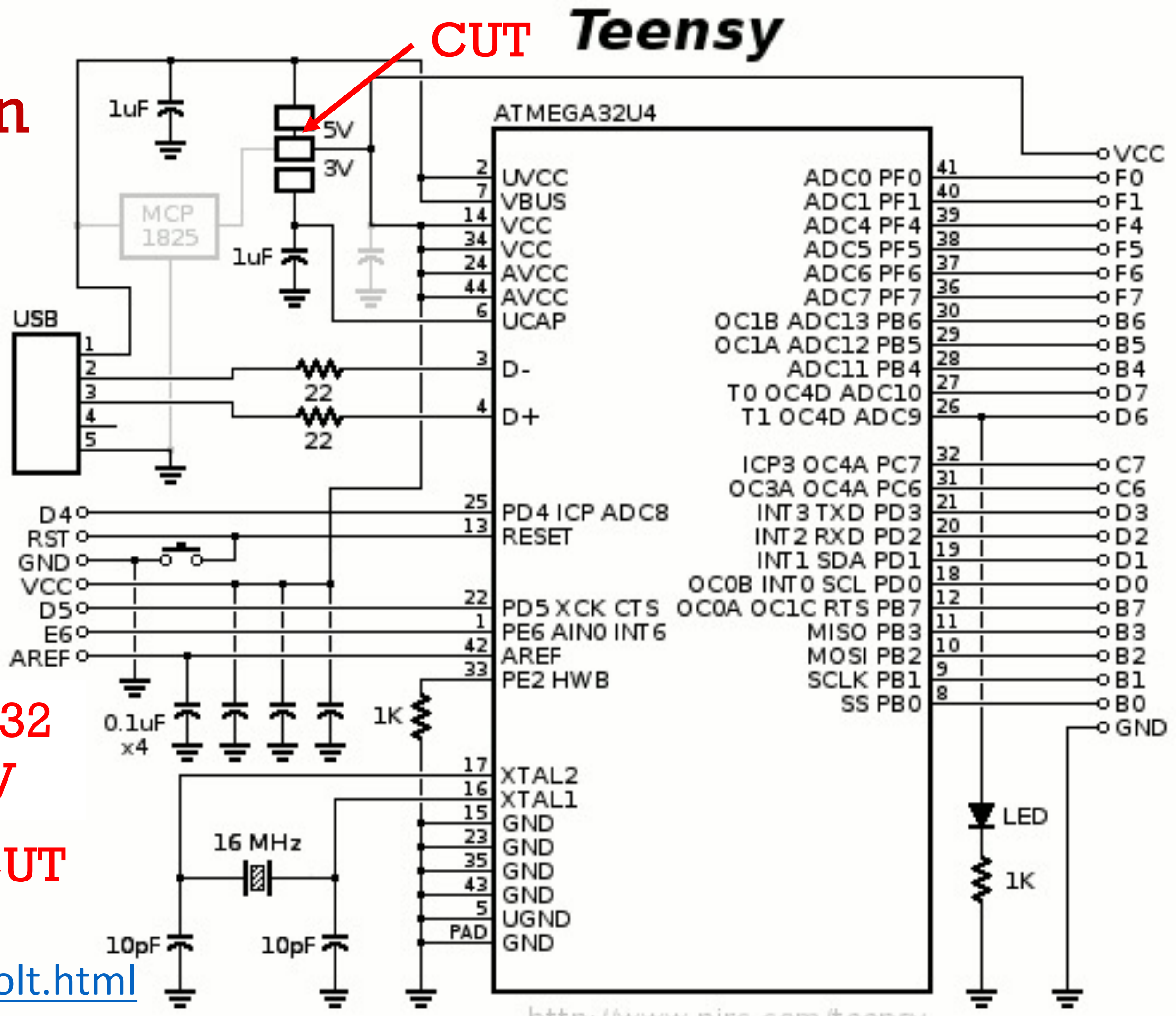
- Can use ESP32 3.3V
- Modify teensy –
  - Cut trace to prevent damaging ESP32 when connecting USB
  - Add solder jumper



ESP32 3.3V

CUT

https://www.pjrc.com/teensy/3volt.html



04

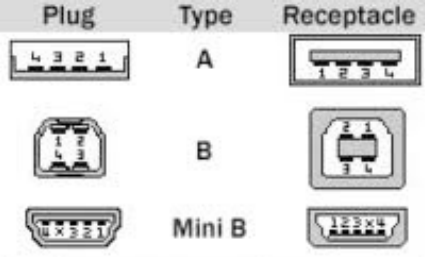
Other Protocols (I^2S, CAN, USB)

I2S (Inter-IC Sound)

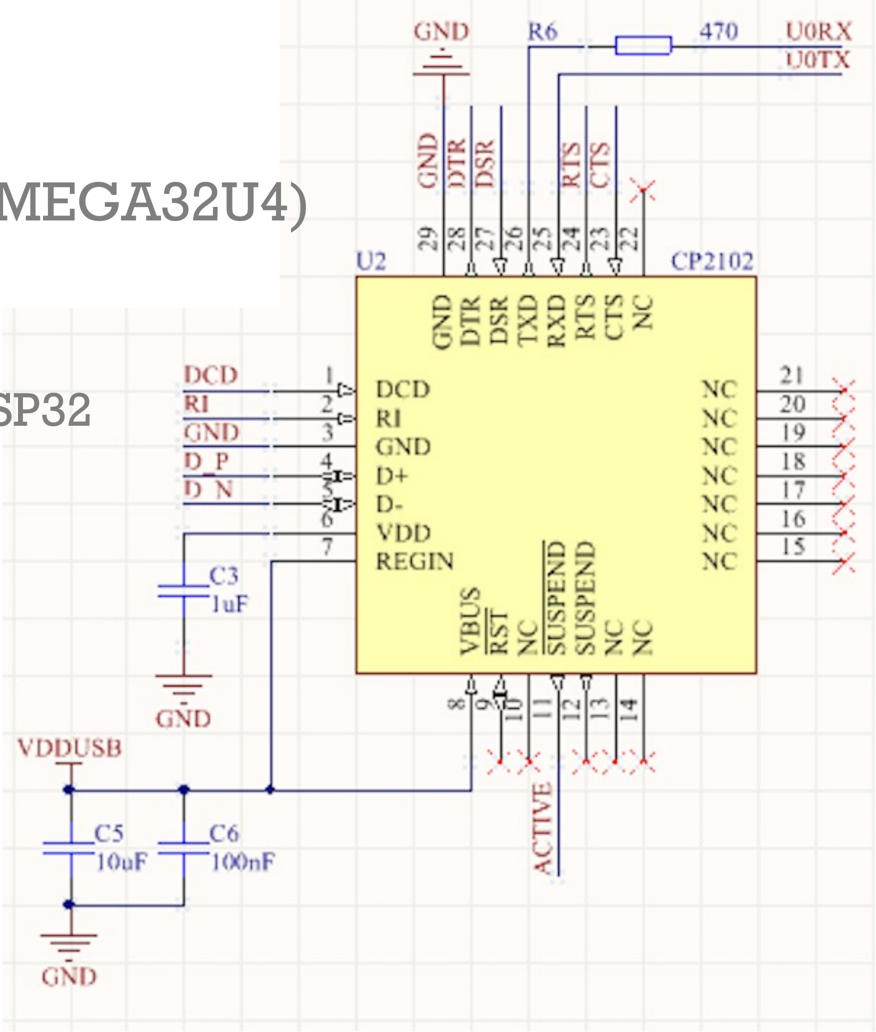
- Closer to SPI than I2C
- Three wire (plus ground) interface
  - Clock (SCK) also called bitclock (BCLK or BCK)
  - Data (SD) also called (DATA, SDATA, SDIN, SDOUT...)
  - Word clock (WS) also called left-right clock (LRCLK), signals left stereo channel data and right stereo channel data
- Faster than I2C (up to 12Mbit/sec) (used for Oscillosorta)
- Made for digital audio files (PCM), but can be used to transmit data.
- Standard hardware interface uses TTL voltage levels (5V)

USB (Universal Serial Bus)

- Either USB included on chip (ex: ATMEGA32U4) or
- Use USB driver bridge
  - to UART (U0TX, U0RX) ex: CP2102 on ESP32
  - to SPI ex: CP2130

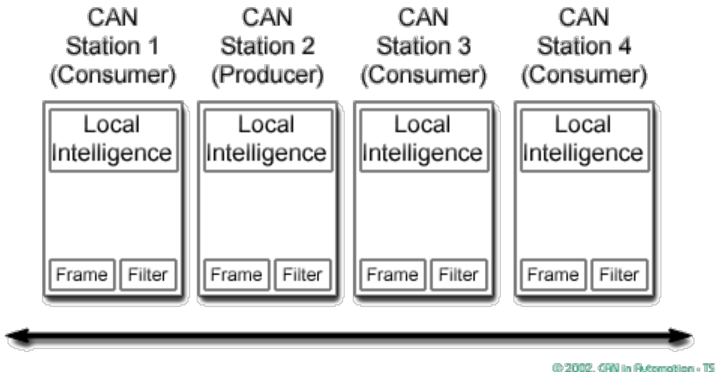


Pin	Signal	Color	Description
1	VCC		+5V
2	D-		Data -
3	D+		Data +
4	GND		Ground



CAN Bus (Controller Area Network)

- Layer 2 (data link layer) of OSI model
  - Packet transmission protocol
  - Collision, error detection (all in hardware!)
- Uses differential pair (e.g., twisted pair)
  - Two lines relative to each other (not gnd)
  - Robust to electric noise (e.g., in a car)
- Up to 1 megabit/sec.



Comparing Serial Protocols

Estimated Parameters

	UART RS232	UART Logic level	I2C	SPI	CAN	USB 2.0
Network Architecture	Singlemaster-Singleslave	Singlemaster-Singleslave	Multimaster-Multislave	Singlemaster-Multislave	Multimaster-Multislave	Singlemaster-Singleslave
# signal lines (add gnd)	2+5 optional full-duplex	2 full-duplex	2 half-duplex	4 full-duplex	2 half-duplex	2 half-duplex
Size of network	Cable Length 15m @ 20kbs	Cable Length ~1m @ 20kbs	7-bit addr limited by 400pF	limited by extra SS lines	11-bit addr	1 to 1
Interface	+/- 12V logic level	logic level	Open drain active master	Push-pull and tristate	Differential pair	Differential pair
Voltage	min -5V and 12V required	typ 3.3V or 5V	1.8 to 5.5V	1.8 to 5V	min 4.5V diff	5V signal
Nominal Speed	115kbit/sec (~2M max)	115kbit/sec (~2M max)	~400kbit/sec (3.4M max)	~12Mbit/sec (60M max)	~1Mbit/sec (5M CAN-FD)	280Mbit/sec (10G 3.2 G2)

Other concerns

- FIFO buffer depth
  - When multi-tasking, MCU’s need some buffer to hold data before processes can get to them. This can be critical for reliable data transfer on a busy MCU
- Line length – noise will become an issue
- Packetizing protocol
  - Often data sent won’t be received, OSI-ISO level 2 handles this and resends if necessary.
- Debugging tools
  - Packet analyzer (sniffer) can be useful
  - Logic Analyzer and/or Oscilloscope

Summary Quiz

Q4 For asynchronous comm, you must set the same __________ on both ends

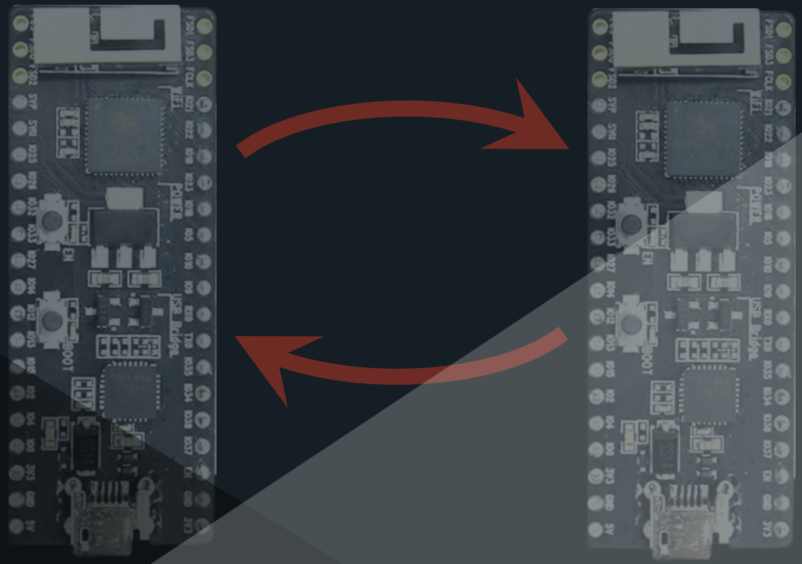
Q5 For synchronous, _________ sets the clock speed (mostly, though I2C has slightly different)

Q6 _______and _____ are two most common *synchronous* protocol used between embedded processors

Q7 ______________ can be used for simple limited state changes

05

ESP-NOW

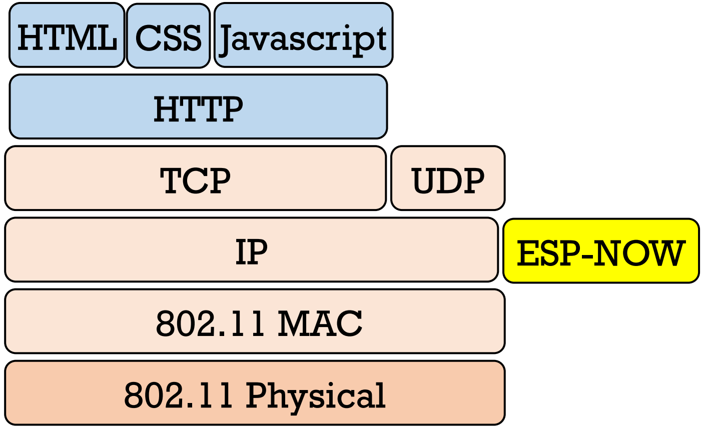


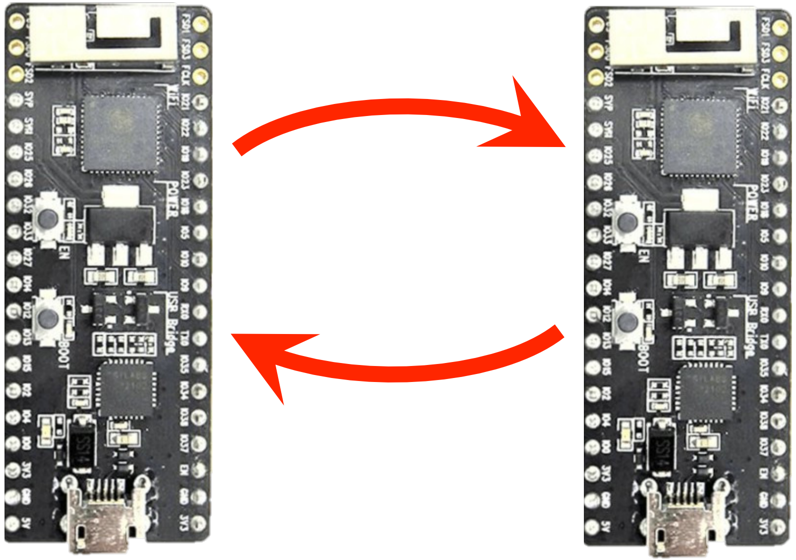
ESP-NOW Quiz

8) [True or False], like UDP, the receiver repeatedly calls a routine to check when a message has come in.
9) [True or False], ESP-NOW is good for low latency short data, bad for streaming lots of data.
10) What info about the receiver does a sender need to know?
11) What info about the sender does the receiver need to know?
12) List in terms of most to least reliable UDP, TCP, ESP-NOW

ESP-NOW overview

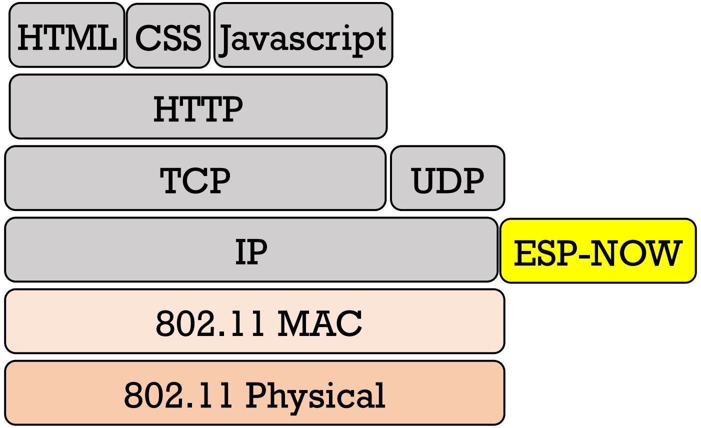
- More reliable than UDP, but not guaranteed like TCP
- ESP to ESP (up to 20 ESPs)
- 250 byte packet max size
- Low latency (low overhead)
- No router, no phone, no web.

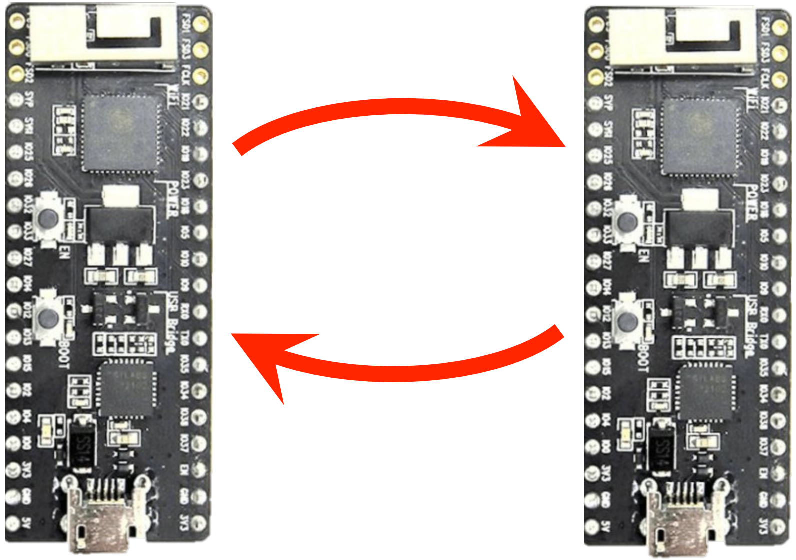




ESP-NOW

- Uses 802.11 Action Frames (not IP, no IP addresses, no connection overhead)
- Sends to MAC address
- Has some form of acknowledge
- Does not check for error or resend





7C:DF:A1:D1:12:23

7C:DF:A1:D1:45:FA

Compare TCP, UDP, ESP-NOW

	TCP	UDP	ESP-NOW
Overhead [bytes]	40+	28	39
Send Acknowledged	Yes	No	Yes
Resend automatic	Yes	No	Kind of
Order guaranteed	Yes	No	With effort
Broadcast to all	Yes	Yes	No

MAC Address (Media Access Control)

- Theoretically every device on a network has a unique MAC address that has 6 pairs of hex digits like this:

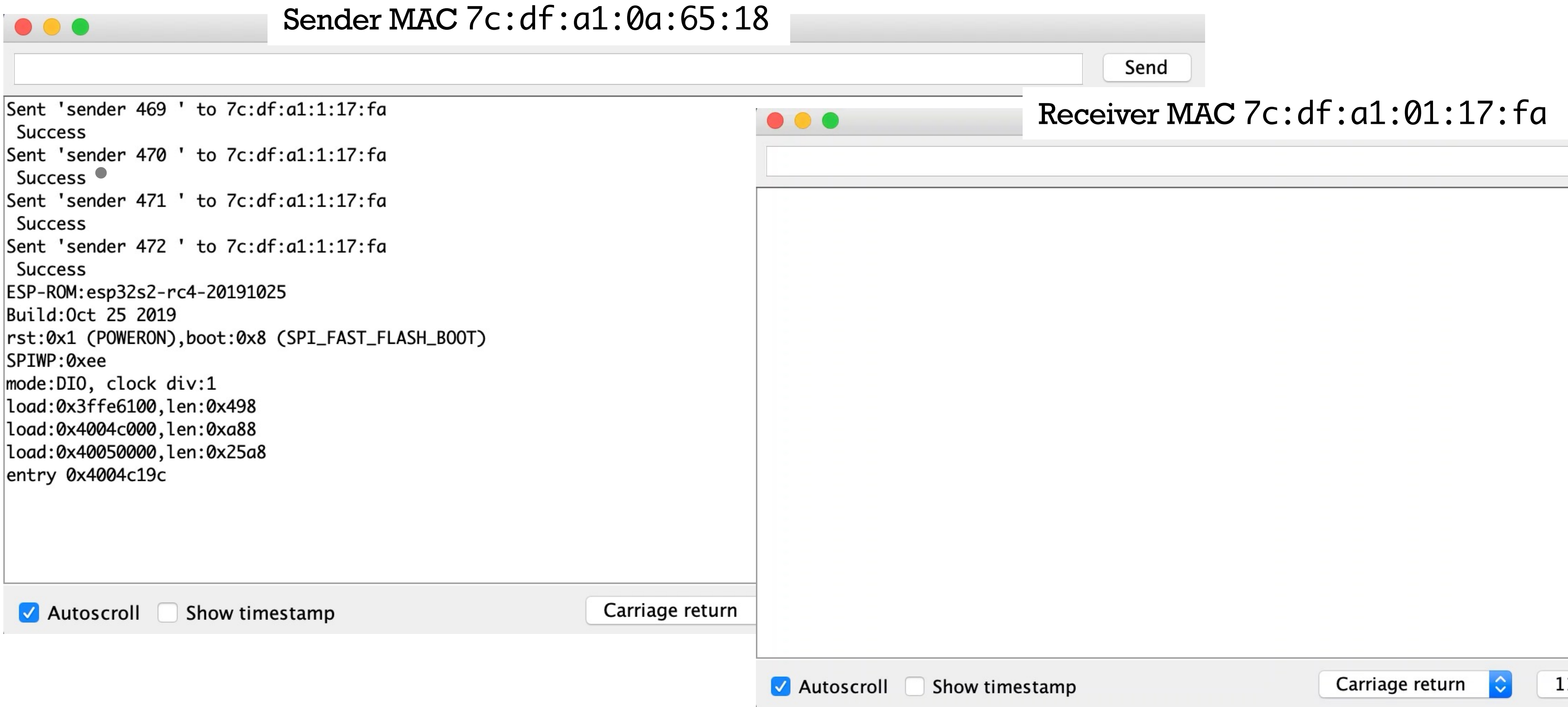
  01:23:45:67:89:AB

  - There are 2^48 = 281,474,976,710,656 unique MAC addresses
- Companies reserve blocks (the first 3 hex pairs) 16,777,216
  - Large list here: https://devtools360.com/en/macaddress/vendorMacs.xml
- Espressif has reserved 92 blocks = 1,543,503,872
- Each ESP32 actually has 4 MAC addresses
  - Base (used for Station mode)
  - Base+1 (used for Access Point mode)
  - Base+2 (used for Bluetooth)
  - Base+3 (used for Ethernet (wired))
- You can change the MAC address on the ESP32 (but don't)

Initial Testing

- Speed tests:
  - Can send 250 byte packets 100 times/sec even with four transmitters sending to one receiver simultaneously.
- Loopback tests (sending 20 bytes then sending back the data received – check to see if data is the same)
  - Sending 20 times/sec has 0 errors after 100,000 sends
  - Sending 50 times/sec has 58 errors after 100,000 sends (two senders) (sometimes double messages)
- Safe and reliable to run at 10Hz.

ESP-NOW simple send/receive demo



ESP-NOW Receiver

```
#include <esp_now.h>
#include <WiFi.h>

// callback on receive
void OnDataRecv(const uint8_t *mac_addr, const uint8_t *data, int data_len) {
  Serial.print(" Data: ");
  Serial.println( (char *)data);  // assume data is ascii string
}

void setup() {
  Serial.begin(115200);
  WiFi.mode(WIFI_STA);
  Serial.print("ESPNow Receiving MAC: ");  Serial.println(WiFi.macAddress());

  if (esp_now_init() != ESP_OK) {
    Serial.println("ESPNow Init Failed");
  }
  esp_now_register_recv_cb(OnDataRecv);
}

void loop() {
}
```

ESP-NOW Sender Setup [Part 1 / 2]

```
#include <esp_now.h>
#include <WiFi.h>

esp_now_peer_info_t peer1 = {
  .peer_addr = {0x7C, 0xDF, 0xA1, 0x01, 0x17, 0xFA}, // receiver MAC address
  .channel = 1,          // channel can be 1 to 14, channel 0 means current channel.
  .encrypt = false,
};

void setup() {
  Serial.begin(115200);
  WiFi.mode(WIFI_STA);
  Serial.print("STA MAC: "); Serial.println(WiFi.macAddress());
  if (esp_now_init() != ESP_OK) {
    Serial.println("init failed");    while (1) ;  // stop
  }

  esp_now_register_send_cb(OnDataSent);          //optional send callback
  if (esp_now_add_peer(&peer1) != ESP_OK) {      // must add peer to send
    Serial.println("Pair failed");    while (1) ;  // stop
  }
}
```

ESP-NOW Sender Loop [Part 2/2]

```
// optional callback when data is sent
void OnDataSent(const uint8_t *mac_addr, esp_now_send_status_t status) {
  if (status == ESP_NOW_SEND_SUCCESS) Serial.println ("Success ");
  else Serial.println("Fail ");
}


void loop() {
  static int count;
  uint8_t message[200];
  sprintf((char *) message, "sender %d ", count++);    // make a message

  if (esp_now_send(peer1.peer_addr, message, sizeof(message))==ESP_OK)
      Serial.printf("Sent '%s' to %x:%x:%x:%x:%x:%x \n", message,
peer1.peer_addr[0],peer1.peer_addr[1],peer1.peer_addr[2],peer1.peer_addr[3],peer1.peer_
addr[4],peer1.peer_addr[5]);
  else Serial.println("Send failed");

  delay(100);
}
```

ESP-NOW Summary Quiz

1. [True or False], like UDP, the receiver repeatedly calls a routine to check when a message has come in.
2. [True or False], ESP-NOW is good for low latency short data, bad for streaming lots of data.
3. What info about the receiver does a sender need to know?
4. What info about the sender does the receiver need to know?
5. List in terms of most to least reliable UDP, TCP, ESP-NOW

Answer in CHAT

Answer how you feel about each topic below with:

1. I don't understand this topic at all
2. I don't know now, but know what to do to get by
3. I understand some, but expect to get the rest later
4. I understand completely already

A. Ranging sensors
B. Master-Slave vs Peer-Peer, Async vs Sync, Half/Full duplex
C. Using I2C on ESP32 if needed.