Communication Protocols

Notes and/or Reference

Joe Steinmeyer

6.111 October 24, 2017

Overview of Today

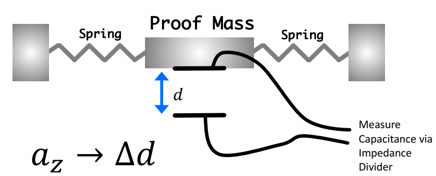
- How does an Accelerometer Work and how can we use it?
- Discussion of Particular Chip-to-Chip Communication Protocols
- Discussion of Device-to-Device Communication Protocols

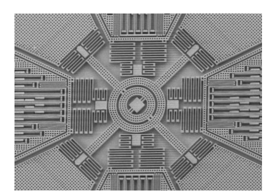
Case Study

- 9 axis IMU (Inertial Measurement Unit)
  - Accelerometer
  - Gyroscope
  - Magnetometer
- One of the only real MEMS (MicroElectroMechanical Systems) applications that has gone full-scale (others might be TI's DMD, gyroscopes, microphones, some microfluidics, Si resonators, Piezoelectrics from Inkjets, etc…)

Accelerometers

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





SEM of two-axis accelerometer

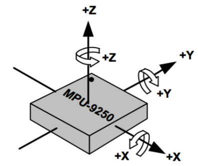
Uses of Acceleration Measurements:

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

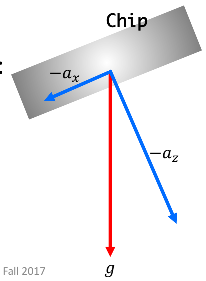
$$a_T = \sqrt{a_x^2 + a_y^2 + a_z^2}$$

- Use gravity and trig to find orientation:

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



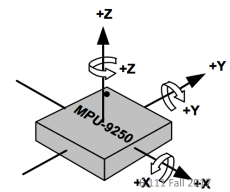
Accelerometer directions
+X, +Y, +Z

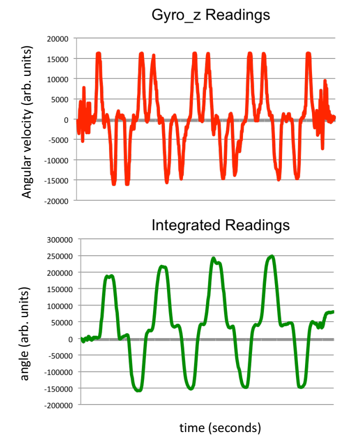


Bring in Gyroscopes

- 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):





Problems

- Accelerometers have huge amounts of high-frequency noise
- To fix, usually Low Pass Filter the raw signal (IIR 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

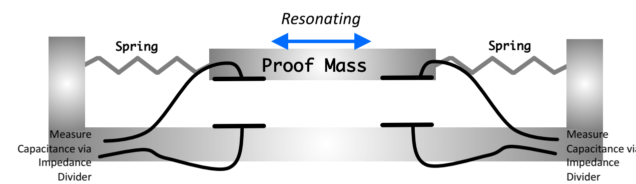
a_z z acceleration

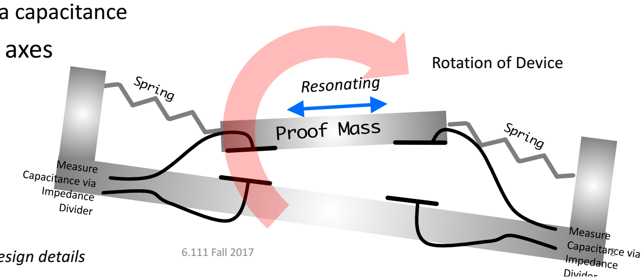
$0 < \beta < 1$ Filter Coefficient

θ_y Angle estimate around y axis

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:

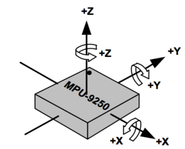
- Because of Drift (low frequency noise/offset) you want to avoid doing much long-term integration with a gyro reading
- Having beta less than unity ensures any angle that comes from gyro reading will eventually disappear, but in short term it will dominate
- Depending on time step:

$$\theta_g[n] = \beta\theta_g[n-1] + Tg_y[n-1]$$

$0 < \beta < 1$ Filter Coefficient g_y Gyro y reading

$\beta \approx 0.95$ starting point T Time Step

Merge the signals



- **Complementary Filter:**

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

$0 < \beta < 1$ Filter Coefficient g_y Gyro y reading a_x X acceleration

T Time Step $\beta \approx 0.95$ good starting point a_z z acceleration

- Could also do Kalman Filter (LQE) if desired (or others)

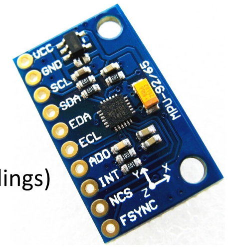
What to do?

- 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?

How to get Access to the signals in first place?

- Some accelerometers are analog out (can therefore read them with an A-to-D converter) (ADXL335, for example)
- These have limited functionality…and also it is analog so there's the whole noise issue....which is not nice
- Most flavors of sensors are digital

MPU-9250



Board: $5.00 from Ebay
Chip: $3.00 in bulk

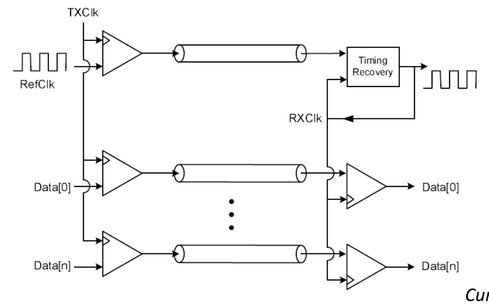
- 3-axis Accelerometer (16-bit readings)
- 3-axis Gyroscope (16-bit readings)
- 3-axis Magnetic Hall Effect Sensor (Compass) (16 bit readings)
- SPI or I2C communication (!)…no analog out
- On-chip Filters (programmable)
- On-chip programmable offsets
- On-chip programmable scale!
- On-chip sensor fusion 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! ☺

Common Chip-Chip Communication Protocols

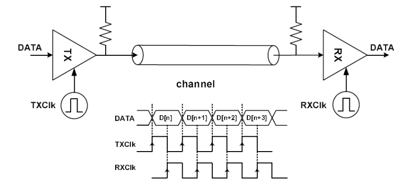
- Parallel (not so much anymore).
- Serial (UART) (still common in some communication and GPS devices)
- SPI (Serial Peripheral Interface) very common
- I2C (Inter-Integrated Circuit Communication) very common
- I2S (Inter-Integrated Circuit Sound Bus) common

Serial and Parallel at High Level

Parallel Link:



Serial Link:



Currently pushing 10-30Gb/s …

Serial Communications

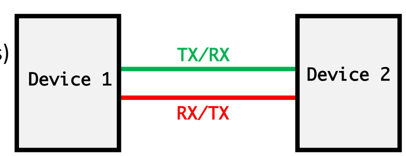
- Sending information one bit at a time vs. many bits in parallel
  - Serial: good for long distance (save on cable, pin and connector cost, easy synchronization). Requires "serializer" at sender, "deserializer" at receiver
  - Parallel: issues with clock skew, crosstalk, interconnect density, pin count. Used to dominate for short-distances (eg, between chips).
  - BUT modern preference is for parallel, but independent serial links (eg, PCI-Express x1,x2,x4,x8,x16) as a hedge against link failures.
- A zillion 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
  - Networking: RS232, Ethernet, T1, Sonet
  - Computer Peripherals: USB, FireWire, Fiber Channel, Infiniband, SATA, Serial Attached SCSI

Common Chip-Chip Communication Protocols

- Parallel (not so much anymore).
- Serial (UART) (still common in some communication and GPS devices)
- SPI (Serial Peripheral Interface) very common
- I2C (Inter-Integrated Circuit Communication) very common
- I2S (Inter-Integrated Circuit Sound Bus) common

Serial (UART)

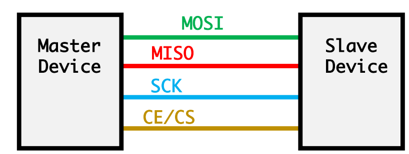
- Stands for Universal Asynchronous Receiver Transmitter
- Requires agreement ahead-of-time between devices regarding things like clock rate (BAUD), etc…
- Two wire communication
- Cannot really share
  - (every pair of devices needs own pair of lines)
- Data rate really < 115.2Kbps



Note on Terminology

- In device-to-device communication, it is common to have one device labeled the ”Master” and one labeled the “Slave”…the Master controls the Slave(s) in these settings.
- Trace history of this naming terminology back to 1940s
- I’ve seen some alternatives suggested: Leader/Follower, Primary/Secondary (other ideas?), but this naming scheme persists in the field and on data sheets
- Movement from this terminology has occurred more readily in software than hardware…Django has transitioned
- Los Angeles actually requested manufacturers to use alternative naming scheme as far back as 2003

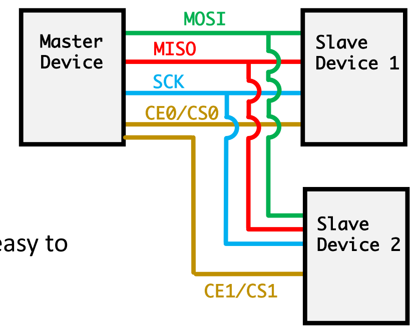
SPI



- Stands for Serial-Peripheral Interface
- Four Wires:
  - MOSI: Master-Out-Slave-In
  - MISO: Master-In-Slave-Out
  - SCK: Serial Clock
  - CE/CS (Chip Enable or Chip Select)
- SCK removes need to agree ahead of time on data rate (from UART)
- High Data Rates: (1MHz up to ~70 MHz clock (bits))

SPI

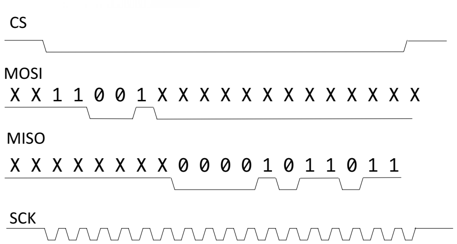
- Can share MOSI/MISO Bus
- Addition of multiple slaves requires additional select wires
- Hardware/firmware for SPI is pretty easy to implement:
  - Wires are uni-directional
  - Classic “duh” sort of approach to digital communication, but very robust.



SPI Example

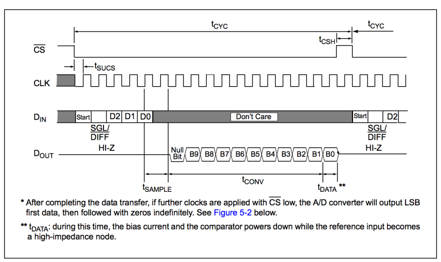
CMOD-A7-35T

MCP3008



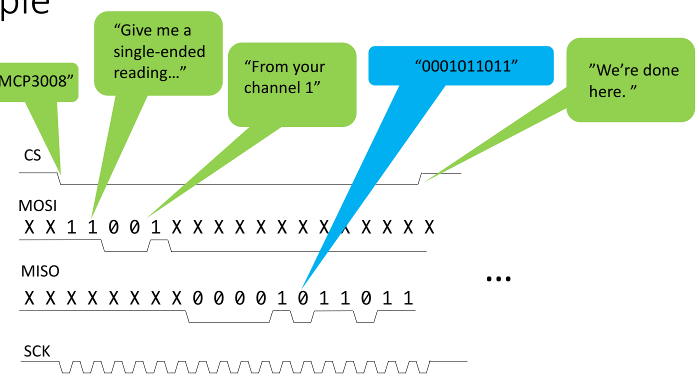
…

From MCP3008 Datasheet



Here I am talking to a MCP3008 10 bit ADC

SPI Example

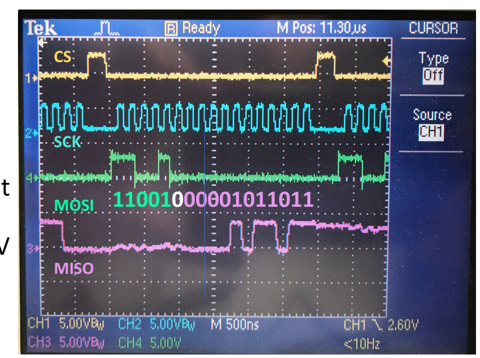


Artix-7 (Master Device) Dialog

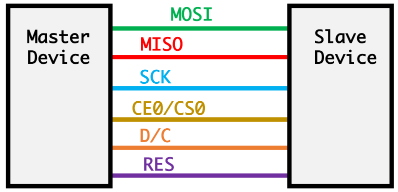
MCP3008 (Slave Device) Dialog

SPI In Real Life

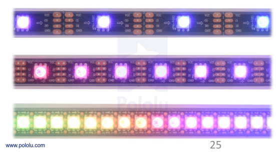
- Here I am talking to the same chip I was daydreaming about talking to on the previous slide.
- Dreams do come true
- I’m saying, give me your measurement on Channel 1, and it is responding with 10’b0001011011 mapped to 3.3V or 0.293 V



SPI*

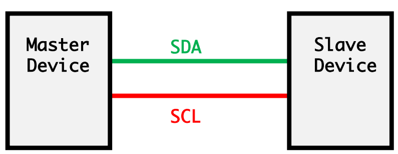


- Six Wires:
  - MOSI: Master-Out-Slave-In
  - MISO: Master-In-Slave-Out
  - SCK: Clock
  - CE/CS (Chip Enable or Chip Select)
  - RES: Reset Device
  - D/C: Data/Command (often seen in devices where you need to write tons of data (i.e. a display)
- Three/Two Wires:
  - If a device has nothing to say, drop MISO:
  - If you assume only one device on bus drop CE/CS



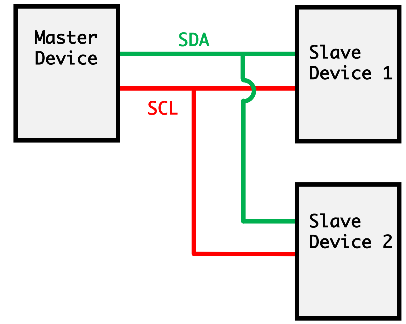
i2C

- Stands for Inter-Integrated Circuit communication
- Invented in 1980s
- Two Wire, One for Clock, one for data (both directions)
- Usually 100kHz or 400 kHz clock (newer versions go to 3.4 MHz)

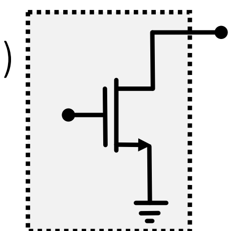


On i2C Multiple Devices Require Same # of Wires

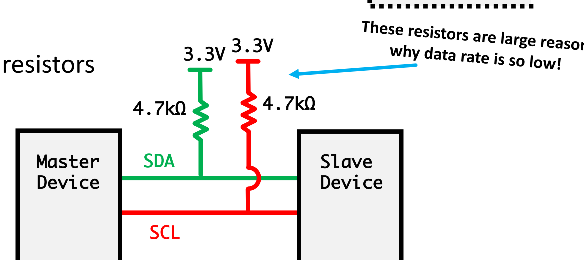
- Devices come with their own ID numbers (originally a 7 bit value but more modern ones have 10 bits)…allows potentially up to 2^7 devices or 2^10 on a bus (theoretically anyways)
- ID's are specified at build, usually several to choose from and you select them by pulling external pins HI or LOW



More to story (need pull-up resistors)



- i2C uses an open drain
- Meaning both Master and Slave Device are either:
  - LOW
  - High-Impedance
- Need external pull-up resistors



Tri-State

- inout cannot be a reg ever, ever…it is closer to a wire…usual way to work with them is the following:

In verilog…

```
inout sda;

reg sda_val;

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


i2C Operation

- Data is conveyed on SDA (Either from Master or Slave depending on point during communication)
- SCL is 50% duty cycle
- SDA generally changes on falling edge of SCL (isn't required)
- SDA sampled at rising edge of SCL
- Master is in charge of setting SCL frequency and driving it

As a result:

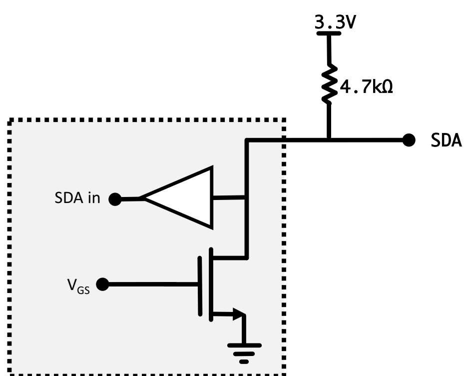
```
inout sda;
reg sda_val;
assign sda = sda_val? 1'bz: 1'b0;
```

Wanna write to SDA?

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

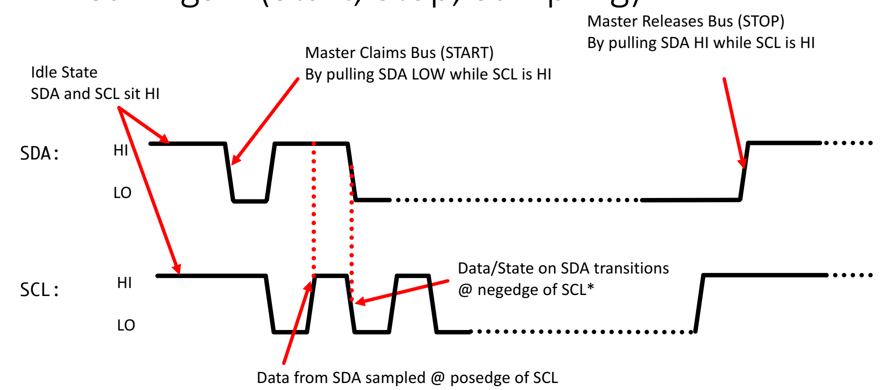
Wanna read to SDA?

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



Mode	Master	Slave
Master Transmit	HiZ (HI) or LOW	HiZ (listening)
Slave ACK/NACK	HiZ (listening)	HiZ (HI) or LOW
Slave Transmit	HiZ (listening)	HiZ (HI) or LOW
Master ACK/NACK	HiZ (HI) or LOW	HiZ (listening)

Meanings I: (Start, Stop, Sampling)



*not specified but probably easiest spot to do

Meanings II Address

- First thing sent by Master is 7 bit address (10 bit in more modern i2C…has some leading 11111’s in it..don’t worry about that)
- If a device on the bus possesses that address, it acknowledges (ACK/NACK=0) and it becomes the slave
- All other devices (other than Master/Slave Devices) will ignore until STOP signal appears later on.

Meanings III (Read/Write Bit)

- After sending address, a Read/Write Bit is specified by Master 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 Slave will start sending data out, with the Master 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 (@posedge of SCL):
  - If LOW, then receiver acknowledges data
  - If remains HI, no acknowledgement
- Transmitter/Receiver act accordingly

Meanings V

- For Master Device to write to Slave Device:
  - START
  - Send Device Address (with Write bit)
  - Send register you want to write to
  - Send data…until you’re satisfied
  - STOP
- For Master Device to read from Slave Device:
  - START
  - Send Device Address (with Write bit)
  - Send register you want to read from
  - ReSTART communication
  - Send Device Address (With Read bit)
  - Read in bits
  - After every 8 bits, it is Master’s job to acknowledge Slave…continued acknowledgement leads to continued data out by Slave.
  - Not-Acknowledge says “no more data to Slave”
  - STOP leads to Master ceasing all communication

Implementing i2C on FPGA with MPU9250:

- Made master i2C controller 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 Master

- Redundant states (repeated READ/WRITE, ADDRESS, ACK/NACK, etc…)
- ARM manual describes ~20 state FSM
- Included code on site for reference/starting point
- Diagram: on next page for reference

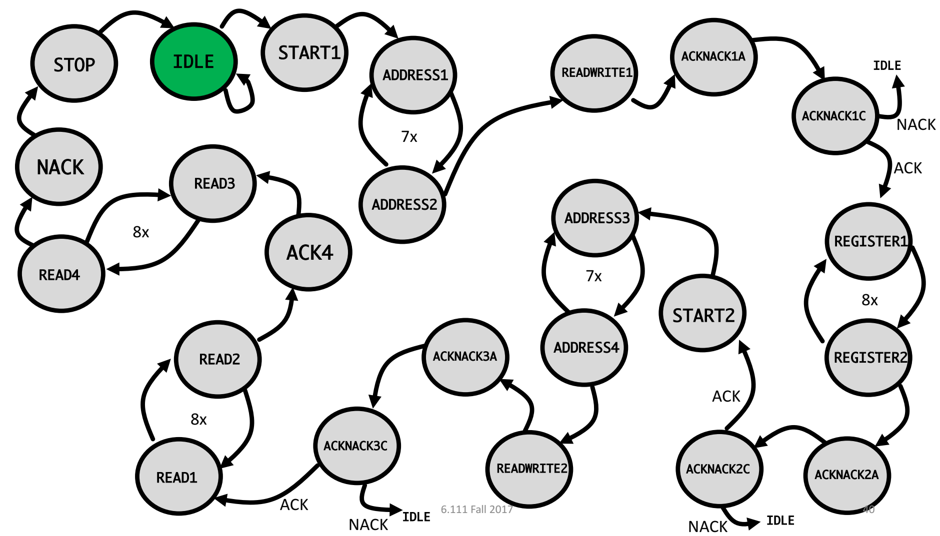
```
always @(posedge clock_for_sys)begin //update only on ri
    if (reset &&(state !=IDLE))begin
        state <= IDLE;
        count <=0;
    end else begin
        case (state)
            IDLE: begin
                if (reset) state <= IDLE;
                else if (count == 60)begin
           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

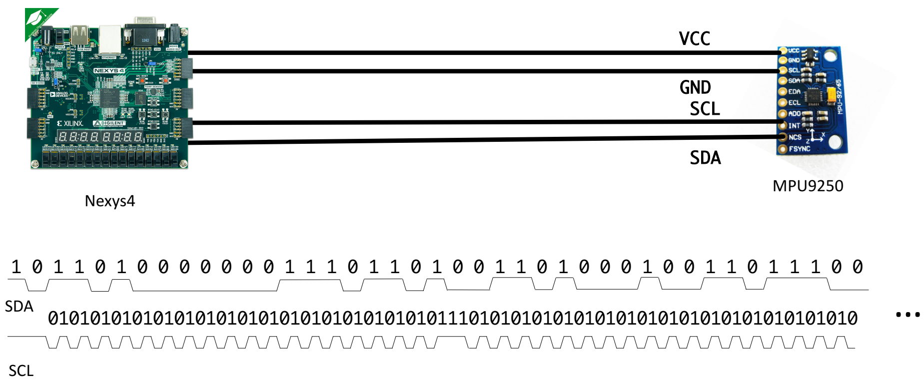
State-Machine Implementation of i2C Master

- 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

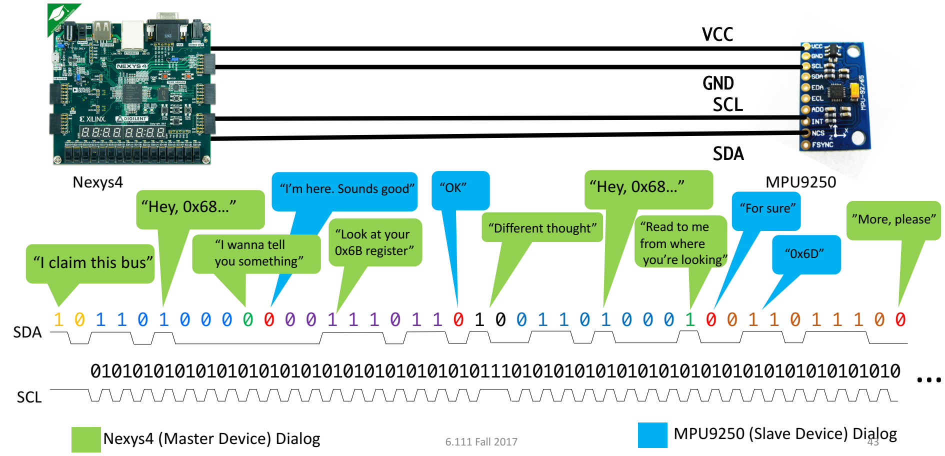
```
module i2c_master(input clock,
    input reset,
    output reg [15:0] reading,
    inout sda,
    inout scl,
    output [4:0] state_out,
    output  sys_clock);

    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 HI
```

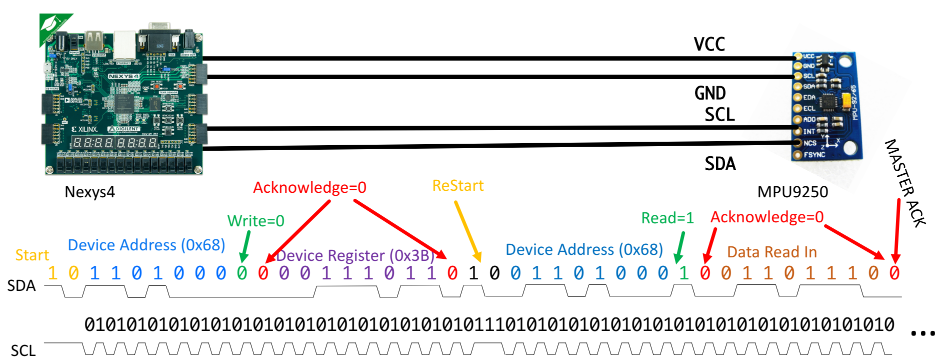
Communication Part



Communication Part

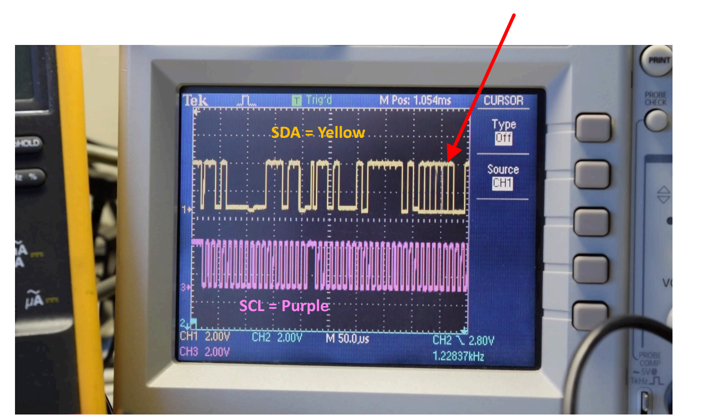


Communication Part



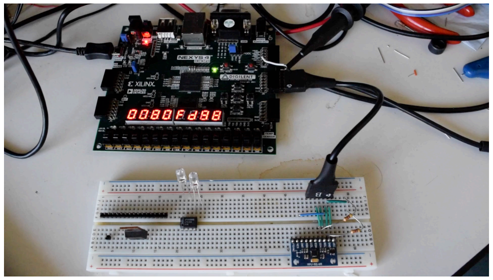
Communication in Real-Life:

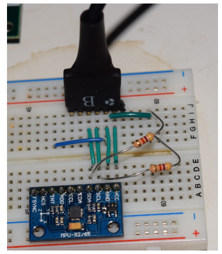
Data being sent from MPU9250



Triggered on leaving IDLE state

Running and reading X acceleration:





HOOKUP

Horizontal:

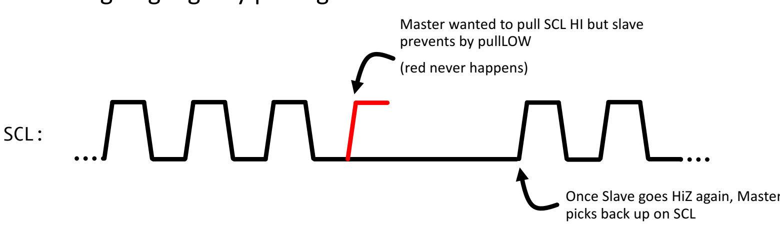
16'hFD88 = 16'b1111_1101_1000_1000 (2's complement)
Flip bits to get magnitude: 16'b0000_0010_0111_0111
=-315
Full-scale (default +/- 2g)
-315/(2**15)*2g = -0.02g ☺ makes sense

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!

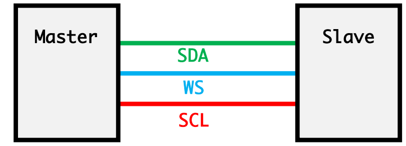
Clock-Stretching (Cool part of i2C!!!) 😻

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



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

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

i2S

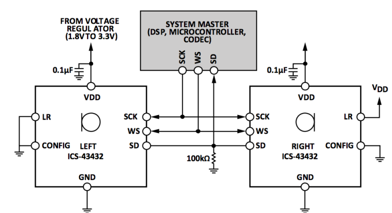
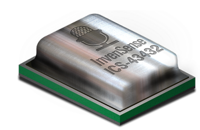


Figure 10. System Block Diagram



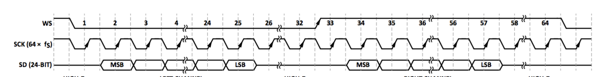


Figure 11. Stereo Output I²S Format

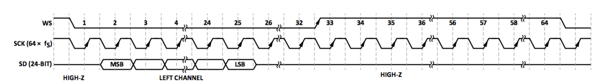


Figure 12. Mono Output I²S Format Left Channel (LR = 0)

Implementation

- You’ve built a UART/serial module already…it was pretty short/easy
- Vivado has IP cores for i2C Master and i2S Masters
- SPI is much more open to interpretation and loose on its specs so no default core that I can find:
  - I put some generic skeleton code on github/site with a FIFO buffer that can get folks started if they need it.
  - I put/will put that i2C MPU9250 code on the site as well

Compare and Contrast?

- Generally the fewer the wires the more rigid the protocol
- 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

Which to Choose?

- SPI is generally easier and more flexible to implement, but only certain devices use it since it takes up a lot of pins (and pins are expensive/limited)
- ”Slow” and “Fast” data rates are relative too…i2C is not as much of a compromise now as it was fifteen years ago, particularly with high-speed i2C (or even now that 400 kHz rates are common)
- Remember, these are all meant for chip-to-chip communications!
- Check out the example i2C code from this lecture for the IMU, and a generic SPI master I wrote up as well…see if you can add clock-stretching! (not required)

Going Between boards

- Previous protocols are meant for device-to-device communication
- There is no cabling standard for these protocols
- Distances are not specified for i2C, SPI, i2S, but think in terms of inches
- Open-Drain protocols are particularly susceptible to parasitics so keep leads short where possible!
- To go between devices we must use other protocols!

RS232 (aka “serial port”)

- Labkit: simple bidirectional data connection with computer.
- Characteristics
  - Large voltages => special interface chips
    (1/mark: -12V to -3V, 0/space: 3V to 12V)
  - Separate xmit and rcv wires: full duplex
  - Slow transmission rates (1 bit time = 1 baud); most interfaces support standardized baud rates: 1200, 2400, 4800, 9600, 19.2K, 38.4K, 57.6K, 115.2K
  - Format
    - Wire is held at 1/mark when idle
    - Start bit (1 bit of “0” at start of transmission)
    - Data bits (LSB first, can be 5 to 8 bits of data)
    - Parity bit (none, even, odd)
    - Stop bits (1, 1.5 or 2 bits of 1/mark at end of symbol)
    - Most common 8-N-1: eight data bits, no parity, one stop bit

RS232 interface

- Transmit: easy, just build FSM to generate desired waveform with correct bit timing
- Receive:
  - Want to sample value in middle of each bit time
  - Oversample, eg, at 16x baud rate
  - Look for 1->0 transition at beginning of start bit
  - Count to 8 to sample start bit, then repeatedly count to 16 to sample other bits
  - Check format (start, data, parity, stop) before accepting data.

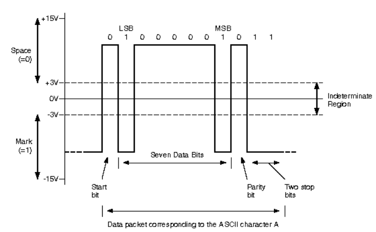
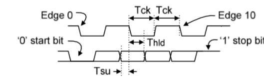


Figure from
http://www.arcelect.com/rs232.htm

Should look familiar from Lab 2!

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

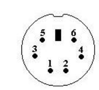


Symbol	Parameter	Min	Max
T_{CK}	Clock time	30us	50us
T_{SU}	Data-to-clock setup time	5us	25us
T_{HLD}	Clock-to-data hold time	5us	25us

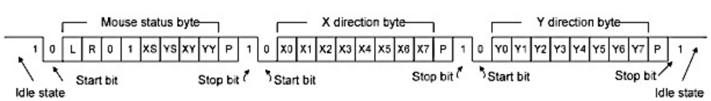
Figures from digilentinc.com

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

IDE Bus – Serial ATA (SATA)

40-Pin IDE Connector PinOut

Pin #	Signal Function	Pin #	Signal Function
1	Reset	2	Ground
3	Data 7	4	Data 8
5	Data 6	6	Data 9
7	Data 5	8	Data 10
9	Data 4	10	Data 11
11	Data 3	12	Data 12
13	Data 2	14	Data 13
15	Data 1	16	Data 14
17	Data 0	18	Data 15
19	Ground	20	Key
21	DMARQ	22	Ground
23	DIOW-	24	Ground
25	DIOR-	26	Ground
27	IORDY	28	CSEL
29	DMARK-	30	Ground
31	INTRQ	32	IOCS16-
33	DA1	34	PDIAG-
35	DA0	36	DA2
37	CS1FX-	38	CS3FX-
39	DASP-	40	Ground

SATA

Pin	Name
1	GND
2	A+
3	A-
4	GND
5	B-
6	B+
7	GND

2-wire (+,-) for high-speed

SATA 1: 1.5Gb/s
SATA 2: 3Gb/s
SATA 3: 6Gb/s

USB: Universal Serial Bus

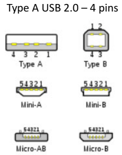
- USB 1.0 (12 Mbit/s) introduced in 1996
- USB 2.0 (480 Mbit/s) in 2000
- USB 3.0 (5 Gbit/s) in 2012
- USB-C 2016.
- USB 3.2 (30 Gbit/s) in July 20, 2017
- Created by Compaq, Digital, IBM, Intel, Northern Telecom and Microsoft.
- Uses differential bi-direction serial communications

Insert correctly

On third try

Credit: Reddit

Type A USB 2.0 – 4 pins

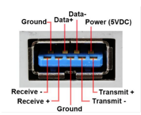


Pin	Name	Cable color	Description
1	VCC	Red	+5 V
2	D−	White	Data −
3	D+	Green	Data +
4	GND	Black	Ground

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

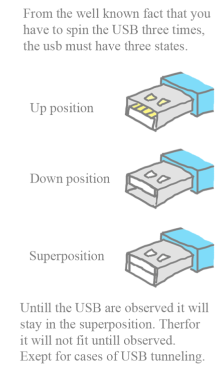
Mini/Micro Pinout



USB 3.0

USB: Universal Serial Bus

- More defined layers than your other things we’ve seen
- The 2000 version of USB spec was 570 pages long
- Current USB 3.2 (9/22/2017 release!...so new! so fresh!)
  - spec is 103 MB zip file*
  - Approximately 8,000 pages long at this point
  - I’ll summarize in a few slides



*and hosted on web page that has painfully slow DL speeds and looks like it is from 2000

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):

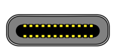
- USB uses twisted wire pairs and there is no CLOCK wire
- All data is transmitted using Non-Return-Zero-Inverted (NRZI) encoding:
  - A 0 is encoded as a value change
  - A 1 is encoded by no change
- After initial synchronization byte, the receiver extracts the clock from the on-average probability of 0's in the data (which give transitions) using local oscillator and Phase-Locked Loops (Maybe talk about in Lecture 15 because it is really cool actually)
- 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
- Capable of up to 30 Gbit/s
  - ~2 HD movies per second

USB - C

- Universal connector for power and data – first product MacBook Air – one and only port!
- Symmetrical – no 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..
- New adapters required for DisplayPort, HDMI, power, USB, and VGA – omg!

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

Potential Problems

- If we all followed the laws life would be grand
- Not everyone can read all 8,000 pages
- Not everyone *wants* to read all 8,000 pages
- Difference between 5V and 20V going into your laptop is now based on software handshakes between two devices.
- Do you trust your devices?
- Solution is now to do hardware verification prior to any power delivery using table of approved-devices for via 128 bit encryption (mid 2016)
- It'll be interesting to see how quickly this gets hacked

Getting data back to the board…

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
- Extremely common

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

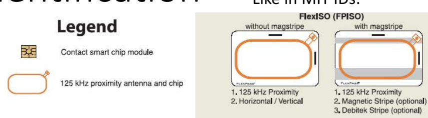
Human Interface Device Classes

- Complex, yet implementable communication protocol that utilizes widely accepted protocol:
- Have a device and/or FPGA directly run implement that part of the USB stack
- Can implement in ~10 state FSM or so
- Appear as a "mouse" or a "keyboard" or a "webcam", etc…
- Medium speeds…really need specialized hardware for the super speeds

RFID: Radio Frequency Identification

- Used to provide remote interrogation/identification
- Frequency bands:
  - 125 - 134 kHz [MIT ID]*
  - 13.56 MHz [US Passports]*
  - 400 – 960 MHz UHF
    [EZPASS 915mhz ~ 1 mw]**
  - 2.45 GHz
  - 5.8 GHz

Like in MIT IDs:



Transmitting antenna

Battery

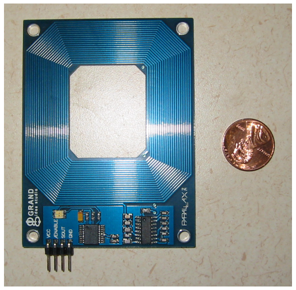
EZ Pass Internals

* excitation/broadcast powered

** battery powered

*http://groups.csail.mit.edu/mac/classes/6.805/student-papers/fall04-papers/mit_id/#specs

125khz RFID



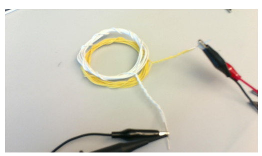
125khz transmitter

Receiver

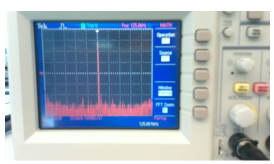
Powered by 125khz broadcast signal

MIT RFID

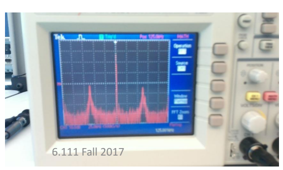
- 125 kHz carrier
- 62.5 kHz modulating wave phase-shifts every 16 cycles:
  - π shift indicates a 1
  - No shift indicates a 0
- …so we've got:
- Phase-shift-encoded Non-Return-to-Zero-Mark Encoding (NRZ-M)



Stimulating and Receiving Coils



FFT of Pickup on Receiving Coil while Stimulating Coil has 125 kHz driven into it and NO CARD in between (Spike is 125 kHz centered)



FFT of Pickup on Receiving Coil while Stimulating Coil has 125 kHz driven into it and CARD is in between (LOOK AT THAT SIDEBAND ACTION!!!)

Next Monday

- Potpourri…
- FFTs
- Bluetooth, BLE discussion
- Maybe PLLs
- Some other stuff