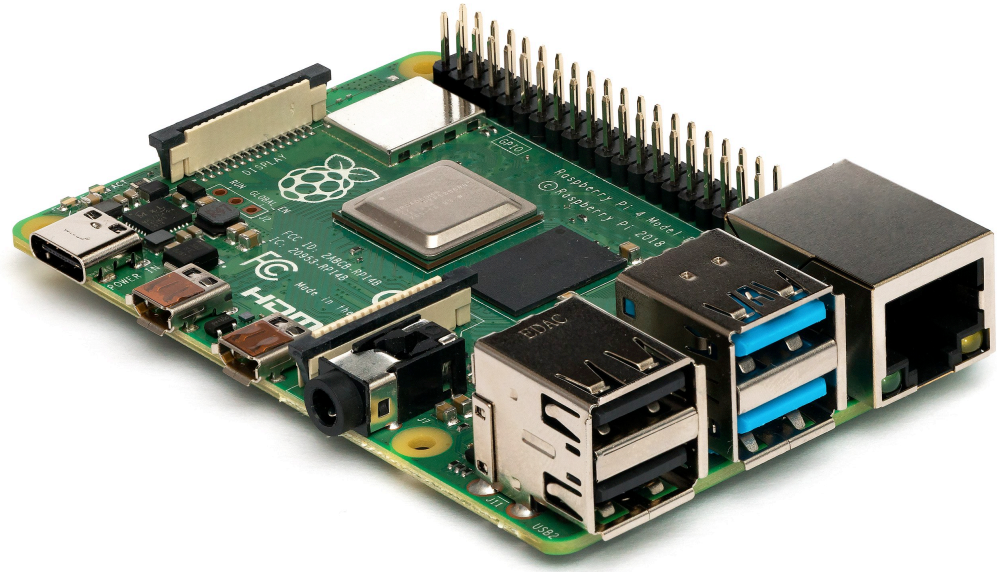


Raspberry Pi and Interfacing

Linux

Python

Interfaces

The Point

- Experiments often mean measuring and recording data
  - sense
  - digitize
  - communicate
  - automate
  - store
  - analyze
  - publish
  - fame and glory?

Focus on Accessible

- Oceans of possibilities for data acquisition/interface
- Raspberry Pi is:
  - cheap (you can have your own)
  - cheap (software is free)
  - cheap (low-cost accoutrements, like ADC)
- Other RPi benefits:
  - familiarizes with Linux & Python
    - means Pi can run very advanced/sophisticated code, if needed
  - supports loads of modern interfaces
    - I^2C, SPI, serial, GPIO, USB
  - can play "nice" with research-grade interfaces
    - telnet, ssh, other network interfaces

Linux (Unix) Environment

- Command-line interface (terminal session)
- Will want to find and work through tutorials
- Essential commands:
  - `cd` (and meaning of `.`, `..`), `mkdir`, `ls` (and `ls -l`), `cp`, `rm`, `mv`, `pwd`, `vi` or `nano`, `less`, `head`, `tail`, `cat`, `grep`, `wc` (word count), `|` (pipe), `>` (stuff into file), `<` (source from file), `chmod`, `passwd`, `exit`, etc.
  - familiarize yourself with at least these (and associated arguments/flags)
  - use "man" (manual) pages for details:
    - `man mkdir`
- Mac computers have Unix foundation, so prevalent OS

Raspberry Pi Access

- Pi4 units in lab; one per bench; “headless”
- Access via ssh or putty on lab machines
- hostname: bench1, bench2, etc.
- username: bench1, bench2, etc. (matches unit/bench)
- password: bench1, bench2, etc.
  - temporary: suggest changing after you & partner establish your bench (share/decide with partner)
  - command: `passwd`

Python Language

- Prevalent in Physics/Astro
- Interpreted (slower than compiled)
- Easy syntax (high level, readable)
- Exceptionally good at string parsing/handling
- Libraries provide powerful functionality
  - numpy: math on vectors/arrays
  - scipy: special functions, optimization
  - matplotlib (pylab): plotting, a la MatLab
  - boatloads of others (many included in standard installation: math, sys, os, time, re, as a start)

Python Tutorials

- Finding your own resources, learn how to:
  - run interactively to explore syntax; use `dir()` and `help()`
  - use lists, tuples, dictionaries; list comprehension
  - perform math: `import math`; `dir(math)`
  - create/invoke/run program (next slide)
  - control flow: `if`/`else`; `for`/`do`/`while`
  - format print statements: `%s`, `%d`, `%5.2f`, etc.
  - use command line arguments: `float(sys.argv[3])`, e.g.
  - read from file: `open()`; `for line in file_handle`; `close()`
  - write to file: `file_handle.write(`formatted_string`)`
- Example: Google: python list comprehension tutorial

Example Python Creation/Execution

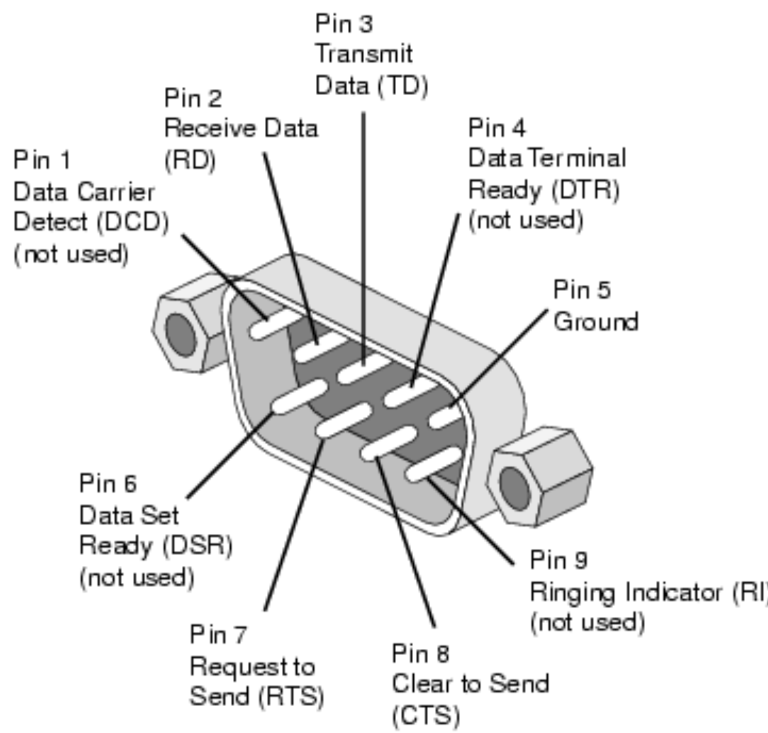
```
$ mkdir sandbox                 (create place to mess around)
$ cd sandbox                    (navigate into directory)
$ vi test.py                    (or edit using nano, emacs, etc)
#!/usr/bin/env  python          #top line of file; invoke Python
import sys                      #so we can use command line arg.
name = sys.argv[1]              #not checking to verify exist.
print "Hello, %s" % name        #formats personalized output
(save and quit)
$ chmod +x test.py              (do once: make file executable)
$ ./test.py Tom                 (run with ./ and incl. argument)
Hello, Tom                      (output)
$                               (prompt)
```

Interfaces

- A moving target, as technology changes
  - serial (RS-232), USB, I2C, SPI are common
    - Raspberry Pi does these, plus GPIO (Gen. Purp. Input/Output)
  - GPIB, CAMAC, VME/VXI, PCI cards (DAQ) for lab environ.

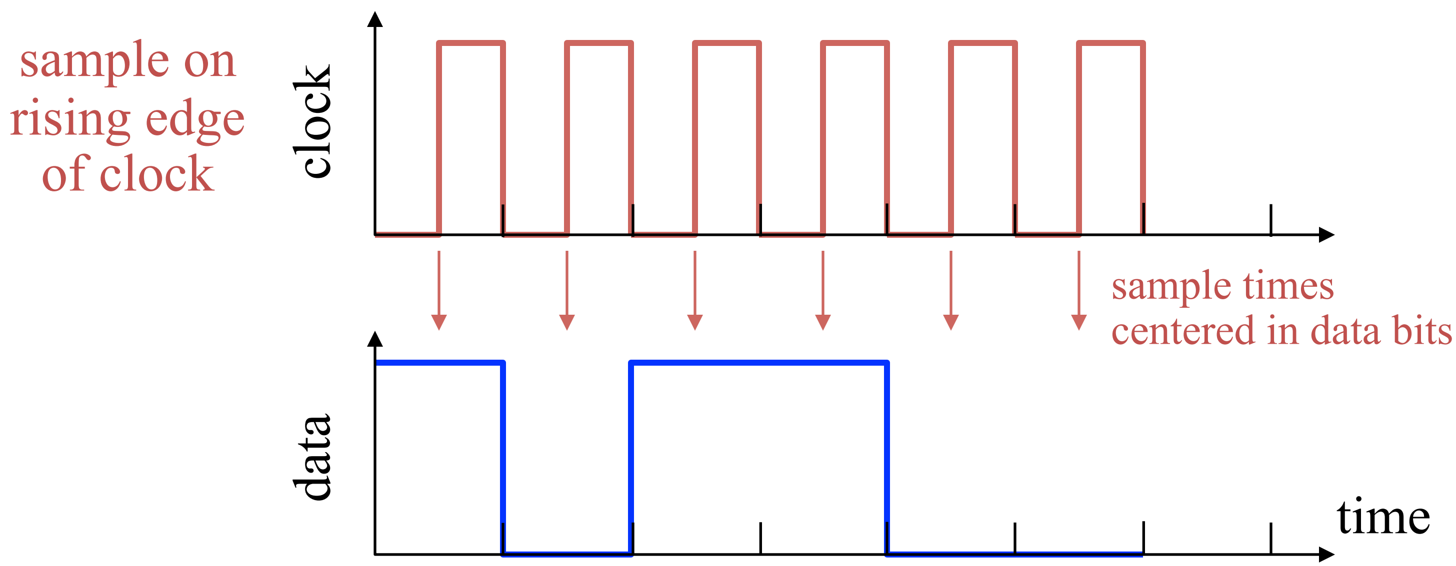
Serial Communications

- Most PCs have a DB9 male plug for RS-232 serial asynchronous communications
  - we'll get to these definitions later
  - often COM1 on a PC
- In most cases, it is sufficient to use a 2- or 3-wire connection
  - ground (pin 5) and either or both receive and transmit (pins 2 and 3)
- Other controls available, but seldom used
- Data transmitted one bit at a time, with protocols establishing how one represents data
- Slow-ish (most common is 9600 bits/sec)



Time Is of the Essence

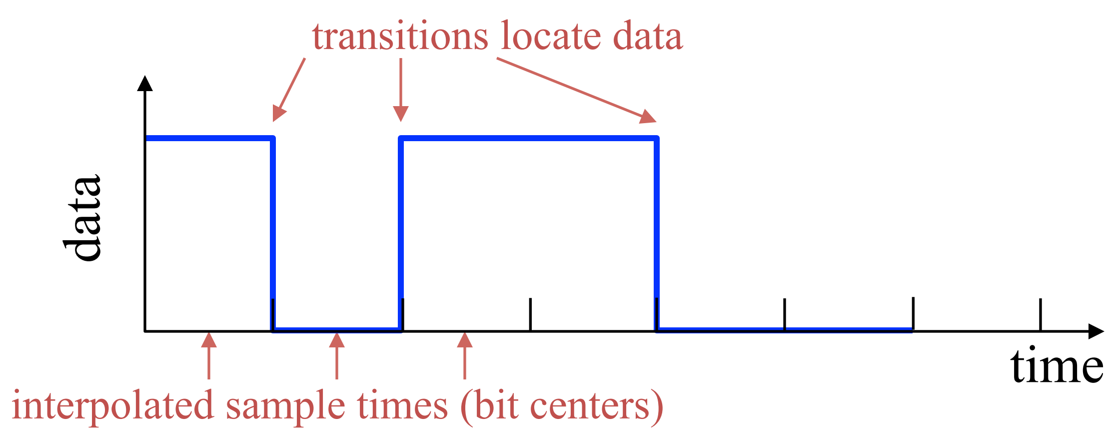
- If provided separate clock and data, the transmitter *gives* the receiver timing on one signal, and data on another
- Requires two signals (clock and data): can be expensive (but I^2C, SPI does this)
- Data values are arbitrary (no restrictions)
- As distance and/or speed increase, **clock/data skew** destroys timing



slide courtesy E. Michelsen

No Clock:
Do You Know Where Your Data Is?

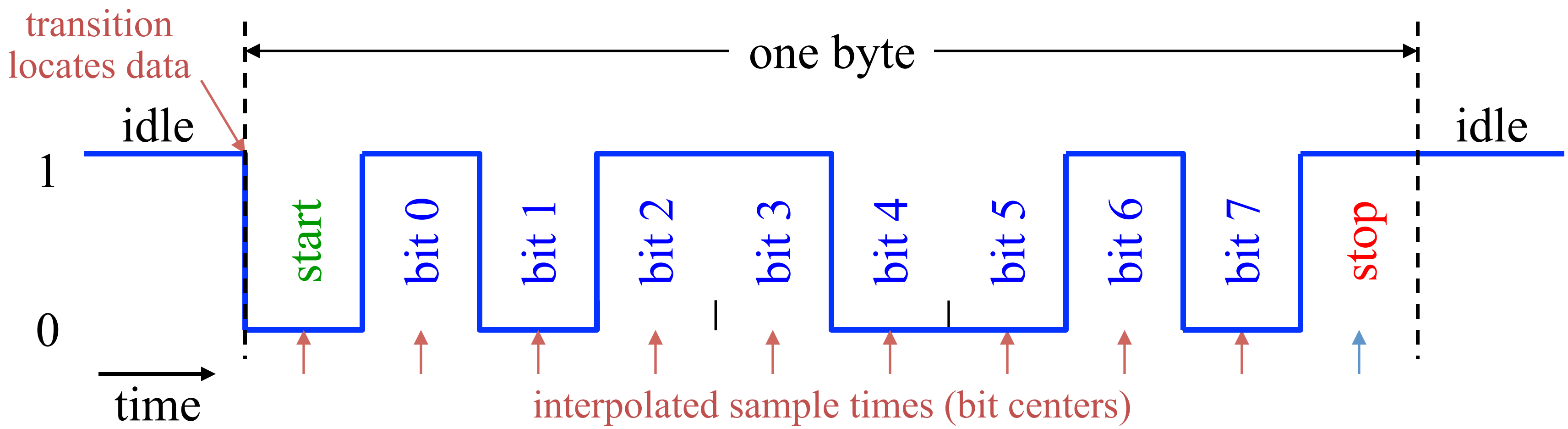
- Most long-distance, high speed, or cheap signaling is **self timed**: it has no separate clock; the receiver recovers timing from the signal itself
- Receiver knows the *nominal* data rate, but requires **transitions** in the signal to locate the bits, and interpolate to the sample points
- Two General Methods:
  - Asynchronous: data sent in short blocks called **frames**
  - Synchronous: continuous stream of bits
    - Receiver *tracks* the timing continuously, to stay in synch
    - Tracking requires sufficient **transition density** throughout the data stream
    - Used in all DSLs, DS1 (T1), DS3, SONET, all Ethernets, etc.



slide courtesy E. Michelsen

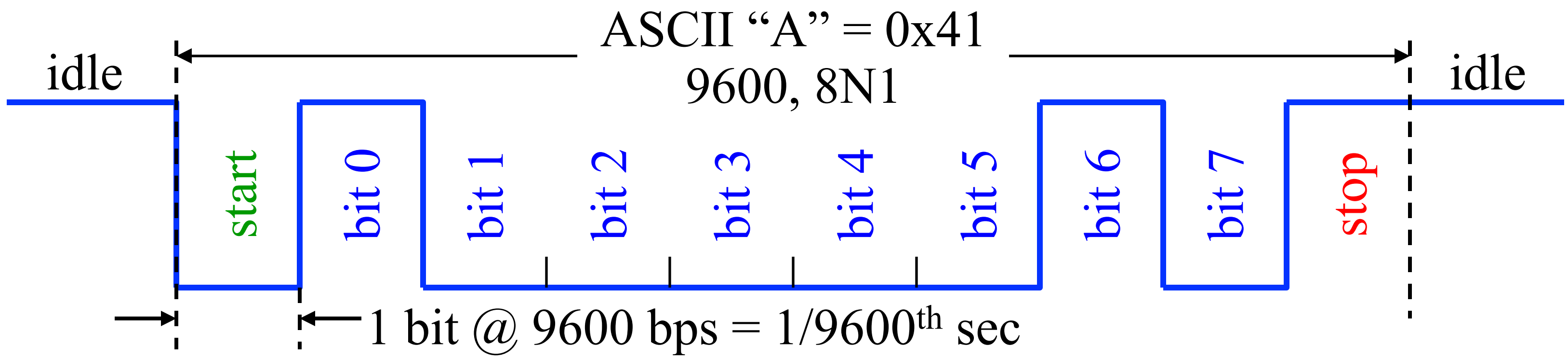
Asynchronous: Up Close and Personal

- **Asynchronous**
  - technical term meaning "whenever I feel like it"
- Start bit is always 0. Stop bit is always 1.
- The line "idles" between bytes in the "1" state.
- This guarantees a 1 to 0 transition at the start of every byte
- After the leading edge of the start bit, if you know the data rate, you can find all the bits in the byte



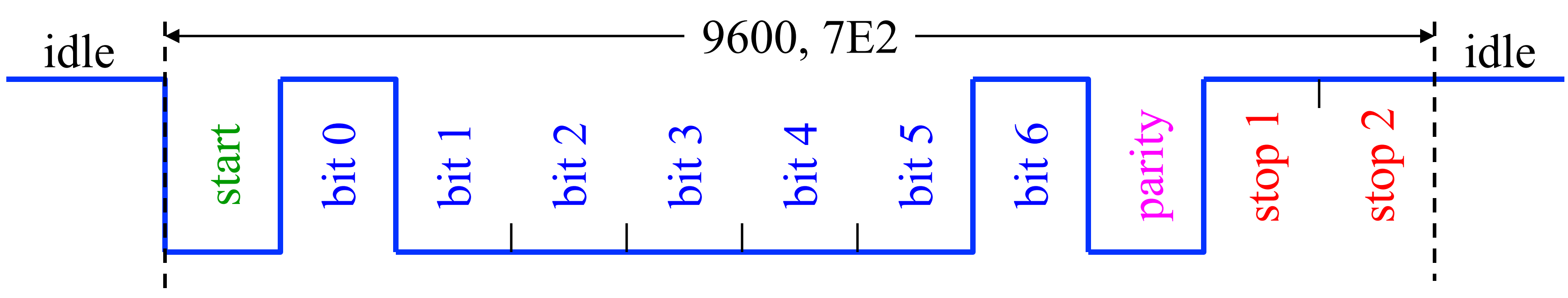
slide courtesy E. Michelsen

Can We Talk?



- If we agree on 4 asynchronous communication parameters:
  - Data rate: Speed at which bits are sent, in bits per seconds (bps)
  - Number of data bits: data bits in each byte; usually 8
    - old stuff often used 7
  - Parity: An error detecting method: None, Even, Odd, Mark, Space
  - Stop bits: number of stop bits on each byte; usually 1.
    - Rarely 2 or (more rarely) 1.5: just a minimum wait time: can be indefinite

Note: LSB sent first



slide courtesy E. Michelsen

RS-232: most common implementation

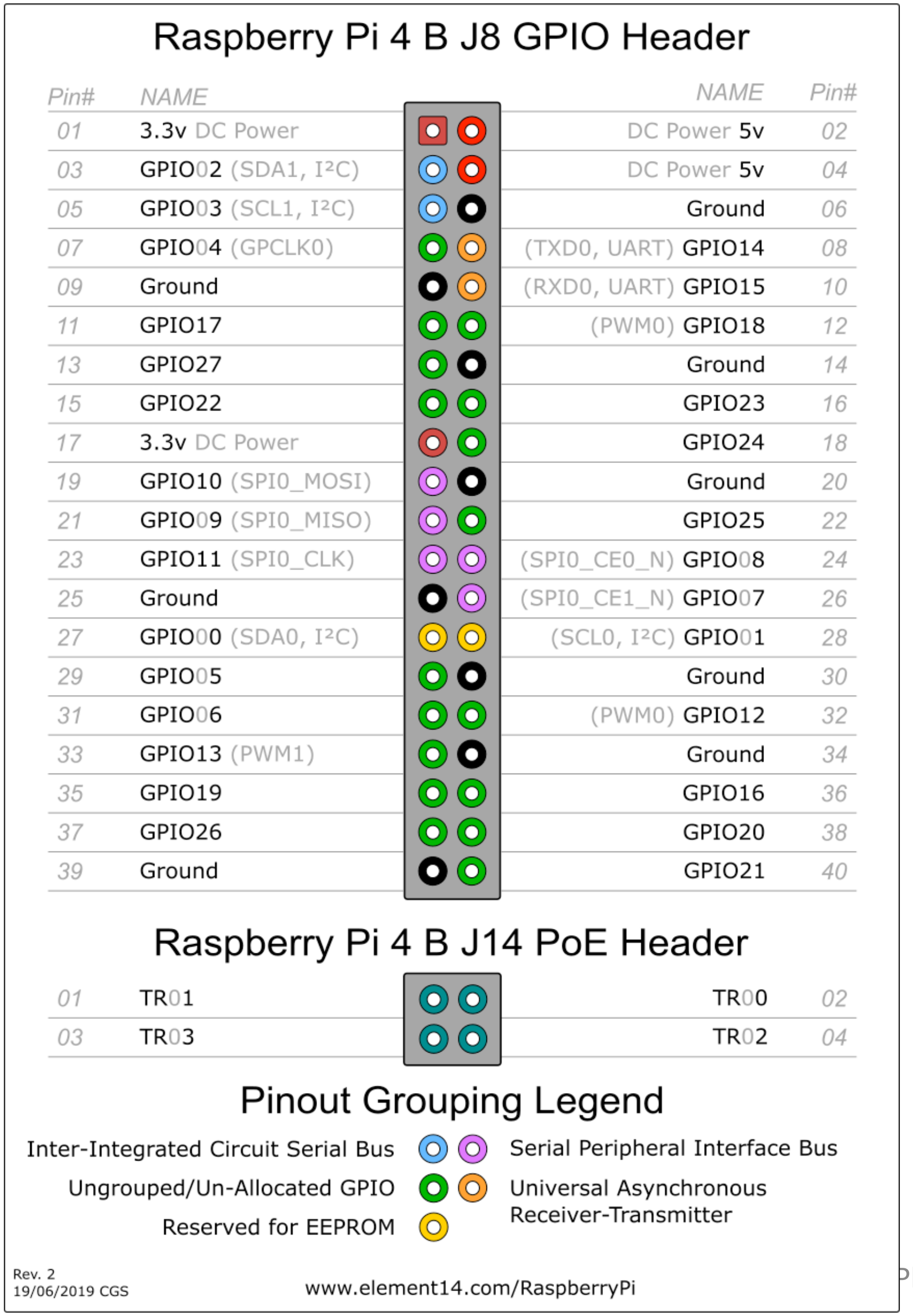
- RS-232 is an electrical (physical) specification for communication
  - idle, or "mark" state is logic 1;
    - -5 to −15 V (usually about −12 V) on transmit
    - -3 to −25 V on receive
  - "space" state is logic 0;
    - +5 to +15 V (usually ~12 V) on transmit
    - +3 to +25 V on receive
  - the dead zone is from −3 V to +3 V (indeterminate state)
- Usually used in asynchronous mode, defined by parameters on prev. slide
  - so idles at −12; start jumps to +12; stop bit at −12
  - since each packet is framed by start/stop bits, guaranteed a transition at start
  - parity (if used) works as follows:
    - even parity guarantees an even number of ones in the train
    - odd parity guarantees an odd number of ones in the train
- UART: Universal Asynchronous Receiver/Transmitter
  - common term/label for a serial interface

GPIB (IEEE-488)

- An 8-bit parallel bus allowing up to 15 devices connected to the same computer port
  - addressing of each machine (either via menu or dip-switches) determines who's who
  - can daisy-chain connectors, each cable 2 m or less in length
- Extensive handshaking controls the bus
  - computer controls who can talk and who can listen
- Many test-and-measurement devices equipped with GPIB
  - common means of controlling an experiment: positioning detectors, measuring or setting voltages/currents, etc.
- Can be reasonably fast (1 Mbit/sec)

Data Acquisition

- A PCI-card for data acquisition is a very handy thing
- The one pictured at right (National Instruments PCI-6031E) has:
  - 64 analog inputs, 16 bit
  - 2 DACs, 16 bit analog outputs
  - 8 digital input/output
  - 100,000 samples per second
  - on-board timers, counters
- Breakout box/board recommended



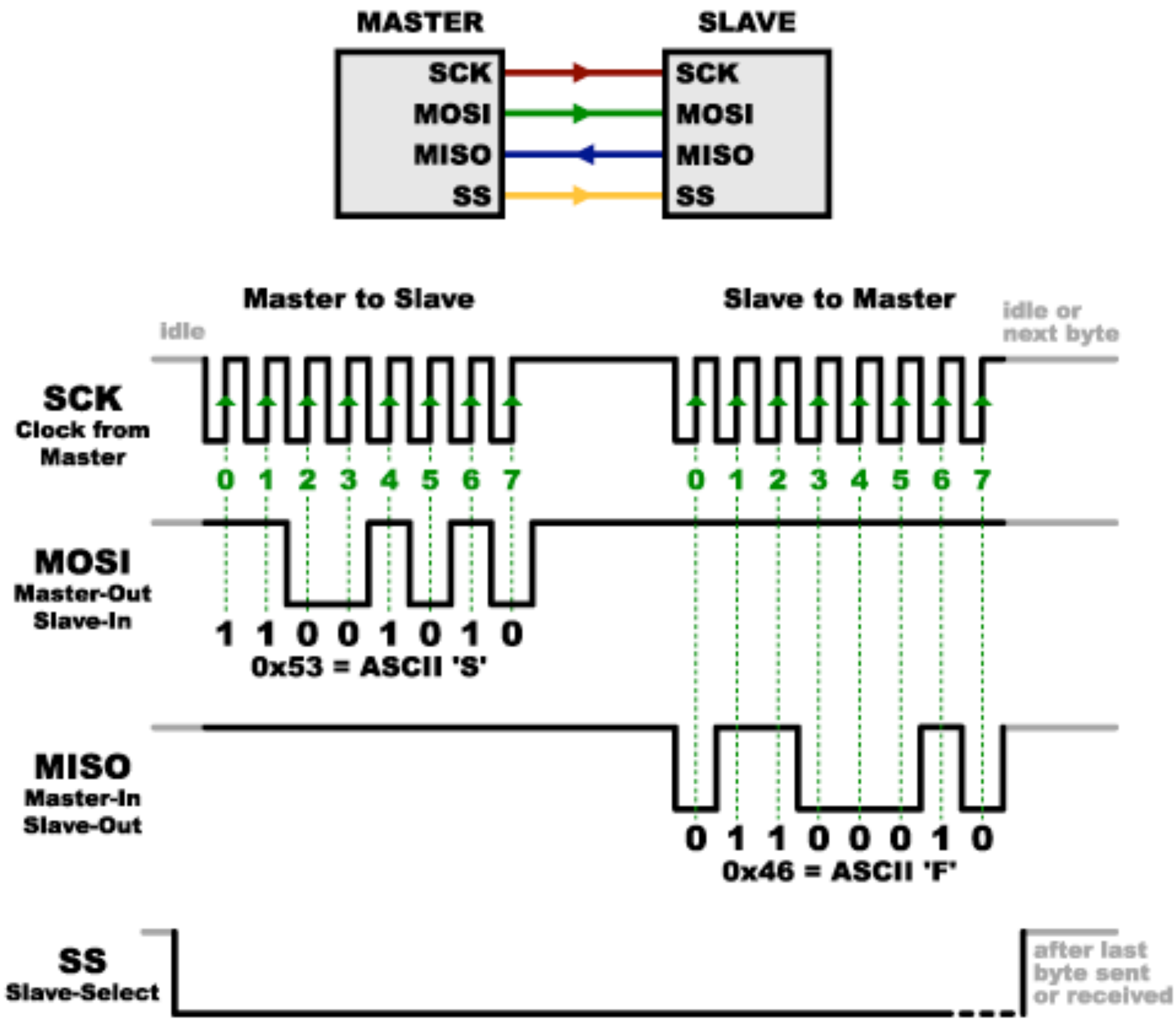
RPi Interface

- 40-pin header on side of RPi
- serial is orange (UART)
- I^2C is light blue
- SPI is purple
- GPIO is green
  - and can also use any pin labeled GPIOxx

SPI: Serial Peripheral Interface

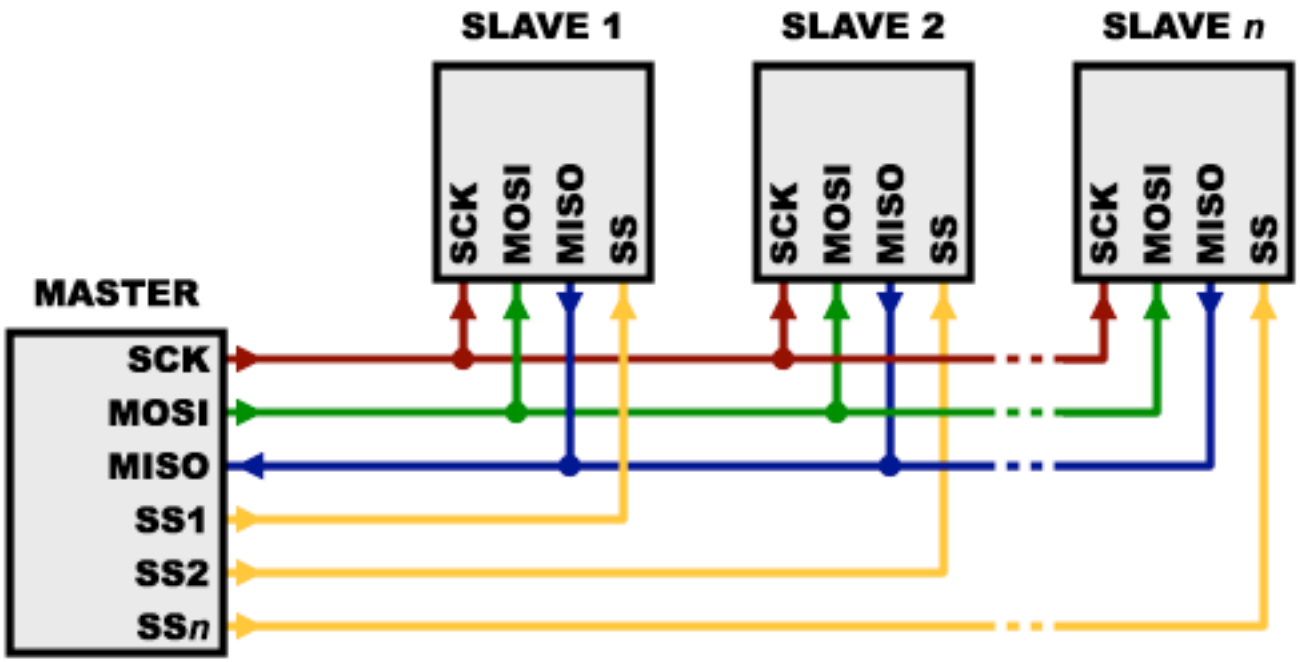
- 4 lines (plus ground reference, as always)
  - clock (CLK)
  - data "in" (MISO: master in, slave out)
  - data "out" (MOSI: master out, slave in)
  - chip enable (CE#_N: usually active low)
    - RPi has two CE lines
    - sometimes called chip select (CS) or slave select (SS)
- Synchronous Form
- Naming resolves ambiguity about data direction
  - TX/RX always confusing: according to which device?

SPI Scheme



from sparkfun.com

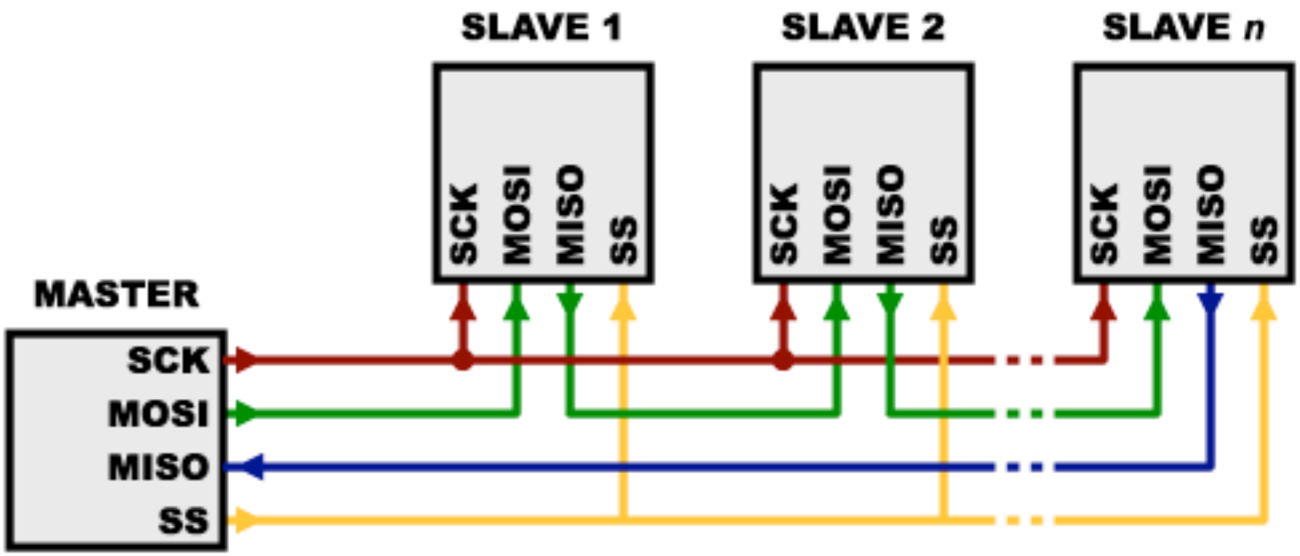
Multiple Devices



from sparkfun.com

Device only listens when its CE/CS/SS line is pulled low

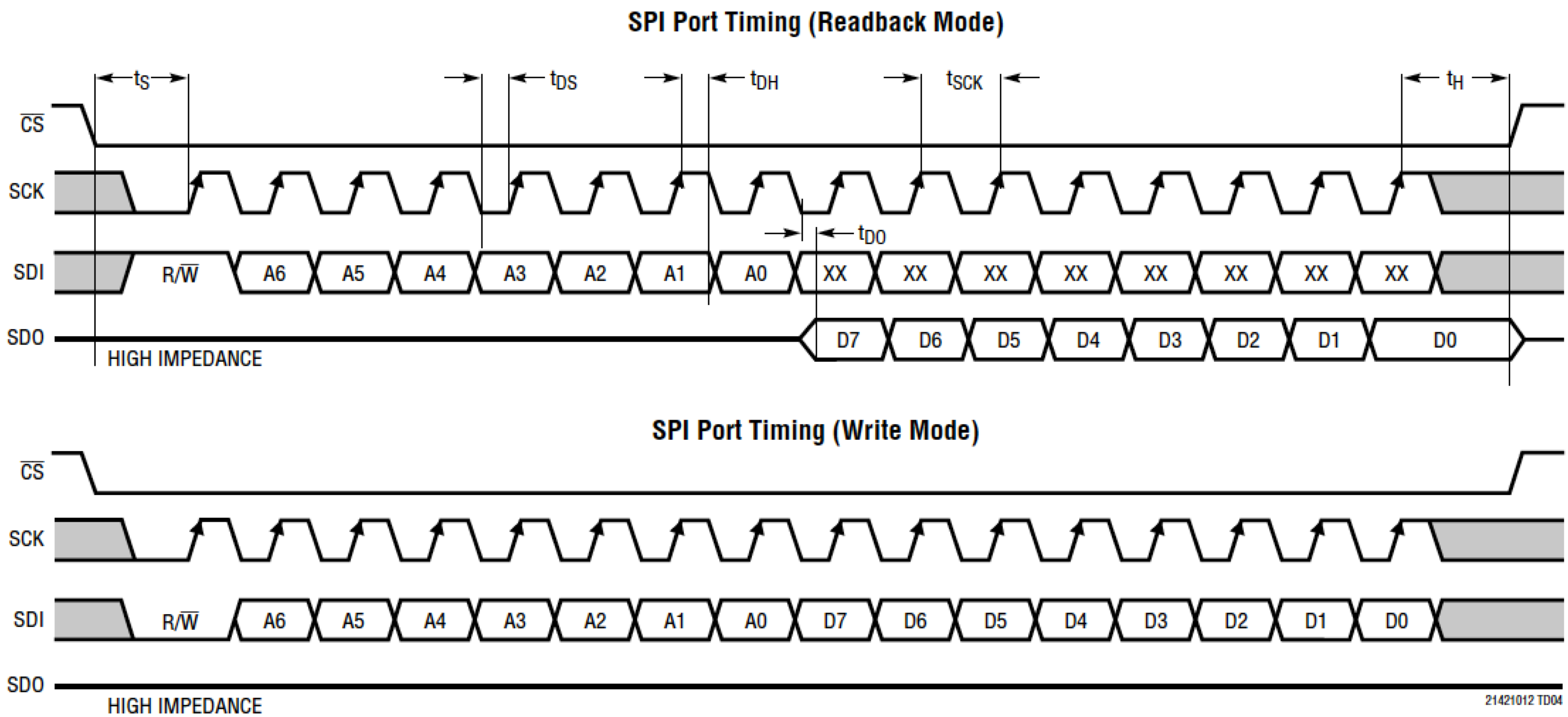
Also Possible to Daisy Chain



from sparkfun.com

Each device passes message on to next; common for LED strings

Example from LTC2141 (ADC) datasheet



Notes: MSB first; MOSI = SDI (slave data in); MISO = SDO (slave data out)
looks at SDI (MOSI) or SDO (MISO) on upward clock transition
R/$\overline{W}$ high means read; low (note bar) means write
first write address, then either read or write data
chip enable asserted low for whole exchange

Example Register on LTC2141

REGISTER A4: DATA FORMAT REGISTER (ADDRESS 04h)

D7	D6	D5	D4	D3	D2	D1	D0
X	X	OUTTEST2	OUTTEST1	OUTTEST0	ABP	RAND	TWOSCOMP

Bit 7-6 Unused, Don't Care Bits.

Bits 5-3 **OUTTEST2:OUTTEST0** Digital Output Test Pattern Bits
000 = Digital Output Test Patterns Off
001 = All Digital Outputs = 0
011 = All Digital Outputs = 1
101 = Checkerboard Output Pattern. OF, D11-D0 Alternate Between 1 0101 0101 0101 and 0 1010 1010 1010
111 = Alternating Output Pattern. OF, D11-D0 Alternate Between 0 0000 0000 0000 and 1 1111 1111 1111
Note: Other Bit Combinations Are Not Used

Bit 2 **ABP** Alternate Bit Polarity Mode Control Bit
0 = Alternate Bit Polarity Mode Off
1 = Alternate Bit Polarity Mode On. Forces the Output Format to Be Offset Binary

Bit 1 **RAND** Data Output Randomizer Mode Control Bit
0 = Data Output Randomizer Mode Off
1 = Data Output Randomizer Mode On

Bit 0 **TWOSCOMP** Two's Complement Mode Control Bit
0 = Offset Binary Data Format
1 = Two's Complement Data Format

To set register 4 to ABP and 2's comp., would write 0x04, 0x05 over SPI

A quick note on hexadecimal

decimal value	binary value	hex value
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	a
11	1011	b
12	1100	c
13	1101	d
14	1110	e
15	1111	f

Hexadecimal, continued

- Once it is easy for you to recognize four bits at a time, 8 bits is trivial:
  - 01100001 is 0x61
  - 10011111 is 0x9f
- Can be handy because the ASCII code is built around hex:
  - 'A' is 0x41, 'B' is 0x42, …, 'Z' is 0x5a
  - 'a' is 0x61, 'b' is 0x62, …, 'z' is 0x7a
  - '^A' (control-A) is 0x01, '^B' is 0x02, '^Z' is 0x1A
  - '0' is 0x30, '9' is 0x39

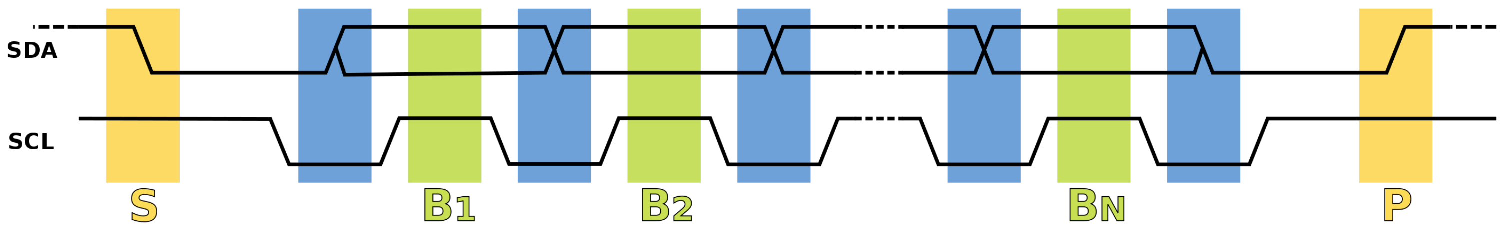
Core Python SPI Code

```
import spidev                          # module with SPI cmds

spi = spidev.SpiDev()                  # instantiate device
spi.open(0,0)                          # selects CE0
spi.max_speed_hz = 122000              # 122 kHz*

def readRegister(regAddr):
    address = 0x80 | regAddr           # sets read bit
    resp = spi.xfer2([address,0x00])   # xfer2 keeps CE low
    return resp[1]                     # result is in second byte
def writeRegister(regAddr,data):
    spi.xfer2([regAddr,data])          # simply write (write bit low)

writeRegister(0x04,0x05)               # sets register 4 to 0x05
result = readTegister(0x04)            # if want to confirm reg. 4 setting
```

* options for speed are: 7629, 15200, 30500, 61000, 122000, 244000, 488000, 976000, 1953000, 3900000, 7800000, 15600000, 31200000, 62500000, 125000000

I^{2}C: Inter-Integrated Circuit

- Pronounced I-squared-C or I-two-C
- Two signal lines (plus ground):
  - clock (SCL)
  - data (SDA; bi-directional)

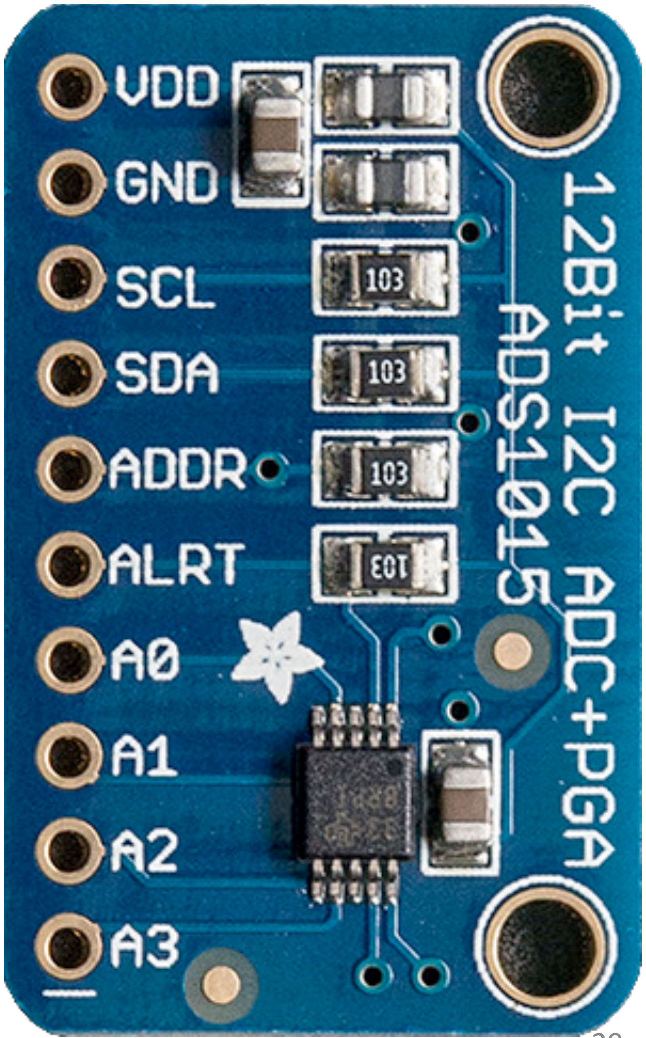


  - Starts when SDA pulled low while SCL still high
  - stoPs when SDA pulled high while SCL restored to high
  - data read/valid while SCL high (updated when SCL low)
  - data line can contain read/write and acknowledge bits

A Real Example for Lab 3: ADS1015

- Texas Instr. ADS1015
  - 12-bit ADC, 4 channels
  - V_{DD} 2.0 to 5.5 V
  - I^2C Interface
- Device address depends on what ADDR connects to:

ADDR Pin to:	Full Address (7 bit)
GND	1001000
VDD	1001001
SDA	1001010
SCL	1001011



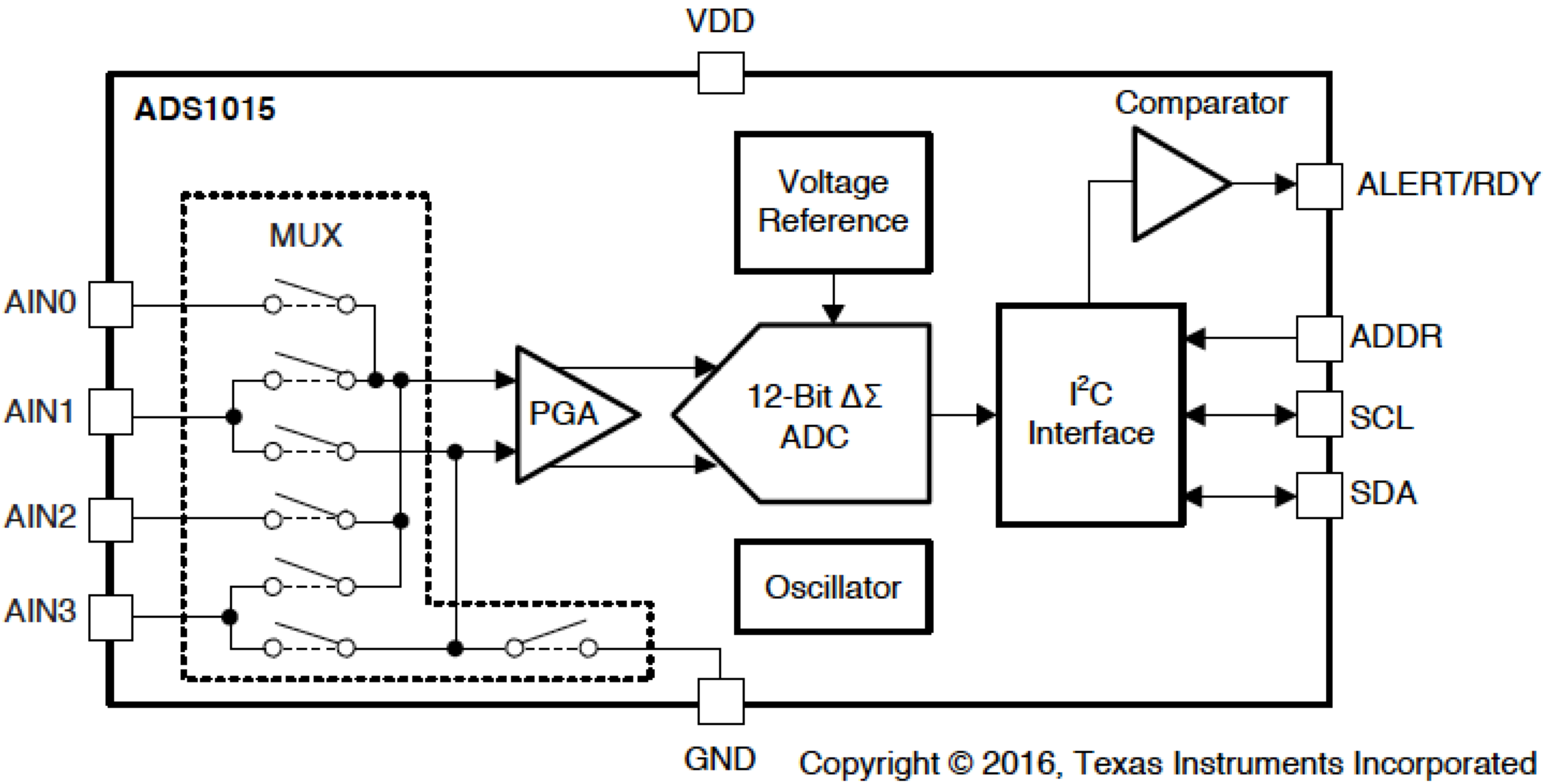
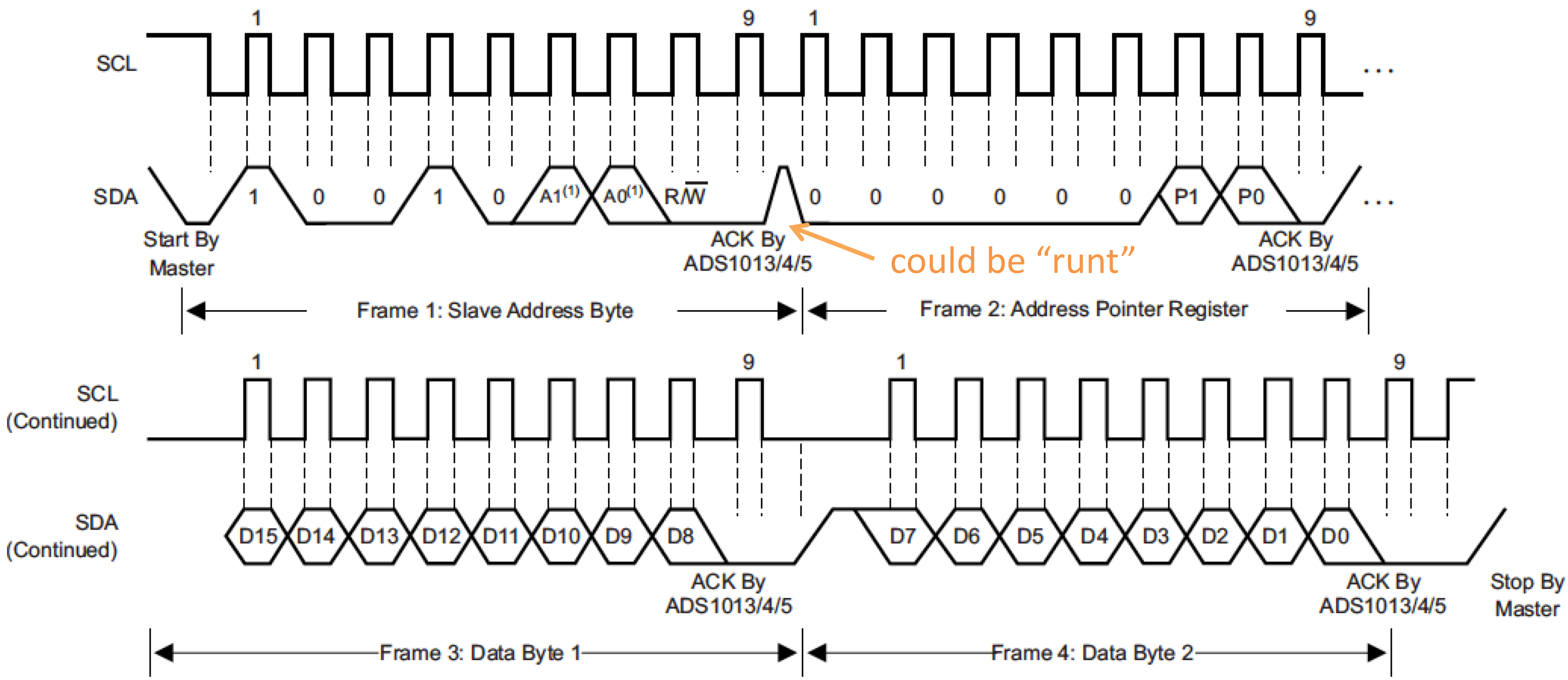


Figure 7. ADS1015 Block Diagram

- Can configure inputs various ways using MUX (close two switches)
- Variable gain (range) via PGA (programmable gain amplifier)
- I^2C for interface
- Optional comparator action to control ALERT pin



(1) The values of A0 and A1 are determined by the ADDR pin.

Figure 16. Timing Diagram for Writing to ADS101x

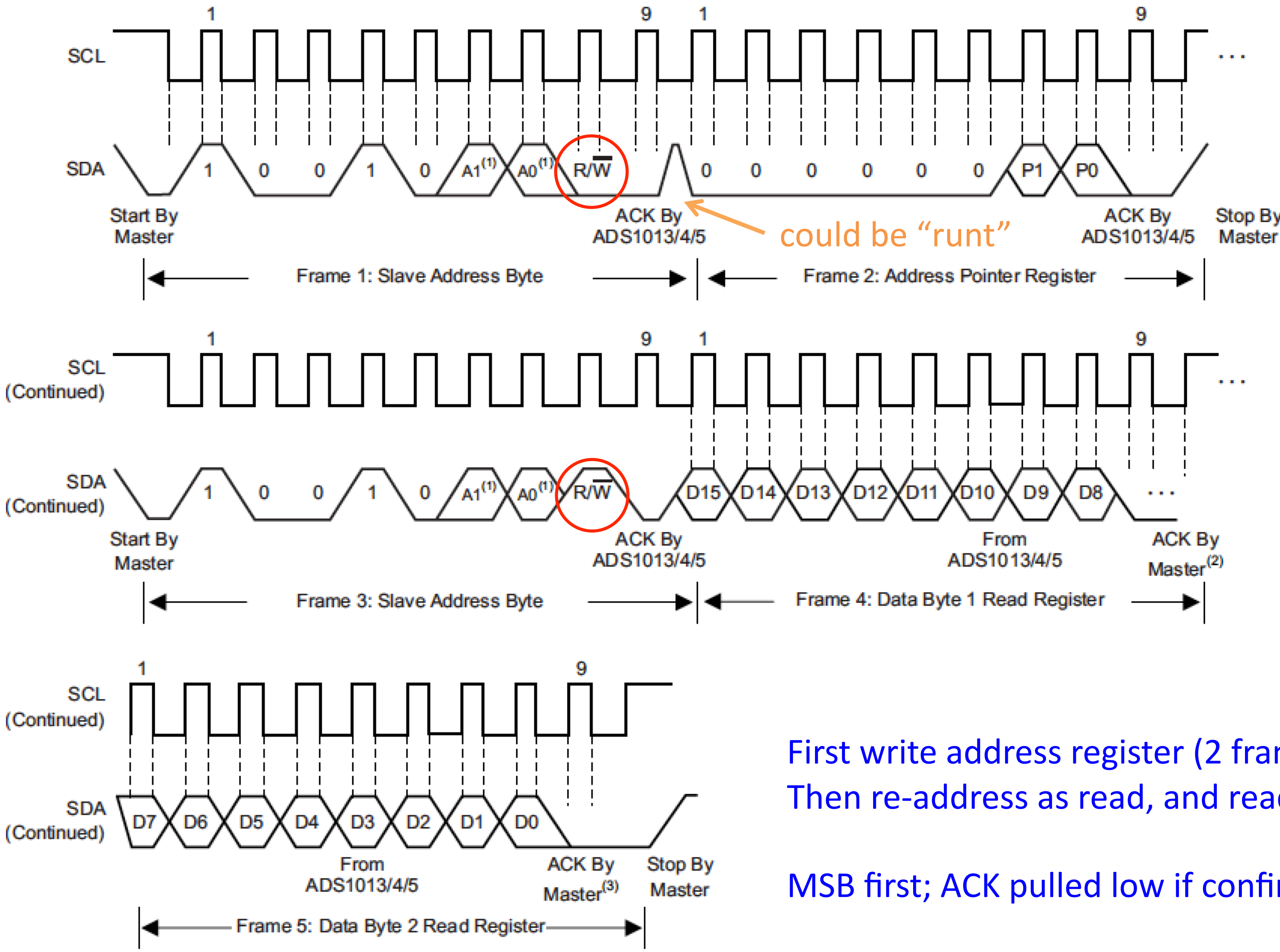
Four frames (bytes plus R/W and acknowledge):

target address; register to access; then two bytes of data

Notes: first frame instructs whether read or write (here write)

ACK pulled low means device confirms communication

MSB first, LSB last



(1) The values of A0 and A1 are determined by the ADDR pin.

(2) Master can leave SDA high to terminate a single-byte read operation.

(3) Master can leave SDA high to terminate a two-byte read operation.

Figure 15. Timing Diagram for Reading From ADS101x

First write address register (2 frames);
Then re-address as read, and read 2 bytes

MSB first; ACK pulled low if confirmed comm.

Register Mapping

Figure 19. Address Pointer Register

7	6	5	4	3	2	1	0
0	0	0	0	0	0	P[1:0]	
W-0h	W-0h	W-0h	W-0h	W-0h	W-0h	W-0h	

LEGEND: R/W = Read/Write; R = Read only; W = Write only; -n = value after reset

Table 4. Address Pointer Register Field Descriptions

Bit	Field	Type	Reset	Description
7:2	Reserved	W	0h	Always write 0h
1:0	P[1:0]	W	0h	**Register address pointer** 00 : Conversion register 01 : Config register 10 : Lo_thresh register 11 : Hi_thresh register

- We'll just care about first two registers (00 and 01)
- 12-bit conversion register (00) arranged in 2 bytes as:
  - D11 D10 D9 D8 D7 D6 D5 D4 and D3 D2 D1 D0 0 0 0 0
- Configuration register is pretty busy…

Configuration Register

Figure 21. Config Register

15	14	13	12	11	10	9	8
OS	MUX[2:0]			PGA[2:0]			MODE
R/W-1h	R/W-0h			R/W-2h			R/W-1h
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
DR[2:0]			COMP_MODE	COMP_POL	COMP_LAT	COMP_QUE[1:0]	
R/W-4h			R/W-0h	R/W-0h	R/W-0h	R/W-3h	

LEGEND: R/W = Read/Write; R = Read only; -n = value after reset

- ADS1015 datasheet takes 2 pages to detail options
  - controls Operating State (e.g., start conversion)
  - MUX: 4 single-ended or 2 differential measurements
  - sets voltage range for conversion (Prog. Gain Amplifier)
  - single shot or continuous MODE
  - Data Rate (if continuous sampling)
  - COMParator operation for controlling ALERT operation

Example Python

```
import smbus                  # module for i2c

i2cbus = smbus.SMBus(1)       # instantiate: can name whatever

ADDR = 0x48                   # default 1001000 if ADDR->GND

# write to config register (1) default values
i2cbus.write_i2c_block_data(ADDR,1,[0x85,0x83])

# read from conversion register (0) 2 bytes and combine
data = i2cbus.read_i2c_block_data(ADDR,0,2)
val_twos_comp = (data[0] << 4) + ((data[1] & 0xf0) >> 4)
```

Result will be single differential conversion of A0 minus A1 in ±2.048 V range

All the work is in figuring out how to manipulate the config register to get the results you want (in single mode, each conversion needs a configure command)

Refer to ADS1015 datasheet for full details on register configuration options

Result is in 2's complement

- Binary representation for signed integers
  - makes binary math easy/natural (single set of rules)
- Positive numbers look "normal"
  - 0000 0000 = 0; 0000 0001 = 1; 0100 1101 = 77
- Negative numbers have the MSB "lit", then other bits inverted, then add 1
  - Ex: −3; start with 0000 0011; MSB → 1 and invert others (1111 1100), then add 1: 1111 1101
  - now −3 added to +3 in binary will give 1 0000 0000 (zero if ignoring overflow bit)

Recovering 2's complement value

```
def twos(val,bits):                          # bits in represent.
  if (val & (1 << (bits - 1))) != 0:         # check if MSB=1
    val = val - (1 << bits)                  # subtract 2^bits
  return val
```

- Must specify number of bits in representation
  - in previous slide, used 8; for ADS1015, it's 12
- The `if` statement checks MSB
  - `<<` is left-shift by some # of places; `&` is bit-wise AND operation
    - Example: 0001 0110 << 2 becomes 0101 1000
    - Example: 0110 1101 & 1010 1010 becomes 0010 1000 (only 1 if both bits 1)
- When MSB is lit (not equal zero)
  - subtract off 1 0000 0000 (in 8-bit example)
- Our −3 example: 1111 1101 is literally 253 in unsigned binary
  - subtract 256 (1 0000 0000) and left with -3
- Perhaps you see the "complement" aspect
  - the "other" part of 2^N, once the negative part is removed

Application for Lab 3

- We'll read multiple temperature sensors
  - RTDs (resistive temperature devices)
  - signal conditioning (turn resistance into voltage)
  - analog-to-digital conversion (ADS1015)
  - interface to Raspberry Pi
  - programming Python to collect and store data

Temperature measurement

- A variety of ways to measure temperature
  - thermistor
  - RTD (Resistive Temperature Device)
  - AD-590 (current proportional to temperature)
  - thermocouple
- Both the thermistor and RTD are resistive devices
  - thermistor not calibrated, nonlinear, cheap, sensitive
  - platinum RTDs accurate, calibrated, expensive
- We'll use platinum RTDs for this purpose
  - small: very short time constant
  - accurate; no need to calibrate
  - can measure with simple ohm-meter
  - $R = 1000.0 + 3.85\times(T - 0^\circ\text{C})$
    - so 20°C would read 1077.0 Ω
    - "tempco" of 0.385% per °C (3.85 Ω/°C)

Problem: Measuring Resistance

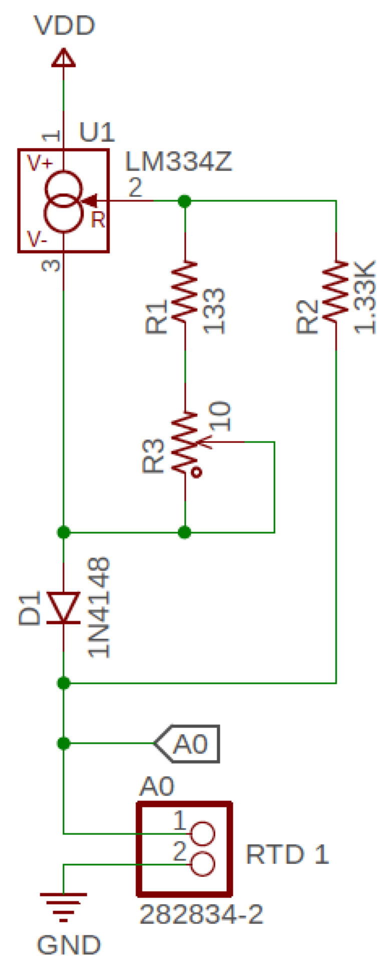
- The ADC (ADS1015) reads a *voltage*, not a resistance
- How can we measure a resistance using the ADC?
  - how do we do it right/well?
  - what issues might arise?

Current Source

- Provide stable 1.00 mA to RTD, so 1.00 kΩ → 1.00 V
  - a fine range for measuring using ADC
  - if 5 V range, get approx. 1 mV resolution at 12 bits
    - 1 mV is at 1 mA corresponds to 1 Ω change in RTD
    - translates to about 0.25 degrees, and not limiting factor
    - RTD calibration, and subtle gradients tend to be larger errors

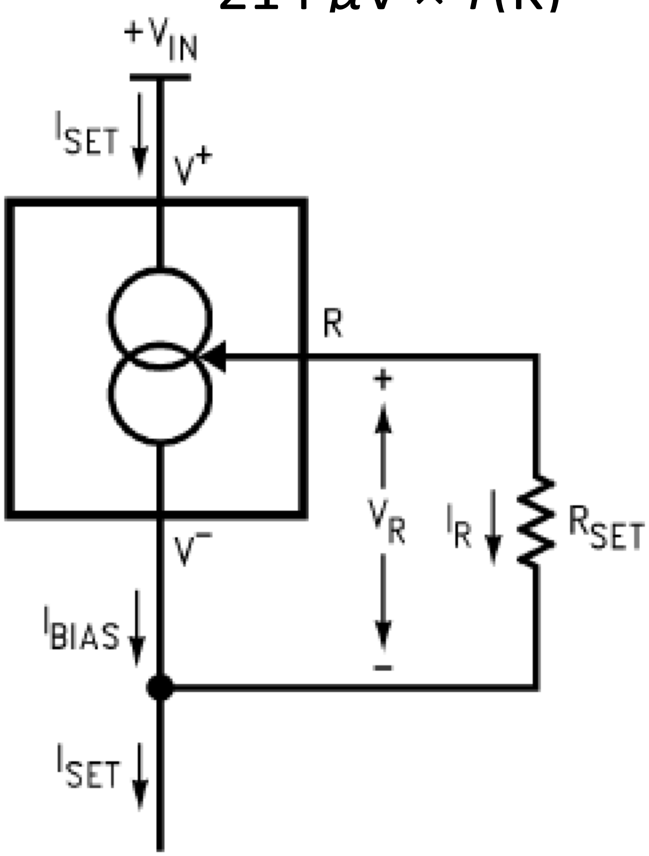
Implementation

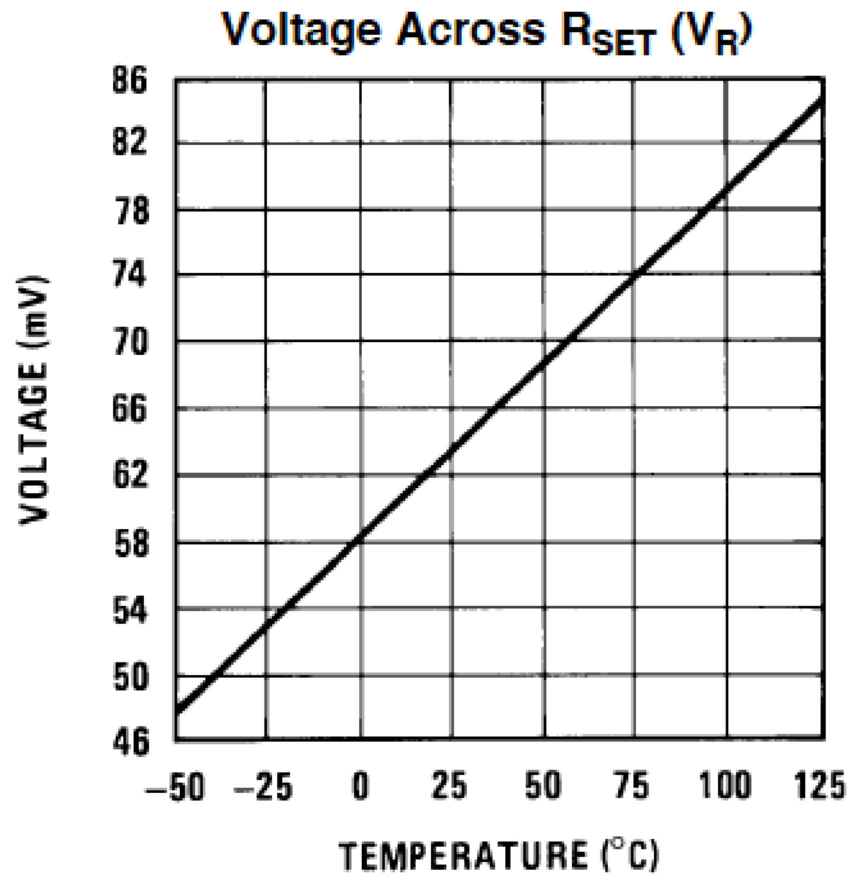
- LM334 current source
  - resistors configure current output
    - datasheet Figs 13 & 15
  - diode performs temperature compensation (hold close to LM334) so current steady as ambient temperature changes
  - RTD attached in series and voltage measurement at top end goes to ADC



Inner Workings of the LM334

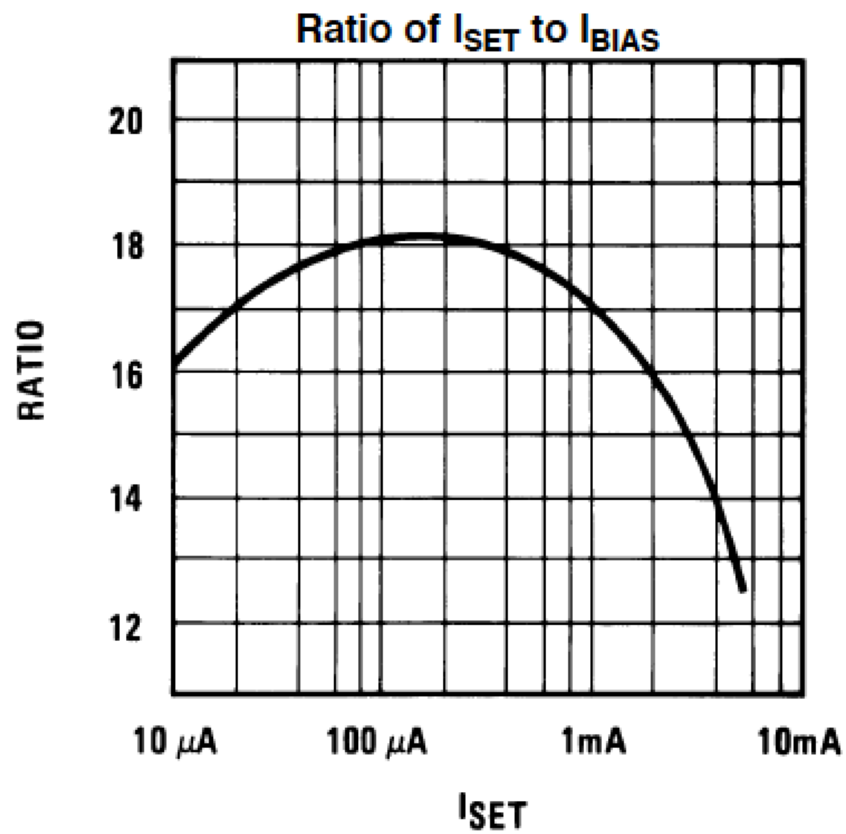
- V_R held to ~64 mV
  - across R_{SET} gives I_{SET}
  - strong linear temp. dep.
  - 214 μV × T(K)





Meanwhile I_{SET}/I_{BIAS} Ratio Well-Behaved

- At 1 mA, a ratio of ~17
- Result of math is that:
  - $I_{SET} = V_R/R_{SET} \times n/(n-1)$
  - n is ratio
  - V_R is 214 μV × T(K)
    - about 64 mV at room T
  - I_{SET} = 227 μV × T(K)/R_{SET}
  - so to get 1 mA at 300 K:
    - R_{SET} wants to be 68 Ω



Diode Compensation

- The "tempco" of the LM334 is 0.227 mV/C
  - 0.33% per degree; RTD is 0.385% per degree
  - same sign, so almost doubles d*V*/d*T* of ambient rise
- Typical diodes have a tempco about ten times higher, and opposite sign (−2.5 mV/C)
- The resistor ratio is roughly 10× to effect compensation
  - see data sheet for associated calculations
- Relies on similar temperature for both components
  - therefore good to put close together, touching, even encase

Lab 3 Flow

- Log on to Pi; reset group/bench password
- Mess around with Linux/Unix
- Mess around with Python
- Establish I^2C communication to ADS1015
  - including oscilloscope verification
- Build breadboard RTD current source
- Make program to collect RTD data
- Expand to multiple RTD channels
  - can breadboard or use pre-built modules

Announcements

- If no Unix/Linux familiarity
  - encouraged to look at Lab 3 before Wed.
  - find tutorials, and explore essential commands listed earlier
  - ideal if you can try on terminal
    - Mac Terminal; can use lab Pi as well
- If no Python familiarity
  - encouraged to look at Lab 3 before Wed.
  - find tutorials, and learn to write and execute simple programs
  - ideal if able to run Python interactive session and also try executing programs
    - Mac Terminal; can use lab Pi as well
- Lab 3 will be combined with Lab 4 for single write-up, due Oct. 30