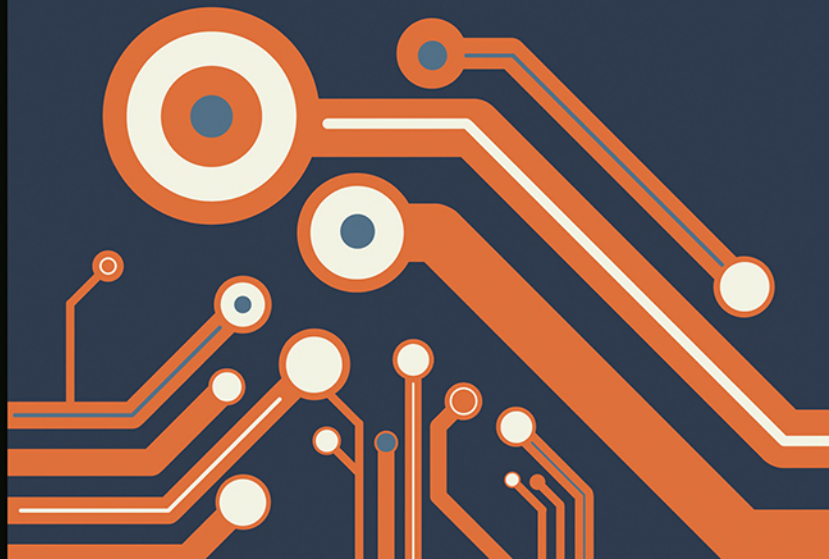


ELECTRONICS AND MICROPROCESSING FOR RESEARCH, 2ND EDITION YOU CAN MAKE IT

DAVID DUBINS



Electronics and
Microprocessing
for Research,
2nd Edition

Electronics and Microprocessing for Research, 2nd Edition:

You Can Make It

By

David Dubins

Cambridge
Scholars
Publishing



Electronics and Microprocessing for Research, 2nd Edition:
You Can Make It

By David Dubins

This book first published 2019

Cambridge Scholars Publishing

Lady Stephenson Library, Newcastle upon Tyne, NE6 2PA, UK

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

Copyright © 2019 by David Dubins

All rights for this book reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the copyright owner.

ISBN (10): 1-5275-3929-6

ISBN (13): 978-1-5275-3929-7

*To Robert B. Macgregor, Jr., Gregory Man Kai Poon, Rashid
Abu-Ghazalah, and to our late lab nights, past and future.*



Epigraph by Remy Dubins

TABLE OF CONTENTS

List of Figures.....	xvii
List of Tables.....	xxx
Acknowledgements	xxxiv
Preface.....	xxxv
Section 0.....	1
Introduction and Course Objectives	
Introduction.....	1
Why Microprocessing?	2
Course Objectives	3
Section 1	5
Introduction to Electricity	
What is Electricity?.....	5
Charge.....	6
Voltage.....	7
Power	10
The Generalized Power Law	11
Resistance	11
Ohm's Law	12
Resistors	13
Measuring Voltage, Resistance, and Current.....	14
Using a Multimeter to Analyze Your Complicated Circuit.....	14
Measuring Overall Circuit Power Consumption and Overall Circuit Resistance	15
Electrical Ground.....	17
DC Ground.....	17
AC Ground.....	18
Different Ground Symbols	19
Types of Returns	19
Voltage Sources: Series vs. Parallel.....	22
Batteries in Series.....	22

Batteries in Parallel	22
Circuit Configurations	23
Kirchhoff's Voltage Law (KVL)	24
The Voltage Divider Equation	26
Kirchhoff's Current Law (KCL).....	28
The Current Divider Equation.....	31
Calculating Current-Limiting Resistor Values for LEDs	33
Anode vs. Cathode: Devices with Polarity.....	34
Introduction to Switches	35
Breadboarding.....	37
Circuit Diagram Etiquette Example: Light Theremin	38
Activity 1-1: 9V Battery + LED + 10K Resistor	39
Activity 1-2: 9V Battery + 10K Resistor + 100K Resistor	41
Demo: Light Theremin	42
Learning Objectives for Section 1	42
Section 1 - Station Content List	43
Section 2	44
Capacitance, Power and Logic	
Capacitors	44
Capacitor Circuit Diagram Symbols	45
Capacitor Ratings	45
Capacitors in Series and Parallel	46
Capacitors: Typical Uses.....	46
Capacitor Equations	48
Charging a Capacitor through a Resistor.....	48
Discharging a Capacitor Through a Resistor.....	50
Voltage Divider Design: 10% Rule	50
Other Options for Delivering Lower Voltage.....	52
Datasheet Example: LM317 (Variable Linear Voltage Regulator)	53
How Hot Will My Chip Get? Heat Dissipation Calculations.....	56
Thévenin's Theorem	57
Thévenin's Theorem by Measurement (Using a Multimeter)	58
Mesh Current Method	61
Thévenin's Theorem Method (Theoretical)	64
Integrated Circuits (ICs)	67
PDIP/DIP.....	67
Surface Mount Technology	69
Logic Circuits	69
AND Gate: (e.g. 74HC08).....	69
OR Gate: (e.g. 74HC32).....	72

NOT Gate: (e.g. 74HC04)	73
Combining Logic Circuits	74
Activity 2-1: Capacitor Charging and Discharging	78
Activity 2-2: LM317 Voltage Regulator	79
Activity 2-3: Logic Gates	81
Learning Objectives for Section 2	82
Section 2 - Station Content List	83
Section 3	84
Introduction to Programming in the Arduino C++ Environment	
Introduction to the Arduino Uno Microcontroller Board	84
Connecting a Serial LCD Module to the Arduino Uno	85
Your First Sketch	88
Basic Programming Concepts	89
Commenting Your Code	89
Storing and Accessing Data in Variables	91
Declaring and Using Variables	93
Integers	93
Long Integers	95
Global Space, Setup Function, and Loop Function	96
Float Variables	97
If...Then...Else Statements (and Logical Expressions)	98
Bool Variables	101
Boolean Operators	102
Byte Variables	103
String and Char Variables	103
Casting Variable Types	105
Arrays of Variables	107
Char Array	108
Data Types: More Complicated Conversions	109
Defining Programming Loops in Arduino	111
For Loops	111
C++ Shorthand Increment Expressions	112
Do...While Loops	112
While Loops	114
For, Do...While, or While?	115
Ommitting Curly Brackets	115
The Break Command	115
Switch Case	116
General Programming Tips	118
Activity 3-1: Programming Challenge	119

Learning Objectives for Section 3	120
Section 3 - Station Content List	121
Section 4	122
Arduino Pins, and Writing Functions	
Byte Variables and Digital Pins	122
What is a Digital Pin?.....	124
Digital OUTPUT Mode Example.....	125
Pulse Width Modulation (PWM) Example	126
Digital Input Mode Example.....	128
Analog Pins.....	131
Using Analog Pins as Digital Output Pins	132
Analog Read Example.....	132
External Analog Reference: AREF Pin.....	134
Arduino Pin Conflicts	135
Arduino Digital and Analog Pins: Summary Tables.....	136
The Serial Monitor.....	137
The Serial Plotter	138
Subroutines and Functions	138
Properties of Functions.....	139
Void Functions	139
Call-by-Value vs. Call-by-Reference	141
Float Functions.....	142
Integer (and other) Functions	144
Function DOs and DON'Ts.....	144
#define and #ifdef Statements	145
General Programming Etiquette.....	147
Activity 4-1: NTC Thermistor Circuit	148
Calibrating a Thermistor	149
Two-Term Exponential Thermistor Equation	149
Learning Objectives for Section 4	153
Section 4 - Station Content List.....	154
Section 5	155
Switching Higher Power Devices: Relays, Transistors, TRIACs	
Voltage and Current Limitations of the Arduino Uno	155
Relays	157
High Side Switching vs. Low Side Switching.....	160
Powering a Relay with a Separate Supply.....	161
Vin Pin: Arduino Uno	161
Diodes (P-N Junction, or Rectifier Diodes).....	162

Transistors.....	165
Bipolar Junction Transistors (BJTs).....	165
NPN Transistors: Selecting a Base Resistor Value	167
NPN Transistors in the Active Region	170
Darlington Pairs	172
Current Gated vs. Voltage Gated	173
MOSFETs	174
TRIACs.....	177
BT139-600E (TRIAC)	180
Protecting your Circuit from DC Motors.....	180
Protection Diode.....	180
Reducing DC Motor Noise with Capacitors.....	181
Activity 5-1: Hot Plate Thermostat.....	182
Activity 5-2: Transistor as a Switch for a DC Motor.....	184
Parsing Serial Data.....	187
Activity 5-3: MOSFET as a Switch for a DC Motor	187
Learning Objectives for Section 5	188
Section 5 - Station Content List.....	190
Section 6.....	191
Process Control	
When “Close Enough” Isn’t Close Enough	191
How a DC Motor Works.....	192
Using an H-Bridge to Control Motor Speed and Direction.....	192
L298N H-Bridge Motor Driver Module.....	194
Stepper Motors.....	197
28BYJ-48 Stepper Motor with ULN2003 Motor Driver.....	198
Nema-17 Stepper Motor with A4988 Motor Driver.....	200
Servo Motors.....	205
System Control Strategies.....	206
Open-Loop Control.....	206
Feed Forward Control	208
Feedback Control.....	211
On-Off Controller.....	212
Proportional (P) Controller.....	213
Proportional-Integral (PI) Controller.....	217
Proportional-Integral-Derivative (PID) Controller.....	219
Combining Feedback Strategies.....	222
Activity 6-1: L298N Motor Driver Controlling a DC Motor.....	224
Activity 6-2(a): 28BYJ-48 Stepper Motor	225
Activity 6-2(b): Nema-17 Stepper Motor	226

Activity 6-3: SG90 Servo Control	228
Activity 6-4: PID Control of a 12V CPU Fan.....	230
Learning Objectives for Section 6	231
Section 6 - Station Content List, Activities 6-1 & 6-3	233
Section 6 - Station Content List, Activities 6-2(a,b), 6-4.....	234
 Section 7	 235
Operational Amplifiers	
Introduction.....	235
Open Loop Configuration (Comparator)	236
Closed Loop Configuration.....	237
Buffer	237
Op-Amp Characteristics.....	239
Output Short-Circuit Current.....	239
Gain in dB (decibels).....	239
Headroom.....	239
Slew Rate	242
Unity Gain Bandwidth	243
Inverting Amplifier	243
Biasing the Output of an Inverting Amplifier	246
Non-Inverting Amplifier.....	247
Biasing the Output of a Non-Inverting Amplifier	249
Differential Amplifier	250
Summing Amplifier (Inverting).....	251
Summing Amplifier (Non-Inverting).....	252
Summing Amplifier (Non-Inverting) Equations Solved	253
Negative Voltage?.....	260
Solution 1: Using a Virtual Ground.....	260
Solution 2: Negative Voltage Generator	261
Solution 3: Negative Supply Line from an ATX Power Supply ..	262
Op-Amps Can Do Calculus	262
Signal Attenuation: Reducing the Voltage.....	263
Activity 7-1: Load Cell Scale	263
Activity 7-2: pH Meter.....	267
Learning Objectives for Section 7	271
Section 7 - Station Content List, Activity 7-1	272
Section 7 - Station Content List, Activity 7-2.....	273
 Section 8.....	 274
Data Filtering, Smoothing, and Logging	
Data Filtering.....	274

Low-Pass Filters (LPFs).....	275
High-Pass Filters (HPFs).....	280
Inverting AC Amplifier.....	281
Blocking the DC in your Signal: Charge Coupling.....	283
Higher Order Filters.....	284
Band-Pass Filters.....	284
Second Order Low-Pass and High-Pass Filters.....	286
Operational Amplifiers: Practical Considerations.....	287
Impedance Considerations: Op-Amp Inputs.....	288
Impedance Considerations: Op-Amp Output.....	288
Measuring Output Impedance.....	290
Measuring Input Impedance.....	290
Practical Strategies to Reduce Signal Noise.....	291
Measuring Noise.....	294
Data Smoothing.....	295
Mean Filter.....	295
Median Filter.....	296
Mode Filter.....	297
Mean Filter with Threshold Rejection.....	298
Data Logging.....	300
Arduino TimeLib.h Library.....	301
Using millis() Instead of delay().....	302
Logging through the Serial Port.....	304
Logging to an External microSD Card.....	305
Logic Shifters.....	305
Activity 8-1: Noise Reduction.....	307
Activity 8-2: Data Smoothing.....	307
Activity 8-3: Data Logging to an SD Card.....	309
Learning Objectives for Section 8.....	311
Section 8 - Station Content List.....	312
Section 9.....	313
Design Project Guidance.....	
Introduction.....	313
Design Project Selection.....	313
Design Project Assessment.....	313
What if my design project doesn't work?.....	317
Code Snippets and Examples.....	318
Serial Monitor Menu.....	318
Using EEPROM: Memory that Doesn't Forget!.....	320
Generating Beeps to Alert your User: Arduino Tone Library.....	323

Programming One Button with Multiple Functions	324
Measuring Light Intensity	325
Photoresistors	325
Photodiodes	328
Phototransistors	329
Integrated Packages	330
Measuring Time Duration with Interrupts	336
Op-Amp Comparator with Bias Voltage: Turning an Analog Signal into a HIGH or LOW Digital Level	337
Matrix Keypads and LED Matrix Displays	338
Charlieplexing LEDs	341
Need More Digital Pins?	343
Shift-Out Registers	343
Shift-In Registers	345
Bareduino – Running the ATmega328 Alone	348
Learning Objectives for Section 9	351
Section 10	352
Advanced Topics in Programming	
Controlling MCU Registers, Interrupts and Timers	352
Bitwise Operations	352
Bitwise AND (&)	353
Bitwise OR ()	354
Bitwise NOT (~)	355
Bitwise XOR (^)	356
Shifting Bits with “<<” and “>>”	357
Bitwise Operators: Short Forms	358
Introduction to Port Manipulation	360
Worked Example: Fast Analog Read	361
Fast Digital Read and Write	365
Interrupts	368
Internal (Pin Change) Interrupts	370
Never Miss a Button Push Again	371
Rules for Writing an Interrupt Service Routine	372
Customized Frequencies for PWM	373
Timer 0	374
Timer 1	377
Timer 2	379
Timing your Interrupt Service Routines with CTC Mode	382
Sleep Mode	386
Wake on Pin Change	386

Wake on Timeout of Watchdog Timer	387
Resetting the MCU	389
Reset with a Watchdog Timer	389
Hard Wiring a Digital Pin to the RESET Pin	390
Advanced Formating and Variable Type Conversions	391
Secrets of Serial.print()	391
Additional String Conversion Commands.....	394
Comparing Strings.....	394
Arrays of Strings and Arrays of Char Arrays.....	395
Using Special Characters	396
Char Arrays: Advanced Functions	397
Structures	399
Unions.....	402
Increment Operators as Array Index Values.....	403
Appendix	405
Troubleshooting Guide	405
Troubleshooting Flowchart	409
Troubleshooting Zones.....	409
UTF-8 and ASCII Tables	410
Tips to Optimize Sketch Memory	414
Using a 555 Timer as an External Clock	420
Common Fixed Resistor and Capacitor Values	422
.ino Files	423
triacDimmer.ino (Section 5).....	423
Thermostat.ino (Section 5).....	423
4WStepper.ino: 4-Wire Stepper Control (Section 6).....	425
4WStepper_noLib: 4-Wire Stepper Control (no library required).....	425
PID.ino (Section 6).....	427
QuickStats.h (Section 8).....	429
TimedISR_N.ino (Section 10).....	432
Derivation for $V_{in(+)}$ (Section 7)	433
Arduino Uno Pin-out Diagram.....	436
ATmega328 Pin-out Diagram	437
ATtiny85 Pin-out Diagram	437
Ohm's Law Equation Table	437
List of Circuit Diagram Symbols.....	438
Variable Type Conversion Chart	439
List of Abbreviations	440

Bibliography 442

Index 451

LIST OF FIGURES

Figure 1-1. Mobile valence electron in the outer shell of a copper atom.....	5
Figure 1-2. Different ways of illustrating wire connections on a circuit diagram.	6
Figure 1-3. A battery provides a constant voltage source.....	7
Figure 1-4. Conventional current vs. actual flow of electrons.....	8
Figure 1-5. Cathode and anode reactions of an alkaline battery. (Besenhard 1999).....	8
Figure 1-6. Circuit symbols and voltage vs. time diagrams for Direct Current (DC) and Alternating Current (AC) voltage sources.	10
Figure 1-7. Using DC batteries in series.....	10
Figure 1-8. Resistance is the proportional current of electrons induced by a difference in voltage. For an ohmic device like a fixed-value resistor, this relationship is linear.	12
Figure 1-9. Circuit diagram symbols for fixed-value resistors.	13
Figure 1-10. Measuring voltage (left), resistance (middle), and current (right) using a digital multimeter.	14
Figure 1-11. Measuring the overall voltage, current, and power consumption of your circuit.	15
Figure 1-12. Circular and linear methods of drawing the same circuit.....	18
Figure 1-13. AC household power outlet (North America). AC cable wire colours are indicated for North America and the United Kingdom.	18
Figure 1-14. Common symbols for earth ground (left), chassis ground (middle), and floating ground (right).....	19
Figure 1-15. Floating return, drawn as it would be wired (left), using circular circuit diagram format (middle), and linear circuit diagram format (right).....	20
Figure 1-16. Chassis return. Note that optionally, the chassis can also be connected to earth ground, to reduce shock hazard in case of an accidental short circuit.	21
Figure 1-17. Earth return. Even though most battery-powered devices are floating, circuit diagrams tend to use the earth ground symbol for them regardless.....	21
Figure 1-18. Voltage is additive, and total current capacity remains constant when batteries are wired in series.	22

Figure 1-19. Current capacity is additive, and voltage remains constant when batteries are wired in parallel.	23
Figure 1-20. Three ways of connecting electronic components: basic, in series, and in parallel.	23
Figure 1-21. Kirchoff's Voltage Law applied to resistors in series.	24
Figure 1-22. Derivation of the voltage divider equation.	25
Figure 1-23. The voltage divider equation. The voltage across each resistor is proportional to the ratio of its contribution to the total resistance, R_1+R_2	26
Figure 1-24. Worked example for the voltage divider equation.	27
Figure 1-25. Kirchoff's Current Law example.	28
Figure 1-26. KCL applied to resistors in parallel. Note that the drawing on the left is equivalent to the drawing on the right electrically. Resistors in parallel can be represented either way.	28
Figure 1-27. KCL for two resistors in parallel.	30
Figure 1-28. The current divider equation, for two resistors in parallel.	31
Figure 1-29. Worked example for the current divider equation.	32
Figure 1-30. Calculating the resistance of a current-limiting resistor for an LED. The LED symbol is labelled D_1 (for diode 1) in the circuit diagram.	33
Figure 1-31. LED & resistor circuit, with anodes and cathodes labeled.	34
Figure 1-32. Circuit diagram symbol (left) and example (right) of a latching on/off switch.	35
Figure 1-33. Momentary switch connections.	36
Figure 1-34. Breadboard layout (top). Transparent arrows show how the power rails are connected by row, and middle pins are connected by column. The adhesive backing was peeled from the underside of a breadboard, revealing directionality of the internal rails (bottom).	37
Figure 1-35. Light theremin circuit diagram. If a legend is used, values next to the circuit diagram symbols may be omitted.	38
Figure 1-36. Schematic for Activity 1-1. Note: the <i>longer</i> wire on the LED is the positive side (anode).	39
Figure 1-37. Schematic for Activity 1-2.	41
Figure 1-38. Section 1 station setup.	43
Figure 2-1. Electron build-up and flow upon capacitor charging and discharging.	45
Figure 2-2. Circuit symbols for different types of capacitors.	45
Figure 2-3. Electrolytic (left) and ceramic (right) capacitors.	46
Figure 2-4. Calculating total capacitance of capacitors in parallel and in series.	46

Figure 2-5. A charge-discharge circuit. Holding down SW ₁ charges the capacitor. Once charged, holding down SW ₂ discharges the capacitor.	47
Figure 2-6. Capacitors can remove high frequencies from a signal (e.g. in an RC low-pass filter), low frequencies (e.g. in a CR high-pass filter), rectify an AC signal to DC, and reduce fluctuations in a noisy power supply.....	47
Figure 2-7. A circuit to illustrate charging and discharging a capacitor through a resistor.	48
Figure 2-8. It takes about three time constants ($3 \times \tau$) for a capacitor to charge through a resistor.	49
Figure 2-9. It takes about three time constants ($3 \times \tau$) for a capacitor to discharge through a resistor.	50
Figure 2-10. Solving for R ₂ using the voltage divider equation.	50
Figure 2-11. Calculating the value of the bleed resistor, using the 10% Rule.	51
Figure 2-12. Solved circuit using the 10% Rule.	52
Figure 2-13. Split supply.	53
Figure 2-14. LM317 pin-out diagram (left) and pin assignments (right)...	54
Figure 2-15. LM317 used as an adjustable regulator circuit with improved ripple rejection. (Texas Instruments Inc 2016a)	55
Figure 2-16. LM317 used as a current limiter. (Texas Instruments Inc 2016a).....	56
Figure 2-17. Circuit diagram symbol for a 10 mA current source.....	56
Figure 2-18. Heat sink for TO-220 package. The silicon layer and plastic nut keep the body of the package insulated from the heat sink, to reduce the chances of a short circuit.	56
Figure 2-19. Thévenin's Theorem in a nutshell. Any complicated network of resistors, capacitors, and sources from the perspective of a single component (at connections a and b) may be represented as a voltage source in series with a resistor.....	58
Figure 2-20. Example circuit for Thévenin's Theorem. First step: identify two terminals of interest, and label them a and b	59
Figure 2-21. Remove the load from the example, and measure V _{ab}	59
Figure 2-22. Measure i _{sc} across terminals a and b.	59
Figure 2-23. Thévenin Equivalent Circuit.	60
Figure 2-24. Thévenin Equivalent Circuit with load replaced.....	60
Figure 2-25. Short all voltage sources, remove all current sources, then measure R _{TH}	61
Figure 2-26. Norton Equivalent Circuit.....	61
Figure 2-27. Example for the Mesh Current Method.	62

Figure 2-28. Number each inside loop, draw current arrows, and label each junction.....	62
Figure 2-29. Perform KCL on junction A.....	63
Figure 2-30. Perform KCL on junction B.....	63
Figure 2-31. Identify points a and b around the load.....	65
Figure 2-32. Remove the load, then calculate the voltage difference between points a and b.....	65
Figure 2-33. Circuit diagrams can be deceptive. These four circuits all depict two resistors in parallel, from points <i>a</i> to <i>b</i>	66
Figure 2-34. Thévenin Equivalent circuit, with load replaced.....	67
Figure 2-35. DIP chips are great for prototyping with breadboards.....	67
Figure 2-36. Pin numbering for DIP chips runs counterclockwise, starting from the bottom left pin.....	68
Figure 2-37. Logic symbol for an AND gate.....	69
Figure 2-38. Pinout diagram for the 74HC08 AND chip.....	70
Figure 2-39. Circuit diagram to test out the functionality of an AND gate. Pull-down resistors protect against floating pin states when the DIP switches are open.....	70
Figure 2-40. Venn diagram for a 2-input AND gate.....	71
Figure 2-41. Logic symbol for a 2-input OR gate (top). Pinout diagram for the 74HC32 OR chip (bottom).....	72
Figure 2-42. Circuit diagram to test out the functionality of an OR gate.....	72
Figure 2-43. Venn diagram for a 2-input OR gate.....	73
Figure 2-44. Logic symbol for a NOT gate (top). Pinout diagram for the 74HC04 NOT chip (bottom).....	73
Figure 2-45. Schematic to test out the functionality of a NOT gate.....	73
Figure 2-46. Venn diagram for a NOT gate.....	74
Figure 2-47. An AND gate combined with a NOT gate is equivalent to a NAND gate.....	74
Figure 2-48. Venn diagram for a 2-input NAND gate.....	74
Figure 2-49. An OR gate combined with a NOT gate is equivalent to a NOR gate.....	75
Figure 2-50. Venn diagram for a 2-input NOR gate.....	75
Figure 2-51. XOR gate, meaning “exclusive OR”.....	75
Figure 2-52. Venn diagram for a 2-input XOR gate.....	76
Figure 2-53. XNOR gate, meaning “NOT exclusive OR”, or “exclusive NOR”.....	76
Figure 2-54. Venn diagram for a 2-input XNOR gate.....	76
Figure 2-55. Symbolic logical representation of Example 1(a).....	77
Figure 2-56. Venn diagram for Example 1(a).....	77
Figure 2-57. Symbolic logical representation of Example 2(a).....	77

Figure 2-58. Venn diagram for Example 2(a).....	77
Figure 2-59. Circuit diagram for Activity 2-1.	78
Figure 2-60. Schematic for Activity 2-2. Note: Pin 3 of the 1K trim is not connected.	80
Figure 2-61. Circuit diagram for Activity 2-3 (from Figures 2-38 and 2-39).....	81
Figure 2-62. Section 2 station setup.	83
Figure 3-1. Arduino Uno R3 board layout (DIP version).	85
Figure 3-2. Connecting the MCU to a serial LCD module.....	86
Figure 3-3. Arduino Uno (SOIC version) connected to serial LCD module and laptop.	86
Figure 3-4. Arduino IDE sketch window.....	87
Figure 3-5. Our first Arduino Uno sketch, with section and line comments.	91
Figure 4-1. Digital and analog pins of the Arduino Uno MCU.	123
Figure 4-2. Schematic to connect an LED to the Uno's digital pin 6, using a breadboard.	125
Figure 4-3. Circuit diagram for schematic in Figure 4-2.	126
Figure 4-4. Pulse width modulation (PWM), illustrated with different duty cycles.....	127
Figure 4-5. Gaining and shifting the sin() function to the working PWM range [0-255].	128
Figure 4-6. LED circuit (left) and momentary switch circuit (right).	130
Figure 4-7. LM35 temperature sensor. (Texas Instruments Inc. 2017) ...	132
Figure 4-8. A potentiometer can be set up as a variable resistor (rheostat), or a voltage divider. The wiper (middle pin) sweeps across a length of resistive material.	133
Figure 4-9. LED circuit (left) and potentiometer set up as a voltage divider (right).	133
Figure 4-10. The AREF pin lets you set the voltage of the highest div (1023). Range: 0-5V.	134
Figure 4-11. Thermistor + LED circuit. Resistor R_1 is the sense resistor.	151
Figure 4-12. Section 4 station setup.	154
Figure 5-1. A high voltage induces a magnetic field, pulling a switch closed.	156
Figure 5-2. Relay circuit diagram symbol (Single Pole, Double Throw - SPDT).....	157
Figure 5-3. Two common types of relays: DPST and DPDT.	158
Figure 5-4. Internal circuit diagram of a SPDT relay module. An opto-coupler isolates the Arduino Uno from the relay supply.	

Some relay modules allow for a separate ground for the supply powering the relay, although in the lab we will power the relay module using the Arduino +5V pin. (ELECTFREAKS wiki 2015).....	158
Figure 5-5. Using a SPDT relay as a normally open (NO) switch vs. a normally closed (NC) switch. The switch terminals have been sketched into the relay module boxes to help illustrate switching direction.....	159
Figure 5-6. The relay module on the left is wired as a high side switch (above the load). The relay module on the right is wired as a low-side switch (below the load).	160
Figure 5-7. The relay module on the left is powered using the Arduino Uno, which can introduce a lot of switching noise to your measurements. The relay module on the right is powered using a separate +5V DC adapter.	161
Figure 5-8. The Vin pin gives you access to the power supply voltage used to power the Uno, before the on-board 5V voltage regulator.	162
Figure 5-9. Diode symbols: LED (left), and general diode symbol (right).	162
Figure 5-10. Anatomy of a diode. Silicon has 4 outer valence electrons. When doped with boron (3 outer valence electrons), a space is formed capable of temporarily accepting an electron (left). When doped with phosphorus (5 outer valence electrons), there is an extra electron, capable of flowing (right).	163
Figure 5-11. When conventional current tries to flow from anode to cathode, a diode is forward biased, and electrons can jump across the P-N junction.	163
Figure 5-12. When conventional current flows from cathode to anode, a diode is said to be reverse biased, and electrons can no longer cross the P-N junction.	164
Figure 5-13. 1N4007 diode. A stripe indicates the brick wall (location of N-terminal).	164
Figure 5-14. Circuit to illustrate how a transistor works. Q ₁ is a 2N2222 NPN transistor in a TO-92 package. (ON Semiconductor Corp 2013).....	165
Figure 5-15. Worked example for the 10% Current Rule.....	168
Figure 5-16. An NPN transistor configured as a common emitter amplifier.	170
Figure 5-17. Calculating the voltage changes around the three terminals of a bipolar junction transistor used as a common emitter amplifier in Figure 5-16.....	171
Figure 5-18. A Darlington pair of NPN transistors.....	172
Figure 5-19. A regular LED as a light sensor.	173

Figure 5-20. When $V_{GS(th)}$ is applied to the gate terminal, an N-channel MOSFET allows conventional current to flow from drain to source.	175
Figure 5-21. Pin-out diagram for 2N7000 N-channel MOSFET	176
Figure 5-22. Circuit diagram symbol for a TRIAC	177
Figure 5-23. A TRIAC, controlled by microcontroller digital pin for a resistive load connected to 60 Hz, 120V AC power. This circuit can interrupt the hot wire in the middle of an extension cord, to create a high-side switch	177
Figure 5-24. TRIAC with zero-crossing detector, for an inductive or resistive load. This circuit is a low-side AC dimming switch	179
Figure 5-25. Forward phase dimming (left) and reverse phase dimming (right) with a TRIAC. The TRIAC is switched on and off strategically to chop an AC supply into narrower widths to dim a resistive load. (Coleman 2015)	179
Figure 5-26. Protection diode on a DC motor. In this configuration, the DC motor can only spin in one direction	181
Figure 5-27. Noise-reducing capacitor on “Bob”, the laboratory video rover. These capacitors fixed his nasty resetting problem	182
Figure 5-28. Circuit diagram for Activity 5-1: hot plate thermostat	183
Figure 5-29. Schematic for Activity 5-2: NPN transistor-controlled DC motor	185
Figure 5-30. Floppy drive connector of an ATX power supply, capable of supplying up to 3 amps. ATX power supplies are salvageable from old desktop computers	185
Figure 5-31. Circuit diagram for Activity 5-3: MOSFET-controlled DC motor	188
Figure 5-32. Section 5 station setup	190
Figure 6-1. The rotor coil of a motor (left) will spin according to Fleming’s left hand rule (right)	192
Figure 6-2. Simple H-Bridge configuration	193
Figure 6-3. Protection diodes for an H-bridge	194
Figure 6-4. L298N H-bridge module. This module comes with an on-board regulator that you can also use to power the logic side of your circuit. Check the top and underside of the module to confirm pin and screw terminal identities, as some modules may vary	195
Figure 6-5. Pin jumpers in the <i>open</i> position (top), and in the <i>closed</i> position (bottom), L298N module	196
Figure 6-6. A Nema-17 bipolar stepper motor with 5mm shaft coupler	197
Figure 6-7. Circuit diagram symbols for unipolar (left) and bipolar (right) stepper motors. A unipolar stepper motor can be controlled by 4	

switches, whereas a bipolar stepper motor requires two H-bridges to operate.....	198
Figure 6-8. A 28BYJ-48 stepper motor, with ULN2003 driver module..	198
Figure 6-9. Connecting a Nema-17 to a microprocessor with an A4988 stepper motor driver. Build this circuit with the power off.....	201
Figure 6-10. A4988 motor driver. White arrow points to trim potentiometer.	202
Figure 6-11. MG995 servo with shaft attachments.....	205
Figure 6-12. Functional block diagram of toaster control system.	207
Figure 6-13. Toaster setting dial – an example of open-loop feedback control.	208
Figure 6-14. Adding a sensor to a system on an INPUT stream and using that information to adjust DRIVE is a feed-forward strategy...	209
Figure 6-15. Example of a feed forward control system.	210
Figure 6-16. Example of a feedback control system.....	211
Figure 6-17. Example of a proportional feedback control system.	213
Figure 6-18. Undamped feedback response.....	214
Figure 6-19. Over-damped feedback response.	215
Figure 6-20. Under-damped feedback response, illustrating the concepts of rise time, overshoot, and settling time.	216
Figure 6-21. Finding a critically-damped feedback response.	217
Figure 6-22. Over-damped feedback response illustrating the effects of an integral gain. This response overshoots a little, because of integral wind-up.	218
Figure 6-23. Adjusting the derivative gain to attain the setpoint.	221
Figure 6-24. A simple PID control algorithm.	222
Figure 6-25. Circuit diagram for Activity 6-1.	224
Figure 6-26. Circuit diagram for Activity 6-2(a): 28BYJ-48 stepper motor with ULN2003.	225
Figure 6-27. Circuit diagram for Activity 6-2(b): Nema-17 stepper motor with A4988.	227
Figure 6-28. Circuit diagram for Activity 6-3.	229
Figure 6-29. Circuit diagram for Activity 6-4.	230
Figure 6-30. Functional block diagram for Activity 6-4.....	231
Figure 6-31. Station setup for Activities 6-1 and 6-3. Not shown: ATX power supply.....	233
Figure 6-32. Station setup for Activities 6-2 and 6-4. Not shown: ATX power supply.....	234
Figure 7-1. Circuit diagram symbol for an operational amplifier.	236
Figure 7-2. Open-loop configuration of an operational amplifier.	236
Figure 7-3. Buffer (or voltage follower) configuration of an op-amp.	237

Figure 7-4. The voltage output of an ideal buffer follows the voltage input ($V_{out}=V_{in}$).	238
Figure 7-5. Maximum peak output voltage vs. frequency for the TL07x op-amp series ($R_L=2k\Omega$, $T=25\text{ }^\circ\text{C}$). (Texas Instruments Inc 2017).	240
Figure 7-6. The op-amp on the right can swing higher because larger voltages are provided to the power rails.	240
Figure 7-7. Op-amp headroom means that the output of the op-amp can't swing all the way to the power rails. This is an illustration of the TL07x series, when the op-amp is supplied with $\pm 5V$.	241
Figure 7-8. The output of a typical op-amp will not be able to swing all the way to ground, if the negative rail is connected to ground.	241
Figure 7-9. A rail-to-rail op-amp is able to swing its output within microvolts of the power rails (virtually no headroom).	242
Figure 7-10. Inverting amplifier configuration of an op-amp.	243
Figure 7-11. An op-amp's V_{out} is constrained by the voltage connected to its power rails.	244
Figure 7-12. Input and output of an inverting amplifier (gain of -2).	245
Figure 7-13. Configuration (left) and performance (right) of an inverting amplifier (unity gain).	245
Figure 7-14. Biasing an inverting amplifier.	246
Figure 7-15. Using a voltage divider to generate a bias voltage for an inverting amplifier.	247
Figure 7-16. Configuration (left) and example performance (right) of a non-inverting amplifier.	248
Figure 7-17. Calculating V_{out} for a non-inverting amplifier.	248
Figure 7-18. Configuration (left) and example performance (right) of a non-inverting operational amplifier, with a bias voltage.	249
Figure 7-19. Differential amplifier configuration (left) and equations (right).	250
Figure 7-20. Calculating V_{out} example for a differential amplifier.	251
Figure 7-21. Inverting summing amplifier configuration (left) and equations (right).	251
Figure 7-22. Calculating V_{out} example for an inverting summing amplifier.	252
Figure 7-23. Non-inverting summing amplifier configuration (left) and equations (right).	252
Figure 7-24. Calculating V_{out} example for a non-inverting summing amplifier.	253
Figure 7-25. This non-inverting summing amplifier shifts and gains the signal, V_{in} .	253
Figure 7-26. Voltage at the non-inverting input, $V_{in(+)}$.	255

Figure 7-27. Shifted and gained pH electrode voltage signal. V_b shifts the signal positive, and after amplifying, the total shift is +1.65V.	256
Figure 7-28. Solved non-inverting summing amplifier for pH meter.	257
Figure 7-29. The bias voltage is generated using a voltage divider, and then buffered before adding it to the (buffered) probe voltage.	259
Figure 7-30. How do you supply negative volts?	260
Figure 7-31. Splitting a power supply (top) and using a voltage divider (bottom) for supplying a negative voltage to the bottom rail of an op-amp.	261
Figure 7-32. ICL7760 wired as a negative voltage generator (-5V out on pin 5).....	261
Figure 7-33. Differentiator (left) and integrator (right) op-amp configurations.	262
Figure 7-34. Attenuating a probe signal using a voltage divider, then buffering the output.....	263
Figure 7-35. Structure of a Wheatstone bridge.....	263
Figure 7-36. Circuit diagram for Activity 7-1 (load cell scale).	265
Figure 7-37. Experimental setup for Activity 7-1 (load cell scale).....	266
Figure 7-38. Circuit diagram for Activity 7-2 (pH meter).....	269
Figure 7-39. Station setup for Activity 7-1.....	272
Figure 7-40. Station setup for Activity 7-2.....	273
Figure 8-1. The effect of a carefully designed low-pass filter (LPF).....	275
Figure 8-2. Bode Magnitude Plot for a first-order LPF.....	277
Figure 8-3. Calculation of the gain of an LPF, one decade higher than the cutoff frequency.	278
Figure 8-4. Worked example for an LPF, $f_c=200$ Hz. Left: passive LPF, right: active LPF with unity gain.	279
Figure 8-5. The effect of a carefully designed high-pass filter (HPF).	280
Figure 8-6. Bode Magnitude Plot for a first-order HPF.....	281
Figure 8-7. Inverting AC amplifier. Capacitor C_1 filters out frequencies two decades below f_c . If $V^+=+5V$, the output signal will be gained by R_F/R_1 , and oscillate about +2.5V.....	282
Figure 8-8. Inverting amplifier example: electret microphone preamplifier.....	282
Figure 8-9. A coupling capacitor transmits AC and blocks DC.	283
Figure 8-10. An example band-pass filter (top) and corresponding Bode Magnitude Plot (bottom).....	285
Figure 8-11. Twin-T Notch Filter (left) and Bode Magnitude Plot (right). (Carter 2006, 19-26)	286
Figure 8-12. Passive second-order HPF.	286
Figure 8-13. Non-inverting, amplifying second-order LPF.....	287

Figure 8-14. Properties of an ideal op-amp.	287
Figure 8-15. Balancing the inputs of an inverting op-amp with a compensating resistor.....	288
Figure 8-16. Matching the output impedance of an op-amp with the input impedance of the next stage (in this case, the MCU's analog pin). ...	289
Figure 8-17. The 16 MHz crystal oscillator on the Arduino Uno, responsible for microprocessor speed.	300
Figure 8-18. RTC module for keeping track of epoch time.....	301
Figure 8-19. Circuit diagram (left) of a logic shifter (photo right), safely bridging a 3.3V microSD card module to the hotter 5V logic-level MCU.	306
Figure 8-20. Section 8 station setup.	312
Figure 9-1. Piezoelectric elements.....	323
Figure 9-2. Piezoelectric buzzer plugged directly into the Arduino Uno.	323
Figure 9-3. Photoresistors of various sizes (5 to 12 mm).	325
Figure 9-4. Voltage divider for a photoresistor.	326
Figure 9-5. A transimpedance amplifier (left) converts current, I_{in} to voltage, V_{out} with a gain of R_F . An example (right) shows the LM358 amplifying the current through an LDR, with a bias voltage V_b to raise the signal into the output voltage swing of the amp (~0.1 V – 3.9V for op-amp supplies $V^+=+5V$ and $V^-=GND$).....	327
Figure 9-6. The PD638C photodiode.....	328
Figure 9-7. A photodiode wired in the voltage divider configuration with sense resistor (left), and as an input for a transimpedance amplifier (right).....	328
Figure 9-8. The PT334-6C phototransistor (right), wrapped in electrical tape (left).	329
Figure 9-9. A phototransistor wired in the voltage divider configuration with sense resistor (left), and as an input for a transimpedance amplifier (right).....	329
Figure 9-10. TSL235R light-to-frequency converter.....	330
Figure 9-11. Toshiba TCD1304AP linear CCD.	332
Figure 9-12. BH1750 Light Sensor module.....	333
Figure 9-13. Low-cost laser light receiver.....	334
Figure 9-14. TCS3200 colour detection module.....	334
Figure 9-15. A comparator op-amp with a carefully-selected bias voltage changes a small signal response into a digital signal (HIGH or LOW).	338
Figure 9-16. Mapping the pins of a matrix keypad to rows and columns.	338

Figure 9-17. A custom LED matrix display.	340
Figure 9-18. A SN74HC595 shift-out register, with digital output pins Q0 – Q7.	343
Figure 9-19. A SN74HC165 shift-in register, wired to send the pin states from digital input pins Q0 – Q7. In this configuration, pin 15 has been wired to ground, so the chip is constantly enabled, requiring one less control wire from the MCU.	346
Figure 9-20. Configuring the ATmega328 chip outside the Arduino Uno. Pin numbers follow the physical layout of the DIP chip (Pin 1 is RESET).	349
Figure 9-21. USBtinyISP chip programmer, useful for re-burning the bootloader onto an MCU, uploading a sketch through the ICSP header on the Arduino Uno, or programming an MCU directly through the appropriate pins.	350
Figure 10-1. Using port registers to set and read digital pin 12 (PB4), in Bank B.	367
Figure 10-2. Momentary switch connected to INT0 (digital pin 2).	369
Figure A-1. Troubleshooting flowchart: developing resilience.	409
Figure A-2. Potential areas to consider troubleshooting. Many problems are multifaceted, involving more than one area.	409
Figure A-3. A 50% duty cycle square wave generator configuration for the 555 timer (LMC555 CMOS version required above ~400 kHz). Ranges for f_{out} were obtained experimentally using a LMC555 timer and monolithic capacitors.	421
Figure A-4. Voltage divider network at an op-amp input: solving for the input voltage.	433
Figure A-5. Pin-out diagram for a generic Arduino Uno r3. Pin numbers (2 to 28) correspond to the pin numbers on the ATmega328 chip. (Atmel Corporation 2016).	436
Figure A-6. Pin-out diagram for the ATmega328 28-pin (DIP). Pin labels for the Arduino IDE are indicated in blue (~ indicates PWM-capable). (Atmel Corporation 2016).	437
Figure A-7. Pin-out diagram for the ATtiny85 8-pin (DIP version). Pin labels for the Arduino IDE are indicated in blue (~ indicates PWM-capable). (Atmel Corporation 2013).	437
Figure A-8. List of circuit diagram symbols used in this text. Note that many variants of these symbols are common. LDRs, LEDs, photodiodes, transistors, and MOSFETs often appear in circuit diagrams without circles.	438

Figure A-9. Summary conversion chart for common variable types. Grey lines: *casting* (see Table 3-5). Black lines: see Table 3-6 for examples. See Table 10-23 for advanced formatting with Strings..... 439

LIST OF TABLES

Table 1-1. Typical voltages and current capacities for various battery types and chemistries. (Wenzel 2017)	16
Table 1-2. Current capacity for common wire gauges. (Scherz and Monk 2016).....	17
Table 1-3. Summary chart: resistors in series vs. resistors in parallel.	32
Table 1-4. Measured and calculated values for Activity 1-1.	40
Table 1-5. Measured and calculated values for Activity 1-2.	41
Table 2-1. Recommended operating conditions for the LM317 voltage regulator (TO-220 package). (Texas Instruments Inc 2016a)	54
Table 2-2. Some electrical characteristics of the NE555 chip. (Texas Instruments Inc 2014b)	68
Table 2-3. Logic tables for a 2-input AND gate. “0V” is the same as “ground”, and this table assumes that <i>logic level</i> for your circuit is +5V.	71
Table 2-4. Two-input AND logic table.....	71
Table 2-5. Two-input OR logic table.....	73
Table 2-6. NOT logic table.....	74
Table 2-7. Two-input NAND logic table.....	74
Table 2-8. Two-input NOR logic table.....	75
Table 2-9. Two-input XOR logic table.....	75
Table 2-10. Two-input XNOR logic table.....	76
Table 2-11. Logic table for Example 1(a).....	77
Table 2-12. Logic table for Example 2(a).....	77
Table 2-13. Experimental results from Activity 2-1.....	79
Table 3-1. Bit depth size chart.....	92
Table 3-2. Mathematical operators and functions in C++.....	94
Table 3-3. Relational operators in C++.....	100
Table 3-4. Table of logical (Boolean) operators.....	102
Table 3-5. How to cast between common variable types, with examples	106
Table 3-6. More complicated variable conversions involving String and char.....	110
Table 3-7. C++ shorthand.....	112
Table 3-8. Three major types of loops performing the same task.....	115
Table 4-1. Wiring a momentary switch to a digital input pin.	129

Table 4-2. Digital pin summary table.....	136
Table 4-3. Analog pin summary table.....	136
Table 4-4. Selectable baud rates (bps) in the Arduino IDE serial monitor.....	137
Table 4-5. Common types of functions.....	139
Table 4-6. Example sketches for call-by-value (left) vs. call-by-reference (right).....	142
Table 4-7. DOs and DON'Ts in writing functions.....	145
Table 4-8. Advantages and disadvantages of thermistors.....	148
Table 4-9. Table for Experimental Results in Activity 4-1.....	152
Table 5-1. NPN and PNP Bipolar Junction Transistors (BJTs).....	166
Table 5-2. h_{FE} values of P2N2222A (TO-92) (ON Semiconductor Corp 2013).....	169
Table 5-3. N-Channel MOSFET vs. P-Channel MOSFET.....	174
Table 6-1. An H-bridge can switch the polarity across a motor by changing the states of four switches.....	193
Table 6-2. L298N H-bridge control of two independent DC motors.....	195
Table 6-3. Suggestions for 4-Wire stepper motor colour codes.....	201
Table 6-4. V_{ref} settings (in mV) for A4988 motor drivers.....	202
Table 6-5. The process of toasting bread in a toaster, described without (left) and with (right) control system terminology.....	207
Table 7-1. Calculating the expected op-amp output across the expected input range.....	247
Table 7-2. Gaining and shifting a pH probe signal to a convenient range for the analogRead() function (longer method).....	255
Table 7-3. Gaining and shifting a pH probe signal to a convenient range for the analogRead() function (quicker method).....	258
Table 8-1. First-order passive and active low-pass filters. (Texas Instruments Inc 2013, 1-26).....	275
Table 8-2. First-order passive and active high-pass filters. (Texas Instruments Inc 2013, 1-26).....	280
Table 8-3. Measuring the output impedance of a signal. (Andy Collinson 2018).....	290
Table 8-4. Measuring the input impedance of the input pin of a device (e.g. a microprocessor). (Andy Collinson 2018).....	291
Table 8-5. Ten practical tips for noise reduction and prevention.....	291
Table 8-6. Calculating the median of a data set.....	297
Table 8-7. Calculating the mode of a data set.....	297
Table 8-8. Comparison of four data smoothing methods.....	299
Table 9-1. Some Design Project Ideas.....	313
Table 9-2. Charlieplexing LEDs.....	342

Table 9-3. Connecting the 6-pin ICSP connector from the USBtinyISP to the ATmega328, or the ATtiny85. The red cable shows which side Vcc is on.....	349
Table 10-1. Example bitwise AND comparison: finding out a pin state.	354
Table 10-2. Example bitwise OR comparison: setting a specific pin HIGH.	355
Table 10-3. Example bitwise AND comparison: setting a pin LOW.	355
Table 10-4. XOR logic table.....	356
Table 10-5. Example bitwise XOR comparison: toggling a pin state HIGH.....	356
Table 10-6. Example bitwise XOR comparison: toggling a bit state LOW.....	357
Table 10-7. Boolean operators vs. bitwise operators in C++.....	358
Table 10-8. Short forms for common bitwise operations, with examples.	359
Table 10-9. Structure of the ADCSRA, the ADC control and status register (Atmel Corporation 2016).....	362
Table 10-10. Setting the ADC Prescaler value using the ADPS0, ADPS1, and ADPS2 bits (Atmel Corporation 2016).....	362
Table 10-11. ATmega328 pin banks. (Atmel Corporation 2016).....	365
Table 10-12. DDR, PORT, and PIN registers for the three pin banks of the ATmega328. Bit DDD1=1 if the serial monitor is needed, otherwise it can be cleared. (Atmel Corporation 2016).....	366
Table 10-13. Trigger options for interrupt service routines.....	372
Table 10-14. Fast mode PWM frequencies in Hz for Pin 5 (Timer 0).....	375
Table 10-15. Fast mode PWM frequencies in Hz for Pin 6 (Timer 0).....	376
Table 10-16. Fast mode PWM frequencies in Hz for Pin 10 (Timer 1)...	378
Table 10-17. Fast mode PWM frequencies in Hz for Pin 9 (Timer 1).....	378
Table 10-18. Fast mode PWM frequencies in Hz for Pin 3 (Timer 2).....	380
Table 10-19. Fast mode PWM frequencies in Hz for Pin 11 (Timer 2)...	381
Table 10-20. Some C++ escape sequences you can use in strings and char variables.	391
Table 10-21. Custom output formats for Serial.print() and Serial.println().	392
Table 10-22. Additional commands that can help you manipulate Strings.	392
Table 10-23. Functions to convert other variable types to Strings.	394
Table 10-24. Commands that test the contents of a char variable.	399
Table A-1. Troubleshooting questions and suggested follow-up actions.	406

Table A-2. UTF-8 character table, with decimal, hexadecimal, and binary codes. The ASCII column is the corresponding ASCII character resulting from the same code. 410

Table A-2. UTF-8 and ASCII character table (continued). 411

Table A-2. UTF-8 and ASCII character table (continued) – extended character set. 412

Table A-2. UTF-8 and ASCII character table (continued) – extended character set. 413

Table A-3. ATmega328 variable sizes, and the ranges of values they can store. 415

Table A-4. Some microprocessors and their memory limits. (Arduino.cc 2018; Stör et al. 2017; Paul Stoffregen 2019)..... 419

Table A-5. Common fixed resistor values. 422

Table A-6. Common fixed capacitor values. 422

Table A-7. Pin-out of 20-pin and 24-pin versions of the ATX main power connector. (Fisher 2019) 435

Table A-8. Ohm’s Law equations, re-arranged for each term. 437

ACKNOWLEDGEMENTS

Many people assisted in the inception and development of this text.

Dr. Robert B. Macgregor, Jr. inspired this work by encouraging me to turn a fixed USP dissolution apparatus into an undergraduate course, and he helped me offer it within a short time span of six months.

Andrew Cooper patiently taught me much of the tacit and practical knowledge in this text, for example, *Practical Strategies to Reduce Signal Noise* (Table 8-5) and providing a virtual ground for a probe (Figure 7-31).

Dr. Will Cluett kindly reviewed and provided comments for the process control content in Section 6.

Many students taking my electronics courses have helped to refine and improve this work immensely: Abigail D'Souza, Celeste Vicente, Brittany Epp-Ducharme, Nabeel Tariq, Jack Bufton, and Hamed Tinafar. Your enthusiasm and keen eyes for detail are warmly appreciated.

I would like to thank Adam Rummens at Cambridge Scholars Publishing, for his knowledge, help and expertise with the first edition.

Lastly, I would like to thank the institution in which I work, the Leslie Dan Faculty of Pharmacy at the University of Toronto for providing a positive and supportive educational environment and infrastructure, and plenty of batteries.

PREFACE

It's not *supposed* to work.

One of the things I hope you will discover throughout this course is that we take our technology for granted. This attitude is deeply embedded in our consumer culture. As end users, we are intentionally blinded to failures during product development. We demand quality, taking good design for granted. We purchase the latest gizmos at our favourite technology stores, and after a year or two of using them, either they break and we buy new ones, or we toss them aside because we are bored and done with them, wanting newer gizmos. We are *accustomed* to the idea that a device is supposed to work. Why shouldn't it? After all, we paid good money for it. It comes in a box, with a guarantee. If it doesn't work, we get a refund after a flustered sales person squints apologetically at the original store receipt.

These gizmos *work* because of careful design, quality parts, automated assembly processes, meticulous QC checks, and market research—something that engineers, scientists, manufacturers, and business people spend their entire careers working on. As a novice then, you will very likely become frustrated early on that the circuit you are building does not light up an LED, measure the temperature, or send text to a screen. Perhaps little components will produce strong-smelling blue smoke. Although a lay person might call this “broken”, “screwed up”, or “horrible” and feel discouraged about their abilities, this frustration is more appropriately called the *design and testing* phase—where things necessarily won't work. Why should they? The circuit doesn't even exist yet, let alone function flawlessly enough to sell in a store.

As a designer then, it is your job to put on an optimistic hat, and sharpen your troubleshooting skills. I can promise you that during the activities in this course, you will hit a dead end where you feel that you have tried everything you can, yet your circuit still doesn't work. You might feel frustrated, discouraged, or defeated. I can promise you this because it happens to me all the time.

However, I can also promise that if you hang in there, when you finally do figure out a critical piece of the puzzle that makes your project spring to life, all that frustration will dissipate (after the urge to kick yourself passes). It will be replaced by a feeling of relief, satisfaction, pride,

happiness, and perhaps the thrill that drew me to this area of study in the first place. Building electronic circuits is addictively fun!

SECTION 0

INTRODUCTION AND COURSE OBJECTIVES

Introduction

We are taught at a very early age to start counting at 1. However, with computer programming in microprocessing, we need to get into the habit of starting to count from zero. It is only fitting then to begin the introduction of this course with Section 0 accordingly—starting *ex nihilo*, from nothing.

As scientists, we tend to use very sophisticated equipment, containing circuits that we barely understand and take for granted. Understanding and designing your own electronic circuits can allow you to control equipment on the microsecond scale. With sensors and switches, you can use electronics to build your own scientific equipment, and to record measurements. This course will introduce you to programming, electronics, data acquisition, system control strategies, and data analysis techniques.

The goal of this course is to introduce you to theoretical and applied concepts in electronic circuitry, for the purpose of collecting and analyzing experimental data. As the curriculum was developed at the Leslie Dan Faculty of Pharmacy, University of Toronto, many of the examples are rooted in pharmaceuticals, the science involved in drug formulation design. The concepts discussed are nonetheless applicable to many quantitative research contexts where measurement and data collection are important. The course discusses introductory circuit design, with an emphasis on how common components work (e.g. resistors, capacitors, diodes, transistors, motors, operational amplifiers, and a variety of sensors) in scientific and manufacturing instrumentation.

Practical and mathematical aspects of circuit design are discussed (e.g. Ohm's Law, voltage dividers, analog vs. digital signals). There is a heavy emphasis on programming in C++ in the Arduino IDE platform, taught at an introductory level, which complements learning activities. Mathematic models are kept intentionally simple, using practical techniques where applicable (e.g. Thévenin's Theorem). Programs are intentionally short.

Why Microprocessing?

With the recent advent of low-cost, consumer-level microprocessors (e.g. the ATtiny and ATmega family, ESP8266, and new microprocessors continually being developed), affordable and accessible microprocessing has empowered researchers with resources to take experimental designs to new heights. Such microprocessors are relatively simple compared to the complexities of today's computers; but have sufficient speed and power to control sophisticated equipment such as scientific instrumentation and 3D printers. Previously, ADCs (Analog-to-Digital Converters) were thousands of dollars, requiring high programming aptitude to bridge the gap between instrument and computer. Serial communication ports were reliable only at slower speeds (e.g. 1200 bps). Serial communication was finicky, requiring access to equipment subroutines not always readily available. However, the climate has now changed for experimental design. Hobbyist platforms such as Arduino, and Raspberry Pi support a growing community. Libraries are readily available facilitating microprocessor control in many languages, such as C++, Python, and MatLab®. Interfaces are more intuitive. Components are inexpensive, and readily available. A large open source community has evolved to support scientists and hobbyists alike. It has *never* been easier to build your own equipment. Knowledge of programming and circuitry will provide a solid foundation not only in experimental design and analysis for research, but in many other areas as well.

The development and commercialization of prototype boards such as Arduino, Raspberry Pi, ESP8266, and Teensy have empowered the electronics community with quicker and easier circuit prototyping. Due to economies of scale, electronic components have become very inexpensive over websites such as eBay, Alibaba, Newegg, and Amazon, and through electronics manufacturers and retailers such as Texas Instruments Inc. and Digi-Key. The Internet of Things has driven the development of modular electronic components marketed for general purposes, compatible with open-source platforms at +3.3V and +5V logic levels (e.g. opto-isolator power relay circuits, H-bridge motor controllers, and frequency-matched RF transmitter/receiver units). Circuits that previously needed to be built from scratch are readily available and packaged as low-cost, ready-to-use modules. This course will examine some of these modules and their usefulness in circuit design.

This text has been improved considerably since the first edition. A new design project section is provided with example sketches for LED and button matrix devices, light sensors, and other useful methods that may find their way into student projects. The appendix grew so large with the

inclusion of topics such as registers, interrupts, and timers, that Section 10 was created to pull together advanced programming concepts. Some embarrassing errors and typos in the first version have been identified and corrected (for instance, a decade above a 440 Hz tone is not an A note, it's much closer to a C#). The fast-paced nature of electronics means that even as you hold this text in your hands brand new, parts of it will already be outdated. However, the fundamentals in this text are not subject to fashion. They won't go out of style. Your investment in learning electronics components, programming, and especially troubleshooting will appreciate over time regardless of the platform or project.

Course Objectives

The course uses mixed teaching methods, with the first half of each lecture as small-group didactic teaching, and the second half as laboratory exercises to experiment with and illustrate the concepts covered. As such, Sections 1 to 8 were designed as three-hour combined lecture/lab periods. These sections loosely follow different “themes” in electronics. Section 9 provides guidance and tips for individual student design projects. Section 10 covers some advanced control over the ATmega328 microprocessor, providing sketches to control MCU registers, interrupts, and timers. It also includes more advanced concepts in programming, such as how to declare a structure, and more complicated variable type conversions. Learning objectives are defined at the end of every section to help guide your studies. Over-arching these learning objectives are the following course objectives.

After taking this course, you will ...

- 1) Be able to properly interpret other people's circuit diagrams and draw your own diagrams in a manner that will enable you to accurately record and share your work with others. You will be able to draw circuit diagrams with enough detail so that others can build the same project. You will develop a foundation for “circuit diagram literacy”. Many diagrams in this manual were intentionally hand-drawn to reinforce that drawing a circuit diagram shouldn't require special software.
- 2) Have a working knowledge of some basic building blocks in electronics: resistors, capacitors, relays, transistors, MOSFETs, motors, op-amps, voltage regulators, etc. This course will not expose you to all the fundamentals (e.g. inductors, Zener diodes, transformers, and diode bridges have been intentionally omitted). However, the course provides enough background to start you off designing your own circuits, hopefully pointing you in a helpful

direction. Whereas a cook follows recipes, a chef creates them. Understanding how electronic components fit together will enable you to go beyond following other people's circuit diagrams, synthesizing your own modules and ideas.

- 3) Become proficient in C++ programming, which is a wonderful segue into learning other programming languages. Once you learn how to code in one language, learning another is often a matter of translation, which is much faster to pick up (e.g. Python, Matlab®, and R-project). You will learn the basics of writing a computer program. You will have many opportunities to write and compile your own code. You will learn proper programming etiquette, including commenting, selecting and working with appropriate variable types, writing subroutines, functions, and declaring local and global variables. After you are finished this course, you will be able to write, compile, and upload code for the Arduino Uno's microprocessor, the ATmega328, using the open-source Arduino IDE. These skills are transferrable to programming other microprocessors.
- 4) Be able to design, create, assemble, and test your own projects and scientific equipment, from raw sensor output to data filtering and logging.
- 5) Be able to develop and refine your troubleshooting skills, and become more confident in your circuit building and testing abilities. You will be empowered to diagnose and solve your own problems. You will recognize the value and complexity in commercial electronics around you.
- 6) By learning the fundamentals, you will no longer see an electronic device as a single black box that works or doesn't work, but rather as a sum of parts that can be repaired, and failing that, at least salvaged for working parts.
- 7) Be able to design and build circuits safely and carefully, always being mindful of the dangers of high voltage.
- 8) Have fun with electronics and programming!

The activities in this course have been carefully designed and prepared assuming that you will work independently. There will be many opportunities to help troubleshoot each other's work; however, you are expected to perform these exercises on your own to help develop your skills at circuit designing and programming.

SECTION 1

INTRODUCTION TO ELECTRICITY

What You'll Be Learning	Lecture: Introduction to electricity and circuit diagrams. Ohm's Law, General Power Law, Power (AC vs. DC). Voltage, current, resistance, and how to measure them. Electrical ground. Kirchhoff's Laws. Voltage and current dividers. Anode vs. cathode. Switches. Introduction to breadboarding.
What You'll Be Doing	Activity 1-1: 9V LED - resistor circuit. Measuring voltage, current, resistance. Calculating power. Adding a momentary switch. Activity 1-2: Set up a simple voltage divider circuit. Confirm the voltage divider equation by measuring the voltage difference across each resistor. Demo: Light Theremin
Files you will need	Not applicable.

What is Electricity?

Electricity, in the sense that we will be thinking about it, can be described as the movement of electrons. Metals that have a portable, or *free* electron, can conduct electricity. Copper is one such metal, and consequently is commonly used to make wire. It has one free electron per atom. A *cloud* of these free electrons holds the metal together, giving rise to metallic bonds. In any conductive metal, the free electrons move in Brownian-like random pathways. When there is a charge gradient across the metal, electrons generally move from an area of negative (–) charge to an area of positive (+) charge, or thinking about it differently, from a higher to lower concentration of negative charge.

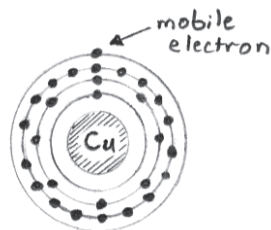


Figure 1-1. Mobile valence electron in the outer shell of a copper atom.

If electrons flow into one end of a copper wire, the outer-most electron of the copper atom leaves the orbit and flows to adjacent copper atoms, causing a chain of electrons hopping from atom to atom. Although the actual flow of electrons is slower, the signal created by electrons moving across a gradient is close to the speed of light.

We use a straight line (with corners, as necessary) to depict a wire. Every point on the wire should ideally have the same electric potential. So, if you used a voltmeter to measure the change in voltage across any continuous wire, it should read zero, or close to it (within measurement noise).

When two bare wires touch each other metal-to-metal, electrons can flow across that junction as if they were a single, continuous wire. There are different ways of illustrating wires crossing on a circuit diagram: (Gibilisco 2013)

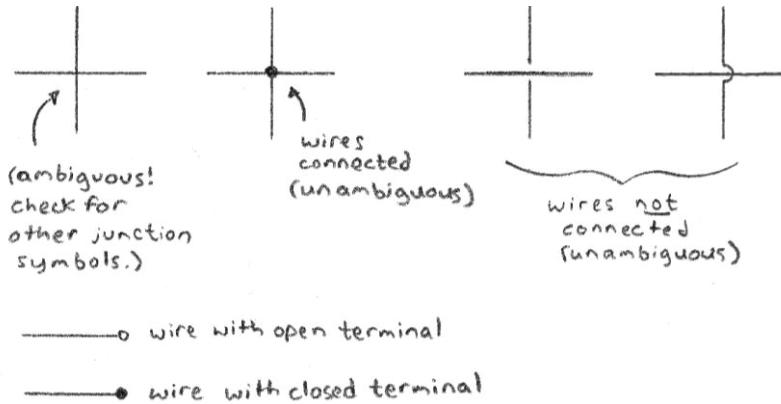


Figure 1-2. Different ways of illustrating wire connections on a circuit diagram.

A circuit diagram shows how electronic components are connected. A wire is our first circuit diagram symbol. For a list of all circuit diagram symbols used in this text, see Figure A-8 in the appendix.

Charge

Electrons carry a tiny amount of charge, which we measure in Coulombs (from Charles Augustin de Coulomb). We use the letter “Q” to represent charge. (Scherz and Monk 2016)

- 1 electron = $1\bar{e} = -1.602 \times 10^{-19} \text{ C}$ (Coulombs) = Q_{electron}
- 1 C = the charge from $6.242 \times 10^{18} \bar{e}$
- A lightning bolt: $\sim 15 \text{ C}$
- AA battery: $5 \text{ kC} = \sim 5000 \text{ C}$

At first glance, it looks like an AA battery has a lot more energy in it than a lightning bolt. However, charge and energy are different ideas, as you will see when we talk about power. An AA battery contains more *charge* than a lightning bolt.

Voltage

To get electrons to flow in a *current*, there needs to be a difference in *voltage* between two points. You can have a difference in voltage without current flowing, but you can't have current flowing without a difference in voltage. This is almost like saying you need water in order to have a waterfall.

If we want our circuit to do any work, we need a *voltage source*. A *battery* is a type of voltage source that maintains a constant voltage across its terminals. It is a DC (direct current) voltage source. Figure 1-3 illustrates the circuit diagram symbol for a battery.

In order to get current to start flowing, we need to make a closed loop (or *circuit*), by connecting a device like a light bulb between the positive and negative terminals of the battery. Below is a very simple circuit: a battery connected to an incandescent light bulb. Which way does the current flow? We always think of the current as flowing from the positive terminal of the battery (*cathode*) to the negative terminal (*anode*). This is called the direction of *conventional current*, based on Benjamin Franklin's understanding of electricity. However, sometime later, Joseph Thomson, discoverer of the electron, realized that current actually flows the other way (from the negative side to the positive side of the battery). Unfortunately, it was too late to fix this misunderstanding, and all the equations developed that far worked just as well backwards. So, we go with Benjamin, pretending that the current of electrons flows from positive to negative in an electronic circuit. This ends up being a source of confusion for new electronics students, and a permanent item at the top of this field's wish list for do-overs. (Scherz and Monk 2016)

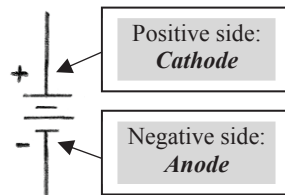
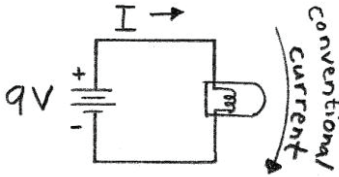


Figure 1-3. A battery provides a constant voltage

Benjamin Franklin: “father of electricity”
(positive charge carriers)



Joseph Thomson: “father of the electron”
(negative charge carriers)

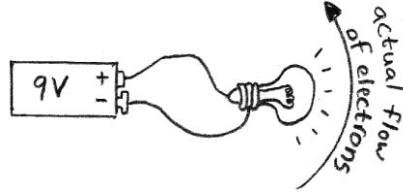


Figure 1-4. Conventional current vs. actual flow of electrons.

In a single-cell alkaline battery, the positive terminal (cathode) is manganese (IV) oxide, and the negative terminal (anode) is zinc. Both metals are in powder form to maximize reaction surface area, suspended in a matrix of potassium hydroxide—which acts as a conductive electrolyte. The chemical reaction that occurs when you connect an alkaline battery to a *load* is:

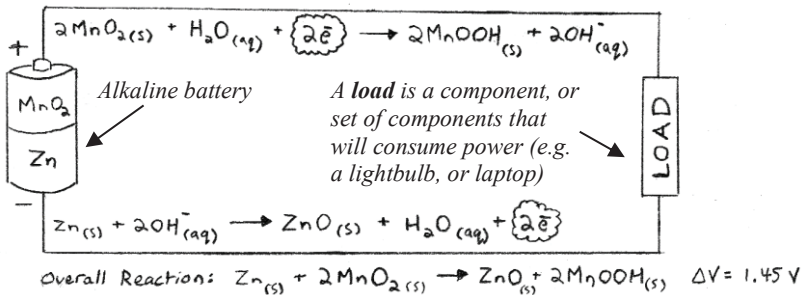


Figure 1-5. Cathode and anode reactions of an alkaline battery. (Besenhard 1999)

This chemical reaction is not reversible. Both reactant metals are consumed. When alkaline batteries leak, we typically think that fluid is battery acid. However, it’s not—this is the potassium hydroxide electrolyte leaking from the battery, causing a white residue, and then corrosion, on the battery terminals and inside your equipment. This misunderstanding likely comes from car batteries, which commonly use sulfuric acid as the electrolyte.

When the circuit in Figure 1-5 is connected, the speed of electrons (called *drift velocity*) is quite slow. At the current capacity limits of the average wire gauges presented in Table 1-2, this works out to about 0.208

mm/s for copper wire. (Scherz and Monk 2016) However, the signal (electrons moving across a gradient vs. electrons moving randomly) is propagated very close to the speed of light. Think of a big conga line stopping all at once when the music stops. This is the signal we will make use of most in this course (“ON vs. OFF”).

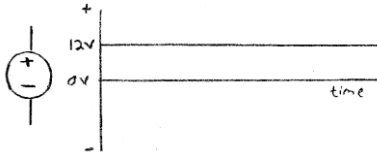
James Joule was responsible for characterizing the unit for voltage, which can be thought of as the amount of energy per unit charge. We measure voltage in Volts (in honour of Italian physicist Alessandro Volta), and use the letter “V”. (Maloberti and Davies 2016) Voltage is a relative measure, which makes it somewhat unique and interesting. Voltage is expressed as a difference, measured between two points. In this manual, when we solve for or measure “V” (in volts), the measurement implies a difference in voltage (ΔV), usually the difference from where the circuit is at its lowest relative voltage. The voltage of a single point with no reference has no meaning. This might not make any sense at face value, but think about it this way: it’s a bit like gravity. If I were to ask you, “which way is up?” you might point towards the sky. However, that “up” is relative to the ground. Which way is up when you are floating in space? Gravity provides a great analogy, because things fall towards the ground, and that’s the direction we picture conventional current flowing. In a DC circuit, we measure the voltage difference of a given point in the circuit relative to the lowest voltage point in the circuit (the negative terminal), and we somewhat arbitrarily call the lowest voltage point **ground**. A voltage measurement is the amount of energy a coulomb’s worth of electrons will change as they travel between those two points of interest.

For our specific purposes, voltage is the difference in energy per unit charge of the electrons in two different points of a circuit:

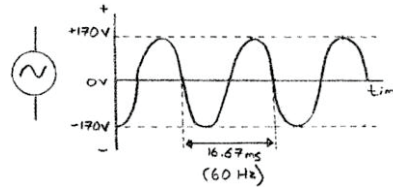
$$\mathbf{V = Voltage (Volts) = Joules / Coulomb}$$

$$1 \text{ V} = 1 \text{ J/C}$$

Voltage sources create a difference in voltage between their terminals. There are two main types of voltage sources: DC (Direct Current) and AC (Alternating Current).

Direct Current (DC)

- “Safer”, less noisy
- Great for logic (*on* or *off* is easy to measure)
- High loss of power over long distances (every wire has some resistance)
- Not so great for supplying power (eg: $12\text{V} \times 1\text{A} = 36\text{W} \rightarrow$ not a whole lot of power, for a lot of current)

Alternating Current (AC)

- In Canada, 120V (RMS), 60 Hz
- In Europe, 220V (RMS), 50 Hz
- Less energy loss over great distances
- Great at supplying power (eg: $120\text{V} \times 1\text{A} = 120\text{W} \rightarrow$ lots of power)
- $V_{\text{Peak}} = V_{\text{RMS}} \times \sqrt{2}$
- Voltage can vary by region and time of day

Figure 1-6. Circuit symbols and voltage vs. time diagrams for Direct Current (DC) and Alternating Current (AC) voltage sources.

DC voltage sources are additive in series:

- You usually call **ground** the lowest voltage point in the circuit ($V=0$).
- Some circuits require “negative” voltage. You can define your ground in the middle of these two batteries, and call it a **common return** or **virtual ground**. This strategy is called a **split supply**.

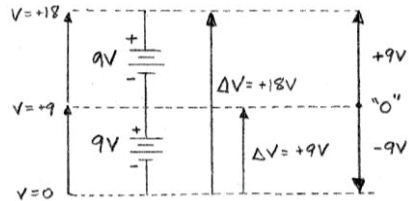


Figure 1-7. Using DC batteries in series.

Power

Power can be thought of in terms of how much of it your circuit requires to run properly, or conversely how much of it your voltage source can supply. Power is the rate of energy transferred. We measure power in Watts, from James Watt, and use the symbol “P”:

$$\mathbf{P = Power (Watts) = Joules / Time}$$

$$1 \text{ W} = 1 \text{ J/s}$$

If you remember the units for Volts (J/C) and Amps (C/s) then you will never forget the following very easy (and very handy) relationship.

The Generalized Power Law

$$\begin{array}{rcl} \text{Power} & = & \text{Voltage} \times \text{Current} \\ P & = & V \times I \\ \text{Watts} & & \text{Volts} \quad \text{Amps} \\ (J/s) & & (J/C) \quad (C/s) \end{array}$$

Example 1: A 10W halogen lightbulb will be connected directly to a 12V DC voltage. What will the current be?

Answer: We can re-arrange the Power Law:

$$P = V \times I \rightarrow I = P/V = 10W/12V = 0.833A$$

Example 2: A laboratory hot plate, powered by 120V AC power, is labelled “750W”. What is the current draw of the hot plate?

Answer: AC voltage is usually specified as RMS (root mean square) of the voltage vs. time curve over one period, out of convenience for a quick power calculation. You can think of the RMS voltage (V_{RMS}) as how much a resistive load “feels” in an alternating current. The AC Power Law for a purely resistive load powered by AC uses the RMS voltage and current (Scherz and Monk 2016):

$$P = V_{RMS} \times I_{RMS} \rightarrow I_{RMS} = P/V_{RMS} = 750W/120V = 6.25 A$$

Note that for a sinusoidal curve shape, the actual peak voltage for 120V AC will be:

$$V_{Peak} = V_{RMS} \times \sqrt{2} = 120V \times 1.414 = 170V$$

and that the AC voltage will swing from -170 to +170V (see Figure 1-6).

Resistance

The path of electrons through a circuit can be a bumpy one. Electrons collide with other electrons, impurities in the metal, and lattice ions—these interactions can impede the flow of electrons. With many materials, there is a predictable relationship of how much current flows through a material when a voltage difference is applied across it, and we call this *resistance*. We measure resistance in Ohms (Ω), from Georg Simon Ohm, and use the symbol “R”:

$$\text{Resistance } (\Omega) = \text{Joule}\cdot\text{second/Coulomb}^2$$

If the current through a device varies linearly with the voltage difference applied (which isn't always the case), we call it an *ohmic device*, because it follows Ohm's Law.

Ohm's Law

Resistance = Voltage / Current

$$R = \frac{V}{I}$$

Resistance is a property of the material or device, which is why the equation above has resistance on the left side. In ohmic devices, resistance is independent of voltage, as long as you respect the ratings of the device. However, Ohm's Law is more frequently written in terms of voltage:

Voltage = Current \times Resistance

$$V = I \times R$$

Volts (V) Amps (A) Ohms (Ω)

We know that electrons can flow when there is a voltage difference, and the bigger the voltage difference, the larger the current. We can think of resistance as a bottleneck in the stream. A material or device with a *low* resistance will result in a much larger current than a material or device with a *high* resistance. Figure 1-8 illustrates the voltage vs. current plots for various fixed-value resistors. Resistance is the slope of this graph.

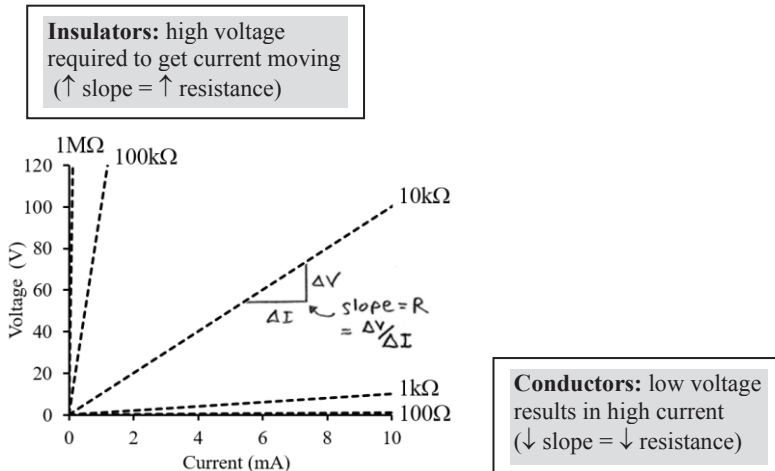


Figure 1-8. Resistance is the proportional current of electrons induced by a difference in voltage. For an ohmic device like a fixed-value resistor, this relationship is linear.

The linear relationships in Figure 1-8 only apply to ohmic materials (e.g. a diode does not follow Ohm's law). However, many devices (e.g. fixed-value resistors) do follow Ohm's law, as long as you use them within their maximum ratings. Most devices have documented limits on voltage, current, and power.

Resistors

Resistors are semiconductors. The flow of electrons is somehow impeded as intentional impurities are introduced in conductive materials like metal. For instance, carbon composition resistors (the type we will be using in class) involve mixing different proportions of finely powdered carbon and non-conductive ceramic. A lower carbon ratio means more resistance to electron flow (and consequently a higher resistance value). A **fixed-value resistor** (Figure 1-9) has a constant resistance value. A **potentiometer** allows you to change its resistance value by rotating a knob or pushing a slider. To a certain extent, *every* component in your circuit will act at least somewhat like a resistor. Even copper wire has a non-zero resistance, which can be important if you plan on running a large length of it.

Figure 1-9 illustrates two ways of drawing fixed-value resistors on circuit diagrams. Decimal points are difficult to see on circuit diagrams, particularly when there is a lot going on. To clarify a resistance value, a one-letter unit replaces the decimal point, based on the resistance value unit. For instance, a resistance of $2.4\ \Omega$ could be written as "2R4" on a circuit diagram, $1.3\ \text{k}\Omega$ as "1K3", and $3.3\ \text{M}\Omega$ as "3M3".

You can count on a resistor to only allow specific, and predictable current through it depending on the voltage across the resistor. If you hook a resistor up to a given difference in voltage, you can readily predict the current using Ohm's law. Some devices (e.g. LEDs and laser diodes) don't behave the same way. They can burn out quite quickly when a voltage is applied across them. You need to limit the current going through them, using a carefully selected resistor. The resistor will then act as a current limiting device, letting through an amount of current that is safe for the non-ohmic component. For example, a 1K resistor could be used to limit current

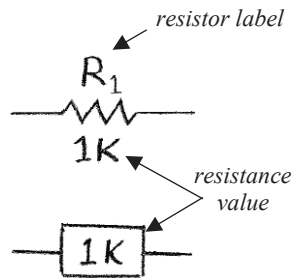


Figure 1-9. Circuit diagram symbols for fixed-value resistors.

through a typical LED with a voltage source of 5V, although larger resistance values would work as well. See *Calculating Current-Limiting Resistor Values for LEDs* for a more rigorous method.

Measuring Voltage, Resistance, and Current

Measuring voltage, current, and resistance is relatively simple, using a **digital multimeter**. We will be using multimeters in class. If a multimeter is set to measure voltage, we call it a **voltmeter**. If it is set to measure resistance, we call it an **ohmmeter**. If it is set to measure current, we call it an **ammeter**. The following symbols are used to represent these devices in circuit diagrams:

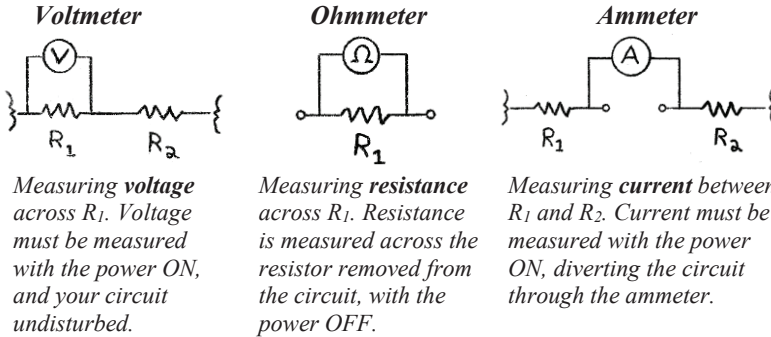


Figure 1-10. Measuring voltage (left), resistance (middle), and current (right) using a digital multimeter.

Using a Multimeter to Analyze Your Complicated Circuit

We can make use of these relationships and a multimeter to measure (and calculate) some very useful properties of a complicated DC circuit. We can measure the total voltage of the battery, for a few reasons. Firstly, even though a 9V battery has “9V” printed on it, it will likely not give a 9V reading. Why?

- As the battery ages, it has a natural drain rate. The voltage will slowly drop over time.
- A fresh, unused 9V battery should read approximately 9V. As the battery gets used, the voltage across its terminals will drop. When the voltage falls below 8V, it likely won’t provide enough current

for your circuit, and should probably be replaced, although your circuit might work a while longer.

Measuring Overall Circuit Power Consumption and Overall Circuit Resistance

Example: Let's say you measure the voltage across the 9V battery in Figure 1-11 with a voltmeter, and find that the voltage is 8.86 V. Then with an ammeter, you measure a current of 0.412 A. What is the total power the circuit is drawing from the battery?

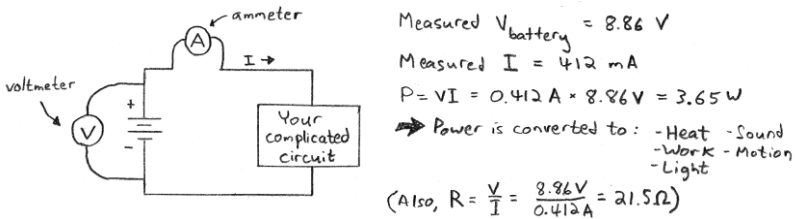


Figure 1-11. Measuring the overall voltage, current, and power consumption of your circuit.

To solve this problem, we can use the Generalized Power Law ($P=VI$).

If we substitute Ohm's Law into the Generalized Power Law, we also have two new ways to calculate power:

$$V = IR \rightarrow P = (IR)I \rightarrow P = I^2 R$$

$$I = \frac{V}{R} \rightarrow P = V \left(\frac{V}{R} \right) \rightarrow P = \frac{V^2}{R}$$

There are many useful ways to re-arrange Ohm's Law. See *Ohm's Law Equation Table* in the appendix for common mathematical re-arrangements.

Using Ohm's Law, we calculate $P=VI = 8.86 \text{ V} \times 0.412 \text{ A} = 3.65 \text{ W}$. We can also calculate the total resistance of the circuit, as $R=V/I = 8.86 \text{ V} / 0.412 \text{ A} = 21.5 \Omega$. We therefore already know a lot about this complicated circuit without even knowing the identities of the components, or how they are connected.

One common question arises, particularly when you design a battery-powered circuit: *how long* will the device last with fresh batteries installed? This becomes an important design parameter in a practical sense, because replacing batteries is expensive, and annoying. To estimate the answer, we need to know the **current capacity** of the voltage source, or how many mAh (milliamp-hours) are provided by the batteries we are using. One alkaline

9V battery can typically provide ~400 mAh. To estimate total operating time, divide the battery mAh by total current drawn:

$$\text{Operating time} = \frac{\text{current capacity}}{\text{required current}} = \frac{400 \text{ mAh}}{412 \text{ mA}} = 0.97 \text{ h}$$

Our circuit in Figure 1-11 would not last very long if powered using a 9V battery, and left on continuously. A 9V DC power adapter might be a better idea, one that can provide at least 500 mA (power adapters always list their voltage and maximum current on the label). Table 1-1 provides typical current capacities for some common battery types. Actual capacities will vary between brands, battery chemistry, temperature, and also with drain rate.

Table 1-1. Typical voltages and current capacities for various battery types and chemistries. (Wenzel 2017)


<i>Battery Type</i>	<i>Chemistry</i>	<i>Typical Current Capacity (mAh)</i>	<i>Voltage (V)</i>
6V Lantern	alkaline	11000	6
D	alkaline	13000	1.5
C	alkaline	6000	1.5
18650 rechargeable	NiMH	3400	3.7
AA	alkaline	2400	1.5
AA rechargeable	NiMH	2000	1.2
AAA	alkaline	1000	1.5
AAA rechargeable	NiMH	800	1.2
N-type	alkaline	650	1.5
9V rechargeable	NiMH	600	8.8
9V	alkaline	400	9
CR2032	alkaline	~240	3
LR44 button cell	alkaline	~160	1.5

It's not a good idea to power a circuit with two different types of batteries concurrently or mix fresh batteries with partially discharged or dead ones.

Now that we know a little about power and current, it's important to mention that every device has limits, even copper wire. Limits of a device may be specified in terms of voltage, current, power, and often all three. If your circuit will be using a higher current, you need to choose an appropriately thicker gauge wire, or the wire could overheat and melt—potentially causing smoke, fire, or burns on your fingers. Table 1-2 will help you select a wire gauge, measured in AWG (American Wire Gauge). Similar to needles, a larger gauge number means a thinner wire. For most

of the applications throughout this course, solid core breadboard hookup wire (AWG 22) is appropriate, unless otherwise specified. It can handle close to one amp of current, which is sufficient for most of the applications in this course.

Table 1-2. Current capacity for common wire gauges. (Scherz and Monk 2016)

	<i>Wire Size (AWG)</i>	<i>Diameter (mm)</i>	<i>Ohm/1000ft (@25°C)</i>	<i>Current Capacity (A)</i>
<i>larger</i>  <i>smaller</i>	10	2.588	0.99987	14.834
	12	2.052	1.588	9.327
	14	2.628	2.524	5.87
	18	1.024	6.386	2.32
	20	0.813	10.128	1.463
	22 (breadboard wire)	0.643	16.200	0.914
	24	0.511	25.67	0.577
	28	0.320	162.00	0.227
	32	0.203	1079.00	0.091

Electrical Ground

It is difficult to talk about circuits without an understanding of electrical *ground*. This concept is important, and somewhat abstract at first. We have already mentioned ground as being a reference point for voltage. There is more to it than just that. This section will focus on different types.

DC Ground

In a DC circuit, *ground* is usually defined as the lowest voltage reference point, however as mentioned before, with a split supply (depicted in Figure 1-7) you can set your ground in the middle of the supply so that you can make use of “negative” voltage (or, less abstractly, current running the other way). There are two common ways to draw a simple DC circuit, shown in Figure 1-12.

In a circle: (the traditional way) **In a vertical line:** (the quick way)

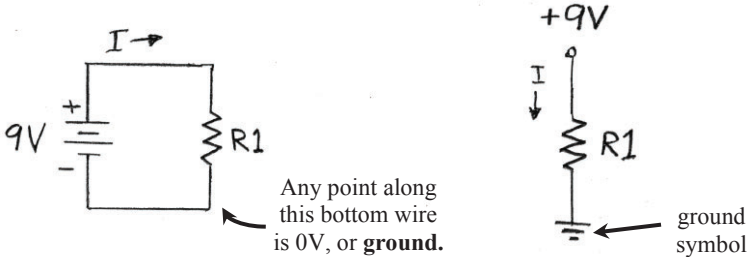


Figure 1-12. Circular and linear methods of drawing the same circuit.

Either way of drawing a circuit is valid. The ground symbol on the right is the traditional symbol for *earth ground*, meaning that a metal rod is driven into the ground and connected to the circuit; however, people tend to use this as an all-purpose ground symbol. A DC circuit rarely has a true earth ground, and is typically *floating* (not connected to the earth at all). Sometimes, people draw the ground symbol on the left diagram regardless, attached to the bottom of the circuit.

AC Ground

Ground for AC circuits is a bit different. A household power outlet in North America usually looks like this:

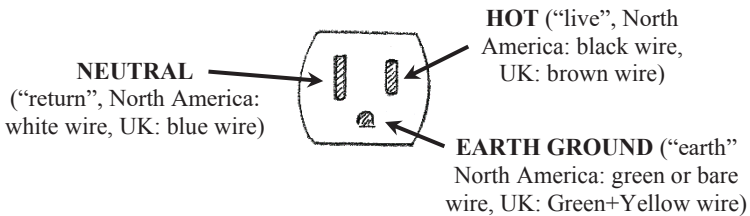


Figure 1-13. AC household power outlet (North America). AC cable wire colours are indicated for North America and the United Kingdom.

Earth ground is defined as a reference point for zero voltage. Earth ground should be connected to a rod driven into the earth (as described above). In older houses, earth ground is sometimes electrically connected to the large network of copper plumbing pipes. Ground is necessarily not the lowest voltage point in an AC circuit, because AC voltage swings from

negative to positive (120V AC swings from $\sim -170\text{V}$ to $+170\text{V}$). Ground is in the middle (see Figure 1-6).

Inside your electrical equipment, the ground wire is often connected to the equipment's frame. This is called a **chassis ground**. The point of grounding equipment properly is that if you get a **short circuit** (e.g. a hot wire touches the equipment frame), the current has a safer pathway to go, rather than going through your body to the floor.

We usually don't need an earth ground for DC circuits, which is convenient, because we tend to like our DC devices to be portable. AC circuits use much higher voltages than the DC circuits we will be tinkering with in class, and as such they are more dangerous. Be extremely careful when handling AC power. Never tinker with an AC circuit when it is plugged in to wall power.

What's a Short Circuit?

A **short circuit**, (also called a **short**) is what happens when a high voltage point in your circuit finds another way to ground that is shorter than the one you intended. It doesn't travel through the devices you planned. Connecting a high voltage point directly to ground is the absolute shortest path, and it should not be done! We will discuss the consequences of short circuits later in this text.

Different Ground Symbols

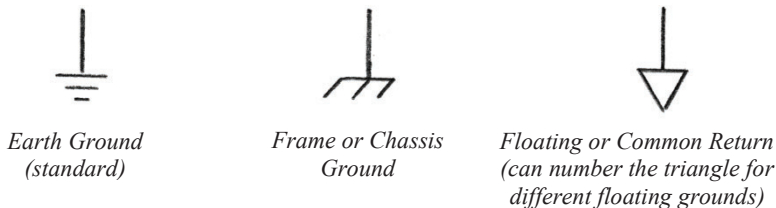


Figure 1-14. Common symbols for earth ground (left), chassis ground (middle), and floating ground (right).

Types of Returns

We call the type of ground in a circuit a **return**, because we think about this as the last step in conventional current: the end of the line (when in fact, it's the beginning!). There are different types of returns. In this course, we

will mainly building DC circuits, so the returns will mostly be floating. The earth ground symbol is used throughout this text for DC floating ground.

When you are powering your circuit using a laptop, if the laptop is plugged into the wall using a 3-pronged laptop charging cable, your system will be earth-grounded. If it is running on battery power, it will be floating. You can often see a difference in sensor behaviour between the two.

Floating Return

In a *floating return*, the ground wire is not connected to an external earth ground, or anything else for that matter. Floating returns are used in portable DC circuits.

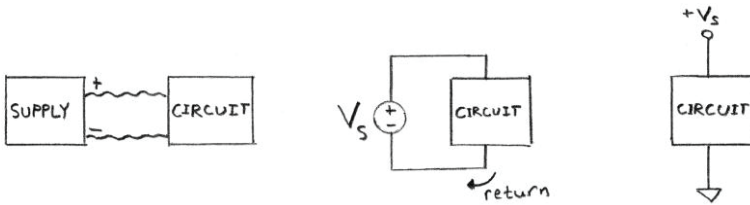


Figure 1-15. Floating return, drawn as it would be wired (left), using circular circuit diagram format (middle), and linear circuit diagram format (right).

Chassis Return

In a *chassis return*, the ground wire is connected to the metal frame of the equipment. This provides a more consistent ground if you are measuring something sensitive. The chassis itself can provide some shielding against external noise. Chassis grounds are present in both DC and AC circuits. A car's electrical system is powered by a large 12V DC car battery. The negative terminal is chassis grounded to the car's steel frame.

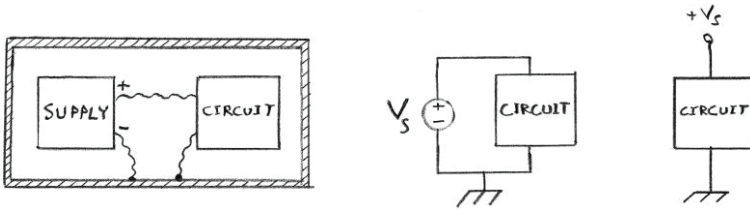


Figure 1-16. Chassis return. Note that optionally, the chassis can also be connected to earth ground, to reduce shock hazard in case of an accidental short circuit.

Earth Return

The safest type of return is an *earth return*, where the circuit ground is connected to the third prong of an AC power outlet. It may look odd, but with some DC circuits, a single wire connected to the third prong of a power outlet can help reduce signal noise. Earth returns are used most commonly with AC circuits.

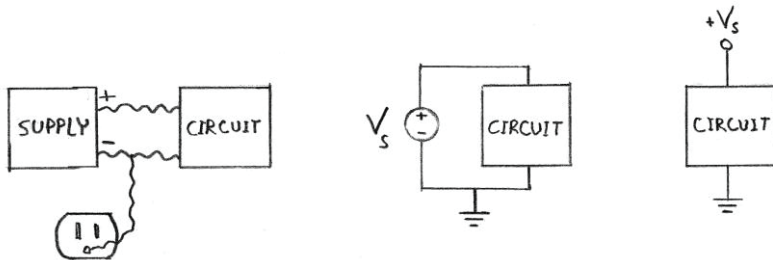


Figure 1-17. Earth return. Even though most battery-powered devices are floating, circuit diagrams tend to use the earth ground symbol for them regardless.

Single Point vs. Ground Bus

If you are planning a chassis ground, it's important to connect all planned ground points to the same point on the chassis. Otherwise, the distance between grounded points can accidentally cause grounds to be at different voltage levels—because the frame won't be a perfect conductor. This is known as *ground looping*, and can cause noise in your system.

Alternately, if the equipment layout makes a single chassis ground point inconvenient, you can also make use of a *ground bus*, which is a thick wire for grounding of low resistance, connected to the chassis and dedicated to ground. Then, a wire connected to the ground bus will effectively be at the

same voltage. The “blue” line on a breadboard is essentially a ground bus (more on breadboards later).

Voltage Sources: Series vs. Parallel

You have likely noticed that lots of your battery-powered devices run on more than one battery. You’ve probably noticed it more when you put a battery in the wrong way, and the device doesn’t power on. Arranging batteries in series and parallel have different effects on the voltage and current they can provide to a circuit.

Batteries in Series

When batteries are connected *in series* (one after the other) the voltages in the batteries are added together, however, their combined *current capacity* remains the same (Figure 1-18).

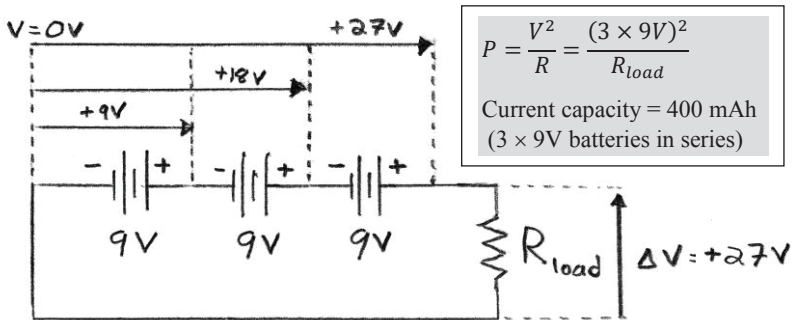


Figure 1-18. Voltage is additive, and total current capacity remains constant when batteries are wired in series.

Batteries in Parallel

Batteries can also be connected *in parallel* (positive terminals connected to each other, and negative terminals connected to each other). When this happens, the voltages are not added. In the example in Figure 1-19, the highest voltage is still 9V. However, the current capacities are added, and the circuit will run four times longer.

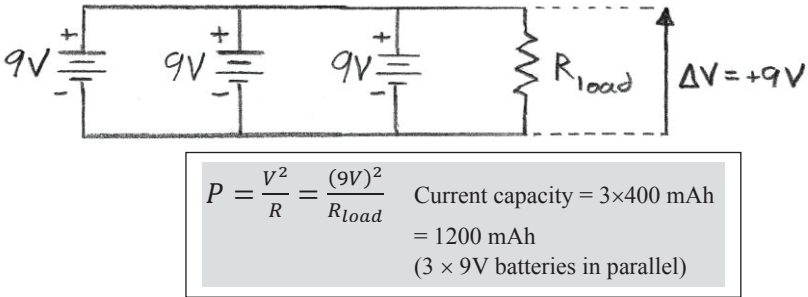


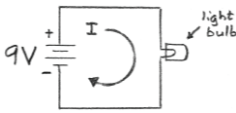
Figure 1-19. Current capacity is additive, and voltage remains constant when batteries are wired in parallel.

Make sure when wiring batteries in parallel that they are of the same type, and that they are all fresh batteries.

Circuit Configurations

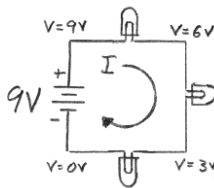
There are many ways to connect circuit components other than batteries. However, there are three basic ways to connect single components to a power supply. Let's look at different ways of connecting incandescent lightbulbs to a DC power supply (not LEDs—they would require current-limiting resistors):

Basic



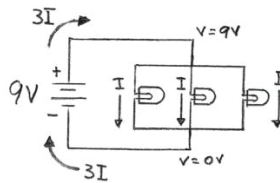
- Connecting the light bulb will turn on the light.

In Series



- Each bulb is 1/3 as bright.
- Bulbs draw 1/3× current of basic circuit from the battery.

In Parallel



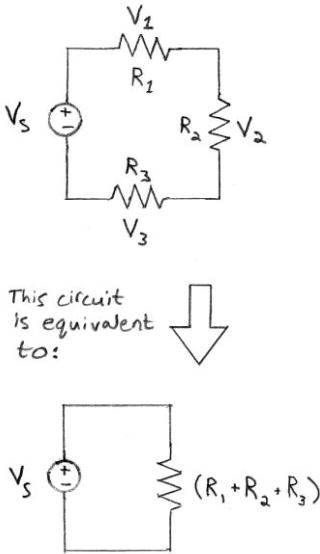
- Each bulb is as bright as the basic circuit bulb.
- Bulbs draw 3× current of basic circuit from the battery.

Figure 1-20. Three ways of connecting electronic components: basic, in series, and in parallel.

An incandescent lightbulb acts like a resistor when powered. As the current flows through the tungsten element, potential energy (voltage) is converted to kinetic energy, in the form of light and heat. However, the lightbulbs behave quite differently when connected in basic, series, or parallel configurations.

Why do resistive loads behave differently in series and in parallel? Gustav Kirchhoff, a German physicist, addressed this question, when he came up with a set of laws named after him. (Maloberti and Davies 2016) In order to solve the voltage drop across components in series, we will start with Kirchhoff's Voltage Law.

Kirchhoff's Voltage Law (KVL)



Kirchhoff's Voltage Law (KVL) is essentially a conservation of voltage. The idea is that if you look at the voltage gains and drops around a closed loop in a circuit, then the sum of them should be equal to zero. (Scherz and Monk 2016) Worded another way: if you were to start at one point of your circuit, and go around a loop, adding or subtracting the voltage differences across the components as you went along, then by the time you get back to where you started, all of those voltage differences should cancel out leaving you at the exact same voltage level as you started. For example, for the following circuit with a battery supplying V_s , and three resistors in series:

$$(1) \text{ KVL: } \sum_{i=1}^n V_i = 0$$

Going around clockwise:

$$(2) +V_s - V_1 - V_2 - V_3 = 0$$

$$(3) V_s = V_1 + V_2 + V_3$$

Figure 1-21. Kirchhoff's Voltage Law applied to resistors in series.

From (3), we can see the *total* voltage drop across *all three resistors* is equal to V_s . We can then calculate the total resistance of the circuit, *as if there were only one resistor*:

Ohm's Law: (4) $V = IR$

$$(4) \rightarrow (3):$$

$$(5) IR_{total} = I_1 R_1 + I_2 R_2 + I_3 R_3$$

It's worthwhile to note at this point that the current going through each separate resistor is equal to the current going through the loop. In other words, in a single closed loop:

$$(6) I_1 = I_2 = I_3 = I$$

$$(6) \rightarrow (3):$$

$$(7) R_{total} = R_1 + R_2 + R_3$$

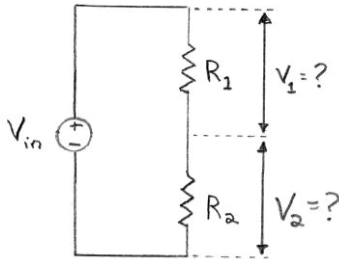
Equation (7) is generalizable to any n number of resistors in series:

$$R_{total} = R_1 + R_2 + \dots + R_n = \sum_{i=1}^n R_i \quad \text{Resistors in Series}$$

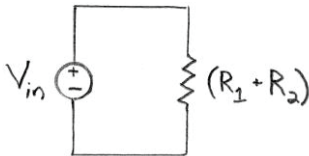
This equation has a practical consequence. If you are hunting through your resistor box and you can't find a 2K resistor, you can just wire two 1K resistors in series, and their combined resistance value will be equal to 2K.

Let's look at a simpler example, with only two resistors. We now know that the combined resistance of two resistors in series (R_1 and R_2) will be equal to their sum. But what about the voltage drop across each resistor? Let's consider the simple circuit in Figure 1-22. It has one DC voltage source, connected to two resistors in series.

What is the voltage drop across each resistor in this circuit? We can use KVL to solve it.



This circuit is equivalent to:



Firstly, we can draw the equivalent circuit, below it, knowing from the previous example that *resistance in series is additive*.

We can now solve for the total current (I) through the circuit:

$$(1) V = IR$$

$$\rightarrow (2) I = \frac{V}{R} = \frac{V_{in}}{(R_1 + R_2)}$$

As mentioned before, the current going through a single closed loop is constant, so the current calculated in (2) will be flowing through each resistor. Since a resistor is an ohmic device, if we know the current flowing through each resistor, we can calculate the voltage drop using Ohm's law:

Figure 1-22. Derivation of the voltage divider equation.

$$V_1 = I \times R_1 = \left(\frac{V_{in}}{R_1 + R_2} \right) R_1 = V_{in} \left(\frac{R_1}{R_1 + R_2} \right)$$

$$V_2 = I \times R_2 = \left(\frac{V_{in}}{R_1 + R_2} \right) R_2 = V_{in} \left(\frac{R_2}{R_1 + R_2} \right)$$

To check our work, our solved values for V_1 and V_2 should respect KVL in this single loop:

$$(3) \sum_{i=1}^n V_i = 0$$

$$(4) V_{in} - V_1 - V_2 = 0$$

$$(6) V_{in} = V_1 + V_2$$

$$(7) V_{in} = \frac{V_{in}}{(R_1+R_2)} R_1 + \frac{V_{in}}{(R_1+R_2)} R_2 = V_{in} \left(\frac{R_1}{R_1+R_2} + \frac{R_2}{R_1+R_2} \right) = V_{in} \left(\frac{R_1+R_2}{R_1+R_2} \right)$$

The math works out. Both voltages V_1 and V_2 add up to V_{in} , and we can say the sum of all voltages is equal to 0. We have now expressed the voltage drops across R_1 and R_2 completely in terms of our known values, V_{in} , R_1 , and R_2 . Here is our final “solved” circuit. We have just derived the **voltage divider equation**.

The Voltage Divider Equation

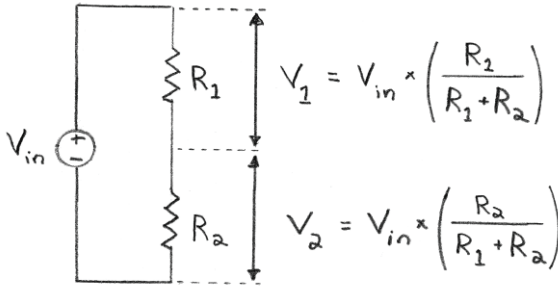


Figure 1-23. The voltage divider equation. The voltage across each resistor is proportional to the ratio of its contribution to the total resistance, R_1+R_2 .

We can now also calculate the power loss across each resistor:

$$(9) P_1 = V_1 I$$

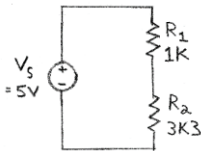
$$(10) P_2 = V_2 I$$

$$(11) P_{total} = V_{in} I = P_1 + P_2$$

Or alternately,

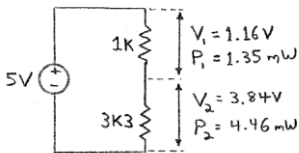
$$(12) P_{total} = P_1 + P_2 = I^2 R_1 + I^2 R_2$$

Example: Calculate the voltage and power drop across each resistor when placed in series, with a 5V DC voltage supply:



Note: 3K3 = 3.3K
(letter takes the place of the decimal)

Solution



Voltage:

$$\begin{aligned} V_1 &= V_{in} \left(\frac{R_1}{R_1 + R_2} \right) \\ &= 5V \left(\frac{1K}{1K + 3.3K} \right) \\ &= 5V \times \frac{1}{4.3} \\ &= 1.16V \end{aligned}$$

Current:

$$\begin{aligned} V_s &= I R_{total} \\ I &= V_s / R_{total} \\ &= 5V / 4.3K \\ &= 1.16 mA \end{aligned}$$

Power:

$$\begin{aligned} P_1 &= V_1 I \\ &= 1.16V \times 1.16 mA \\ &= 1.35 mW \end{aligned}$$

$$\begin{aligned} V_2 &= V_{in} \left(\frac{R_2}{R_1 + R_2} \right) \\ &= 5V \left(\frac{3.3K}{1K + 3.3K} \right) \\ &= 5V \times 3.3 / 4.3 \\ &= 3.84V \end{aligned}$$

(Note: KVL $V_1 + V_2 = 5V$)

Power:

$$\begin{aligned} P_2 &= V_2 I \\ &= 3.84V \times 1.16 mA \\ &= 4.46 mW \end{aligned}$$

Figure 1-24. Worked example for the voltage divider equation.

It's interesting to note that it's the *ratio* of $R_2/(R_1+R_2)$ that's important, so *any pair* of resistors with the same ratio of resistance above (e.g. $R_1=10K$, $R_2=33K$) will produce the same voltage split as in the example. However, higher value resistors will result in less current (I) through the circuit.

Voltage dividers are great for providing reference voltages, and useful in many other contexts. In general, although it may be tempting, you should not use a voltage divider to step down the voltage in order to *power* a circuit. There are several reasons for this:

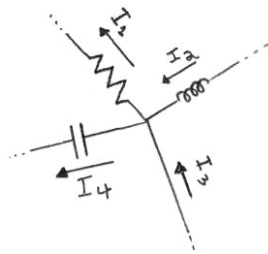
- It's a waste of power. It's the least efficient way of lowering voltage, as the R_1 resistor will convert the volts to heat. Whereas linear voltage regulators can be packaged to handle that heat dissipation, lower power-rated resistors will roast and smoke.
- The divider can act unpredictably, since whatever you are hooking up between R_1 and R_2 will also have its own resistance and upset the voltage split.
- This strategy is unregulated, meaning that if the supply voltage spikes, dips, or ripples, so will V_2 .

- There are better strategies to supply the correct voltage, like voltage regulators, buck/boost converters, and finding a supply/battery at the desired voltage.
- If you must use a voltage divider to supply power, see *Voltage Divider Design: 10% Rule*.

Kirchhoff's Current Law (KCL)

Kirchhoff's current law is the *conservation of electric charge*, which amounts to a conservation of current. If we can picture one branching point (or node) in our circuit diagram, the sum of all currents entering the node should be equal to the sum of all currents leaving. Picture a river, splitting into two paths. The water must go somewhere.

To illustrate this, consider the following somewhat complicated intersection in a circuit:



KCL: $\sum_{i=1}^n I_i = 0$

$$I_2 + I_3 - I_1 - I_4 = 0$$

(currents for I_1 and I_4 are flowing out of the node, so they are negative)

$$I_2 + I_3 = I_1 + I_4$$

Figure 1-25. Kirchoff's Current Law example.

Using KCL, we can calculate the currents that flow through resistors in parallel:

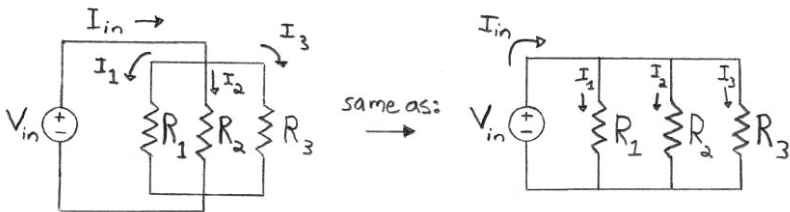


Figure 1-26. KCL applied to resistors in parallel. Note that the drawing on the left is equivalent to the drawing on the right electrically. Resistors in parallel can be represented either way.

At the top node where the labeled currents intersect, I_{in} is entering the node, and I_1, I_2, I_3 are leaving the node. Applying KCL, we can write:

$$(1) \text{ KCL: } \sum_{i=1}^n I_i = 0$$

$$(2) I_{in} = I_1 + I_2 + I_3$$

$$(3) \text{ Ohm's Law: } V=IR \rightarrow I=V/R$$

$$(3) \rightarrow (2): (5) \frac{V_{in}}{R_{Total}} = \frac{V_1}{R_1} + \frac{V_2}{R_2} + \frac{V_3}{R_3}$$

However,

$$(4) V_1 = V_2 = V_3 = V_{in}$$

The voltage drop across each resistor is the same, since they all connect the same two wires. In fact, anything you connect directly to the positive and negative side of the battery will have a voltage drop equal to V_{in} . That's why it's very important not to short circuit the battery (more on that later).

$$(4) \rightarrow (5): (6) \frac{V_{in}}{R_{Total}} = \frac{V_{in}}{R_1} + \frac{V_{in}}{R_2} + \frac{V_{in}}{R_3}$$

$$\div V_{in}: (7) \frac{1}{R_{Total}} = \frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3}$$

Taking the reciprocal of this equation:

$$(7) R_{Total} = \left(\frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3} \right)^{-1}$$

Equation (7) is generalizable to any n number of resistors leaving the node:

$$R_{Total} = \left(\frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3} + \dots + \frac{1}{R_n} \right)^{-1} = \left(\sum_{i=1}^n \frac{1}{R_i} \right)^{-1} \quad \text{Resistors in Parallel}$$

For only two resistors, the equation simplifies somewhat, and we can express R_1 in parallel with R_2 as:

$$R_{Total} = R_1 || R_2 = \left(\frac{1}{R_1} + \frac{1}{R_2} \right)^{-1} = \frac{R_1 R_2}{R_1 + R_2}$$

The double-pipe symbol “||” in this context means that the two components are wired in parallel. This equation is worth memorizing. Like the voltage divider equation, it comes up again and again in electronics.

Note: A resistor in parallel with itself results in *half* the value:

$$R_1 || R_1 = \frac{R_1 R_1}{R_1 + R_1} = \frac{R_1^2}{2R_1} = \frac{R_1}{2}$$

There is a simplified formula for three resistors in parallel as well:

$$R_{Total} = R_1 || R_2 || R_3 = \left(\frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3} \right)^{-1} = \frac{R_1 R_2 R_3}{R_1 R_2 + R_1 R_3 + R_2 R_3}$$

For two resistors in parallel, what is the current across each resistor (I_1 and I_2)?

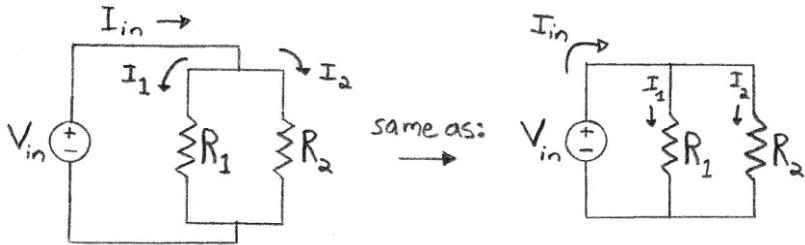


Figure 1-27. KCL for two resistors in parallel.

- (1) **KCL:** $\sum_{i=1}^n I_i = 0$
- (2) $I_{in} = I_1 + I_2$
- (3) **Ohm's Law:** $V = IR \rightarrow I = V/R$
- (3) \rightarrow (2): (4) $I_{in} = V_1/R_1 + V_2/R_2$

However,

- (5) $V_1 = V_2 = V_{in}$
- (5) \rightarrow (4): (6) $I_{in} = V_{in}/R_1 + V_{in}/R_2$
- (7) $I_{in} = V_{in} \left(\frac{1}{R_1} + \frac{1}{R_2} \right)$

Re-arranging (7) to solve for V_{in} :

$$(8) V_{in} = \frac{I_{in}}{\left(\frac{1}{R_1} + \frac{1}{R_2} \right)} = I_{in} \left(\frac{1}{R_1} + \frac{1}{R_2} \right)^{-1} = I_{in} \left(\frac{R_1 R_2}{R_1 + R_2} \right)$$

Now if we write **Ohm's Law** just across R_1 :

- (9) $V_1 = V_{in} = I_1 R_1$
- (8) = (9): (10) $I_1 R_1 = I_{in} \left(\frac{R_1 R_2}{R_1 + R_2} \right)$

$$\text{Simplify: (11) } I_1 = I_{in} \left(\frac{R_2}{R_1 + R_2} \right)$$

$$\text{Similarly: (12) } I_2 = I_{in} \left(\frac{R_1}{R_1 + R_2} \right)$$

We have now expressed the current split into R_1 and R_2 completely in terms of our known values, I_{in} , R_1 , and R_2 . Here is our final “solved” circuit, and we have derived the **current divider equation**.

The Current Divider Equation

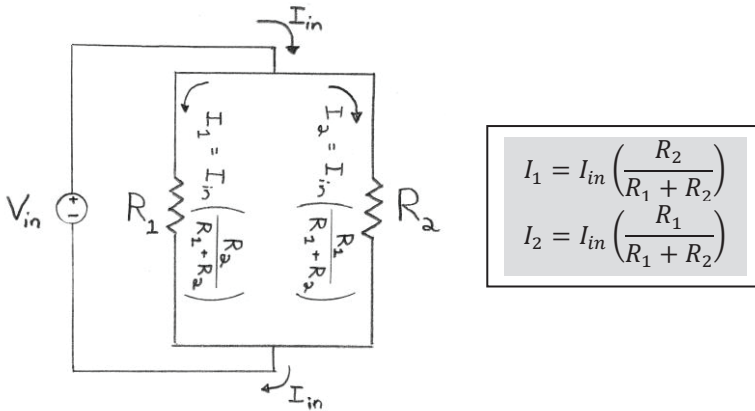


Figure 1-28. The current divider equation, for two resistors in parallel.

We can now also calculate the individual power losses across each resistor:

$$P_1 = V_1 I$$

$$P_2 = V_2 I$$

$$P_{total} = V_{in} I_{in} = V_1 I_1 + V_2 I_2 = V_{in} I_1 + V_{in} I_2$$

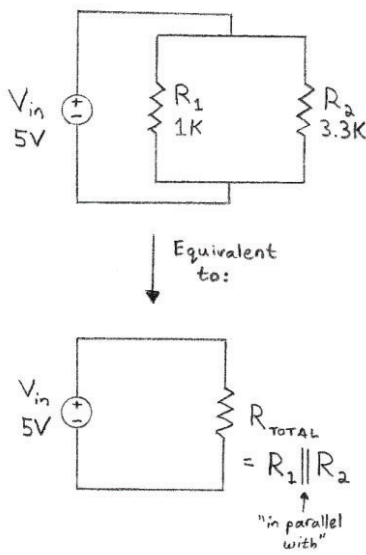
Power across R_1 :

$$P_1 = V_{in} I_1 = V_{in} I_{in} \left(\frac{R_2}{R_1 + R_2} \right)$$

Power across R_2 :

$$P_2 = V_{in} I_2 = V_{in} I_{in} \left(\frac{R_1}{R_1 + R_2} \right)$$

Example: Calculate the voltage and power drop across each resistor when placed in parallel, with a 5V power source:



$$R_1 \parallel R_2 = \left(\frac{1}{R_1} + \frac{1}{R_2} \right)^{-1} = \frac{R_1 R_2}{R_1 + R_2}$$

$$= \frac{1\text{K} \cdot 3.3\text{K}}{1\text{K} + 3.3\text{K}}$$

$$= 0.7674\text{K}$$

$$V_{in} = I_{in} R_{TOTAL}$$

$$\rightarrow I_{in} = \frac{V_{in}}{R_{TOTAL}} = \frac{5\text{V}}{0.7674\text{K}} = 6.515\text{mA}$$

Current Divider Equation:

$$I_1 = I_{in} \left(\frac{R_2}{R_1 + R_2} \right) = 6.515\text{mA} \left(\frac{3.3}{1 + 3.3} \right)$$

$$= 5.0\text{mA}$$

$$I_2 = I_{in} \left(\frac{R_1}{R_1 + R_2} \right) = 6.515\text{mA} \left(\frac{1}{1 + 3.3} \right)$$

$$= 1.515\text{mA}$$

$$(KCL: I_2 = I_{in} - I_1 = 6.515\text{mA} - 5\text{mA})$$

Power:

$$P_1 = V_1 I_1 = 5\text{V} \times 5\text{mA} = 25\text{mW}$$

$$P_2 = V_2 I_2 = 5\text{V} \times 1.515\text{mA} = 7.575\text{mW}$$

$$P_{TOTAL} = P_1 + P_2 = 25\text{mW} + 7.575\text{mW}$$

$$= 32.58\text{mW}$$

Figure 1-29. Worked example for the current divider equation.

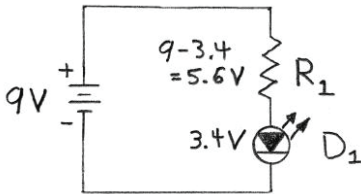
Table 1-3. Summary chart: resistors in series vs. resistors in parallel.

Resistors in Series	<ul style="list-style-type: none"> • The current through each resistor in series is the same. • The voltage is split proportionally to the resistor values (higher resistance values get larger voltage drops across them). • The equivalent overall resistance is the sum of all the resistance values in series. • Two equal value resistors in series have a total resistance of $2 \times R$.
Resistors in Parallel	<ul style="list-style-type: none"> • The voltage drop across each resistor in parallel is the same. • The current is split proportionally to the resistor values (lower resistance values have higher currents through them). • The equivalent overall resistance is <i>less than</i> any single resistor separately. • Two equal value resistors in parallel have a total resistance of $R/2$.

Calculating Current-Limiting Resistor Values for LEDs

LEDs (light emitting diodes) are not ohmic devices. They require a current limiting resistor to keep them from burning out. What resistor value is appropriate? The answer depends on the power supply, and the rating of the LED. The following worked example illustrates some of the design considerations involved in selecting a resistor for this specific application.

Example: An LED has an advertised current of 20 mA, and a forward voltage drop of 3.4V. What resistor should you use to limit the current, if you are using a 9V battery to power the LED?



Answer: The voltage drop across the resistor will be $9V - 3.4V = 5.6V$ (KVL). The required resistance will be:

$$R = \frac{V}{I} = \frac{5.6V}{0.020A} = 280\Omega$$

Figure 1-30. Calculating the resistance of a current-limiting resistor for an LED. The LED symbol is labelled D_1 (for diode 1) in the circuit diagram.

A resistor value of 280Ω would be appropriate.

Now that we have an “answer”, there is the practical matter of selecting a fixed resistor value, its **maximum power rating**, and its **tolerance**, so we can purchase the correct resistor for our circuit. *Common Fixed Resistor and Capacitor Values* in the appendix lists some common fixed resistor values. Notice that there isn’t a 280Ω resistor in the chart. The closest values are 270Ω , and 300Ω . Which value should you select? It would be more conservative to select 300Ω , as it would result in a lower (safer) current to the LED. This means the current in the circuit won’t be limited by a 280Ω resistor, it will be limited by a 300Ω resistor. Our actual current then will be $I = V/R = 5.6V/300\Omega = 18.7\text{ mA}$.

We can calculate that the power dissipated by this resistor will be $P = VI = 5.6V \times 0.0187A = 0.1045W$. To select an appropriate power rating for the resistor, *double this value* to be conservative ($0.209W$). Resistors are commonly available in $1/8W$, $1/4W$, $1/2W$, and $1W$ power ratings. The larger power rating resistors are physically larger, because they need to dissipate more heat. For this application, a $1/4W$ resistor would be best suited. A lower power rating could end up overheating the resistor. Going

with a higher power rating would still work, but the higher value will take up more space than necessary, and would be a more expensive component.

We also need to select the **tolerance** of a resistor. The concept of tolerance is somewhat analogous to drug potency. Just like a 200 mg Advil will not contain exactly 200 mg of ibuprofen, a resistor will not have a measured resistance value exactly equal to its label claim. Resistor tolerances are typically available at 1% and 5% tolerances. If precision around the resistance value isn't critical, you would purchase a 5% tolerance resistor, and that would be fine. This means the actual resistance of the resistor would fall between $95\% \times 300 \Omega$ and $105\% \times 300 \Omega$ (or 285 to 315 Ω). The 5% tolerance fixed resistors in our lab are beige, with coloured stripes to indicate their resistance values. If you require a more precise resistance value (e.g. for an op-amp gain—more on that later), then you would use a 1% tolerance resistor. This means the actual resistance of the resistor would fall between $99\% \times 300\Omega$ and $101\% \times 300\Omega$ (or 297-303 Ω). For this example, a 5% tolerance is appropriate, since we are already rounding up from 280 Ω to 300 Ω , and we do not require more precision to power LEDs.

Final answer: An LED with a current of 20 mA and forward voltage of 3.4 V would require a 300 Ω , 1/4 W, 5% tolerance resistor if powered using a 9V battery. If you find the LED is too bright, it is safe to use a higher value resistor (e.g. 1 K or 10 K). This will result in longer LED and battery life for this circuit.

Anode vs. Cathode: Devices with Polarity

There are devices for which the direction of current flow does not matter. For instance, fixed resistors are reversible. They work the same way wired in either direction. However, an LED is a **polarized** device, meaning that just like a battery, it is designed with an intended polarity in mind. It will only light up when the current flows through it in the correct direction. We will go into more detail about this in our discussion about diodes. For now, we will focus on identifying a device's anode and cathode, a concept which can be quite confusing. A **cathode** is thought of as a **charge emitter**, in the sense of conventional

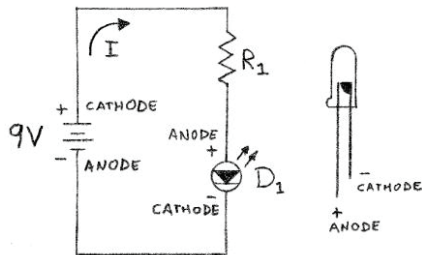


Figure 1-31. LED & resistor circuit, with anodes and cathodes labeled.

current. The cathode is the part of a device that emits “positive charge”.¹ We can picture positive charge leaving the battery in Figure 1-31 from its positive terminal. This terminal is known as the battery’s *cathode*. The other side of the battery is where the “positive charge” flows into—this is known as the *anode*. From the perspective of the LED, the roles are switched. Have a look at the LED in the circuit diagram of Figure 1-31. The charge emitter then is still the *cathode*. The LED is not producing energy, it’s using it up. The anode and cathode terminals therefore reverse signs. The LED’s anode is positive, taking on the role as the charge acceptor, and the LED’s cathode is negative, taking on the role of the charge emitter. The roles are the same, but the polarity is reversed.

How do we know which way to hook up the LED, if only one way works? The convention in electronic devices is to make one wire (or leg) of a polarized device longer than the other. In general (but not always) the *longer* wire means “please hook me up to positive”. Most LEDs and polarized capacitors use this convention. If the legs are snipped equally, the cylindrical ridge on the side of the LED is flattened on the cathode side, to help you distinguish polarity. If you find it confusing to identify the anode and cathode, just remember that the longer wire is connected to the higher (positive) voltage, or if all else fails, just try reversing the LED terminals if it doesn’t light up. On a circuit diagram, the little triangle inside an LED symbol points in the direction of conventional current.

Introduction to Switches

We will be going into more detail for switches. Switches can be quite complicated. However, the simplest of circuits usually have an *on/off switch*. When the switch is *open*, there a physical gap (or discontinuity) in the wire, making the circuit no longer a closed circle, but an open dead end. This results in essentially no electrons flowing, zero current, and no power being drawn from the source—because the circle is broken, and electrons generally don’t jump through the air very well. When the switch is *closed*, terminals inside the switch touch,

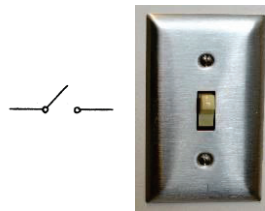


Figure 1-32. Circuit diagram symbol (left) and example (right) of a latching on/off switch.

¹ We know that the actual direction of current is opposite – the positive side of a battery (cathode) *receives* electrons, and the negative side (anode) emits them. However, we are using conventional current definitions for this discussion.

allowing current to flow through. We call this a **closed switch**. This is somewhat counter-intuitive, because when we shut off a light in a room, we are really opening the switch (and consequently the circuit).

A switch is typically represented in a circuit diagram as a break in the wire. The switch in Figure 1-32 is drawn as a **latching**, or **toggle switch**. This means when you flip the switch, it stays in the position to which it is flipped (like a light switch on your wall).

For a power switch, we generally interrupt the positive terminal of the power source on a DC circuit, and the hot (or live) wire on an AC circuit. This cuts power to the whole circuit when the switch is in the off position, reducing the chances of a short circuit when the switch is open. This is called **high side switching**. We will discuss this more in detail in Section 5.

Another type of basic, and useful switch is a **momentary switch**, the simplest and most popular type being “normally open, held closed” (NOHC). This switch only completes the circuit as long as the switch is held closed, or in other words, as long as the button is pushed down. The moment you let go, the switch opens again. A momentary switch can be used to power a circuit for a short period of time, or transmit information (such as a user command). A popular example of a NOHC switch is a car horn.

Figure 1-33 shows the four-pin momentary switches we will be using throughout this course. In Figure 1-33, pushing the momentary switch electrically connects all four of the pins together, effectively making them all the same continuous wire. When the button is not pushed, only the opposite pins are connected.

This can be somewhat confusing at first. How can you tell the difference between the pins when looking at the switch? See if you can figure this out by inspecting Figure 1-33. If in doubt, the switch will always be wired correctly as a momentary switch if you connect it to the circuit using diagonally-opposing pins, and there is only one way a momentary switch will physically fit nicely across the ravine of a breadboard (shown in Figure 1-34).

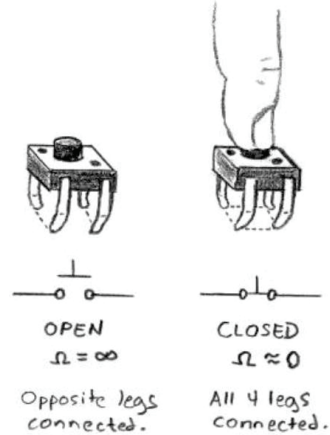


Figure 1-33. Momentary switch connections.

Breadboarding

If you are trying to design and test a circuit for the first time, using a **breadboard** is a great way to go. Circuit connections are made by plugging your wires and devices into a temporary scaffolding. This means you don't have to solder any wires to test a circuit, and changes are easily made.

When looked at lengthwise, the inner *columns* of a breadboard (also called **terminal strips**) are electrically connected. To connect a wire, push the bare end deep in the pin hole of a column straight in, without bending the tip. Now, any other hole in the same column will be connected to that wire, making it easy for you to connect two or more wires or devices.

The top half of the breadboard is not electrically connected in any way to the bottom half. The top and bottom halves are separated by a big groove, also called a **ravine**. When connecting an integrated circuit or component with 0.1" (2.54mm) pin spacing to a breadboard, to access all the pins, you place the component lengthwise across the ravine (as pictured in Figure 1-34). This allows you to connect wires to every single pin of the chip, and avoids shorting out pins across from each other.

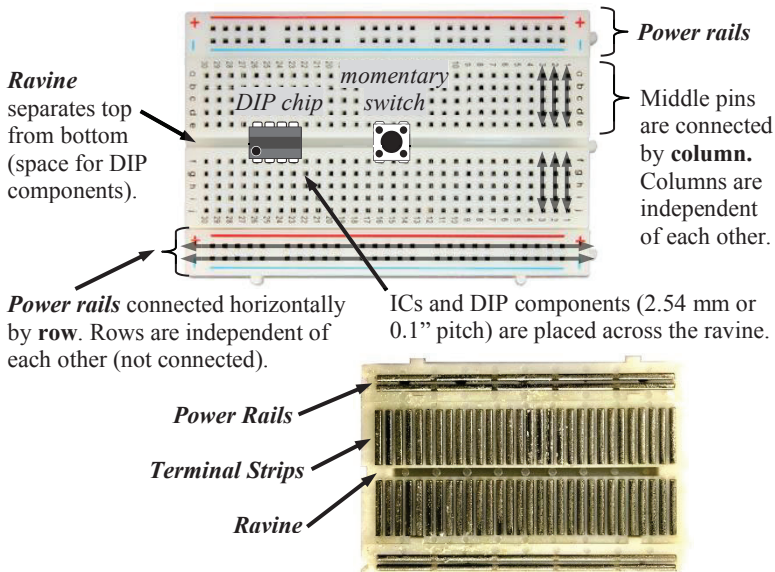


Figure 1-34. Breadboard layout (top). Transparent arrows show how the power rails are connected by row, and middle pins are connected by column. The adhesive backing was peeled from the underside of a breadboard, revealing directionality of the internal rails (bottom).

The power rails run horizontally. You would normally connect the positive terminal of a battery or voltage source to a red (+) rail, and the negative terminal or ground to a blue (-) rail. Since the top set of rails are not connected to the bottom set, you need to connect them with wires if you'd like to use them as power rails. Once the power rails are connected to the battery, you can conveniently connect components from any column to the battery from anywhere on the closest rail. It takes a little practice, but this breadboard layout makes putting circuits together very convenient. Be careful not to bend pins on momentary switches or integrated circuits, as they are fragile and are easily damaged.

Putting all these concepts together, we can now look at the basic structure of a circuit diagram. An example of a fun project, a light theremin, is provided in Figure 1-35. This circuit emits a tone that changes pitch based on the amount of light received by a photoresistor. (Steve Hobley 2012)

Circuit Diagram Etiquette Example: Light Theremin

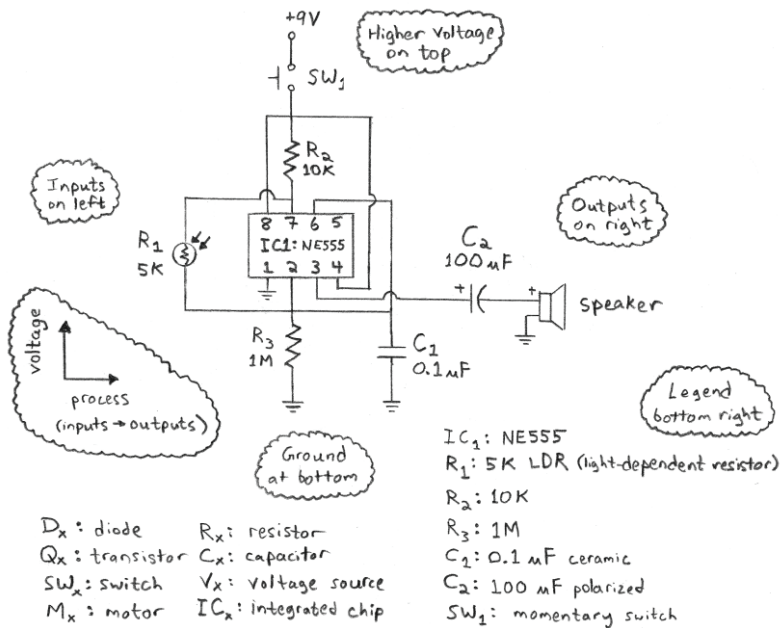


Figure 1-35. Light theremin circuit diagram. If a legend is used, values next to the circuit diagram symbols may be omitted.

A circuit diagram should contain enough detail for someone to assemble the device described using the correct components, resulting in a functional circuit. In many ways, a circuit diagram is a visual protocol. The convention is for higher voltages to be drawn above lower voltages. Process inputs are drawn on the left (e.g. a button or sensor), and outputs are drawn on the right (e.g. a light or speaker). Ground is at the bottom, although for simplicity's sake, sometimes short ground symbols are drawn from a single pin (e.g. pin 1 of IC₁, above). (Gibilisco 2013) Circuit diagrams reside at the fun interface of art and science. Your ability to draw them will affect how well you share your ideas. There are many ways to represent the same component in a circuit diagram. What is most important when drawing a circuit diagram is that the reader understands your ideas, and is able to make the correct connections with the appropriate parts.

Many of the circuit diagrams in this manual have been intentionally hand-drawn. There are a variety of free software packages available for drawing circuit diagrams. However, when it comes to drawing small circuits, a ruler, mechanical pencil, and unlined sheet of paper will serve you well.

Activity 1-1: 9V Battery + LED + 10K Resistor

Goal: In this activity, we will introduce a simple circuit for you to assemble and test, using a breadboard.

Materials:

- Digital multimeter
- 1 x Breadboard
- 1 x 9V Battery + Snap-on Connector
- 1 x 10K resistor
- 1 x LED
- 4 or 5 Male/Male Jumpers
- 1 x Momentary Switch

Procedure:

- a) Measure and record the voltage across the disconnected battery with a voltmeter.
- b) Measure the resistance across the disconnected resistor with an ohmmeter.
- c) Build the circuit above, connecting the battery last.
- d) With the LED light on, measure the voltage across the LED.

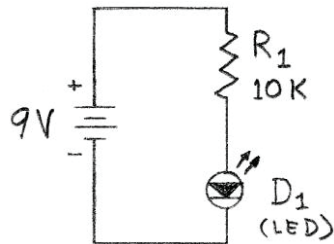


Figure 1-36. Schematic for Activity 1-1. Note: the *longer* wire on the LED is the positive side (anode).

- e) Measure the current at different parts of the circuit (e.g. before resistor, after resistor, after LED) and verify the current is constant/ the same everywhere in the loop.
- f) Complete the Table 1-4 with your measured and calculated values.

Table 1-4. Measured and calculated values for Activity 1-1.

	<i>Resistor R_1</i>	<i>LED</i>	<i>Battery</i>
Voltage (V) (measured)			
Current, I (mA) (measured)			
Power, P (mW) (Calculated: $P=VI$)			

- g) Replace the 10K resistor with a 100K resistor. What happens to the brightness of the LED?
- h) What happens to the overall power of the circuit when you replace the 10K resistor with a 100K resistor? **Hint:** Battery voltage \times current = power supplied by the battery.
- i) Try adding a momentary switch to your circuit *in series* with the LED and resistor, to turn on the LED when pushed.

HELP! My circuit isn't working!

Right about now, you likely have a small pile of components in front of you that aren't doing what you expected. Here are some general tips on building and troubleshooting for your first activity:

- Try to make connections as simple as possible (fewer connections = less chances for errors).
- Build slowly and carefully, with the circuit diagram in front of you. **HIGHLIGHT** the connections you make on the circuit diagram with a highlighter, as you go along. This will help you keep track of what's left to do.
- If you plug both legs of an LED into the same column, the current won't flow through the LED.
- If you forget to wire the 10K resistor in series with the LED, the LED will burn out.

- Connecting your battery **last** will give you a chance to correct mistakes you make along the way, and protect your components from damage. You should never build a circuit with the power connected.
- Have a look at the *Troubleshooting Guide* in the appendix for some helpful tips on getting your circuit working.
- *Advice for later:* Don't try to build your circuit all at once. If your design is modular, work on and test one section at a time.

Activity 1-2: 9V Battery + 10K Resistor + 100K Resistor

Goal: In this activity, we will confirm the voltage divider equation.

Materials:

- Digital multimeter
- 1 x Breadboard
- 9V Battery + Snap-on Connector
- 1 x 100K resistor
- 1 x 10K resistor
- 4 or 5 Male/Male Jumpers

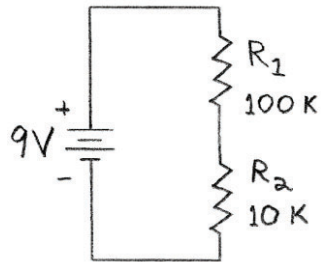


Figure 1-37. Schematic for Activity 1-2.

Procedure:

- Measure and record the actual resistance values of the 10K and 100K resistors.
- Assemble the circuit according to the circuit diagram, connecting the battery last.
- Measure the voltage across the components separately: battery, R_1 , and R_2 .
- If the remaining current capacity of the 9V battery is 400 mAh, calculate how long this circuit will run on the battery provided.

Table 1-5. Measured and calculated values for Activity 1-2.

		Battery	R_1	R_2
Resistance, R (Ω)	Measured:			
Voltage (V)	Theoretical: (Calculated)			
	Measured:			

Current, I (mA)	Theoretical:	
	Measured:	
Operating Time (h)	$= \frac{\text{Current Capacity}}{\text{Required Current}}$	

Demo: Light Theremin

There will be an in-class demonstration of the light theremin circuit at the end of this section.

Learning Objectives for Section 1

After having attended this class, the student will be able to:

- 1) Define and explain the difference between AC and DC power.
- 2) Use the generalized power law to calculate the power consumption of a circuit.
- 3) Predict the voltage drop across resistors using Ohm's Law.
- 4) Select an appropriate power source for a DC circuit.
- 5) Predict how long a circuit would run on fresh batteries given the current requirements of the circuit.
- 6) Represent resistors, voltage sources, wire, grounds, and switches in a circuit diagram.
- 7) Properly draw a basic circuit in the proper orientation (process flow: left→right, voltage: down→up)
- 8) Select an appropriate wire gauge for a circuit, based on the required current.
- 9) Use Kirchoff's Voltage Law (KVL) to predict the voltage drops across a voltage divider.
- 10) Use Kirchoff's Current Law (KCL) to predict the current split through a current divider.
- 11) Identify the actual vs. conventional direction of current in a basic DC circuit.
- 12) Correctly interpret a simple circuit diagram, and breadboard the appropriate connections to make the circuit work.
- 13) Appropriately measure voltage, current, and resistance using a digital multimeter.
- 14) Correctly position a high-side power switch in an AC or DC circuit.
- 15) Select an appropriate current-limiting fixed resistor for an LED.

16) Add a momentary switch to a circuit, observing the correct pin orientation.

Section 1 - Station Content List

- Digital multimeter
- 1 x Breadboard
- 1 x 9V Battery + Snap-on Connector
- 1 x 10K resistor
- 1 x 100K resistor
- 1 x LED
- 4 or 5 Male/Male Jumpers
- 1 x Momentary Switch

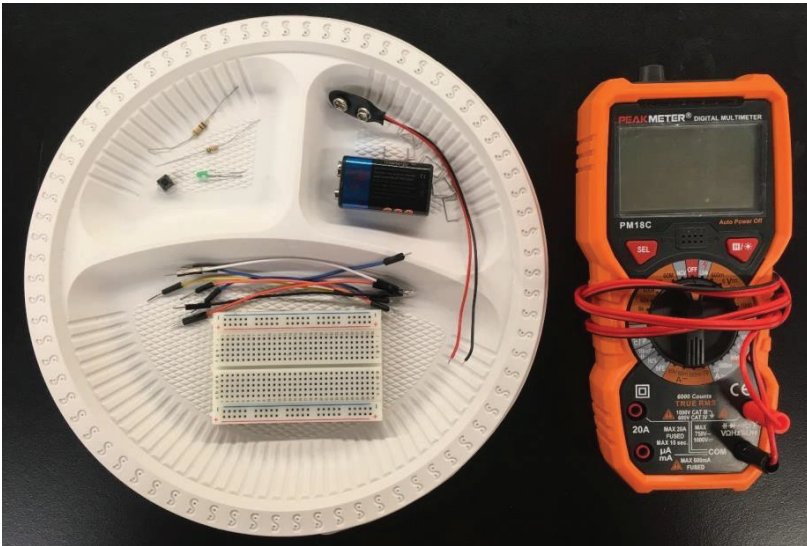


Figure 1-38. Section 1 station setup.

SECTION 2

CAPACITANCE, POWER AND LOGIC

What You'll Be Learning	Lecture: Capacitors - charging/discharging time. Voltage dividers - 10% rule. Voltage and current regulators. Integrated circuits and pin numbering conventions. Thévenin's Theorem. Logic gates - 74XX Series Logic Gates: 7408 (AND), 7432 (OR), 7404 (NOT). Venn diagrams and logic tables.	
What You'll Be Doing	<i>Select two of the following 3 activities:</i> Activity 2-1: Capacitor charging/discharging circuit. Activity 2-2: 5V voltage regulator circuit (LM317). Activity 2-3: Building and testing logic gates. DIP switches - generating and verifying logic tables (requires Activity 2-2).	
Files you will need	All course files are available for download at: http://pb860.pbworks.com	<ul style="list-style-type: none">• <i>Voltage Divider.xlsx</i>• <i>LM317.xlsx</i>

Capacitors

A capacitor can be thought of as a temporary rechargeable battery, that charges up or discharges when exposed to a difference in voltage. When a capacitor is connected to a DC voltage source, it charges almost instantaneously. When the capacitor is shorted, it discharges almost instantaneously. The most common construction of a capacitor is two conductive plates separated by a thin non-conductive layer, which can be ceramic, air, or another material, depending on the type of capacitor.

We measure **capacitance** in Farads (from Michael Faraday's pioneering work in capacitance), and use the symbol "F".

- Capacitance (Farads) = Charge (Coulombs) / Volts

$$C = \frac{Q}{V} = \frac{c}{J/c} = \frac{c^2}{J}$$

Capacitors are rated in Farads. The range of typical capacitors in DC applications is typically on the pF (1×10^{-12} F) to μ F (1×10^{-6} F) scale. See *Common Fixed Resistor and Capacitor Values* in the appendix for common

fixed capacitor values. A capacitor with a higher Farad rating means it can hold more charge.

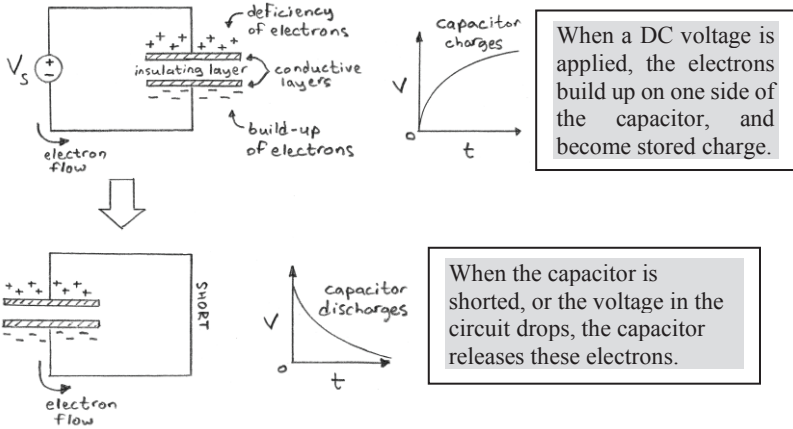


Figure 2-1. Electron build-up and flow upon capacitor charging and discharging.

Capacitor Circuit Diagram Symbols

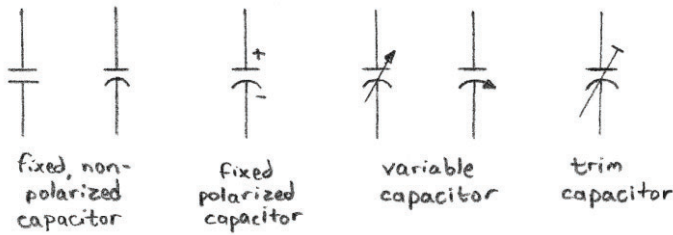


Figure 2-2. Circuit symbols for different types of capacitors.

Capacitor Ratings

The capacitance rating of an *electrolytic capacitor* is usually printed directly on the capacitor in small letters with units (e.g. 100 μF).

The rating of a nonpolarized (e.g. ceramic) capacitor is indicated using a **3-digit code**. As the components are so small, the code can be difficult to see. To translate this number into a capacitance, use the following scheme:

Translating a 3-digit Capacitor Code

- 1) Write down the first 2 digits as they appear on the capacitor.
- 2) The third digit is the number of zeros to add after that number.
- 3) The answer is in pF.
- 4) If there are only 1 or 2 digits, that number is the capacitance in pF.

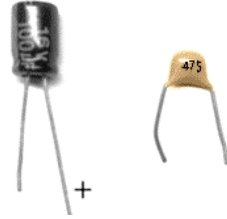


Figure 2-3. Electrolytic (left) and ceramic (right) capacitors.

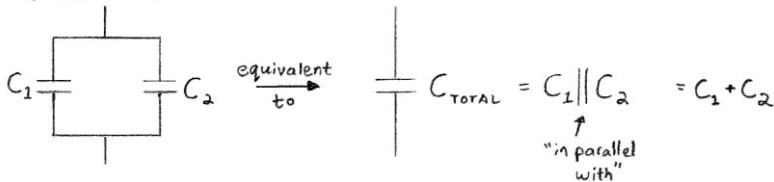
Example: The 3-digit capacitor code in Figure 2-3 is 475. What is the capacitance value?

- 1) The first two digits are 47.
- 2) The third digit is 5, so write down five zeros in front of 47.
- 3) The capacitance is 4,700,000 pF = 4,700 nF = 4.7 μ F.

Capacitors in Series and Parallel

When capacitors are wired together, the total combined capacitance is calculated *oppositely* to resistors. Capacitance values are added when capacitors are wired in parallel. When capacitors are wired in series, the total capacitance is equal to the inverse of the sum of inverted capacitance values.

Capacitors in parallel:



Capacitors in series:

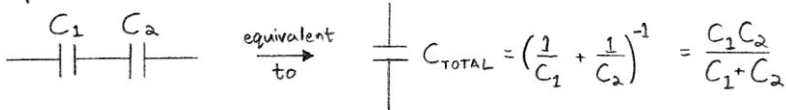


Figure 2-4. Calculating total capacitance of capacitors in parallel and in series.

Capacitors: Typical Uses

- Charge and discharge cycle for power (e.g. camera flash):

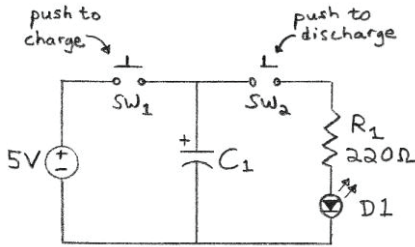


Figure 2-5. A charge-discharge circuit. Holding down SW₁ charges the capacitor. Once charged, holding down SW₂ discharges the capacitor.

- Signal filtering and smoothing:

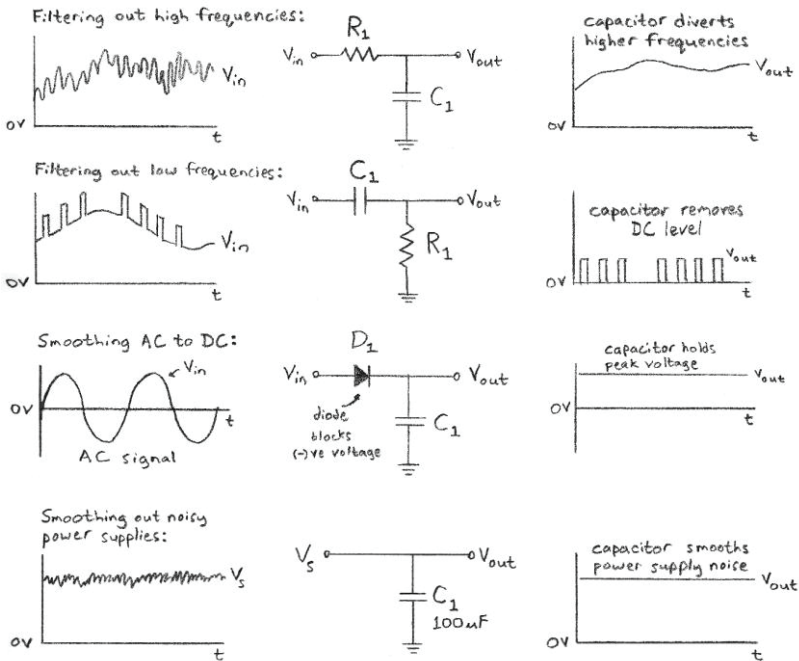


Figure 2-6. Capacitors can remove high frequencies from a signal (e.g. in an RC low-pass filter), low frequencies (e.g. in a CR high-pass filter), rectify an AC signal to DC, and reduce fluctuations in a noisy power supply.

Capacitor Equations

Energy Stored by a Capacitor

$$E_{cap}(J) = \frac{CV^2}{2} = \frac{\text{Farads} \times \text{Volts}^2}{2}$$

Current Across a Capacitor

The current across a capacitor is proportional to the rate of change of voltage (dV_c/dt):

$$I_c = C \times \frac{dV_c}{dt}$$

Voltage Across a Capacitor

The voltage across a capacitor is related to the integral of current with respect to time:

$$V_c = \frac{1}{C} \int I_c dt$$

Charging a Capacitor through a Resistor

A resistor slows down how quickly a capacitor charges by limiting the current (larger resistance = longer charging time).

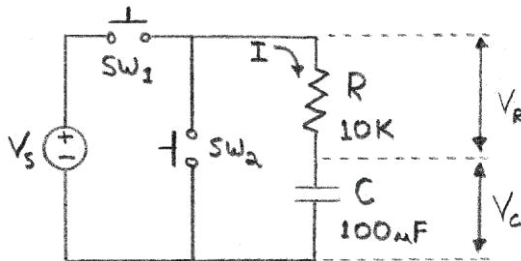


Figure 2-7. A circuit to illustrate charging and discharging a capacitor through a resistor.

When SW_1 in Figure 2-7 is first closed, the following equations describe the current and voltage going through the resistor and capacitor with respect to time:

$$(1) I = \frac{V_S}{R} e^{-t/RC}$$

$$(2) V_R = IR = \frac{V_S}{R} e^{-\left(\frac{t}{RC}\right)} \times R = V_S \times e^{-t/RC}$$

$$(3) V_C = \frac{1}{C} \int I dt = V_S \left(1 - e^{-t/RC}\right)$$

Let $\tau = RC$ (time constant, similar to half-life):

Re-arranging:

$$(4) \frac{t}{\tau} = -\ln\left(\frac{IR}{V_S}\right)$$

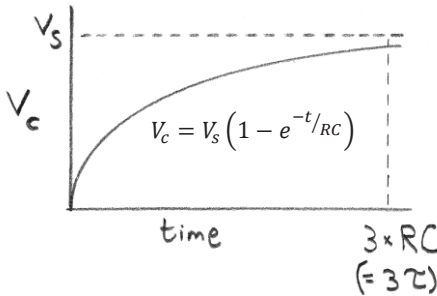
$$(5) \frac{t}{\tau} = -\ln\left(\frac{V_R}{V_S}\right)$$

$$(6) \frac{t}{\tau} = -\ln\left(\frac{V_S - V_C}{V_S}\right)$$

The most important equation here is:

$$(7) V_C = V_S \left(1 - e^{-t/RC}\right) = V_S \left(1 - e^{-t/\tau}\right)$$

So when a capacitor is charging through a resistor (RC configuration), the voltage vs. time curve will look like the curve shown in Figure 2-8.



Units: $R \times C = \tau$

$\Omega \times F = s$

$$\frac{J \cdot s}{C^2} \times \frac{C^2}{J} = s$$

(we can express τ in seconds)

Figure 2-8. It takes about three time constants ($3 \times \tau$) for a capacitor to charge through a resistor.

When the capacitor is completely charged, the current stops flowing through it. The difference in voltage across the capacitor is equal to the voltage source (V_S), and the voltage difference across the resistor is zero.

Example: How long should it take to charge the capacitor in Figure 2-7 ($R=10K, C=100 \mu F$)?

$$\tau = R \times C = 10,000\Omega \times 100 \times 10^{-6}F = 1s$$

$$3\tau = 3 \times 1s = 3s$$

It would take ~ 3 seconds for the capacitor to charge.

Discharging a Capacitor Through a Resistor

When SW₂ is held down in Figure 2-7, the capacitor begins to discharge—the opposite process. The voltage vs. time curve will look like the curve shown in Figure 2-9.

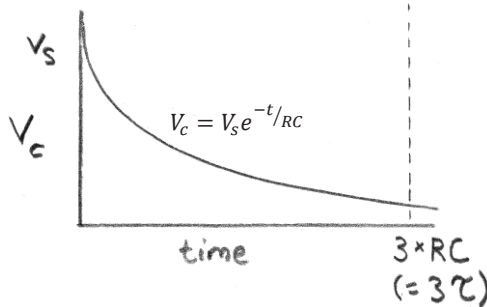
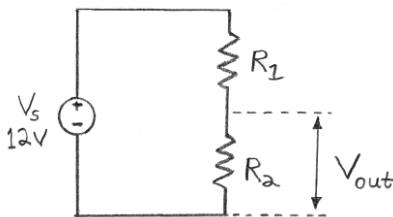


Figure 2-9. It takes about three time constants ($3 \times \tau$) for a capacitor to discharge through a resistor.

Voltage Divider Design: 10% Rule

There was a warning in Section 1 about using a voltage divider to supply power. Although it may not be your first choice, you can still make this idea work, using the 10% Rule. Let's say, for example, that you would like to power a 7V lightbulb, and all you can find is a 12V DC power supply. According to the **voltage divider equation**,



$$V_{out} = V_{in} \left(\frac{R_2}{R_1 + R_2} \right)$$

$$\rightarrow R_2 = \frac{R_1}{\left(\frac{V_{in}}{V_{out}} \right) - 1}$$

$$R_2 = \frac{R_1}{\left(\frac{12V}{7V} \right) - 1} = 1.4R_1$$

Figure 2-10. Solving for R_2 using the voltage divider equation.

So according to the voltage divider equation, *any* combination of two resistors can give $V_{out}=7V$, as long as $R_2 = 1.4 \times R_1$. So which two resistance values should we pick?

- $R_1 = 10 \Omega, R_2 = 14 \Omega ?$
- $R_1 = 1 \text{ k}\Omega, R_2 = 1.4 \text{ k}\Omega ?$
- $R_1 = 2 \text{ M}\Omega, R_2 = 2.8 \text{ M}\Omega ?$

The “answer” depends on how much *current* you need at 7V. The larger the resistance values, the lower the current. You also need to consider that whatever you connect up to V_{out} will have a resistance value as well (R_{load}), which will in effect lower the total resistance below R_1 , because it will be in parallel with R_2 . One strategy is to intentionally pick values of R_1 and R_2 that will result in 90% of the current flowing through the load (called the **load current**, or I_{load}), and 10% of the current flowing through R_2 (called I_{bleed}). This is best understood using an example. Let’s say the 7V light bulb in the example is known to draw 40 mA:

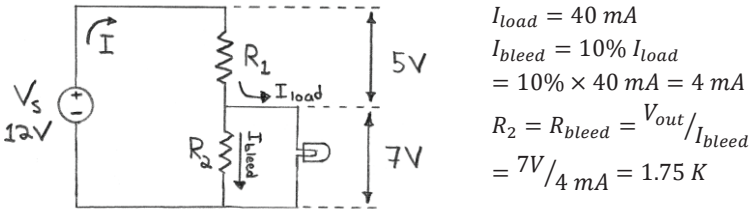


Figure 2-11. Calculating the value of the bleed resistor, using the 10% Rule.

We now know that the bleed resistor, R_2 , should have a resistance of 1.75 k Ω . We can calculate the resistance of the load (the lightbulb):

$$I_{load} = 40 \text{ mA}$$

$$V_{out} = 7V$$

$$R_{load} = \frac{V_{out}}{I_{load}} = \frac{7V}{40 \text{ mA}} = 0.175 \text{ K}$$

We now have a more complete picture of the circuit we’d like, and we can calculate the total resistance below R_1 :

$$R_2 || R_{load} = \left(\frac{R_2 R_{load}}{R_2 + R_{load}} \right) = \left(\frac{1.75 \text{ K} \times 0.175 \text{ K}}{1.75 \text{ K} + 0.175 \text{ K}} \right)$$

$$= 0.1591 \text{ K}$$

Now that we know the total resistance below R_1 , we can calculate the required resistance of R_1 from the requirement $R_2 = 1.4 \times R_1$:

$$R_1 = \frac{R_2}{1.4} = \frac{0.1591 \text{ K}}{1.4} = 0.1136 \text{ K} = 113.6 \Omega$$

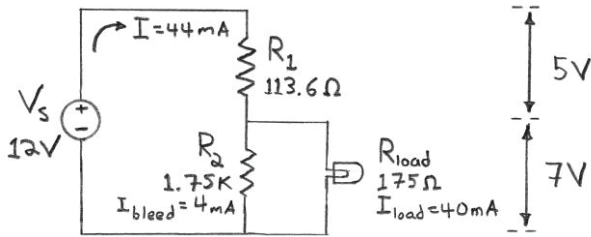


Figure 2-12. Solved circuit using the 10% Rule.

Based on the 10% rule, a value of $R_1=113.6 \Omega$, and $R_2=1.75 \text{ k}\Omega$ should provide enough current to power the bulb (40 mA). Another way of approaching this method is to calculate (or look up) R_{load} , and then multiply it by 10 to obtain a value for R_{bleed} . According to the current divider equation, this will result in 10% of the current flowing through I_{bleed} . Then, calculate R_1 as we did here.

How much power is used by each resistor? Let's check:

$$P = VI$$

$$R_1: 5V \times 0.044 A = 0.22 W \text{ (can use } 1/4 W \text{ resistor or higher)}$$

$$R_2: 7V \times 0.004 A = 0.028 W \text{ (can use } 1/8 W \text{ resistor or higher)}$$

$$R_L: 7V \times 0.040 A = 0.28 W$$

Voltage dividers are ok for low current applications (e.g. LEDs and operational amplifiers). However, they are very inefficient at providing higher current. Have a look at the power across each resistor above: R_1 takes a beating at 0.22W. The resistor literally burns up almost the same amount of power as the load, just to lower the voltage to 7V. For a battery powered system, this wastes energy and shortens battery life. Also, as batteries get used up, the voltage drops considerably before the battery dies—this means the load will get a lower voltage than it needs well before the battery wears out. A voltage divider doesn't regulate (or adjust) the output voltage. There are much better options to supply lower voltage.

Other Options for Delivering Lower Voltage

- **Power adapter:** Find a wall power adapter at the voltage you need (12V, 9V, 7V, and 5V DC power adapters are common).
- **Batteries:** Is your voltage requirement a multiple of 1.5V or 1.2V? (D, C, AA, AAA alkaline batteries: 1.5V, NiMH: 1.2V) Consider running the lower voltage portion of your device using a battery or combination of batteries in series.

- **Split supply:** take a line from the middle battery in a series of batteries, at the voltage level you need (Figure 2-13). This way energy isn't wasted in order to lower voltage.
- **Voltage regulator:** regulates voltage down to a designed value (fixed and variable regulators are available).
- **Buck converter (or step-down converter):** converts higher voltage to lower voltage more efficiently (without burning as much energy).
- **Boost converter (or step-up converter):** converts a lower voltage to a higher voltage, at the cost of a smaller current.

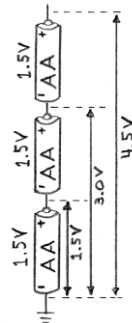


Figure 2-13.
Split supply.

Datasheet Example: LM317 (Variable Linear Voltage Regulator)

The LM317 is a pretty solid general-purpose linear voltage regulator. Depending on how you set it up, it can step down pretty high voltages to something more useable. The TO-220 package of this regulator makes it easy to add a heat sink—something that is required if you are stepping down lots of volts. It isn't very energy efficient compared to a step-down converter, but it *regulates* the output voltage, meaning that it can correct V_{out} if V_{in} fluctuates or changes. As long as the voltage source is above a certain level, the output of the LM317 will be right on target.

A product **datasheet** for the LM317 provides all the information you will need to use this regulator properly. (Texas Instruments Inc 2016a) A datasheet usually has the following sections:

- Overview/description
- Features
- Pin assignments/Pin-out Diagrams
- Applications
- Technical specifications (with maximum power, current, voltage ratings, etc.)
- Example circuits diagrams for common uses
- Required equations for common calculations
- Plenty of graphs (voltage, power, performance, etc.)

From the Texas Instruments LM317 datasheet, we can find out the **pin assignments** (very important if we are going to use it!). Pins are usually labelled from *left to right*, with the front of the device (the face with the part number label) facing you:

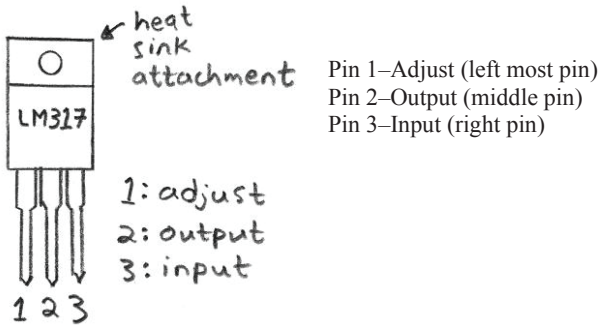


Figure 2-14. LM317 pin-out diagram (left) and pin assignments (right).

Under “Recommended Operating Conditions”, we find the following information:

Table 2-1. Recommended operating conditions for the LM317 voltage regulator (TO-220 package). (Texas Instruments Inc 2016a)

		<i>MIN</i>	<i>MAX</i>	<i>UNIT</i>
V_O	Output voltage	1.25	37	V
$V_I - V_O$	Input-to-output differential voltage	3	40	V
I_O	Output Current		1.5	A
T_J	Operating virtual junction temperature	0	125	°C

This information gives us a few ideas about using the LM317. Firstly, the minimum voltage differential ($V_{in} - V_{out}$) is 3V. This is also called the **dropout voltage**, or **headroom**. We learn the LM317 would not be good at regulating a 9V battery down to 7V, because that is less than a 3V difference. They recommend at least a 3V headroom (so regulating down from 9V to 5V would be ok). Furthermore, this package is good up to 1.5 A. That’s a decent amount of current. The datasheet provides the following circuit diagram:

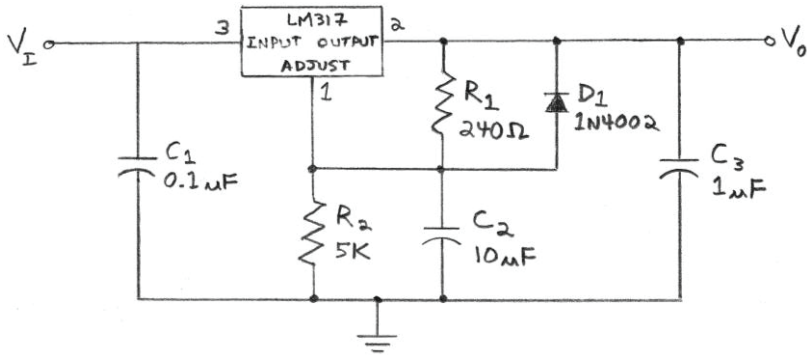


Figure 2-15. LM317 used as an adjustable regulator circuit with improved ripple rejection. (Texas Instruments Inc 2016a)

The capacitors (C_1 , C_2 , and C_3) and diode (D_1) in this diagram are optional to reduce *ripple* (another word for fluctuation) in voltage. The datasheet recommends $R_1=240\ \Omega$. The value of R_2 will determine the output voltage of the regulator, based on the equation:

$$V_{out} = 1.25V \left(1 + \frac{R_2}{R_1} \right)$$

Re-arranging this equation, we can solve for R_2 :

$$R_2 = \frac{R_1(V_{out} - 1.25)}{1.25} = \frac{240\ \Omega \times (V_{out} - 1.25)}{1.25}$$

$$\rightarrow R_2 = 192\ \Omega \times (V_{out} - 1.25)$$

Example: a 6V DC motor driven by a 9V battery would require $R_2=192\ \Omega \times (6V-1.25) = 912\ \Omega$

In another configuration, the LM317 can be used as a **current regulator** (delivering a constant *current*, rather than a constant *voltage*). It can therefore act as a **current limiter**, and to some extent, a **current source**. This can be handy in charging a battery, or powering high-powered LEDs, which require a constant current to run (rather than a fixed voltage). Letting your power source limit your current can drain the battery faster, and also burn out devices that require a limited current. The current-limiter configuration in the LM317 datasheet is shown in Figure 2-16.

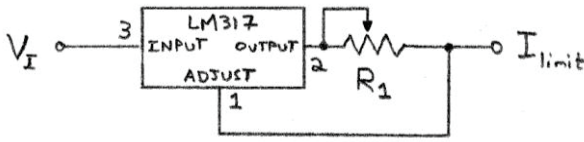


Figure 2-16. LM317 used as a current limiter. (Texas Instruments Inc 2016a)

The circuit diagram symbol for R_1 is one way of representing a variable resistor (also called a *potentiometer*). This circuit works using the formula:

$$I_{out} = \frac{1.25V}{R_1}$$

Re-arranging to solve R_1 :

$$R_1 = \frac{1.25V}{I_{out}}$$

Example: A 50 mA high intensity LED strip would require $R_1 = 1.25V / 0.050A = 25\Omega$.

A circuit diagram symbol for a current source is a circle with an arrow inside, pointing towards the direction of (conventional) current (Figure 2-17).

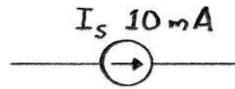


Figure 2-17. Circuit diagram symbol for a 10 mA current source.

How Hot Will My Chip Get? Heat Dissipation Calculations

When you are regulating a supply voltage, one of the biggest concerns is heat, especially when high voltages or high currents are expected. A heat sink can be added to help prevent overheating. It is usually a chunk of metal that helps absorb, conduct, and dissipate heat. Larger heat sinks can even have fans attached to them. How can we tell if we need a heat sink on a voltage regulator, or any component, for that matter?

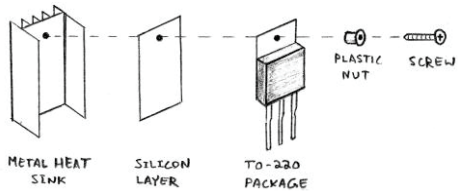


Figure 2-18. Heat sink for TO-220 package. The silicon layer and plastic nut keep the body of the package insulated from the heat sink, to reduce the chances of a short circuit.

Answer 1: Calculate the power dissipated/used by the component, using the power equations ($P=VI$, or $P=I^2R$). If the power is less than 0.25W, most components can dissipate that without needing a heat sink.

Answer 2: To better answer this question, you need to calculate the power dissipation of the regulator:

$$P_D = P_{in} - P_{out} = (V_{in} - V_{out}) \times I_{out} \quad (\text{assume } I_{in} = I_{out})$$

The datasheet for the LM317 lists a “junction to ambient thermal resistance” ($R_{\theta_{JA}}$, or sometimes labeled θ_{JA}) of about 50 °C/W. The formula for this parameter is:

$$\theta_{JA} = \frac{T_j - T_A}{P_D}$$

where T_j is the junction temperature rating, T_A is the ambient temperature, and P_D is the maximum power dissipation. You can use this formula in a number of ways. The most straightforward way is to calculate how hot your component will get.

Example: Let’s say you have an LM317 voltage regulator that is pushing out 1 amp, with an input voltage of 12V and an output voltage of 7V. How hot will it get? First, we calculate the power dissipated by the regulator, P_D :

$$P_D = (V_{in} - V_{out}) \times I_{out} = (12V - 7V) \times 1A = 5W$$

Now we use the junction-to-ambient thermal resistance equation to calculate the temperature increase:

$$\theta_{JA} = \frac{T_j - T_A}{P_D}$$

$$(T_j - T_A) = \theta_{JA} \times P_D = 50 \text{ } ^\circ\text{C}/\text{W} \times 5W = 250 \text{ } ^\circ\text{C}$$

This means that the regulator will heat up 250 °C *above* ambient temperature. We read from the datasheet that the absolute maximum temperature for the regulator is 150 °C, so for this application, a heat sink would definitely be required. Sometimes datasheets will list $T_{j(\max)}$, $\theta_{JA(\max)}$, or $P_{D(\max)}$, which can all be used accordingly as limits for the above calculated parameters.

Thévenin’s Theorem

Thévenin’s Theorem (after Léon Charles Thévenin) is a way of analyzing and *simplifying* a circuit diagram from the perspective of a single component. The theorem states that any network of resistors, capacitors, and sources (current sources, voltage sources) can always be re-written as a single voltage source and resistor. There are a few caveats to the theorem, but it is a useful way of thinking about your circuit, without requiring overly complex equations. (Scherz and Monk 2016)

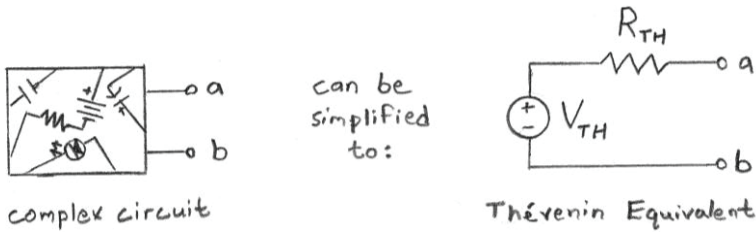


Figure 2-19. Thévenin's Theorem in a nutshell. Any complicated network of resistors, capacitors, and sources from the perspective of a single component (at connections *a* and *b*) may be represented as a voltage source in series with a resistor.

- Thévenin's Theorem lets you simplify complicated parts of a circuit to solve it more easily.
- Non-linear components (e.g. transistors) are excluded from Thévenin's Theorem.
- The theorem lets you model an entire complicated system with only two components: a resistor in series with a voltage source. This can make analysis much easier.

Before we begin our discussion on how Thévenin's Theorem works, we need a few definitions to help us through the calculations. These definitions help us out with the direction that we are calculating a voltage increase or decrease across a component, or any two points in a circuit labeled *a* and *b*:

$$V_{ab} = V_a - V_b$$

$$V_{ba} = V_b - V_a$$

$$V_{ab} = -V_{ba}$$

Thévenin's Theorem by Measurement (Using a Multimeter)

The easiest way to apply Thévenin's Theorem is to build your circuit, and then use a multimeter to determine the Thévenin Equivalent voltage and resistance. Let's apply Thévenin's Theorem to the example circuit in Figure 2-20 to see how it works.

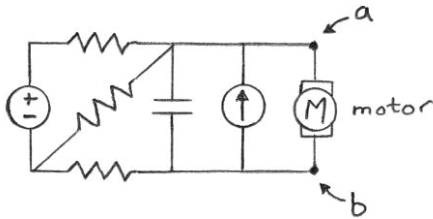


Figure 2-20. Example circuit for Thévenin’s Theorem. First step: identify two terminals of interest, and label them *a* and *b*.

- 1) First, we need to identify the two terminals where Thévenin’s Theorem will be applied. These terminals will usually be the (+) and (–) terminals to a component of interest (e.g. an important resistive load, motor, light, etc.). In the above complicated circuit, we identify the motor (the *load*, or what the circuit is driving) as the main component of interest, so we mark terminals *a* and *b*, above and below the load.
- 2) Remove the load from the circuit, and with the power on, measure the open circuit voltage across *a* and *b*, using a voltmeter ($V_{ab}=V_{TH}$).

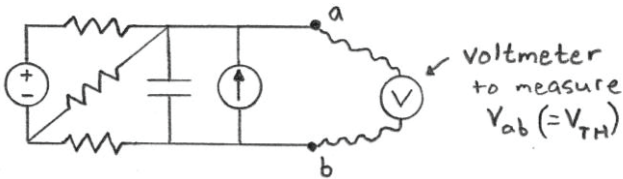


Figure 2-21. Remove the load from the example, and measure V_{ab} .

- 3) Measure the current between *a* and *b*, to find the short-circuit current (i_{sc}):

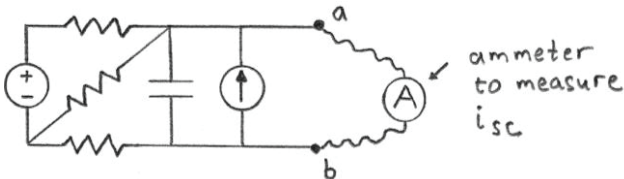


Figure 2-22. Measure i_{sc} across terminals *a* and *b*.

- 4) Calculate the Thévenin Resistance (R_{TH}):

$$R_{TH} = \frac{V_{TH}}{i_{sc}} \quad (\text{Ohm's Law})$$

- 5) We can now simplify this circuit to the Thévenin Equivalent Circuit:

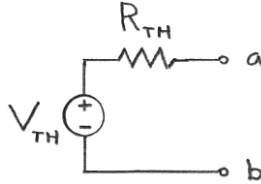


Figure 2-23. Thévenin Equivalent Circuit.

- 6) The current running through the motor (the load current, I_L) would be:

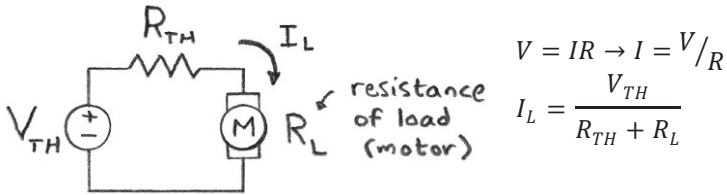


Figure 2-24. Thévenin Equivalent Circuit with load replaced.

Alternate measurement approach (replacing Steps 3 and 4):

You can also measure R_{TH} directly, by removing *all voltage sources* and shorting the gap they leave in the circuit, *removing (opening) all current sources*,² then measuring the resistance across terminals *a* and *b* directly with an ohmmeter:

² Only the independent voltage and current sources (voltage/current sources that do not rely on other voltage/current sources for their values).

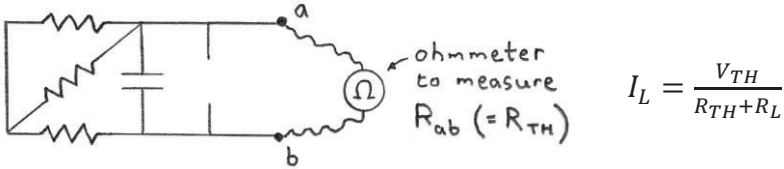


Figure 2-25. Short all voltage sources, remove all current sources, then measure R_{TH} .

The Norton Equivalent Circuit

A variant of the Thévenin Equivalent Circuit is the **Norton Equivalent Circuit** (Figure 2-26).

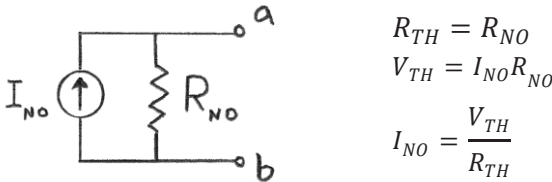


Figure 2-26. Norton Equivalent Circuit.

The Norton Equivalent Circuit is a very similar idea, only the simplified circuit is a resistor in parallel with a current source. Once the Thévenin Equivalent circuit is calculated, it can be converted to a Norton Equivalent Circuit using the formulas in Figure 2-26.

Mesh Current Method

You don't need to physically build your circuit and use a multimeter to solve for a Thévenin Equivalent circuit. We can perform the same analysis theoretically, in order to answer potentially difficult questions on what the voltage or current will be across a component of interest. We will solve the following circuit diagram two ways: first, using a conventional, longer way (using the **Mesh Current Method**) and a quicker way (using **Thévenin's Theorem**), to illustrate how it works.

Let's say we are powering a light bulb (represented as R_L) in the following circuit diagram:

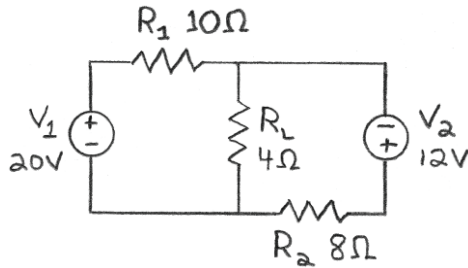


Figure 2-27. Example for the Mesh Current Method.

There are two DC power supplies (batteries) in this diagram, and three resistors (our bulb acting as R_L , for the resistive load of the bulb)—an intentionally complicated circuit diagram.

For the slower way (without Thévenin's Theorem) we will use the **Mesh Current Method** to calculate the voltage and current across R_L . This method gets its name from focusing on where currents “mesh” together, like motor gears coming together. Essentially, the Mesh Current Method systematically applies KVL to every *inner* loop of a circuit, and KCL at every junction. The resultant series of equations is solved simultaneously. Here is how it works:

- 1) A **junction** is a point where three (or more) circuit paths meet. Label each junction, and draw current arrows in all *inside* loops. In the above circuit diagram there are two junctions, **A** and **B**:

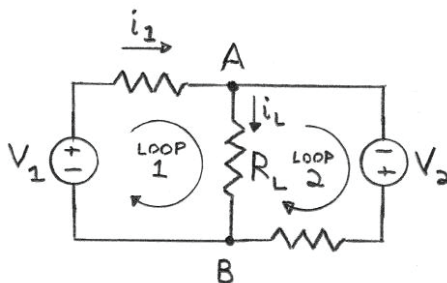


Figure 2-28. Number each inside loop, draw current arrows, and label each junction.

- 2) Let's start at junction **A**. **KCL** says: Σ current entering = Σ current leaving junction **A**:

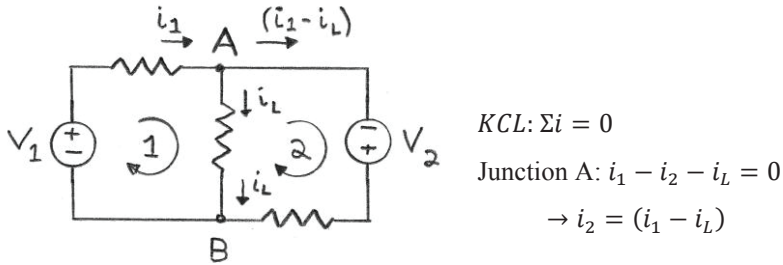


Figure 2-29. Perform KCL on junction A.

- 3) Now apply KCL to junction **B**. Since the current through a resistor is constant, the current flowing into **B** from **A** is equal to i_L . Also, we know that the current going towards V_1 from junction **B** is equal to i_1 , because it's on the same line (or branch). Similarly, the current flowing from V_2 to junction **B** is equal to $(i_1 - i_L)$:

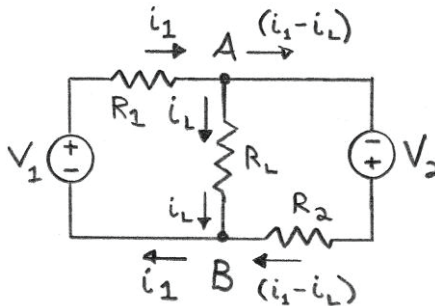


Figure 2-30. Perform KCL on junction B.

- 4) Now apply KVL to loop 1, then loop 2:

Using KVL on Loop 1:Voltage across R_1 :

$$V_{R1} = i_1 R_1 \text{ (Ohm's Law)}$$

Voltage across R_L :

$$V_L = i_L R_L \text{ (Ohm's Law)}$$

$$\mathbf{KVL: \Sigma V = 0}$$

$$V_1 - V_{R1} - V_L = 0$$

$$V_1 - i_1 R_1 - i_L R_L = 0$$

$$20V - i_1(10\Omega) - i_L(4\Omega) = 0$$

$$\mathbf{(1) 20 = 10i_1 + 4i_L}$$

Using KVL on Loop 2:

$$\mathbf{KVL: \Sigma V = 0}$$

$$V_2 - V_{R2} \oplus V_L = 0$$

Positive because the current is going the wrong way (against the direction of the loop)

$$V_2 - (i_1 - i_L)R_2 + i_L R_L = 0$$

$$12V - (i_1 - i_L)8\Omega + i_L 4\Omega = 0$$

$$12 - 8i_1 + 8i_L + 4i_L = 0$$

$$\mathbf{(2) 12 = 8i_1 - 12i_L}$$

Now there are two equations, and two unknowns. Solving:

$$(1): 20 = 10i_1 + 4i_L$$

$$10i_1 = 20 - 4i_L$$

$$i_1 = \frac{20 - 4i_L}{10}$$

$$(1) \rightarrow (2): 12 = 8\left(\frac{20 - 4i_L}{10}\right) - 12i_L$$

$$120 = 8(20 - 4i_L) - 120i_L$$

$$120 = 160 - 32i_L - 120i_L$$

$$152i_L = 40$$

$$i_L = \frac{40}{152} = 0.263 \text{ A}$$

Finally, solving for the voltage across the load:

$$V_L = i_L R_L$$

$$V_L = 0.263A \times 4\Omega = 1.052V$$

We have now solved for the current and voltage across the load.

Thévenin's Theorem Method (Theoretical)

Solving this system using Thévenin's Theorem (working it out theoretically, without measurement) is easier than solving simultaneous equations.

- 1) First, identify the terminals of interest, and mark them as points **a** and **b**. We would like to know the current across the load, so we mark the points on either side of it:

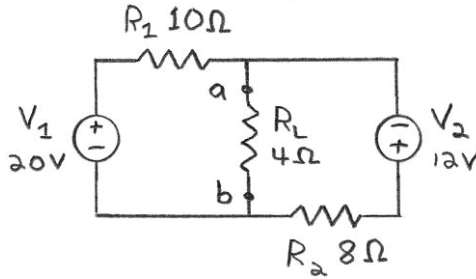


Figure 2-31. Identify points *a* and *b* around the load.

- 2) Remove the load (R_L), then calculate the voltage across the open terminals. At this point you can *remove any capacitors* and *short any inductors* present in your circuit diagram. For this example, when the load is removed, we are left with a single loop. We can use KVL to solve for the difference in voltage between points *a* and *b*.

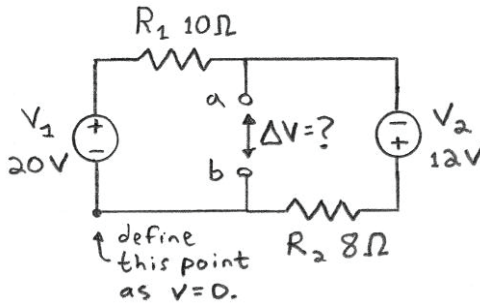


Figure 2-32. Remove the load, then calculate the voltage difference between points *a* and *b*.

Using KVL on the entire outer loop (easier):

Note: Recall that the current in a single loop is constant throughout the entire loop.

KVL: $\Sigma V = 0$

$$V_1 - V_{R1} + V_2 - V_{R2} = 0$$

$$V_1 - IR_1 + V_2 - IR_2 = 0$$

$$V_1 + V_2 = I(R_1 + R_2)$$

$$I = \frac{V_1 + V_2}{R_1 + R_2} = \frac{20V + 12V}{10\Omega + 8\Omega}$$

$$I = 1.7778A$$

We can now calculate $V_{ab} = V_a - V_b$, either by following the current clockwise from **b** to **a** on the **left** side of the circuit:

$$V_a = V_b + V_1(-)(I \times R_1)$$

$$V_a = V_b + 20V - (1.7778A \times 10\Omega)$$

$$V_a = V_b + 2.222V$$

$$\rightarrow V_{ab} = V_a - V_b = 2.222V = V_{TH}$$

Negative because we are defining current as going clockwise, and expect the voltage to **drop** as conventional current flows through R_1 .

OR by following the current clockwise from **a** to **b** on the **right** side of the circuit:

$$V_b = V_a + V_2 - (I \times R_2)$$

$$V_b = V_a + 12V - (1.7778A \times 8\Omega) = V_a - 2.222V$$

$$\rightarrow V_{ab} = V_a - V_b = 2.222V = V_{TH} \text{ (same answer)}$$

It doesn't matter which path you take from **a** to **b**. The voltage change should be the same, according to KVL.

- 3) Short all voltage sources (replace them with a line), and remove all current sources (remove them and leave their connection open), and then calculate the total resistance across the open terminals **a** and **b**. Note that the following four circuits are drawn differently, but are electrically identical:

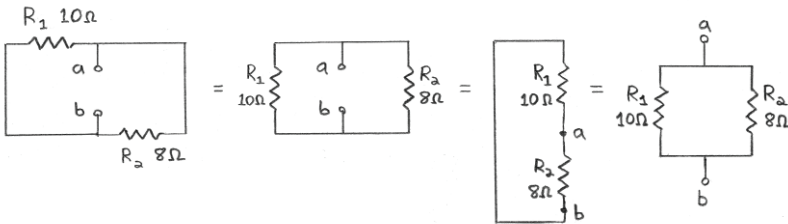
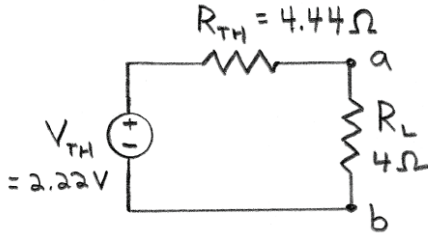


Figure 2-33. Circuit diagrams can be deceptive. These four circuits all depict two resistors in parallel, from points **a** to **b**.

$$R_{ab} = R_{TH} = R_1 || R_2 = \frac{R_1 R_2}{R_1 + R_2} = \frac{10\Omega \times 8\Omega}{10\Omega + 8\Omega} = \frac{80}{18} = 4.44\Omega$$

- 4) We can now write the Thévenin Equivalent circuit from the perspective of the load, and then solve for the current and voltage

across the load, using Ohm's Law and the voltage divider equation, respectively:



$$I_L = \frac{V_{TH}}{R_{TH} + R_L}$$

$$I_L = \frac{2.22V}{4.44\Omega + 4\Omega} = 0.263A$$

Voltage divider equation:

$$V_L = V_{TH} \left(\frac{R_L}{R_{TH} + R_L} \right)$$

$$V_L = 2.22V \left(\frac{4\Omega}{4.44\Omega + 4\Omega} \right)$$

$$V_L = 1.052V$$

Figure 2-34. Thévenin Equivalent circuit, with load replaced.

The load connected to the Thévenin Equivalent circuit will “experience” the same voltage drop and current as if it were connected to the original circuit. Thévenin's Theorem provided the exact same answer as the Mesh Current method, without having to solve simultaneous equations.

Integrated Circuits (ICs)

Certain circuits become quite popular because of their usefulness, and it becomes tedious and annoying to have to rebuild the same circuit if it is of general interest. Here is where integrated circuits (ICs) step in. A circuit is miniaturized into different packages, depending on the application, and marketed by an electronics manufacturer with a datasheet.

PDIP/DIP

For prototyping, the clear winner is the **DIP** package, which we will be using most often in this course. DIP (or sometimes PDIP) stands for (Plastic) Dual Inline Package, and was invented by Bryant (Buck) Rogers in 1964, while at Fairchild Semiconductor. (Dummer 2013) Buck's invention revolutionized electronics. DIP chips can be soldered, and fit nicely into breadboards and

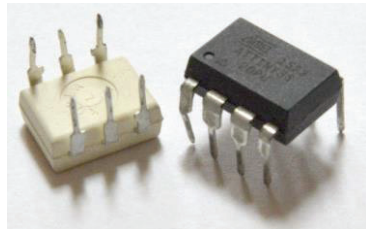


Figure 2-35. DIP chips are great for prototyping with breadboards.

prototyping PCB boards for testing during the design stage.

Two DIP chips are shown in Figure 2-35: the MOC3010 (left), and the ATTiny85 (right). The MOC3010 is an infrared switch used to isolate mains (AC) power from logic. The ATTiny85 is a microprocessor with 8 kb of programmable memory, that you will really want to get to know next if you are enamored of the Arduino Uno.

Another epic DIP chip of historical interest is the 555 timer, invented by Hans Camenzind in 1971, while he was working for Signetics. (Camenzind 1997, 80-85) In monostable mode, it produces a single pulse, the timing of which you can control based on what you hook up to it—so you can use that pulse to time an event. It has two other modes: *astable* (free-running mode-oscillator), and *bistable* (flip-flop mode).

Numbering of the chip pins usually starts at a recessed dot, located on the bottom left hand pin of the chip, and goes counter-clockwise (Figure 2-36). It is critical to get pin assignments right, or you can accidentally fry a chip. A half-moon carved on one side of a chip can also provide a clue for which side of the chip is right-side up, as well as the orientation of the part number label, if you can read it.

You can find the pin assignments, maximum ratings, and thermal characteristics for any chip by looking up the product datasheet. The Texas Instruments product datasheet for the 555 timer provides the information in Table 2-2.

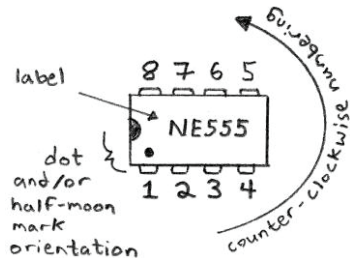


Figure 2-36. Pin numbering for DIP chips runs counterclockwise, starting from the bottom left pin.

Table 2-2. Some electrical characteristics of the NE555 chip. (Texas Instruments Inc 2014b)

Supply Voltage (V_{cc})	4.5 - 16V
Supply Current ($V_{cc} = +5V$)	2 - 6 mA
Supply Current ($V_{cc} = +15V$)	9 - 15 mA
Output Current (max):	± 200 mA
Operating free-air temperature:	0 - 70 °C
θ_{JA} (Thermal Impedance):	90 °C/W
T_J (Maximum operating junction temperature)	150 °C

Surface Mount Technology

Two other common types of ICs are **PLCC** (Plastic Lead-Chip Carrier), and **SOIC** (Small Outline Integrated Circuit). You will see these types of chips on modules throughout the course. Some of them are so small, you need a magnifying glass to read the label. However, they are cumbersome to prototype with, as they do not easily fit into the holes of a breadboard. They are much smaller than DIP chips. It's important to understand the difference between DIP, PLCC, and SOIC when you are ordering chips, because you might get an unpleasant surprise when they are delivered, and aren't compatible with a breadboard.

Logic Circuits

The advent of microcontrollers really did away with the necessity for logic chips. However, an understanding of the fundamentals really helps to introduce concepts in electronics such as input pins, output pins, and gates, as well as more advanced concepts in programming, such as **if** statements, logical expressions, and even neural networks. We take a step back then to look at examples of three fundamental logic gates: **AND**, **OR**, and **NOT**. In all of these examples, a “true” is synonymous with “1”, or in other words, whatever voltage your logic is running at (V_{cc} , usually 5V, or 3.3V). A “false” is synonymous with “0”, or in other words, the ground, or reference state (0V). If your **logic level** is +5V, we therefore interpret +5V as “1”, and GND as “0”. However, there is usually some room for error, or headroom. Keep this in mind when looking at the following diagrams.

AND Gate: (e.g. 74HC08)

The logic symbol for a 2-input AND gate is:

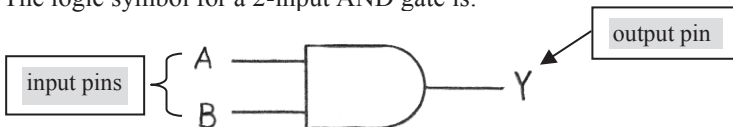


Figure 2-37. Logic symbol for an AND gate.

The 74HC08 is a DIP (dual inline package) chip with four AND gates built in (quadruple gate, 2-input). A pin-out diagram for the DIP version of this chip is provided in Figure 2-38, and was adapted from the Texas Instruments datasheet for this product. (Texas Instruments Inc 2016d)

We will be investigating how an AND gate works. The circuit diagram in Figure 2-39 will test the AND gate functionality. The 10K pull-down resistors keep the input pins from “floating” to an arbitrary value when they aren’t connected (something annoying that happens with most input pins). We will talk about how a pull-down resistor works later. However, what it means in this context is that if we close a dip switch (or turn it on), the state of the pin it is connected to will be HIGH (or +5V, provided our logic level is 5V). If we open a dip switch (or turn it off), the state of the pin it is connected to will be LOW (or 0V), and thanks to the pull-down resistor, it will stay LOW, rather than float randomly because it is disconnected.

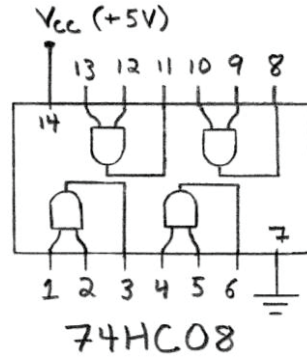


Figure 2-38. Pinout diagram for the 74HC08 AND chip.

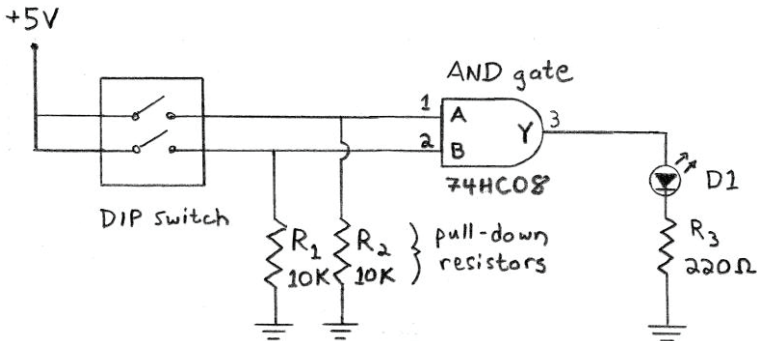


Figure 2-39. Circuit diagram to test out the functionality of an AND gate. Pull-down resistors protect against floating pin states when the DIP switches are open.

What happens to the LED when the DIP switches for A and B are switched on or off? An AND gate will “close” (in other words, $Y=+5V$) when both inputs to the gate are HIGH at the same time (+5V). Otherwise, the gate will “open” (in other words, $Y=0V$). There are four combinations to think about, so we can organize all of the possibilities of the states for A, B, and the result Y in a **logic table** (also called **truth table**), which can be expressed in many ways, all meaning the same thing:

Table 2-3. Logic tables for a 2-input AND gate. “0V” is the same as “ground”, and this table assumes that logic level for your circuit is +5V.

In terms of voltage:				In terms of state:				In terms of logic:			
A	B	Y	LED	A	B	Y	LED	A	B	Y	LED
0V	0V	0V	OFF	LOW	LOW	LOW	OFF	false	false	false	OFF
0V	+5V	0V	OFF	LOW	HIGH	LOW	OFF	false	true	false	OFF
+5V	0V	0V	OFF	HIGH	LOW	LOW	OFF	true	false	false	OFF
+5V	+5V	+5V	ON	HIGH	HIGH	HIGH	ON	true	true	true	ON

To simplify our table, we use a **0** to represent LOW, and a **1** to represent HIGH (Table 2-4). This representation is independent of logic level. If we were powering a chip with $V_{cc}=+3.3V$, then +3.3V would mean “1”. This convention is preserved throughout programming, logic chips, microprocessors, and essentially any chip with a digital pin. This idea will become important later, when we are programming. We can then re-write the logic table with zeroes and ones, as presented in Table 2-4.

Table 2-4. Two-input AND logic table.

A	B	Y=AND(A,B)
0	0	0
0	1	0
1	0	0
1	1	1

The Venn diagram in Figure 2-40 is a graphical representation of the logic table. The numbers in the diagram (X,X) represent the states of A and B, respectively. The state of A is **1** inside the left circle, and **0** outside the left circle. Likewise, the state of B is **1** inside the right circle, and **0** outside the right circle. The output variable Y is represented by shading. The state of Y is **1** where the diagram is shaded, and **0** elsewhere. For an AND gate, the diagram is shaded (or Y=1) only where A and B intersect (where their values are both 1). In set theory, we could write that the solution to an AND gate is equal to $A \cap B$, meaning “A intersect B”. Set theory is immensely useful in computer programming. (Farzan 2018)

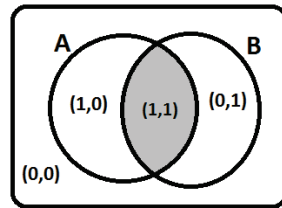


Figure 2-40. Venn diagram for a 2-input AND gate.

OR Gate: (e.g. 74HC32)

Figure 2-41 (top) shows a logic symbol for a 2-input OR gate. (Nexperia 2015b) OR gates can also be drawn with a pointed output terminal, although the output end of the symbol in this figure is semi-circular.

The 74HC32 is a DIP chip with four OR gates built in (quadruple gate, 2-input) that would be compatible with the logic level of the Arduino Uno (+5V). Figure 2-41 (bottom) shows a pin-out for the 74HC32. The location of the input and output pins in this chip are the same as the 74HC08 AND gate. We will also be investigating how an OR gate works. Figure 2-42 shows a circuit diagram to test the OR gate functionality.

An OR gate functions similarly to an AND gate, except that the gate will be closed if *either* A or B is HIGH. The logic table for a 2-input OR gate is provided in Table 2-5. The Venn diagram is depicted in Figure 2-43. Both A and B circles are shaded, so we can write $A \cup B$ to mean “A union B” as the solution to an OR gate.

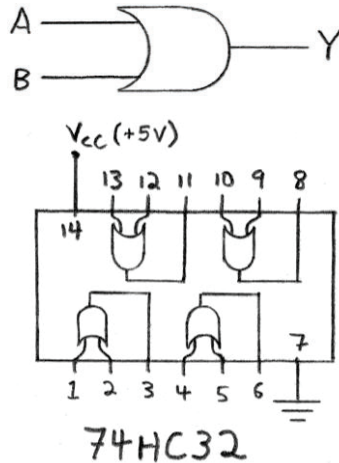


Figure 2-41. Logic symbol for a 2-input OR gate (top). Pinout diagram for the 74HC32 OR chip (bottom).

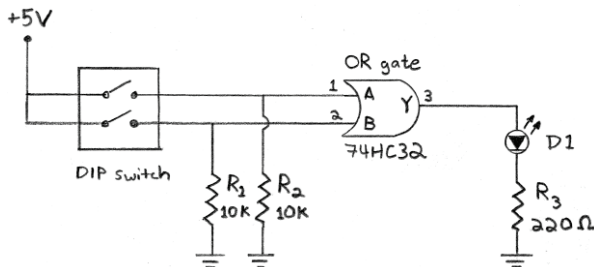


Figure 2-42. Circuit diagram to test out the functionality of an OR gate.

Table 2-5. Two-input OR logic table.

A	B	Y=OR(A,B)
0	0	0
0	1	1
1	0	1
1	1	1

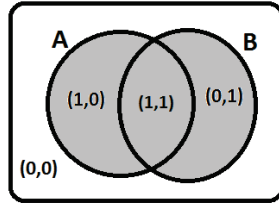
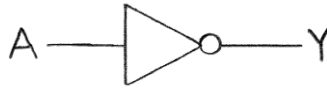


Figure 2-43. Venn diagram for a 2-input OR gate.

NOT Gate: (e.g. 74HC04)

The NOT gate is a single-input *inverter*, called so because it inverts whatever the input is. The output is the opposite of the input. If the input is HIGH, the output is LOW, and vice-versa. Figure 2-44 (top) shows a logic symbol for a single-input NOT gate.



The 74HC04 is a DIP chip with six NOT gates built in (hex gate, single input). Figure 2-44 (bottom) shows a pin-out of the DIP version of this chip. (Nexperia 2015a)

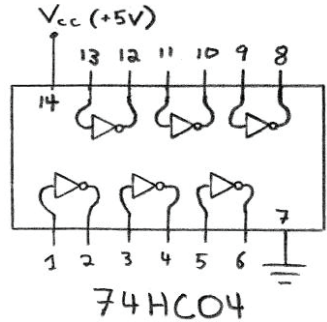


Figure 2-44. Logic symbol for a NOT gate (top). Pinout diagram for the 74HC04 NOT chip (bottom).

We can test the NOT gate using the circuit in Figure 2-45 (only one DIP switch is needed in this circuit).

The logic table and Venn diagram for a NOT gate is presented in Table 2-6, and Figure 2-46 respectively. We could write the solution of a NOT gate is A^c (the complement of A). Although simple, the NOT gate can be combined with other logic gates to form higher order logic gates.

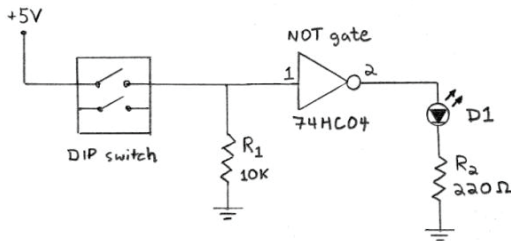


Figure 2-45. Schematic to test out the functionality of a NOT gate.

Table 2-6. NOT logic table.

A	Y=NOT(A)
0	1
1	0

Combining Logic Circuits

You can get into quite complicated logic schemes by combining the basic gates: AND, OR, and NOT. You can also find AND and OR gates with more than two inputs (or construct them with multiple two-input gates). Nonetheless, the building blocks of logic are AND, OR, and NOT. The following higher order gates can be thought of as combinations of these gates.

NAND Gate (AND + NOT)

A common logic gate is the **NAND** gate, which is short for **NOT AND**. The solution of this gate is $(A \cap B)^c$.

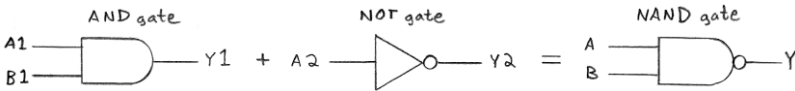


Figure 2-47. An AND gate combined with a NOT gate is equivalent to a NAND gate.

Table 2-7. Two-input NAND logic table.

A1	B1	Y1=AND(A1,B1)	Y2=NOT(Y1)
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

NOR Gate (OR + NOT)

Another common logic gate is the **NOR** gate, which is short for **NOT OR**. The solution of this gate is $(A \cup B)^c$.

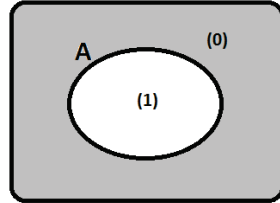


Figure 2-46. Venn diagram for a NOT gate.

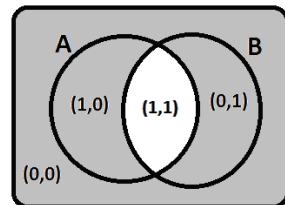


Figure 2-48. Venn diagram for a 2-input NAND gate.

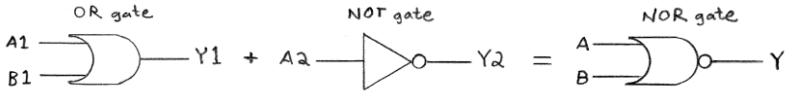


Figure 2-49. An OR gate combined with a NOT gate is equivalent to a NOR gate.

Table 2-8. Two-input NOR logic table.

A1	B1	Y1=OR(A1,B1)	Y2=NOT(Y1)
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

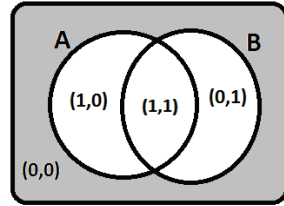


Figure 2-50. Venn diagram for a 2-input NOR gate.

XOR Gate (2 x ANDs, 1 x OR, 1 x NOT)

The **XOR** gate is a bit more complicated. It is short for “exclusive OR gate”, pronounced “ex-or”. It can be constructed in various ways, one of which is shown in Figure 2-51. In set theory, the solution can be written as $(A \cup B) \cap (A \cap B)^c$, or more simply by using the notation $A \oplus B$.

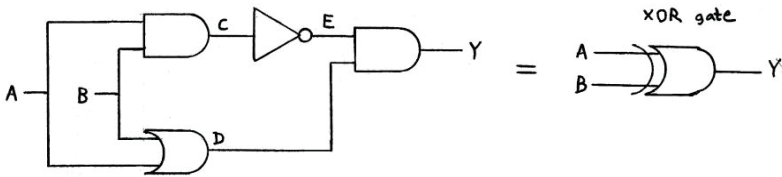


Figure 2-51. XOR gate, meaning “exclusive OR”.

Table 2-9. Two-input XOR logic table.

A	B	C=AND(A,B)	D=OR(A,B)	E=NOT(C)	Y=AND(D,E)
0	0	0	0	1	0
0	1	0	1	1	1
1	0	0	1	1	1
1	1	1	1	0	0

You can see by the Venn diagram in Figure 2-52 that XOR means *either A or B, but not both at the same time*. If one input is HIGH, but not both, then Vout is HIGH.

We will revisit XOR logic, as it is used to invert the state of a pin without disturbing the existing states of other pins (see *Bitwise XOR* (^) in Section 10).

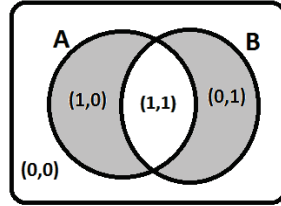


Figure 2-52. Venn diagram for a 2-input XOR gate.

XNOR Gate (1 x XOR, 1 x NOT)

The *XNOR* gate is an XOR gate with an inverter in front of it. The solution of this gate is $(A \oplus B)^c$.

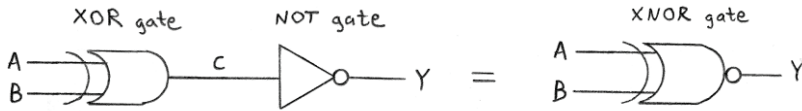


Figure 2-53. XNOR gate, meaning “NOT exclusive OR”, or “exclusive NOR”.

Table 2-10. Two-input XNOR logic table.

A	B	C=XOR(A,B)	Y=NOT(C)
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

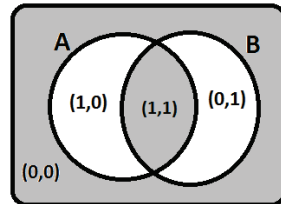


Figure 2-54. Venn diagram for a 2-input XNOR gate.

An XNOR Gate returns HIGH if both gate inputs are HIGH, *or* if neither are HIGH.

These gates are fundamental in logic. You can use them to create higher order logical circuits that do what you want them to, whether the logic is expressed with hardware using electronic chips like the 74HC series, or dealt with in the logical commands of your sketches and programs.

Example: 1. a) How would you express “A and not B” using logic gates?

Answer: First, have a look at the language (which can be ambiguous!). What is implied here is $AND(A, NOT(B))$, or in set theory, $A \cap B^c$. Looking at this syntax, we need an AND gate, and a NOT gate. B is inverted before the AND operation, so the diagram would look like this:

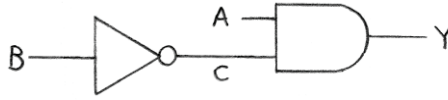


Figure 2-55. Symbolic logical representation of Example 1(a).

b) Write the logic table and Venn diagram for Example 1(a).

Table 2-11. Logic table for Example 1(a).

A	B	C=NOT(B)	Y=AND(A,C)
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	0

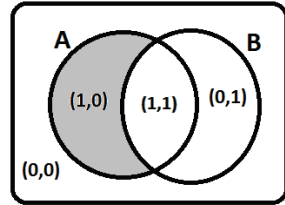


Figure 2-56. Venn diagram for Example 1(a).

Example: 2. a) How would you write “A or not B” using logic gates?

Answer: This can be written as $OR(A, NOT(B))$, or in set theory, $A \cup B^c$. It is the same as Example 1 (a), only an OR gate replaces the AND gate:

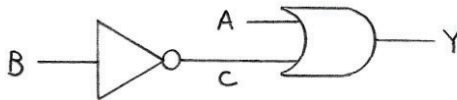


Figure 2-57. Symbolic logical representation of Example 2(a).

b) Write the logic table and Venn diagram for Example 2(a).

Table 2-12. Logic table for Example 2(a).

A	B	C=NOT(B)	Y=OR(A,C)
0	0	1	1
0	1	0	0
1	0	1	1
1	1	0	1

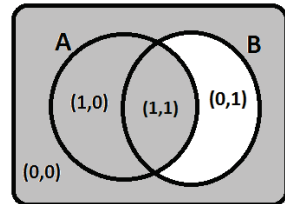


Figure 2-58. Venn diagram for Example 2(a).

Activity 2-1: Capacitor Charging and Discharging

Goal: In this activity, we will be examining the time constant ($\tau=RC$) of a charge/discharge circuit.

Materials:

- 1 x Digital Multimeter
- 1 x Breadboard
- 9V Battery + Snap-on Connector
- 1 x 150 μF Electrolytic Capacitor (Note: longer wire is +)
- 1 x 10 μF Electrolytic Capacitor
- 1 x 10K Resistor
- 1 x 1K Resistor
- 1 x LED (Note: longer wire is +)
- 2 x Momentary Switches
- 2 to 3 x M/M Jumpers
- 1 Stopwatch

Procedure: Assemble the following circuit.

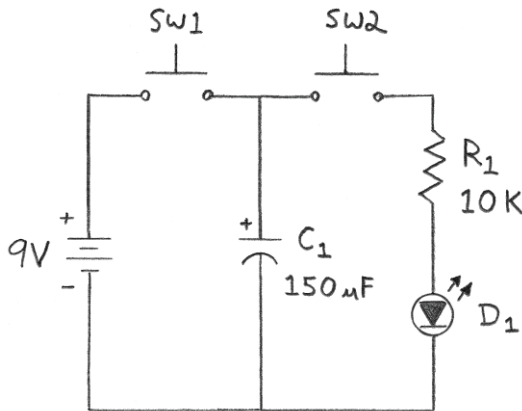


Figure 2-59. Circuit diagram for Activity 2-1.

- a) Hold down SW_1 to charge capacitor, *then* hold down SW_2 to discharge capacitor.
- b) What is the theoretical charging/discharging time for this circuit?
- c) Try testing the following combinations of R_1 and C_1 , completing the following table for each RC pair.
- d) Did this circuit behave as expected? Why might the voltage decay time diverge from theoretical values? **Hint:** the LED also has a resistance.

Table 2-13. Experimental results from Activity 2-1.

R_1	C_1	$\tau(=R_1C_1)$ (s)	$3 \times \tau$ (s)	$t_{discharge}$ (s) (use a stopwatch)
10 k Ω	150 μ F			
1 k Ω	150 μ F			
10 k Ω	10 μ F			
1 k Ω	10 μ F			

Activity 2-2: LM317 Voltage Regulator

Goal: Set up an LM317 voltage regulator circuit to output 5V for V_{out} , from a 9V battery. Try to use as small a space on the breadboard as possible.

Materials:

- 1 x Digital Multimeter
- 1 x Breadboard
- 9V Battery + Snap-on Terminal
- 1 x 0.1 μ F Electrolytic Capacitor
- 1 x 1 μ F Monolithic Ceramic Capacitor (105)
- 1 x 240 Ω Resistor
- 1 x 1K Trim Potentiometer
- 2 x Alligator Wires
- 4 to 6 x Male/Male Jumpers
- 1 x Small Flathead Screwdriver or Microspatula

Procedure: The logic chips in Activity 2-3 require a +5V power supply. We will be setting up the LM317 chip as a regulated +5V DC supply from a 9V battery, using the circuit diagram in Figure 2-60.

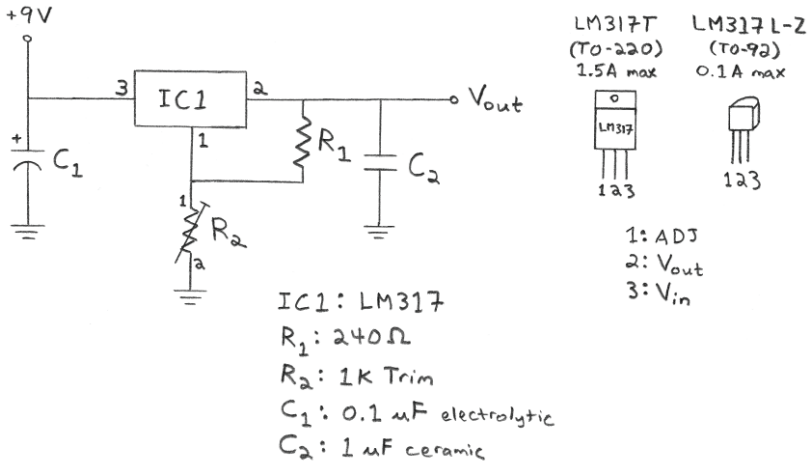


Figure 2-60. Schematic for Activity 2-2. Note: Pin 3 of the 1K trim is not connected.

Quite often, an IC such as this one will be manufactured in different packages, allowing for more flexibility when designing a project. We have the LM317 in two packages in the lab: The **LM317T** (in a TO-220 package that is heat sinkable, and can supply up to 1.5 A current), and the **LM317L-Z** (in a TO-92 package that is not heat sinkable, and can supply up to 0.1 A current).

- Calculate the required resistance of R_2 if the V_{out} required is +5V. Recall: $R_2 = 192 \Omega \times (V_{out} - 1.25)$
- Calculate the power dissipated if the current is expected to be ~20 mA. Is a heat sink necessary? Which chip (the LM317T or the LM317L-Z) is most appropriate for this circuit? Why?
- Build the circuit above, using a voltage regulator appropriate for the current requirements in part (b). Connect the battery as the *last* step. If you see smoke, disconnect the battery immediately.
- Connect the ground and V_{out} pins to a voltmeter using alligator wires. Adjust the trim pot (R_2) with a microspatula until the measured V_{out} is 5.0 V.
- Remove the trim potentiometer, and measure the resistance of the adjusted R_2 (across pins 1 and 2). Was it close to the theoretical value you calculated in Part A? You may use *LM317.xlsx* on the course website to confirm your calculation.

Activity 2-3: Logic Gates

Goal: Using the LM317 5V circuit from Activity 2-2, power and test the functionality of the 74HC08 AND gate.

Materials:

- 1 x Digital Multimeter
- 1 x Breadboard
- 9V Battery + Snap-on Terminal
- DIP switch (≥ 2 switches)
- 2 x 10K Resistors
- 1 x 74HC08 AND Gate
- 1 x 74HC32 OR Gate
- 1 x LED
- 1 x 220 Ω Resistor

Procedure:

- a) In order to provide +5V for this activity, construct the circuit in Activity 2-2.
- b) Using the following circuit diagram, build the AND logic gate (74HC08) circuit:

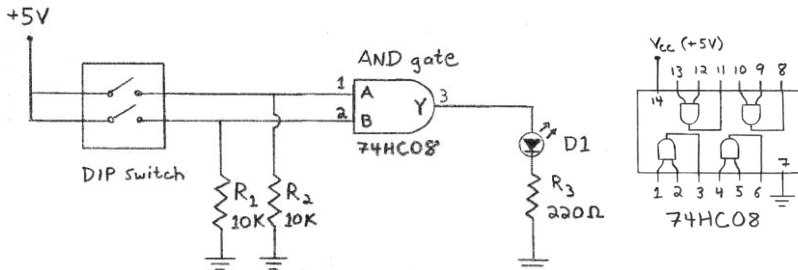


Figure 2-61. Circuit diagram for Activity 2-3 (from Figures 2-38 and 2-39).

Note: Connect +5V to Pin 14 and GND to Pin 7, as shown on the 74HC08 *pin-out diagram*, on the right.

- c) Test the AND gate to verify it produces the expected logic table results, by flipping the DIP switches to different states (HIGH or LOW).
- d) Try swapping out the AND gate with an OR gate (74HC32), and confirm the logic tables. **Hint:** the pin orientation of the OR gate is the same, so there is no need to rewire the circuit.

Learning Objectives for Section 2

After having attended this class, the student will be able to:

- 1) Describe how a capacitor works, and its primary uses: powering, smoothing, and filtering.
- 2) Calculate the charging/discharging time of a capacitor in a resistor-capacitor configuration.
- 3) Calculate the energy stored by a capacitor based on its capacitance in Faraday units.
- 4) Calculate the time constant for a resistor/capacitor circuit, and predict the time required to charge and discharge the capacitor.
- 5) Design a voltage divider circuit based on a given resistive load, R_{load} , and desired voltage, V_{load} , using the 10% rule.
- 6) Appropriately select a voltage source, split voltage supply, voltage regulator, or current regulator depending on the application.
- 7) Calculate the power dissipation and temperature rise based on θ_{JA} , and appropriately decide if a heat sink is needed.
- 8) Use Thévenin's Theorem to simplify a circuit into its Thévenin Equivalent, to calculate the voltage and current for a given component of interest (e.g. the load).
- 9) Correctly identify the pin numbering on DIP chips and transistors.
- 10) Use logic chips in a simple circuit to generate and confirm logic tables.
- 11) Generate logic tables for AND, OR, NOT, NAND, NOR, XOR, and XNOR gates, and be able to determine logic tables for higher order logic circuits.
- 12) Memorize and use logic symbols to construct logic circuits, based on written descriptions of what the circuit should do (e.g. A and B, but not C).
- 13) Use an electronic component datasheet to find and interpret basic information about that component (e.g. pin numbering, maximum power, voltage, and current ratings).

Section 2 - Station Content List

- Digital multimeter
- 1 x Breadboard
- 1 x 9V Battery + Snap-on Connector
- 1 x LM317 (TO-92 Package)
- 1 x LM317 (TO-220 Package)
- 1 x 0.1 μF Electrolytic Capacitor
- 1 x 1 μF Monolithic Ceramic Capacitor (105)
- 1 x 10 μF Electrolytic Capacitor
- 1 x 150 μF Electrolytic Capacitor
- 1 x 220 Ω Resistor
- 1 x 240 Ω Resistor
- 1 x 1K Trim Potentiometer
- 1 x 1K Resistor
- 2 x 10K Resistors
- 1 x LED (Note: longer wire +)
- 2 x Momentary Switches
- DIP switch
- 1 x 74HC08 AND Gate
- 1 x 74HC32 OR Gate
- 2 x Alligator Wires
- 4 to 6 x Male/Male Jumpers
- 1 x Small Flathead Screwdriver or Microspatula
- 1 Stopwatch (phone apps welcome)

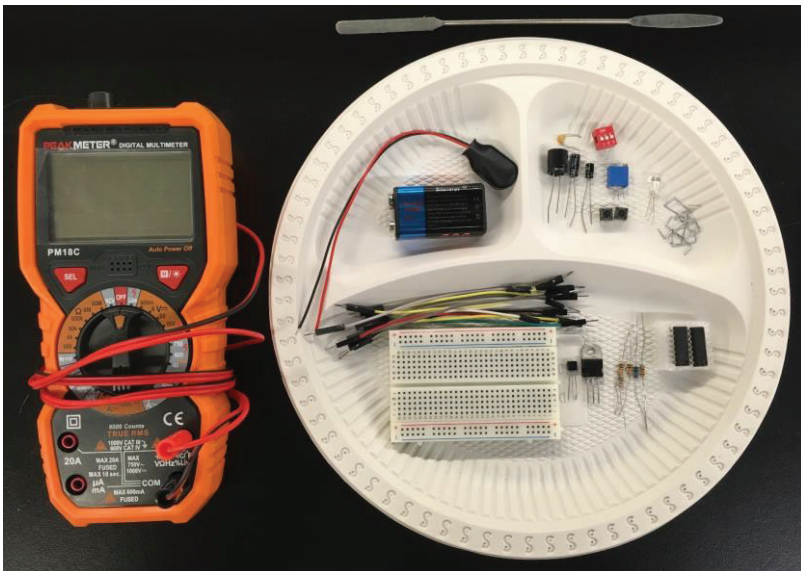


Figure 2-62. Section 2 station setup.

SECTION 3

INTRODUCTION TO PROGRAMMING IN THE ARDUINO C++ ENVIRONMENT

What You'll Be Learning	Lecture: Introduction to programming in C++: The programming environment (Arduino platform). Basic structure of Arduino <i>.ino</i> file: global space, setup function, loop function. Line and section comments. Variable definitions, declaration (global vs. local, integer, long, unsigned, const, float, bool, byte, char, String, arrays), initialization. Casting variable types. Logical expressions and comparison operators. <code>if()</code> , <code>for()</code> , <code>do...while</code> , <code>while</code> , <code>switch case</code> . Programming tips.
What You'll Be Doing	<p>In class, each of you will receive an Arduino Uno board, USB cable, serial LCD module, and jumper wires. The equipment is fragile. Please be careful with it. The activities in this section are interspersed in the lecture. We will be trying out each concept as we go along:</p> <ul style="list-style-type: none">• Connecting a serial LCD module to the Arduino Uno.• Outputting information to the LCD - basic math and logical operations. Example sketches for variable types, math operations, <code>if()</code> statements, loops, etc. <p>Activity 3-1: Programming challenge: spectrophotometer data averaging. Take the Arduino Uno board home and try this activity after class.</p>
Files you will need	To prepare for this section, download and install the Arduino IDE on your laptop computer.

Introduction to the Arduino Uno Microcontroller Board

The Arduino platform was developed at the Ivrea Interaction Design Institute in Italy in 2005, as an inexpensive cross-platform educational prototyping tool. (Mellai 2017a) Microprocessing had existed long before, but the Arduino approach added unprecedented accessibility and ease of programming, which changed the hobbyist landscape and helped jumpstart the internet of things (IoT). The first board developed on this platform was

named the Uno (Italian for “one”). Arduino microcontroller boards can be programmed through a USB connection to a computer, using an open-source platform supported by the Arduino community called the *Arduino IDE* (integrated development environment). There are other software platforms you can program microprocessors with, such as Eclipse, Matlab® and Python. However, the Arduino IDE is a great starting point (available at <https://www.arduino.cc/en/Main/Software/>). Many of the sketches in this section have been inspired by Arduino’s tremendous open source community, accessible at <https://www.arduino.cc/en/Tutorial/Foundations>.

Figure 3-1 provides a quick walk-around of the Arduino Uno microcontroller board. See *Arduino Uno Pin-out Diagram* in the appendix for a more detailed diagram.

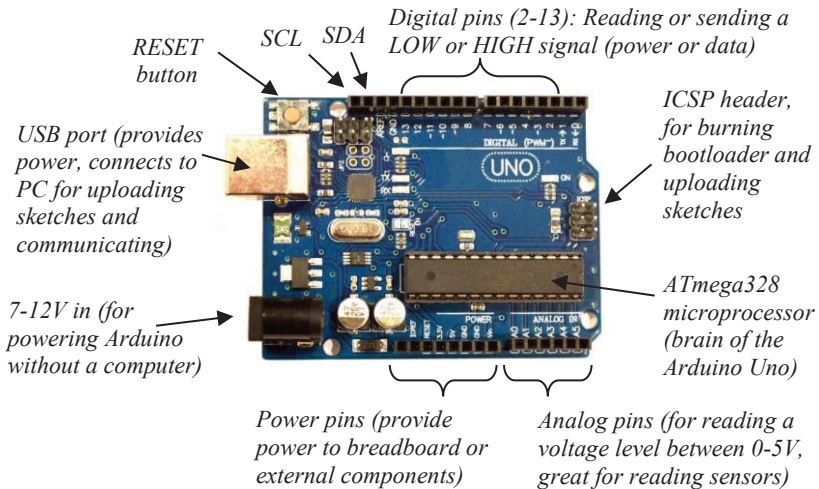


Figure 3-1. Arduino Uno R3 board layout (DIP version).

Connecting a Serial LCD Module to the Arduino Uno

For our first activity, we will connect a serial LCD module to the Arduino Uno. We will refer to the Arduino Uno as an MCU (microcontroller unit) throughout this text, to emphasize that our circuits and sketches are generalizable to other microcontroller boards and chips.

Connect the jumper wires from the MCU to the LCD module first, then plug in the USB cable from the MCU to your computer’s USB port. You will be making the following connections, with four M/F (male-female) jumper wires:

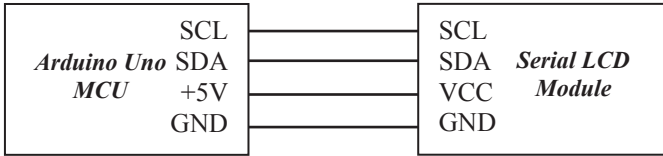


Figure 3-2. Connecting the MCU to a serial LCD module.

Start up the Arduino IDE on your laptop. Your set up should now look like Figure 3-3.

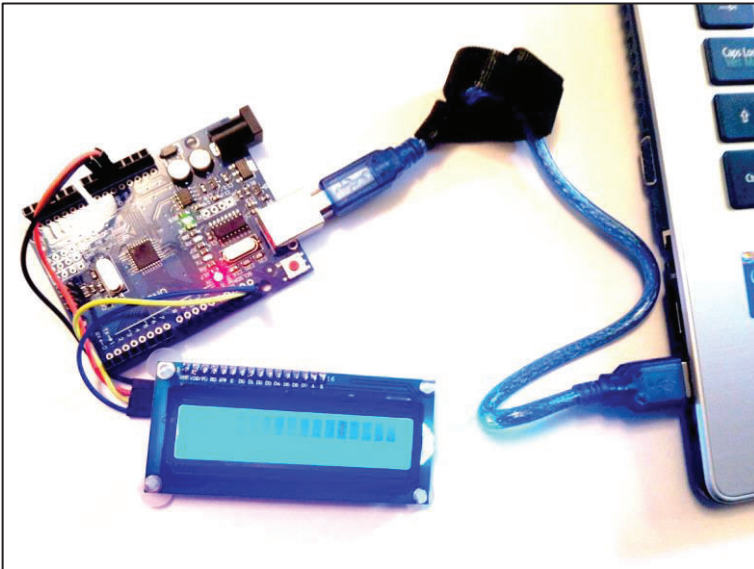


Figure 3-3. Arduino Uno (SOIC version) connected to serial LCD module and laptop.

The serial LCD module will not work without installing and including an external library. We will need to install the *LiquidCrystal_I2C.h* library, by Frank de Brabander. (de Brabander 2017) The following procedure will install the library:

- Click on Tools → Manage Libraries...
- In the “Filter your search...” bar, type “liquidcrystal i2c”. Scroll down, and find “**LiquidCrystal I2C by Frank de Brabander**”.
- Click on this library, then click the “Install” button.

An Arduino program (.ino or .pde file) is called a *sketch*. Every sketch needs to have a `setup()` and `loop()` function defined, even if the sketch does nothing, otherwise it won't compile.

The programming language in the Arduino IDE is a simplified version of the C++ language:

- Every programming statement (except curly brackets), `#define` statements, and `#include` statements) should end with a semicolon.
- There are *many* different ways to code the same thing in C++.
- Your code will be case-sensitive. Upper- and lower-case matters.
- Multiple spaces on a single line are treated the same as single spaces (e.g. “`i = i + 1 ;`” is treated the same as “`i=i+1;`”).
- Empty lines are ignored.

The program window in the Arduino IDE should be similar to Figure 3-4.

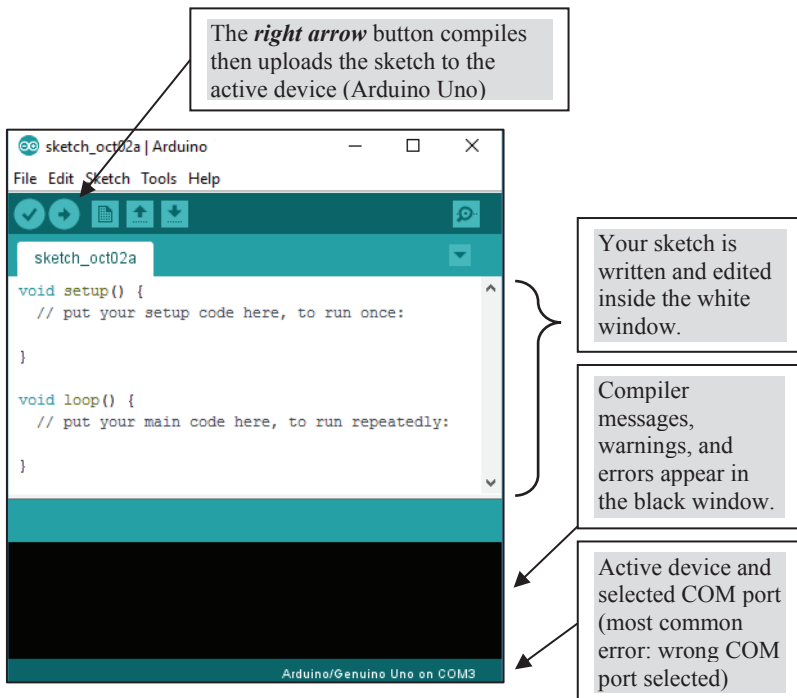


Figure 3-4. Arduino IDE sketch window.

When you run the Arduino IDE program, it will either open a blank sketch (like above), or the last sketch you were working on. When you save a sketch, it will automatically save it in a folder with the same name as the sketch. A sketch named differently than its folder name won't compile. You will instead get a warning or error message. This will help you keep track of the correct and last-compiled version of your program.

Your First Sketch

Type the following commands in the program window:

```
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x27,16,2);

void setup(){
  lcd.init();
  lcd.backlight();
  lcd.clear();
  lcd.setCursor(2,0);
  lcd.print("Hello, world!");
  lcd.setCursor(0,1);
  lcd.print("My first sketch.");
}

void loop(){
}
```

Make sure you select the correct port by clicking Tools→Port→(select the port that is open for the Uno). Now, it's time to compile and upload your code to the Uno, by pressing **Ctrl+U** (for Windows) or **⌘+U** (for MacOS). Another way to compile and upload is to click the little right arrow in the Arduino IDE tool bar. After the sketch uploads, you should clearly see text on the LCD screen.

Note 1: You may need to adjust the contrast of the lcd screen (small blue potentiometer on the underside of the module) using a small screwdriver or microspatula, in order to see the message.

Note 2: Uploading also automatically compiles and saves your sketch. What does *compile* mean? A *compiler* translates code that you write in a computer language (like C++) into machine code that the MCU or CPU can understand.

Note 3: The programming line:

```
LiquidCrystal_I2C lcd(0x27,16,2);
```

initializes the LCD device at bit address 0x27 by default. If your LCD sketch doesn't work, even after checking your wiring and adjusting the contrast by

twisting the dial on the back of the LCD screen, try using 0x3F as the address instead of 0x27, or try running an I²C scanner to find the actual bit address. An I²C scanning sketch is available at <https://playground.arduino.cc/Main/I2cScanner>.

Basic Programming Concepts

Commenting Your Code

Comments are for the programmer (you) for two main reasons:

- 1) To annotate your code and keep track of why you wrote it. It's a breadcrumb trail of your thoughts, since uncommented programs just look like gibberish to someone else, or perhaps to you in the distant future. Get into the habit of adding short comments to your code. They will save you time later.
- 2) To get the compiler to ignore a single line, or section of code. The compiler completely skips over any comments when it compiles the sketch, so you can use a comment to intentionally inactivate a line (or a section) of code, without having to delete it. This is amazingly handy when you are trying to debug your code (find an error), or you would like to stop a particular function in your sketch without having to re-write it later (e.g. to stop an annoying buzzer sound).

Commenting Out One Line

- Using two forward slashes (//) comments out a single line. Everything after the two forward slashes is ignored when you compile the sketch, until the end of that line of code. Comment out the following line in your sketch by placing double forward slashes in front of it, and then upload it to see what happens:

```
//lcd.print("Hello world! ");
```

- Now delete the double forward slashes in front of the `lcd.print()` command, and add an annotation at the end. Then upload it to see what happens:

```
lcd.print("Hello world! "); // "Hello world!" to LCD.
```

- You can comment out more than one line this way. It is handy to keep previous programming ideas commented out in your sketch, if you go off on a tangent that doesn't work.

Commenting Out a Section of Code: Comment Section Brackets

If you need to write a paragraph of comments, it can get tedious using double forward slashes. There is an easier way to comment out a section of text (or code) quickly, using comment section brackets, which can span multiple lines:

- Where you would like the comment section to begin, type: `/*`
- Where you would like the comment section to end, type: `*/`

In the above example, comment out writing text to the LCD screen by adding in comment section brackets:

```
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x27,16,2);

void setup(){
  lcd.init();
  lcd.backlight();
  lcd.clear();
  /*lcd.setCursor(2,0);
  lcd.print("Hello, world!");
  lcd.setCursor(0,1);
  lcd.print("My first sketch.");*/
}

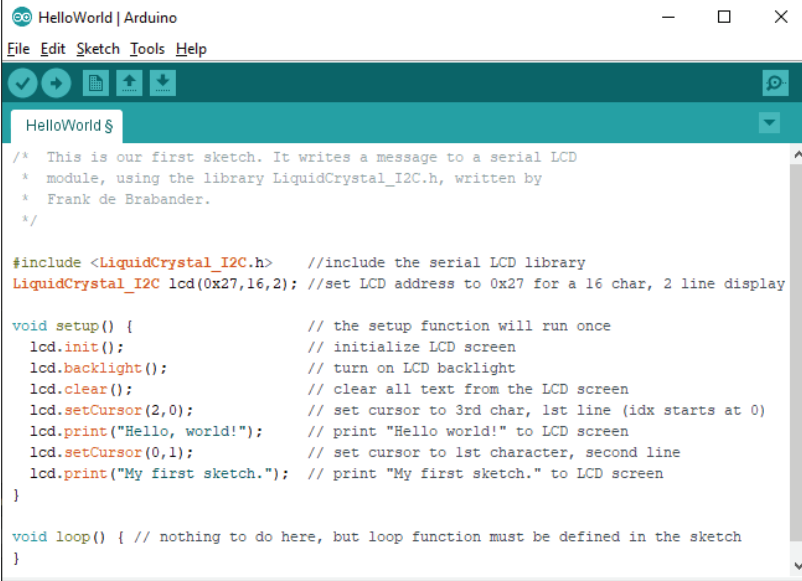
void loop(){
}
```

Now compile and upload, to see what happens. When you comment out text in the Arduino IDE, it changes colour to light grey, to show you that it's just a comment.

It's important if you use section comments to begin and end them at the correct spot. If you forget the end comment section bracket, the compiler will just keep commenting until it reaches the next `*/` if there is one, or it will comment out the rest of your entire sketch.

Comment section brackets are also typically used at the top of a program, to record pertinent information like the program title, filename, author name, date, and even relevant component wiring. Creating a custom header for your sketch can help remind you why you wrote it, and whether or not you successfully completed it. You can also briefly indicate any critical connections you made in your circuit (e.g. connections to the LCD module). Additional notes can help other people understand how to compile and run your sketch, e.g. by providing more information about a library you included.

Remove the commented section brackets in the example above by deleting `/*` and `*/`. Now you can properly annotate your first sketch. At the top of your sketch, add a section style comment to briefly describe what the sketch does. Then, provide a comment on each line (after the semicolon) to describe what that line of code is supposed to do.

The image shows a screenshot of the Arduino IDE interface. The title bar reads 'HelloWorld | Arduino'. The menu bar includes 'File', 'Edit', 'Sketch', 'Tools', and 'Help'. Below the menu bar is a toolbar with icons for saving, undo, redo, and other functions. The main text area contains the following code:

```
/* This is our first sketch. It writes a message to a serial LCD
 * module, using the library LiquidCrystal_I2C.h, written by
 * Frank de Brabander.
 */

#include <LiquidCrystal_I2C.h> //include the serial LCD library
LiquidCrystal_I2C lcd(0x27,16,2); //set LCD address to 0x27 for a 16 char, 2 line display

void setup() { // the setup function will run once
  lcd.init(); // initialize LCD screen
  lcd.backlight(); // turn on LCD backlight
  lcd.clear(); // clear all text from the LCD screen
  lcd.setCursor(2,0); // set cursor to 3rd char, 1st line (idx starts at 0)
  lcd.print("Hello, world!"); // print "Hello world!" to LCD screen
  lcd.setCursor(0,1); // set cursor to 1st character, second line
  lcd.print("My first sketch."); // print "My first sketch." to LCD screen
}

void loop() { // nothing to do here, but loop function must be defined in the sketch
}
```

Figure 3-5. Our first Arduino Uno sketch, with section and line comments.

Storing and Accessing Data in Variables

Variables hold information that can change. Different types of variables store different types of data (and have different limits!). We often forget that memory on a microprocessor (or memory chip) actually occupies physical space on the chip, and that space has limitations. The number of bits the microprocessor uses to store information limits the range of numbers it can store. The processor represents numbers inside the microprocessor in *binary*, so it becomes important to understand how the binary numbering system works.

The smallest variable is 1 *bit* long. In other words, it has only one space (or bit) in microprocessor memory. This space can either be in a high

voltage (“1”) or a low voltage (“0”) state. So a 1-bit variable can hold two different values: 0, or 1.

A 2-bit number has two spaces in microprocessor memory, so it can be used to represent four different numbers: 00, 01, 10, and 11. These numbers in binary are used to represent the base 10 numbers “0, 1, 2, and 3”. Since each bit holds two values, to find out the number of different values a variable can hold, raise 2 to the power of the number of bits (2^n).

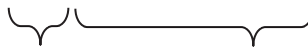
Table 3-1 illustrates how many numbers each type of variable can hold. We will come back to this table, because it ends up being very important when discussing the limits of variable types.

An 8-bit binary number (e.g. 10110010) can be represented like this:

Table 3-1. Bit depth size chart.

# Bits <i>n</i>	# Numbers represented (also called resolution)
1 bit	2^1 2
2 bits	2^2 4
3 bits	2^3 8
4 bits	2^4 16
5 bits	2^5 32
6 bits	2^6 64
7 bits	2^7 128
8 bits	2^8 256
9 bits	2^9 512
10 bits	2^{10} 1,024
11 bits	2^{11} 2,048
12 bits	2^{12} 4,096
15 bits	2^{15} 32,768
16 bits	2^{16} 65,536
31 bits	2^{31} 2,147,483,648
32 bits	2^{32} 4,294,967,296

0b10110010



“0b” is just a header, to let you know the following series of 1s and 0s is a number represented in binary.

“There are 10 kinds of people. Those who understand binary, and those who don’t.”
 –T-shirt at Snow Valley Ski Resort, Barrie ON
 “I’m convinced the only reason we use the base 10 system so much is that we have 10 fingers.” –Robert B. Macgregor, Jr.

To convert this 8-bit number to decimal (base 10):

$$\begin{aligned}
 &1 \times 2^7 = 128 \\
 &+ 0 \times 2^6 = 0 \\
 &+ 1 \times 2^5 = 32 \\
 &+ 1 \times 2^4 = 16 \\
 &+ 0 \times 2^3 = 0 \\
 &+ 0 \times 2^2 = 0 \\
 &+ 1 \times 2^1 = 2 \\
 &+ 0 \times 2^0 = 0 \\
 &\rightarrow 128 + 32 + 16 + 2 = 178.
 \end{aligned}$$

Declaring and Using Variables

Integers

Perhaps the most popular and common variable type in programming is the *integer*. Integers can be positive or negative, and include the number “0”. Only whole numbers can be represented (e.g. 3.14 is not an integer).

An integer-type variable in the ATmega328 chip occupies 2 bytes (or 16 bits) of memory. Looking at Table 3-1, this means an integer defined using the Arduino Uno can store 65,536 different numbers! That’s not bad. However, that doesn’t mean the range of numbers (in base 10) for an integer is 1 to 65,536, for two reasons:

- 1) Zero counts as an integer.
- 2) The first space in the 16-bit number is used by default as a + or – sign. So there are really only 15 bits remaining to describe numbers, half of which will be negative and the other half positive.

The effecting range of numbers you can represent in a 16-bit signed integer (one that can have a + or – sign) is (–32,768 to +32,767). There are different ways of *declaring* (or creating) an integer variable (of course, this is C++, after all). The three most common are:

```
int myVariableName=0;      // signed 16bit integer
int16_t myVariableName=0; // signed 16bit integer
```

Other MCUs can have different storage sizes for integers. For instance, the Arduino Due, with a SAM3X8E ARM Cortex-M3 MCU, stores integers using 32 bits (or 4 bytes).

I would like you to use the declaration statement `int` to declare 16-bit integers. I have included the other ways to declare variables in this section so they don’t confuse you if you see them in someone else’s sketches.

- a) Try writing and running the following sketch, to declare your first integer variable:

```
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x27,16,2);

int answer=0; // declare answer as integer, value=0

void setup(){
  answer=answer+1; // increments answer by 1
  lcd.init();     // initialize LCD screen
  lcd.backlight(); // turn on backlight
  lcd.clear();    // clear LCD screen
  lcd.setCursor(0,0); // set cursor to first row,col
  lcd.print("Answer:"); // print "Answer" to LCD
  lcd.setCursor(0,1); // second row first col
```

```

    lcd.print(answer); // no quotes: print value
  }                    // stored in answer

void loop() {
}

```

This sketch declared an integer called `answer`, and gave it the initial value of 0. In terms of naming the variable, we can use any name we like (e.g. “`int Dave99=0;`” or “`int fluffy_1=0;`”, as long as the name isn’t being used for something else in the program, and there are no spaces or special characters in the variable name. I just chose `answer` arbitrarily. Feel free to call your integer variable something else, as long as you are consistent in the rest of the sketch. Variable names are *case sensitive*, so make sure if you use upper and/or lower case, you are consistent. It’s a good practice to *initialize* your variable with a value in the beginning (like we did here) to clear out any old rubbish in the microprocessor memory. Initializing a variable is important, because the memory may have been previously allocated to something else, and contain other random data. It’s better to start the variable off with the number you want (in this case, 0).

After declaring and initializing the integer variable `answer`, the sketch then performed an operation with it. The command “`answer=answer+1`” means, “increase the current value stored in the variable called `answer` by 1”. Mathematical operations work as you would expect in C++, and brackets work as well. The following symbols and functions are commonly used:

Table 3-2. Mathematical operators and functions in C++.

<i>Operator</i>	<i>Function</i>	<i>Example:</i>
+	add	<code>answer=x+1;</code>
-	subtract	<code>answer=x-1;</code>
*	multiply	<code>answer=x*3;</code>
/	divide	<code>answer=x/2;</code>
%	modulus (computes the remainder after dividing two integers)	<code>answer=x%2;</code> e.g. $19\%8=3$, because 8 goes into 19 twice, with 3 remaining.
<code>pow()</code>	exponent (x^b)	<code>answer=pow(x, 3);</code>
<code>exp()</code>	e^x function	<code>answer=exp(x);</code>
<code>abs()</code>	absolute value	<code>answer=abs(x);</code>
<code>log()</code>	natural log	<code>answer=log(x);</code>
<code>log10()</code>	base 10 log	<code>answer=log10(x);</code>
<code>sq()</code>	square ($x*x$)	<code>answer=sq(x);</code>
<code>sqrt()</code>	square root	<code>answer=sqrt(x);</code>

Note: when you *divide* integers, the decimals will be *dropped* (not rounded). So $5/2$ will return a 2, and $1/2$ will return a 0.

- b) Now replace the line “answer=answer+1;” with “answer=32767+1;”. Then re-run the program. What happens? Why?
- c) What happens if you want to represent a number that is larger than 32767? One of the ways to do this is to declare an ***unsigned integer***. When you are declaring an integer, you can decide to throw away the negative sign if you would like to represent a higher positive number. This is called an “unsigned integer”, which uses **all 16 bits** for positive numbers (so the range is now **0-65535**). Replace the line “int answer=0;” with “unsigned int answer=0;”. Then re-run the program. What happens? Why? Another way to declare an unsigned 16-bit integer is:

```
unsigned int myVariableName=0; //unsigned 16bit int
uint16_t myVariableName=0; // unsigned 16bit integer
word myVariableName=0; // unsigned 16bit integer
```

- d) A ***constant*** is a variable that can’t change. We use constants to protect the data in our variables from ever changing once declared. Replace the line unsigned int answer=0; with:
- ```
const int answer=0;
```
- then try to compile and upload the program. What happens?

### *Long Integers*

Sometimes you will find 16 bits limiting when you are dealing with very large numbers. ***Long*** integers allocate 32 bits per variable (1 byte=8 bits, so long integers are 4 bytes long). Looking back at Table 3-1, a 32-bit binary number can represent 4,294,967,296 numbers, half of which will be negative. So the range of numbers a ***long*** integer can represent is (-2,147,483,648 to +2,147,483,647). An ***unsigned long*** integer can hold numbers between (0 to 4,294,967,295).

Declaring a ***long*** and ***unsigned long*** integer can look like this:

```
long answer = -249502L; // 32-bit signed integer
int32_t answer1=0L; // 32-bit signed integer
unsigned long answer3=56800UL; // 32-bit unsigned int
uint32_t answer2=0UL; // 32-bit unsigned integer
```



You can omit the suffix “L” for long, or “UL” for the numbers above, but if you see either of these at the end of a number, they explicitly tell the compiler these numbers are long, and unsigned long numbers, respectively.

- e) Try out the following program. What happens? What happens if you change the time delay?

```
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x27,16,2);
long answer=0L;

void setup(){
 lcd.init();
 lcd.backlight();
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print("Answer:");
}

void loop(){
 answer++; // shortcut for: answer=answer+1;
 lcd.setCursor(0,1);
 lcd.print(answer);
 delay(250); // 250 msec delay (=0.25 seconds)
}
```

### ***Global Space, Setup Function, and Loop Function***

So far, we have loaded our special libraries and declared our variables at the top of the sketch, and written commands inside the `setup()` function. In the last sketch, we used the `loop()` function for the first time. Until you start writing your own functions, we can say that there are three different “spaces” to write commands. Let’s have a look at the previous sketch again, this time paying attention to *where* things are:

**Global space.** Where you load libraries (#include statements), initialize classes and devices (e.g lcd), and use #define statements. Variables declared here, and for that matter anywhere else *outside* the setup() and loop() functions, will be **global variables** (accessible to the entire sketch).

```
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x27,16,2);
```

```
long int answer=0L;
```

```
void setup() {
 lcd.init();
 lcd.backlight();
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print("Answer:");
}
```

```
void loop() {
 answer++;
 lcd.setCursor(0,1);
 lcd.print(answer);
 delay(250);
}
```

**setup() function.** Any code inside setup() will only be executed *once*, when the MCU first powers up, or each time it is reset (by pressing the reset button on the board). Any variables declared inside setup() will be **local** to setup(), in other words only exist inside the setup() function (in local space), and forgotten after.

**loop() function.** Any code inside loop() will be *repeated* indefinitely. After the MCU runs the last line between the curly brackets, it will “loop up” to the top of the loop() function and start again – running the code over and over. Any variables declared inside loop() will be local to loop(), and won’t exist elsewhere.

The program above keeps on counting up, because “answer++;” is inside the loop function. So, just like real estate, location really matters in coding! You can get into the habit of declaring all your important variables in global space until you need to make your sketch more memory-efficient, or your code more portable.

## Float Variables

As nice looking and useful as whole numbers are, it is often desirable to work with decimals. **Float** variables can bog down processing time, but have a much larger range than long integers, and have up to 7 significant digits. Like long integers, they are stored as 32 bits (4 bytes); however, some of that space is used to represent exponents. So the decimal range of a float variable is:  $(-3.4028235 \times 10^{38}$  to  $3.4028235 \times 10^{38})$ . There is no “unsigned float”, so there is no option to shift this range into positive numbers exclusively.

You could use the following commands to declare float variables:

```
float voltage=0.0; // decimal will mean float
const float pi=3.14159 // declaring constant float
const float Avogadro=6.022e23 // using exponents
float weight=0.f; // f suffix means float #
double frequency=0.D; // same as float
```

f) Try out the following program:

```
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x27,16,2);

float answer=100.0; //decimal tells compiler float#

void setup(){
 lcd.init();
 lcd.backlight();
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print("Answer:");
}

void loop(){
 answer=answer/2.0; //Use 2.0 here. 2 is integer.
 lcd.setCursor(0,1);
 lcd.print(answer);
 delay(250); //250 msec delay (=0.25 seconds)
}
```

g) Replace the line “`lcd.print(answer);`” with “`lcd.print(answer,6);`”, and upload. What happens?

### *If...Then...Else Statements (and Logical Expressions)*

One of the most common tools in the programmer’s tool chest is the *if...then...else* statement. We face *if...then...else* choices all of the time. If I miss the bus, then I will walk to the subway, or else I will stay home. If I pack a lunch, then I will have something to eat around noon, or else I will have to buy lunch. If I forget to drink my morning coffee, then I will stop at a coffee shop; or else I won’t be able to concentrate. These statements are no different when you are programming. Let’s write a simple sketch to illustrate how to test values inside our variables with the *if* statement:

```
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x27,16,2);
```

```
float x=2.0;
float y=5.5;

void setup(){
 lcd.init();
 lcd.backlight();
 lcd.clear();
 lcd.setCursor(0,0);
 if(x<y){
 lcd.print("x is smaller.");
 // put as many commands as you like here.
 } // end of if statement (careful with brackets)
} // end of setup function
```

Logical expression **inside** the round brackets to be tested

If the logical expression is **true**, then run the code between the curly brackets. If the logical expression is **false**, then the code between the curly brackets is ignored.

```
void loop(){
}
```

#### h) Compile and run the program.

If the condition inside the `if()` statement is true, then lines of code between the following curly brackets after the `if()` statement are executed. You can put as many commands as you like in between the curly brackets (ending each line with a semicolon, as usual). What would happen in the sketch above, if we declared `x` to be 50.0 (in other words, `x>y`)? Nothing! The expression would be false, and everything inside the curly brackets would be ignored. What if we'd like to have the sketch do something else if the logical expression is false? Try out this sketch:

```
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x27,16,2);

float x=50.0;
float y=5.5;
void setup(){
 lcd.init();
 lcd.backlight();
 lcd.clear();
 lcd.setCursor(0,0);
 if(x<y){
 lcd.print("x is smaller.");
 }else{
 lcd.print("x is larger.");
 } // end of if
}
void loop(){
}
```

If the logical expression is **true**, this line runs.

If the logical expression is **false**, this line runs.

Can you spot a logical problem in the sketch above? It should compile and run, but what happens if  $x$  and  $y$  are equal? Then it will say that  $x$  is larger, which isn't true. Luckily, we have more than one way to compare values using *relational operators*, listed in Table 3-3.

**Note:** float variables are not *exact* numbers. They can sometimes be off by a tiny amount, especially when dividing, so be careful when comparing them with the `==` expression. Rather than testing to see if two float numbers are equal, e.g.:

```
if (x==y) {
```

it is safer to subtract them and compare the difference to a tolerance:

```
if (abs(x-y)<0.0001) { // absolute value of difference
```

You can *nest* `if()` statements. In other words, you can put one `if()` statement inside of another. Make sure you keep track of conditions with indenting and curly brackets (the Arduino IDE helps you out here).

### Conditional Operator, “?:”

The conditional operator “?:” provides a short form for the *if...then...else* structure in C++, all in a single line: (Smith 2009)

```
alarm==1?lcd.print("Wake up!"):lcd.print("Snooze.");
```

The question mark comes *after* the condition, and the colon separates the *then* outcome from the *else* outcome. This can make coding more concise. For instance, the following code will store the absolute value of  $x$  to  $y$ :

```
if (x<0) {
 y=-x;
}else{
 y=x;
}
```

This code could be re-written more succinctly using the conditional operator. After the conditional operator runs, the result is returned to  $y$ :  
`y=x<0?-x:x; // or you could just use y=abs(x)`

**Table 3-3. Relational operators in C++.**

| <i>Logical Expression Syntax</i> | <i>Meaning</i>                                      |
|----------------------------------|-----------------------------------------------------|
| <code>if (x&lt;y) {</code>       | “if $x$ is <i>less than</i> $y$ ...”                |
| <code>if (x&lt;=y) {</code>      | “if $x$ is <i>less than or equal to</i> $y$ ...”    |
| <code>if (x&gt;y) {</code>       | “if $x$ is <i>greater than</i> $y$ ...”             |
| <code>if (x&gt;=y) {</code>      | “if $x$ is <i>greater than or equal to</i> $y$ ...” |
| <code>if (x==y) {</code>         | “if $x$ is <i>equal to</i> $y$ ...”                 |
| <code>if (x!=y) {</code>         | “if $x$ is <i>not equal to</i> $y$ ...”             |

## Else If

If you would like to test logical expressions sequentially, you can also use a single (or multiple) **else if** statements. The above code could be modified to handle three cases:

```
if(x<y){
 lcd.print("x is smaller.");
}else if(x==y){ // new condition here
 lcd.print("x is equal.");
}else{
 lcd.print("x is larger.");
} // end of if
```

## Bool Variables

A **bool** (or Boolean-type) variable gets its name from George Boole, a 19<sup>th</sup> century English mathematician whose interests included logic, algebra, and probability. Unlike **integer**, **long**, and **float** variables, **bool** variables are only one bit long, thus can hold two different values: 0 or 1. This can also be represented as “false” or “true” (lower case), or as “LOW” or “HIGH” (upper case). To declare a **bool** variable and initialize it with a value of 1, you could write any of the following:

```
bool answer=true; // true must be all lower case
bool answer=1;
bool answer=HIGH; // HIGH must be all upper case
boolean answer=1;
```

Alternately, to declare a **bool** and initialize it with a value of 0, you could write any of:

```
bool answer=false;
bool answer=0;
bool answer=LOW;
boolean answer=0;
```

Remember the logic tables? The same syntax [(0,1), (false, true), (LOW, HIGH)] is used in C++.

One really handy feature about bool variables is that they can completely replace **logical expressions** inside `if()` statements. A bool *is* a logical true or false, so it doesn't even need an operator (e.g. `==`).

Let's try this example:

```
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x27,16,2);
bool showAnswer=true;
int x=1;
int y=3;
int answer=0;
```

```

void setup() {
 answer=(5*x)+(2*y); // brackets work in C++
 lcd.init();
 lcd.backlight();
 lcd.clear();
 lcd.setCursor(0,0);
 if(showAnswer) {
 lcd.print(answer);
 delay(3000);
 } // end of if
 lcd.setCursor(0,1);
 lcd.print("Finished.");
}
void loop() {}

```

A boolean doesn't require a logical comparison within the **if()** statement.  
`if(showAnswer)`  
 is the same as:  
`if(showAnswer==true)`

Statements outside the **if()** statement will **always** run.

- i) Enter and upload the sketch. Then, change the value of `showAnswer` to `false`, and upload. What happens?

### Boolean Operators

If you would like to make decisions based on values stored in one or more bool variables, or combine more complicated logical expressions inside an `if()` statement, the syntax is slightly different. Let's say we declare two boolean variables:

```

bool condition1=true;
bool condition2=false;

```

**true** and **false** are all lower case, and not in quotes.

Remember the logic tables in Section 2? You can treat bool variables as inputs using the logic operators in Table 3-4 (just like logic gates):

**Table 3-4. Table of logical (Boolean) operators.**

| <b>Boolean Syntax:</b>                            | <b>Meaning:</b>                                                                                                                 |
|---------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------|
| <code>if(condition1) {</code>                     | "if condition1 is true..."<br>If you forget this syntax, you can still use the long way:<br><code>if(condition1==true) {</code> |
| <code>if(condition1&amp;&amp;condition2) {</code> | "if condition1 AND condition2 are true..."                                                                                      |
| <code>if(condition1  condition2) {</code>         | "if condition1 OR condition2 is true..."                                                                                        |
| <code>if(!condition1){</code>                     | "if condition1 is NOT true..." Same as:<br><code>if(condition1==false) {</code>                                                 |

|                                          |                                                                                                                              |
|------------------------------------------|------------------------------------------------------------------------------------------------------------------------------|
| <code>if(condition1!=condition2){</code> | “if condition1 is different than condition2...”<br>Note: this is a relational operator, but is useful as a logical XOR test. |
| <code>condition1=!condition1;</code>     | Change the value of condition1. (If condition1=true, switch it to false, and if condition1=false, then switch it to true.)   |

Boolean operators aren’t just for bool variables. You can also use them to combine multiple logical expressions, e.g.:

```
if(x<6&&y>=12){z=x+y;} //if x<6 and y>=12, add them.
```

### *Byte Variables*

If you only need to store a *small* positive integer, a **byte variable** occupies only 1 byte of memory (or 8 bits of data). A byte variable can represent an integer ranging from 0 to 255 inclusive (no negative numbers allowed). The three main ways of declaring a byte variable are:

```
byte myVariableName=0;
uint8_t myVariableName=255;
unsigned char myVariableName=127;
```

Although byte variables are very memory efficient, be careful when using them for math, as it is easy to forget about their very narrow limits. We will discuss byte variables in more detail in Section 4. Out of interest, there is a signed 8-bit integer, which stores a number between -128 and 127:

```
int8_t myVariableName=-10;
```

### *String and Char Variables*

A **String** variable is very flexible, because it can store text as well as numbers. A shortcoming is that Strings tend to be memory pigs, and if a number is stored as a String, you can’t do math with it in that form.

A **char variable** is similar to a String variable, but is 8 bits long and holds only one character, represented by a code from -128 to 127. The way a char variable is displayed depends on the device it is sent to. Without getting too technical, some devices display char variables using ASCII definitions (American Standard Code for Information Interchange). Other devices display char variables using UTF-8 definitions. *UTF-8 and ASCII Tables* are provided in the appendix (Table A-2). When a char variable holds a number from 0-31, these codes map to (non-printable) control characters.



Codes 32-127 map to regular printed characters (numbers, the alphabet, etc.). Negative codes (-128 to -1) map to the extended character definitions in Table A-2 from 128-255. We can write a simple program to show how **String** and **char** variables are declared and used:

```
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x27,16,2);

String msg1="Welcome"; // note capital S on String
char letter; // declare a char
int i=65; // declare integer to count

void setup(){
 lcd.init();
 lcd.backlight();
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print(msg1); //no quotes to print msg1 contents
}

void loop(){
 letter=i; //65 is A in UTF-8 & ASCII
 lcd.setCursor(0,1);
 lcd.print(letter);
 i=i+1;
 delay(400);
}
```

j) Try uploading and running this sketch, to get the hang of using Strings and char variables.

You can combine strings together in a program simply by adding them with a “+” sign, so the following commands would work, as long as they are both string variables:

```
String message1="bigger";
String message2="line";
String message3=message1+message2;
```

You can declare char variables with either a UTF-8 code, or using **single quotes** with the actual letter. The following two commands are equivalent:

```
char myfavletter='A';
char myfavletter=65;
```

The *UTF-8 and ASCII Tables* (Table A-2) are provided in the appendix. For some additional useful commands for Strings and char variables, see *Advanced Formating and Variable Type Conversions* in Section 10.

## *Casting Variable Types*

Occasionally, your sketch won't compile or give the answer you expect because you are trying to perform operations on different types of variables at the same time. For instance, if you were to try to add an integer to a String, the compiler will let you know there's a problem with that.

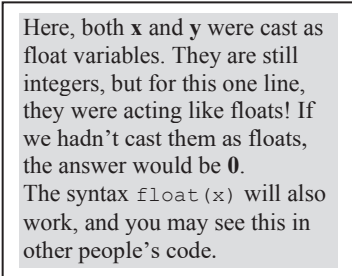
**Casting** a variable takes one variable type and temporarily converts it to another, for calculations and for combining Strings. It does not change the type of the original variable. It only temporarily "casts" it as the different type specified for that one command (like casting a role in a movie).

To cast a variable, write the variable type you want it to behave like in brackets, in front of the variable. Here is a simple example:

```
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x27,16,2);
int x=1;
int y=2;
float answer=0.0;

void setup() {
 answer=(float)x/(float)y;
 lcd.init();
 lcd.backlight();
 lcd.clear();
 lcd.setCursor(0,0);
 lcd.print(answer);
}

void loop() {
}
```



Here, both **x** and **y** were cast as float variables. They are still integers, but for this one line, they were acting like floats! If we hadn't cast them as floats, the answer would be **0**. The syntax `float(x)` will also work, and you may see this in other people's code.

**Table 3-5. How to cast between common variable types, with examples.**

| <i>Initial Variable Type (“A” used to store initial value)</i>            | <i>To cast as a...</i> | <i>Use... (“B” used to store cast value)</i>                                            |
|---------------------------------------------------------------------------|------------------------|-----------------------------------------------------------------------------------------|
| <b>byte</b><br>e.g.:<br>byte A=3;<br>or <b>bool</b><br>e.g.:<br>bool A=1; | int                    | int B=(int)A;                                                                           |
|                                                                           | long                   | long B=(long)A;                                                                         |
|                                                                           | float                  | float B=(float)A;                                                                       |
|                                                                           | String                 | String B=(String)A;                                                                     |
|                                                                           | char                   | char B=(char)A; //UTF8 code<br>char B=(char)(A+48); //1 digit #                         |
| <b>int</b><br>e.g.:<br>int A=3;                                           | byte                   | byte B=(byte)A; //respect byte limits                                                   |
|                                                                           | long                   | long B=(long)A;                                                                         |
|                                                                           | float                  | float B=(float)A;                                                                       |
|                                                                           | String                 | String B=(String)A;                                                                     |
|                                                                           | char                   | char B=(char)A; //UTF8 code<br>char B=(char)(A+48); //1 digit #                         |
| <b>long</b><br>e.g.:<br>long A=3L;                                        | byte                   | byte B=(byte)A; //respect byte limits                                                   |
|                                                                           | int                    | int B=(int)A; //respect int limits                                                      |
|                                                                           | float                  | float B=(float)A;                                                                       |
|                                                                           | String                 | String B=(String)A;                                                                     |
|                                                                           | char                   | char B=(char)A; //UTF8 code<br>char B=(char)(A+48); //1 digit #                         |
| <b>float</b><br>e.g.:<br>float A=3.14;                                    | byte                   | byte B=(byte)A; //rounds down                                                           |
|                                                                           | int                    | int B=(int)A; //rounds down                                                             |
|                                                                           | long                   | long B=(long)A; //rounds down                                                           |
|                                                                           | String                 | String B=(String)A;<br>String B=String(A,3); //A (rounded<br>//to 3 decimals) to String |
|                                                                           | char                   | char B=(char)A-48; //1 digit #<br>(don't subtract 48 for UTF-8 code)                    |
| <b>char</b><br>e.g.<br>char A='3';                                        | int                    | int B=(int)A-48; //1 digit #                                                            |
|                                                                           | long                   | long B=(long)A-48; //1 digit #                                                          |
|                                                                           | float                  | float B=(float)A-48.0 //1 digit #                                                       |
|                                                                           | String                 | String B=(String)A; //# or letter                                                       |

Casting a variable is particularly useful when you are trying to pass the wrong type of variable to a function (more on that later). However, sometimes you can get away with not casting when using different types of variables. For instance, there is no tangible benefit of casting between *byte*, *int*, and *long* variables if the numbers you are intending to store can fit into the *target* variable type. So the following code would not require casting:

```
byte x=1;
int y=2;
long answer=x+y; //3 fits in long, no casting req'd
```

However, we will learn in Section 4 that casting between variable types is required if they are to be used as *input arguments* of functions. Table 3-5 summarizes how to cast between most variable types, with examples.

## Arrays of Variables

Any variable type can be turned into an *array* (even a String). An array is a matrix of data, or a collection of elements all under the same declared variable name. The array is accessed using index numbers. For example, say we wanted to measure the weights of **5** acetaminophen tablets, and store the data to float variables. Instead of declaring 5 different float variables, we can instead make an *array* of float numbers, containing 5 elements:

```
float tabletWeight[5]={0.352,0.314,0.387,0.343,0.308};
```

Here, square brackets are used to declare the array of floats, given the name “tabletWeight”. The 5 means: make the array five *elements* long. Defining the number of array elements is optional. This statement will also work:

```
float tabletWeight[]={0.352,0.314,0.387,0.343,0.308};
```

If you do not specify the number of elements in the array definition, the compiler will pick an appropriate size for you. Once the array is declared and initialized, you lose the ability to fill the entire array with numbers all at once like in the statement above. You can refer to any separate element of the array by using the square brackets. For instance, if we want a separate variable to hold the third value, we could use:

```
float thirdTablet = tabletWeight[2];
```

Even though the array has **5** elements, the index numbers of the array start at **0**, so the array ends at 4, not 5. This can be confusing at first. In our example above, `tabletWeight[0]` is the first float number 0.352, and `tabletWeight[4]` is the last float number, 0.308.

Arrays can have more than one dimension. Let’s say we wanted to store tablet weights (in mg) for *two different tablet batches*, with 10 tablets per batch. If we wanted to declare a two-dimensional array of integers, the syntax would look like this, with the *row index first* and the *column index second*:

```
int tabletInfo[2][10]={
 {325, 323, 326, 339, 337, 348, 318, 327, 319, 342},
 {411, 416, 409, 429, 427, 419, 432, 414, 402, 456}
};
```

We used the structure of the two-dimensional array to store tablet weights from the first batch in the first row, and tablet weights from the second batch in the second row. When declaring a 2D array, pay close attention to where the square brackets, curly brackets, and commas go. Recalling that array index numbers start at zero, then:

```
int tempInt=tabletInfo[1][5];
```

would assign the number 419 (2<sup>nd</sup> batch, 6<sup>th</sup> tablet) to the variable `tempInt`.

Accessing arrays is much easier with *for loops*, discussed in the next section, which are extremely important to programming in general.

**Test your understanding:** What number would `tabletInfo[0][6]` be? What would happen if you tried to read the value `tabletInfo[2][10]`?

### *Char Array*

A *char array* can also be used in place of a String. To make matters slightly confusing, programmers also refer to char arrays as strings (lower case “s”). One thing to note is that the array size of a char array should be one greater than the length of text stored, so that the compiler can add a *null character* to terminate the char array. The null character (ASCII 0) ends the char array so the program knows where the array ends. You don’t have to know what a null character is, just remember to add one to the array size to leave room for it. Either of these declarations will work to declare a char array with a string of literal text:

```
char myMessage[6]="hello"; // array size is 6, not 5
char myMessage[]="hello"; // let compiler choose size
```

**Note:** Double quotes are required for filling an array of char variables in the declaration statement.

The ability to enter an array of characters in one string like this is a privilege that can only happen as you first declare and initialize the char array (on the same line). For instance, the following code would *not* work:

```
char myMessage[6];
myMessage[6]="hello"; // Will not compile. Too late!
```

After a *char array* is defined, you need to either handle array elements one at a time, like this:

```

char myMessage[6];
myMessage[0]='h'; // Note the use of single quotes.
myMessage[1]='e';
myMessage[2]='l';
myMessage[3]='l';
myMessage[4]='o';
myMessage[5]='\0'; // Null character '\0' is last.

```

or use another function, like `strcpy()`:

```

char myMessage[6];
strcpy(myMessage,"hello"); //copy hello to myMessage

```

If you would like to declare a constant array of characters that you want to protect from changing during your sketch, for instance a password or server address, you can declare it as you would expect with the command `const` in front of the variable type:

```

const char myPassword[]="pass12345";

```

Another way that programmers commonly handle this task is by declaring a *pointer*, to point to an array of literal text. (Save 2011) The following commands show equivalent ways of coding this:

```

const char * myPassword="pass12345";
const char *myPassword="pass12345";
const char* myPassword="pass12345";
const char*myPassword="pass12345";

```

The following command will print out the literal string “pass12345”:

```

Serial.println(myPassword);

```

See *Char Arrays: Advanced Functions* in Section 10 for further details.

### *Data Types: More Complicated Conversions*

Casting won’t work in every situation, for example when converting String variables to other variable types. Some conversions require special functions. See Figure A-9, *Variable Type Conversion Chart* in the appendix for an overview of common conversions.

Table 3-6 provides commands for converting variable types to and from Strings and char arrays. In the table, the variable name “A” is used for the data type to be converted from, and “B” is used for the data type to be converted to. “A” and “B” should be replaced with the variable names you need to convert.

**Table 3-6. More complicated variable conversions involving String and char.**

| <i>Initial Variable Type<br/>("A" used to store initial value)</i> | <i>To convert to a...</i>            | <i>Use...<br/>("B" used to store converted value)</i>                                                                                                             |
|--------------------------------------------------------------------|--------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| String<br>e.g.:<br>String A="31";                                  | bool,<br>byte,<br>int,<br>or<br>long | myInput.toInt()<br>e.g.:<br>byte B=A.toInt();<br>int B=A.toInt();<br>long B=A.toInt();                                                                            |
|                                                                    | float                                | myInput.toFloat()<br>e.g.:<br>float B=A.toFloat();                                                                                                                |
|                                                                    | char[]                               | myInput.toCharArray(myOutput, len)<br>e.g.:<br>char B[9]; //big enough array<br>A.toCharArray(B, A.length()+1);<br>or:<br>A.toCharArray(B, sizeof(A));            |
| char[]<br>e.g.:<br>char<br>A[3]="31";                              | byte<br>or int                       | atoi(myInput)<br>e.g.:<br>byte B=atoi(A);<br>int B=atoi(A);                                                                                                       |
|                                                                    | long                                 | atol(myInput)<br>e.g.:<br>long B=atol(A);                                                                                                                         |
|                                                                    | float                                | atof(myInput)<br>e.g.:<br>float B=atof(A);                                                                                                                        |
|                                                                    | String                               | String B=(String)A<br>//works for char[] arrays                                                                                                                   |
| byte<br>e.g.:<br>byte A=31;                                        | char[]                               | itoa(myInput, myOutput, base) //base<br>is #system<br>e.g.:<br>char B[3];<br>itoa(A, B, 10); //(10 for base10)                                                    |
| int<br>e.g.:<br>int A=31;                                          |                                      |                                                                                                                                                                   |
| long<br>e.g.:<br>long A=31;                                        |                                      |                                                                                                                                                                   |
| float<br>e.g.:<br>float A=3.14;                                    | char[]                               | dtostrf(myInput, len, precision, myOutput);<br>//len: length of string to create<br>//precision: #decimals<br>e.g.:<br>char B[4];<br>dtostrf(A, sizeof(A), 2, B); |

See *Additional String Conversion Commands*, Table 10-23 in Section 10 for advanced functions to convert other variable types to Strings.

## Defining Programming Loops in Arduino

A programming loop is a structure that repeats continuously, usually (but not always) testing a condition each time to see if it is time to exit the loop. The most common type of loop is probably the *for loop*, which will run a bit of code a specified number of times, and then exit the loop.

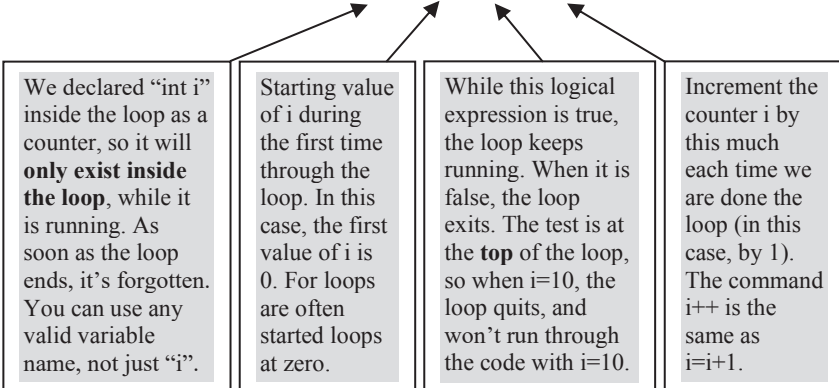
### For Loops

The *for* loop is ubiquitous in programming, so it's important that you understand how it works. The *for* loop starts a counter at a value you specify, increments the counter each time by an amount you specify, and stops when a condition you specify becomes *false*. The following *for* loop would count up from 0 to 9 inclusive, running the loop a *total* of 10 times:

```
for(int i=0;i<10;i++){ // beginning of for loop
 // other commands can go here, inside the for loop
} // end of for (where for loop stops)
```

Let's have a closer look at the *for* loop syntax.

```
for(int i=0;i<10;i++){
```



- a) Let's try writing and uploading the following sketch, which illustrates defining and using an array inside a *for* loop (and also casting an integer to a string):

```
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x27,16,2);
float tabletWeight[5]={0.352,0.314,0.387,0.343,0.308};
```



```

void setup() {
 lcd.init();
 lcd.backlight();
 lcd.clear();
}

void loop() {
 for(int i=0;i<5;i++){
 lcd.setCursor(0,0);
 lcd.print("tablet " + (String)i);//cast i as string
 lcd.setCursor(0,1);
 lcd.print(tabletWeight[i],3);//",3" does 3 decimals
 delay(1000);
 } // end for
}

```

This *for* loop will start at  $i=0$ , and stop at  $i=4$ , running exactly 5 times (0,1,2,3,4). This aligns perfectly with our array index numbers (0 to 4).

## C++ Shorthand Increment Expressions

The  $i++$  command is in part where the C++ language got its name, because it looks snazzy! Programmers are always trying to save time. Here are some other C++ short forms for changing, incrementing, or manipulating values in variables. You can try them out, if you are brave, or want to save keystrokes. See *Increment Operators as Array Index Values* in Section 10 for more details.

There is a whole lot of flexibility with *for* loop syntax in C++. In the above example, we used an integer as a counter, and went up by 1. You can use these shorthand increment expressions in *for* loops, or also feel free to create your own expressions. For example, the following code would also work:

```

for(float Bob=5.0;Bob<50.0;Bob*= 1.1){
 lcd.print(Bob);
 delay(1000);
}

```

**Table 3-7. C++ shorthand.**

| C++<br><i>Shorthand:</i> | <i>Equivalent to:</i> |
|--------------------------|-----------------------|
| $i++$ ;                  | $i=i+1$ ;             |
| $++i$ ;                  |                       |
| $i--$ ;                  | $i=i-1$ ;             |
| $--i$ ;                  |                       |
| $i+=y$ ;                 | $i=i+y$ ;             |
| $i-=y$ ;                 | $i=i-y$ ;             |
| $i*=y$ ;                 | $i=i*y$ ;             |
| $i/=y$ ;                 | $i=i/y$ ;             |

## *Do...While Loops*

Instead of counting a specified number of repetitions and then exiting, *do...while* loops will run until the “while” logical condition becomes false,

and it can be *ANY* logical condition or expression that does the job. This has the drawback of potentially getting your sketch stuck in a loop forever, but also offers more flexibility in programming.

For instance, you may want a sketch to wait until the user presses a button to exit a loop. This is prime “do...while” material. The “while” condition is just a logical expression (or boolean variable).

To make matters confusing, you can write a *do...while* loop to behave exactly as a *for* loop, if you put your own counter in the loop.

b) Let’s write a simple sketch to illustrate how *do...while* works.

Instead of using a *for* loop, we are going to test as many tablets as possible until we find an overweight tablet (defined as mass > 0.380 mg). If we find one, we need to trigger an alarm, and stop our loop.

```
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x27,16,2);
float tabletWeight[5]={0.352,0.314,0.387,0.343,0.308};
bool alarm=false;
```

```
void setup(){
 lcd.init();
 lcd.backlight();
 lcd.clear();
 int i=0;
 do{
 if(tabletWeight[i]>0.380){alarm=true;}
 lcd.setCursor(0,0);
 lcd.print(tabletWeight[i],3);
 i++;
 delay(1000);
 }while(!alarm); // while alarm is false
 if(alarm){
 lcd.setCursor(0,1);
 lcd.print("ALARM:heavy tab");
 } // end of if
}

void loop(){
}
```

**Note:** it’s ok to put the whole *if* statement all on one line, if there is only one command inside it. Make sure you remember the semicolon inside the curly brackets.



This sketch tests each tablet to see if it is overweight, with the threshold 0.380 mg defined as the upper weight limit. If the tablet is overweight, the bool variable “alarm” becomes true, then the `while(!alarm)` term becomes false, so the loop exits. Remember that the *do...while* loop only runs while the condition in the `while()` brackets is true, and the `!alarm` triggered a false to stop it.

**Note:** the *do...while* loop tests the `while()` condition at the *bottom* of the loop, so it will always run through the loop at least once.

**Question:** there are only 5 tablet weights in the array. How would we get the *do...while* loop to stop testing tablet weights after `i=4`?

**Hint:** Have a look at the table of boolean operators (Table 3-4).

## *While Loops*

**While loops** are very similar to *do...while* loops, but the exit condition is tested at the top, so they have the added flexibility of *never running at all* (or running zero times) if the *while* condition is false. This is useful in situations where the sketch needs to react to something bad that's happening, but if nothing bad is going on, then there is no point in going through the loop even once. One of the more useful applications of a **while loop** is to create a dead end in a program. The `setup()` function in a sketch runs only once. The `loop()` function keeps running continuously, as long as the microprocessor is powered on. How do you get the loop function to stop? If you want to, you can try this:

```
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x27,16,2);

void setup(){
 lcd.init();
 lcd.backlight();
 lcd.clear();
}

void loop(){
 for(int i=0;i<1000;i++){
 lcd.setCursor(0,0);
 lcd.print(i);
 delay(10);
 } // end of for
 while(true); //dead end in the program.
}
```

This condition is always **true**, so the *while* loop will repeat forever. This loop never exits. This expression can also be written as: `while(1)`;

The `while(true);` command stops the sketch dead! Of course, you could put the *for* loop code inside of the `setup()` function instead, and it will still just run once, but this construct gives you a work-around if you don't have the choice, and want to stop something in the middle of the `loop()` function.

### *For, Do...While, or While?*

One concept new programmers find tricky is *when* to use each type of loop. The answer is: it really depends on what you want. If you'd like to specify the number of times a loop runs, the **for** loop is best suited. If you would like the option of a loop continuing until a specific condition is met, independent of the number of times a loop is run (e.g. a user pressing a button), then a **do...while** or **while** loop is appropriate. If you would like to only run the contents of a loop if a specific condition is met, then a **while** loop is best because it tests the condition to continue at the top of the loop, and will not run at all unless that condition is true. A **do...while** loop will always run at least once, because it asks the question "should I keep going?" at the bottom of the loop.

However, there can certainly be overlap. The following code in Table 3-8 illustrates how you can use all three loop structures to calculate 5 factorial, as  $5! = 1 \times 2 \times 3 \times 4 \times 5$ . The commands in **boldface** are where each loop would exit, if the condition to continue the loop is *false*.

**Table 3-8. Three major types of loops performing the same task.**

| <i>For Loop:</i>                                                          | <i>Do...While Loop:</i>                                                               | <i>While Loop:</i>                                                                  |
|---------------------------------------------------------------------------|---------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
| <pre>int fact=1; <b>for</b>(int i=1;i&lt;6;i++){     fact=fact*i; }</pre> | <pre>int fact=1; int i=1; do{     fact=fact*i;     i++; }<b>while</b> (i&lt;6);</pre> | <pre>int fact=1; int i=1; <b>while</b> (i&lt;6) {     fact=fact*i;     i++; }</pre> |

### *Omitting Curly Brackets*

If your for loop, do while loop, if statement (etc.) has only one command line inside it, you can omit the curly brackets and the function will stop at the next semicolon. This provides a way to write code succinctly, e.g.:

```
for(int i=0;i<10;i++)a+=i; // a=0+1+...+9
if(a<b)a=0; // if a<b then set a to 0
```

### *The Break Command*

Regardless of the kind of loop you are running (**for**, **do...while**, **while**, etc.), if you use the **break** command, then you can leave the loop you are currently in, right at that point. It's the get-out-of-jail-free card for loops.

This can make for interesting navigation, but it can also save you the headache of coming up with a new exit condition, or stumble through complicated exit logic. It gives you the ability to the exit the loop wherever you like. Here’s how the `break` command could work in a simple sketch:

```
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x27,16,2);
bool annoyUser=true;

void setup(){
 lcd.init();
 lcd.backlight();
 lcd.clear();
 int i=0;
 while(annoyUser){
 i++;
 lcd.setCursor(0,0);
 lcd.print(i);
 if(i==10){
 annoyUser=false;
 break; // breaks out of the while loop here
 } // end if
 delay(2000);
 } //end while
 lcd.setCursor(0,1);
 lcd.print("Warmup complete");
}

void loop(){}

```

This is the same as:  
`while (annoyUser==true)`

Remember the “==” sign here is a logical comparison. It’s a common mistake to write `if (i=10)` here, but a single equal sign means “set the value of *i* to 10”, which is not what we want.

- c) Upload this sketch. At what point does the `break` command leave the loop? Does it exit immediately, or after the last 2-second delay?
- d) What happens when you set `annoyUser=false;` in global space? Does the *while loop* run at all?

### *Switch Case*

Although this chapter doesn’t touch on all the important tools in programming, it covers the main ideas, the last of which (for now) is *switch...case*. An *if...then...else* statement can only test one condition on its own, provide one response for *true* and another optional response for *false*. If you have a number to test with a variety of possibilities for the value, with a variety of different outcomes, you can also use *else if* statements. For instance, let’s say there are 5 “modes” in a device that is

intended to be an automated tablet QC station, and you would like the program to provide a message regarding which mode is currently selected:

```
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x27,16,2);
int choice=2;
String message=""; // initialize with empty String

void setup(){
 lcd.init();
 lcd.backlight();
 lcd.clear();
 lcd.setCursor(0,0);
 if(choice==1){
 message="Tablet sorter";
 // other commands could go here
 }else if(choice==2){
 message="Tablet weighing";
 }else if(choice==3){
 message="Colour detection";
 }else if(choice==4){
 message="Tablet hardness";
 }else if(choice==5){
 message="Friability";
 }else{
 message="Unknown mode";
 }
 lcd.print(message);
}

void loop(){
}
```

The ***switch...case*** command is ideal for this application. The ***switch...case*** command will test each condition, stop until it finds a match, and can also provide a *default action* if none of them are true—which is great if your user selects a wrong option accidentally. Let's try the same sketch again, with ***switch...case***:

```
//switch case example
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x27,16,2);
int myChoice=2;
String message="";

void setup(){
 lcd.init();
 lcd.backlight();
```

```

lcd.clear();
lcd.setCursor(0,0);
switch(myChoice) {
 case 1:
 message="Tablet sorter";
 break;
 case 2:
 message="Tablet weighing";
 break;
 case 3:
 message="Colour detection";
 break;
 case 4:
 message="Tablet hardness";
 break;
 case 5:
 message="Friability";
 break;
 default:
 message="Unknown mode";
 break;
} // end of switch.
lcd.print(message);
}

void loop(){
}

```

The *switched* variable type can be a *byte*, *int*, *long*, or *char*.

**break** commands are required after each action.

Here is the default action if none of the other cases are a match. This section is optional.

- e) Try uploading and running the above sketch, and changing the value of `choice` to 1, 5, and then 10. What happens?

## General Programming Tips

Programming is a skill that takes a little getting used to, but the good news is that once you do, the effect is transformative! The following tips will help you keep your sketches organized and potentially save you time and frustration.

- 1) Comment everywhere. Leave yourself a detailed breadcrumb trail in your work. It will save you lots of work later on.
- 2) Don't try and cram too many ideas in one line. Break complex ideas up into separate, smaller steps. Keep it simple – this will help you debug your program.
- 3) Make your variable names short and informative.
  - Come up with your own naming system. Be consistent with it.
  - Explain the purpose of a variable when you declare it, using comments, on the *same line* if possible (less chance of it getting separated). Suggest a default value or range in the comments, if appropriate.
- 4) Indent each program structure properly (if() statements, loops, etc.)
- 5) Comment what each end curly bracket belongs to. It is frustrating when a curly bracket is missing, and you have to find out where its partner is (or isn't).
- 6) Compiling a program can be your best teacher. Read the compiler errors. They are sometimes cryptic. Google what they mean if you are unsure.
- 7) Once you learn how to code in your first programming language, learning another language is MUCH easier!

The sketches throughout this text have been intentionally stripped down to the essentials with minimal commenting, to keep the code less intimidating to look at and quicker for you to copy into the Arduino IDE. The sketches provided in the appendix are better examples of proper programming etiquette.

C++ is one of the most popular programming languages in the world. You can now add “Programming in C++ (Arduino Platform)” on your resume, which will impress the reader far beyond mentioning word processing and spreadsheet programs.

### Activity 3-1: Programming Challenge

A student has put together a homemade spectrophotometer, and collected the following data for a single standard (after zeroing the spectrophotometer with distilled water):



**(% Transmission at 400 nm)**

14.25%  
 12.62%  
 12.47%  
 13.56%  
 11.78%  
 15.19%  
 18.40%

Write a sketch that:

- Defines one array to hold transmission measurements, and another array to hold the converted absorbance values.
- Calculates and reports the **mean**, **standard deviation**, and **relative standard deviation** (%RSD) of the absorbance values.

**Note:** you will need to use the following functions: `log10()`, `sq()` and `sqrt()`. We will discuss how to use these functions in class.

**Useful equations:**

$$\bar{x} = \sum_{i=1}^n \frac{x_i}{n} \quad OD = -\log(\text{Transmission})$$

$$STDEV = \sqrt{\frac{\sum_{i=1}^n (\bar{x} - x_i)^2}{n-1}} \quad \%RSD = 100 \times \frac{STDEV}{\bar{x}}$$

Transmission in decimal form (e.g. 0.2, not 20%)

**Learning Objectives for Section 3**

After having attended this class, the student will be able to:

- 1) Identify the main parts and their functions on the Arduino Uno:
  - Digital pins, analog pins, reset button, and power pins
- 2) Download and install an external Arduino library, and load it into a sketch.
- 3) Write, compile, and upload a simple sketch to the Arduino Uno MCU using the Arduino IDE platform.
- 4) Hook up an LCD serial display module to the Arduino Uno, and adjust the contrast.
- 5) Write, compile, and upload a simple sketch to print “Hello World!” to the LCD module.
- 6) Start a sketch folder, and become comfortable with the workflow of coding, compiling and uploading.

- 7) Use commenting notation to inactivate a single line of code (using `//`), a section of code (using `/*` and `*/`), and to describe/annotate lines of functional code.
- 8) Select and declare the appropriate data type for a variable, keeping in mind their structure and limits:
  - integers (`int`, `unsigned int`) → 16 bit
  - long integers (`long`, `unsigned long`) → 32 bit
  - float variables → 32 bit, 6-7 significant digits
  - bool variables → 1 bit, true or false
  - byte variables (`byte`) → 8 bit integer, 0-255
  - String and char variables, how to combine strings, UTF-8 and ASCII
  - arrays, and their quirky numbering system: declaring and accessing them
  - casting a variable as another type of variable
  - constants vs. variables
- 9) Appropriately use the three major sections of a sketch: global space, `setup()` function, `loop()` function. Understand the scope and limitations of defining a variable in each section.
- 10) Compare values inside `if()` statements with relational operators (`<`, `<=`, `>`, `>=`, `==`, `!=`)
- 11) Compare logical expressions with boolean operators (`&&`, `||`, `!`)
- 12) Properly apply the following structures while coding, and be able to identify when they terminate: *for loop*, *do...while*, *while*, *switch...case*, *break*
- 13) Write a sketch illustrating many of these important programming concepts.
- 14) Develop good coding habits and programming etiquette so that others can understand, use, modify, and build upon your work.

### Section 3 - Station Content List

- Arduino Uno
- 4 x M/F Jumpers
- 16x2 I<sup>2</sup>C Serial LCD Module

## SECTION 4

### ARDUINO PINS, AND WRITING FUNCTIONS

|                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|-------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| What You'll Be Learning | <b>Lecture:</b> Byte variables. Programming digital and analog pins. Void functions vs. functions that return a value. Power and ground pins. Maximum limits of the Uno. Introduction to serial communications. Digital input vs. output. Analog resolution, and input constraints. Wiring and reading momentary switches: pull-down and pull-up resistors, button debouncing. Various activities throughout the lecture: Button input/LED circuit. PWM example - pulsing LED. Button in - LED state toggle. Potentiometer LED brightness control. Flashing LED (using functions). Average tablet weight (using functions). |
| What You'll Be Doing    | The activities in this section are interspersed throughout the lecture. We will be trying out each concept as we go along.<br><b>Activity 4-1:</b><br>a) Build and calibrate an LCD thermometer (using a thermistor).<br>b) Set up an LED indicator based on temperature/setpoint (LED turns on below threshold temperature).<br>c) Output data in .CSV format to the serial monitor and plotter. Label and store your calibrated thermistor for next class.                                                                                                                                                                |
| Files you will need     | All course files are available for download at:<br><a href="http://pb860.pbworks.com">http://pb860.pbworks.com</a><br>• <i>Bcoeff.xlsx</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

#### Byte Variables and Digital Pins

Since we will be discussing digital pins, the *byte* variable deserves special attention here. A byte is 8 bits of data, so it can represent integers ranging from 0 to 255. You probably wouldn't want to use a byte variable for math, but on the other hand, most microcontrollers have less than 255 pins, so bytes are *excellent* for declaring and holding pin numbers without taking up lots of memory. Bytes are also handy in dealing with logic, for instance sending and receiving data from shift registers, that will be demonstrated later in class.

The Arduino Uno has 13 digital pins and 6 analog pins (but digital pins 0 and 1 are used for serial communication with your computer, so they are off limits!). So 256 numbers is more than enough for defining pin numbers on most small microprocessors.

So far, we've seen different ways of getting data into variables. Often, with byte variables, it becomes handy to be able to express numbers in binary, although for defining a pin number it isn't necessary. Let's say we want to declare a byte variable to refer to digital pin 13. We could use *any* of the following commands:

```
byte myPin=13;
uint8_t myPin=13;
const byte myPin=13;
byte myPin=0b1101;
byte myPin=0B1101;
byte myPin=B00001101;
```

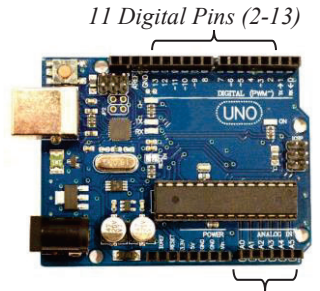
These commands will do almost the same thing: declare a byte variable called “myPin” and assign a value of 13 to it. Like other variables, we could have chosen a name other than “myPin”. Usually you want to select a variable name that gives you a hint what the pin is going to do (e.g. “sensorPin”).

**Note:** One of the conventions in Arduino is to use upper case letters in the middle of variable names, so that two words are easier to discern (e.g. pinNum, instead of pinnum). This compensates for not being allowed to put spaces in variable names, and is part of Arduino programming etiquette.

Digital pin 13 has a special job on the Arduino Uno: it has a built-in LED connected to it on the Uno's board, so that when the *state* of pin 13 is set **HIGH** (at +5V, or “ON”), the light will be on, and when the state of the pin is set **LOW**, it will be off (0V, or GND). This saves you hunting for an LED and resistor when you need one in a pinch. Let's try a simple sketch to define a byte variable for pin 13, and flash the built-in LED on the Uno. This is the first sketch that most people start with (and you don't need to hook up the LCD serial module):

```
// Example: Blink sketch
const byte ledPin=13;

void setup(){
 pinMode(ledPin,OUTPUT);//set ledPin to OUTPUT mode
}
```



6 Analog Pins (A0-A5)  
Figure 4-1. Digital and analog pins of the Arduino Uno MCU.

```
void loop() {
 digitalWrite(ledPin,HIGH); //set ledPin state HIGH
 delay(1000);
 digitalWrite(ledPin,LOW); // set ledPin state LOW
 delay(1000);
}
```

- a) Write and upload this sketch to the Uno. Why are the delays in this program? The processor speed on the Arduino is so fast (16 MHz) that if you commented out the delays, the digital pin would be turning on and off so quickly, the LED would appear to be on without blinking.

How fast is the Uno's processor?

The processor clock is a 16 MHz crystal oscillator, which drums the pace for the processor to run commands.

16 MHz = 16 million assembly instructions per second, so a simple line of code should take about  $1/16 \times 10^6$  seconds to run, or 62.5 nanoseconds. That's **FAST!**

### *What is a Digital Pin?*

A **digital pin** can *write* or *read* two states: HIGH (+5V), or LOW (0V, or ground). It can be set to three different modes: OUTPUT, INPUT, and INPUT\_PULLUP.

In **OUTPUT** mode, the state of the digital pin can be *set* HIGH or LOW. This can be used to transmit digital information, or power a small device like an LED. A digital pin set HIGH will be at +5V, and can provide up to 40 mA of current per digital pin ( $P=VI = 5V \cdot 0.04A = 0.2W$ ). A digital pin set LOW will be at 0V.

In **INPUT** mode, the digital pin can *read* the voltage and report the pin state. For an Arduino Uno running at its typical +5V logic level, if the voltage level connected to a digital pin set to input mode is higher than about +3V, then the pin will return a 1 (or HIGH) state. If the voltage level is less than about +1.5V, it will return a 0 (or LOW) state. In between these two voltages (1.5-3V), the digital pin could register a HIGH or LOW. In input mode, the pins are in a very high impedance state, meaning very little current goes in or out of them, and they are sensitive to very small changes in voltage. This also means that if the input pins aren't connected to anything, they "float"—they can randomly swing from HIGH to LOW states. This is something to keep in mind when you are reading a circuit. If you are trying to read the state of a switch, you will need a pull-down or pull-up

resistor (more on that later), or the input pin will not report the “unpushed” state of the pin faithfully.

It is important to set the digital pin to the correct mode (OUTPUT, INPUT, or INPUT\_PULLUP) before using it, with the `pinMode()` command, usually in the `setup()` function:

```
pinMode(myPin1, INPUT); //set myPin1 to INPUT mode
pinMode(myPin2, OUTPUT); //set myPin2 to OUTPUT mode
pinMode(myPin3, INPUT_PULLUP); //internal pullup
```

We will be discussing the INPUT\_PULLUP mode later in this section.

**Note:** For even faster control of digital pins, see *Bitwise Operations* and *Introduction to Port Manipulation* in Section 10.

### Digital OUTPUT Mode Example

We are going to repeat the “blink” exercise, this time using an external LED of your choice, wired in series with a 220 Ω resistor, on a breadboard:

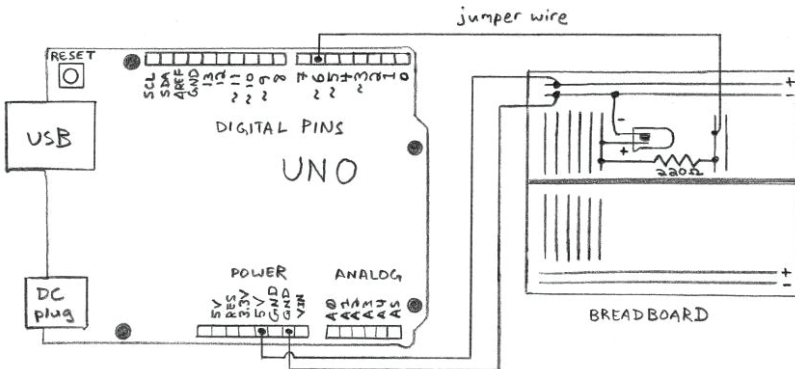


Figure 4-2. Schematic to connect an LED to the Uno’s digital pin 6, using a breadboard.

This schematic is more simply represented by the circuit diagram in Figure 4-3. Note that you can switch the positions of the resistor and LED, and the circuit will function the same (KCL).

- b) Set up the breadboard, change the ledPin number in the previous sketch to 6, then compile and upload to the Uno.
- c) Test your eyes: If you shorten the delays in the sketch, the LED will blink faster and faster, until eventually it will appear to be continuously on (although in reality, it will still be blinking). This effect is called the *persistence of vision*. What delay value does this seem to occur at for you? For many people, a typical range is 16-25 msec (20-30 Hz).

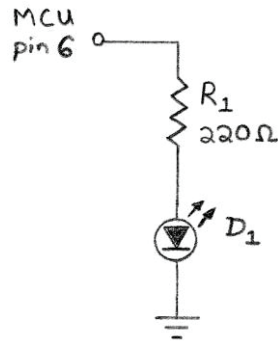


Figure 4-3. Circuit diagram for schematic in Figure 4-2.

### *Pulse Width Modulation (PWM) Example*

The Arduino Uno does not have the ability to output any voltage other than 0V or +5V (from the digital pins), or +3.3V (from the 3.3V power pin). However, since it can switch between 0 and +5V so quickly, we can dim our LED by changing the proportion of time the pin spends on 0V and +5V. By “pulsing” +5V very quickly, it will be similar to having a voltage in between 0V and +5V. If we pulse voltage quickly enough to an LED, it will make the LED appear dim. We call this a *duty cycle*. A duty cycle of 50% means that the digital pin is spending 50% of the time on, and 50% off, but switching so quickly that effectively the output can seem like 2.5V to some components. The fatter the width of the +5V pulse, the higher the “effective” voltage, and the higher the duty cycle. This strategy is called *pulse width modulation* (PWM). Many devices and applications take advantage of PWM.

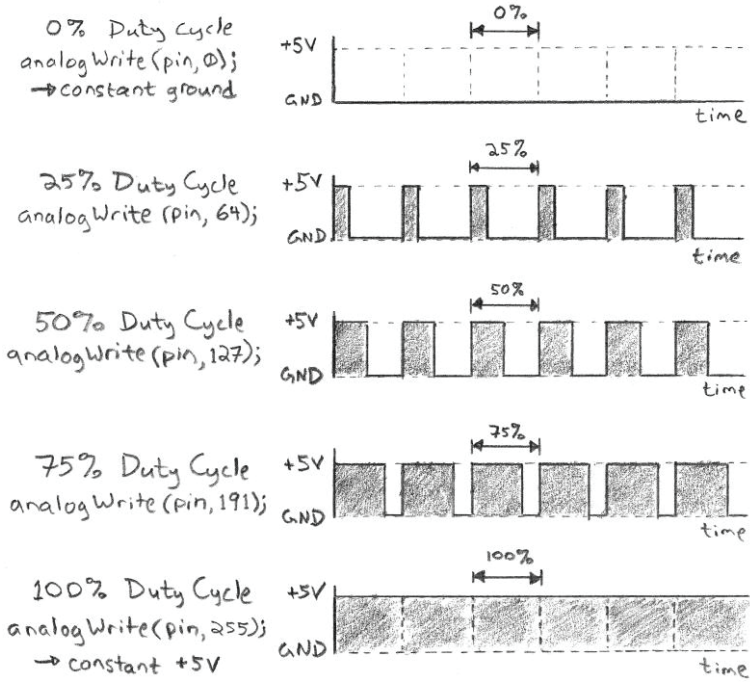


Figure 4-4. Pulse width modulation (PWM), illustrated with different duty cycles.

For the Uno, digital pins 3, 9, 10, and 11 have a PWM frequency of about 490 Hz, and pins 5 and 6 have a PWM frequency of about 980 Hz. (Mellai 2017b) Other digital pins are also capable of switching between LOW and HIGH at these speeds, but PWM-enabled pins have this as a special ability—you can set them on a PWM duty cycle with a single command. These pins will continue to switch on and off at the pulse width programmed using separate internal timers, while your sketch can go on and do other things. If you would like to generate PWM signals at other frequencies, see *Customized Frequencies for PWM* in Section 10.

d) Modify your sketch above by replacing:

```
digitalWrite(ledPin, HIGH);
```

with:

```
analogWrite(ledPin, 127);
```

Confirm that the LED flashes at *half the brightness* as before.



- e) We will be using PWM later to control motors more precisely. In the meantime, try the following sketch. It pulses the LED according to a sin curve function:

```
// Example: Fading an LED
const byte ledPin=6;

void setup(){
 pinMode(ledPin,OUTPUT);
}

void loop(){
 for(float x=0.0; x<6.28; x+=0.01){
 byte y = sin(x)*128.0 + 128.0;
 analogWrite(ledPin,y);
 delay(5); // longer delay=slower fading
 }
}
```

Define a *local* byte variable called *y*, and give it the value  $\sin(x)*128.0+128.0$ . We didn't bother casting here. *float* will be rounded down when converted to bytes.

The  $\sin(i)$  function gives a wave with a range of values from  $[-1$  to  $+1]$ . So we needed to re-map those values within the range for PWM (0-255, where 0 = 0% duty cycle, 255=100% duty cycle). The expression  $x=\sin(i)*128.0+128.0$  “remaps” the range of outputs (gain then shift) from  $[-1\dots1]$  to  $[0\dots255]$ . This useful idea will come in handy later when we discuss sensor ranges. Changing delay time will affect the fading time.

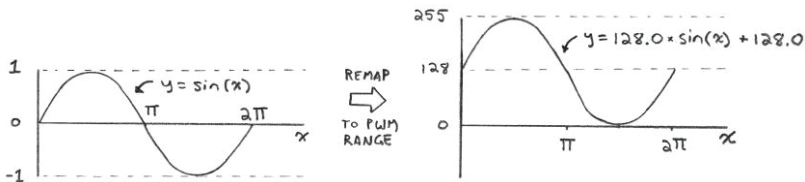


Figure 4-5. Gaining and shifting the  $\sin()$  function to the working PWM range [0-255].

### Digital Input Mode Example

In this next example, a *momentary switch* (Figure 1-33) is used to demonstrate how digital pins work in INPUT mode. A momentary switch is very flexible, since it can be programmed to behave like a toggle, and have different functions triggered by the same switch (e.g. short push = action 1, hold down = action 2, double click = action 3, hold during bootup = action 4). See *Programming One Button with Multiple Functions* in Section 9 for a more advanced example sketch.

**Background: Pull-Up and Pull-Down Resistors**

Different ways to wire a momentary switch to a digital pin are illustrated in Table 4-1. A **pull-up resistor** or **pull-down resistor** (typically 10K) prevents the digital pin from floating while the switch connected to it is open.

**Table 4-1. Wiring a momentary switch to a digital input pin.**

| <b>Momentary Switch with Pull-Down Resistor</b>        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|--------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                        | <p>A <b>pull-down resistor</b> pulls the voltage down to ground when SW<sub>1</sub> is open. Otherwise, when the switch is open, you would get erratic reads from the pin, because it would be “floating” (not connected to anything).</p> <ul style="list-style-type: none"> <li>• When SW<sub>1</sub> open, pin 7 is LOW</li> <li>• When SW<sub>1</sub> closed, pin 7 is HIGH</li> </ul>                                                                                                                                                             |
| <b>Momentary Switch with Pull-Up Resistor</b>          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|                                                        | <p>A <b>pull-up resistor</b> does the opposite: it pulls pin 7 HIGH when SW<sub>1</sub> is open. This reverses the logic.</p> <ul style="list-style-type: none"> <li>• When SW<sub>1</sub> open, pin 7 is HIGH</li> <li>• When SW<sub>1</sub> closed, pin 7 is LOW</li> </ul>                                                                                                                                                                                                                                                                          |
| <b>Momentary Switch with Internal Pull-Up Resistor</b> |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|                                                        | <p>The ATmega328 MCU has built-in (internal) pull-up resistors on its digital pins, that you can activate by setting the pinMode using the following command:</p> <pre>pinMode(myPin, INPUT_PULLUP);</pre> <p>This saves you from needing an external 10K pull-up resistor.</p> <ul style="list-style-type: none"> <li>• When SW<sub>1</sub> open, pin 7 is HIGH</li> <li>• When SW<sub>1</sub> closed, pin 7 is LOW</li> </ul> <p><b>Note:</b> Do not use Pin 13 in INPUT_PULLUP mode, as the built-in LED won't pull the pin higher than ~1.7 V.</p> |

- f) Our next activity has two parts to it: an LED circuit (with current-limiting resistor), and a momentary switch circuit. Leave the LED circuit assembled from part (e). Build the momentary switch circuit in Figure 4-6.

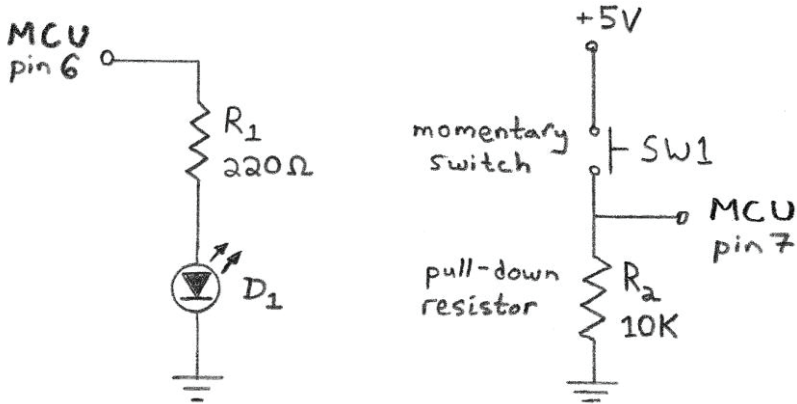


Figure 4-6. LED circuit (left) and momentary switch circuit (right).

- g) Write and upload the following sketch:

```
// Example: Change LED state with button push
const byte ledPin=6;
const byte buttonPin=7;
bool ledState=false;

void setup(){
 pinMode(ledPin,OUTPUT); //pin 6 to OUTPUT mode
 pinMode(buttonPin,INPUT); //pin 7 to INPUT mode
}

void loop(){
 if(digitalRead(buttonPin)==HIGH){ //if pushed
 ledState=!ledState; //toggle ledState
 digitalWrite(ledPin,ledState); //turn on/off led
 delay(50);
 } // end if
}
```

Pins won't work as expected without `pinMode()`.

This is called button debouncing. It gives the user time to let go of the button. Is 50 msec enough?

**Note:** The following commands would turn the LED *on*:

```
digitalWrite(ledPin,HIGH);
digitalWrite(ledPin,true);
digitalWrite(ledPin,1);
bool ledState=1; // declare ledState as 1
digitalWrite(ledPin,ledState); //use bool
```

Similarly, the following commands would turn the LED *off*:

```
digitalWrite(ledPin,LOW);
digitalWrite(ledPin,false);
digitalWrite(ledPin,0);
bool ledState=0; // declare ledState as 0
digitalWrite(ledPin,ledState); //use bool
```

## Analog Pins

The `digitalRead()` function can only report a **LOW** state if the voltage connected to the digital pin is close to zero, or a **HIGH** state if the voltage connected to the digital pin is close to +5V. However, there are situations when we require more resolution than just **LOW** or **HIGH** when measuring a voltage. The ATmega328's analog pins can read 1024 “divisions” (or *divs*) from 0 to +5V, inclusive. This resolution gives us enough information to make meaningful measurements. We need to keep measured voltages within 0 and +5V, or we risk damaging the board. Analog pins do not require a `pinMode()` command. They are just ready to do their jobs. The command to read from an analog pin is:

```
analogRead(myPin);
```

where `myPin` is A0, A1, A2, A3, A4, or A5—one of the Uno's six analog pins. You refer to them in your sketch with the letter “A” then the pin number, e.g.:

```
int reading=analogRead(A5);
```

When you use the `analogRead()` command, it doesn't tell you the voltage. The resolution of the analogPin is 10 bits ( $2^{10}=1024$  numbers). The command returns *divs* (or divisions), a number from 0 to 1023, that is linearly proportional to the voltage from 0 to +5V (0 *divs*=0V, and 1023 *divs*=+5V). You need to convert *divs* to volts if you'd like to know the actual voltage:

```
float volts=0.0; //float variable to hold volts
volts=analogRead(A0)*5.0/1023.0; //divs to volts
```

Once you convert divs to volts, you usually won't stop there. If you are reading the voltage from a sensor, you need to convert the voltage to a measurement in your unit of interest. For instance, the LM35 is an IC that puts out a voltage signal proportional to the temperature in degrees Celsius. The conversion from Voltage to Celsius is  $10 \text{ mV}/^\circ\text{C}$ , with an intercept of 0 (that means  $0 \text{ mV}=0^\circ\text{C}$ ). So the next line in the sketch might be:

```
float Celsius=volts/0.010; //volts to Celsius
```

**Note:** Avoid using pins A4 and A5 to read analog voltages if you are using any devices that require SDA and SCL pins, as these pins will conflict.

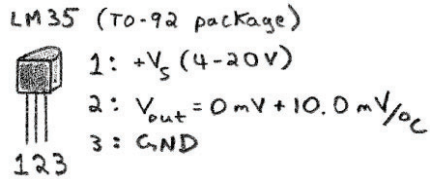


Figure 4-7. LM35 temperature sensor. (Texas Instruments Inc. 2017)

### *Using Analog Pins as Digital Output Pins*

If you run out of digital pins, and your analog pins are still free, you can set them to be digital output pins. Simply declare them to `OUTPUT` mode in `pinMode()`, and then you can use the `digitalWrite()` command on an analog pin:

```
pinMode(A0,OUTPUT); //pin A0 to digital output mode
digitalWrite(A0,HIGH); //set A0 state to +5V
```

Analog pins do not have built-in PWM capability.

### *Analog Read Example*

In this next exercise, we will set up a 10K linear potentiometer as a voltage divider, and “read” the voltage on the middle post (or “wiper”) of the *potentiometer*. We don't need to convert divs to volts in this exercise. We are reading *divs*, then rescaling that reading (0-1023) to the brightness limits of PWM (0-255).

**Background: Potentiometers**

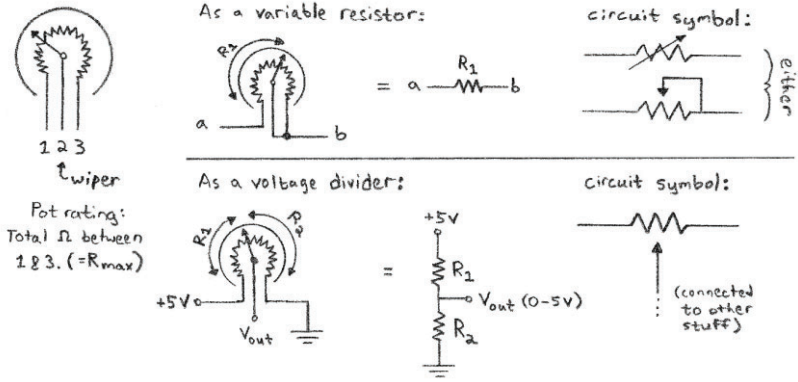


Figure 4-8. A potentiometer can be set up as a variable resistor (rheostat), or a voltage divider. The wiper (middle pin) sweeps across a length of resistive material.

h) Assemble the following circuits:

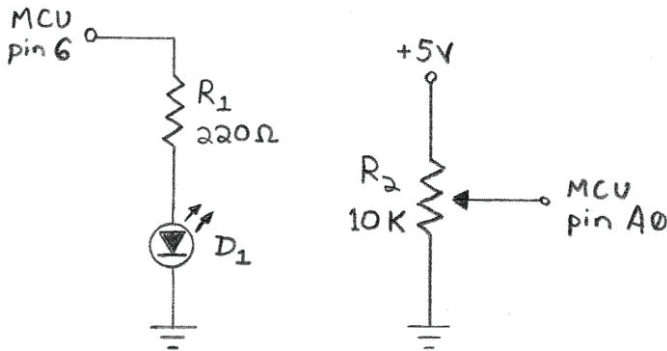


Figure 4-9. LED circuit (left) and potentiometer set up as a voltage divider (right).

i) Write and upload the following sketch:

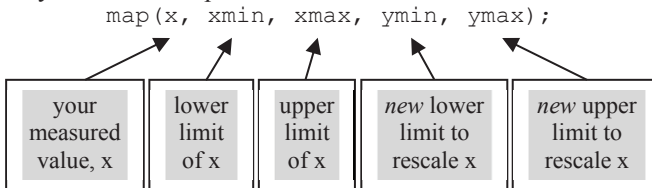
```
// Example: Fade LED with potentiometer as input
const byte ledPin=6; //ledPin is digital pin 6
const byte potPin=A0; //potPin is analog pin A0
int divRead=0; //for storing analog reading
```

```

void setup() {
 pinMode(ledPin,OUTPUT); // set ledPin as OUTPUT
}
void loop() {
 divRead=analogRead(potPin); //read div from pot
 int y=map(divRead,0,1023,0,255); //scale to PWM
 analogWrite(ledPin,y); //write PWM signal
}

```

**Note:** the `map()` function is a useful linear rescaling routine. Basically, it performs linear interpolation on integer-type variables (e.g. byte, int, and long). The syntax of the `map` function is:



The `map()` function will not scale float variables.

Mathematically, the following code would do the same thing as the `map()` command, storing the answer to the integer variable `y`:

```
int y=ymin+((x-xmin)/(xmax-xmin))*(ymax-ymin);
```

### ***External Analog Reference: AREF Pin***

The default resolution on `analogRead` is  $+5V/1024 \text{ divs} = 4.9 \text{ mV/div}$ . This reading takes  $\sim 116 \mu\text{sec}$ , so you can make up to about 8,600 readings per second (8.6 kHz). This is also called *sampling at 8.6 kHz*. See *Worked Example: Fast Analog Read* in Section 10 if you would like to speed this up.

If you would like increased resolution (less mV per step), you can apply the 1024 steps to a narrower range (e.g. 0-3.3V) by physically connecting the upper limit voltage you want to the Uno's analog reference (**AREF**) pin (e.g. connect a jumper from the 3.3V pin to AREF). Then in the `setup()` function, add the following command:

```
analogReference(EXTERNAL);
```

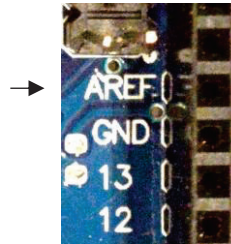


Figure 4-10. The AREF pin lets you set the voltage of the highest div (1023). Range: 0-5V.

The resulting resolution for AREF=3.3V would then be  $+3.3\text{V}/1024$  divs=3.2 mV/div, ranging from (0-3.3V). The highest div (1023) will map to 3.3V. As with other pins on the Uno, voltages connected to AREF should not be outside [0.0-5.0V]. External voltages outside this range could damage the board. (Mellai 2017c)

If you use AREF, remember to adjust your conversion from divs to volts. For example, with a 3.3V analog reference:

```
float volts=divs*3.3/1023.0; //divs->volts,AREF=3.3V
```

A common mistake is declaring the `analogReferene(EXTERNAL);` command, and then forgetting to connect an external voltage to the AREF pin. If this happens, the AREF pin will be left in a floating state, producing very erratic readings.

## Arduino Pin Conflicts

Many pins on the Arduino Uno are multifunctional, and you can accidentally run into conflicts while allocating pins. For instance, if you use an LCD module in your project, it operates using the I<sup>2</sup>C communication pins SDA and SCL. These are wired in parallel with analog pins A4 and A5, respectively, so you can't take analog readings from these pins while using the LCD module. Similarly, MOSI, MISO, and SCK (also called CLK on some boards) are wired to pins 11, 12, and 13 respectively. How would you know this? You can look up the microprocessor datasheet (Atmel Corporation 2016) or check out a pin-out diagram for the Arduino Uno (provided in Figure A-5). These types of diagrams are invaluable when you are trying to figure out connections to a microcontroller.



## Arduino Digital and Analog Pins: Summary Tables

**Table 4-2. Digital pin summary table.**

|                                       |           |                                                                                                                                                                                                                                                                                                                        |
|---------------------------------------|-----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Digital Pin:<br><b>INPUT</b><br>Mode  | Code:     | <code>pinMode(myPin, INPUT);</code><br><code>pinMode(myPin, INPUT_PULLUP);</code><br><code>digitalRead(myPin);</code>                                                                                                                                                                                                  |
|                                       | Used for: | <ul style="list-style-type: none"> <li>• Receiving information (0 or 1)</li> </ul>                                                                                                                                                                                                                                     |
| Digital Pin,<br><b>OUTPUT</b><br>Mode | Code:     | <code>pinMode(myPin, OUTPUT);</code><br><code>digitalWrite(myPin, HIGH);</code><br><code>digitalWrite(myPin, LOW);</code><br><code>analogWrite(myPin, #);</code><br>(# is an integer from 0-255,<br>0=0% duty cycle, or LOW<br>127=50% duty cycle, or HIGH half the time<br>255=100% duty cycle, or HIGH all the time) |
|                                       | Used for: | <ul style="list-style-type: none"> <li>• Sending information (0 or 1)</li> <li>• Powering low-power devices (&lt;0.2 W)</li> <li>• Serial communications (e.g. I<sup>2</sup>C)</li> <li>• PWM (Pulse-Width Modulation), pins 3,9,10,11: 490 Hz, pins 5,6: 980 Hz.</li> </ul>                                           |

**Table 4-3. Analog pin summary table.**

|                                      |           |                                                                                                                                                                                                                                                                                                                                                                                                          |
|--------------------------------------|-----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Analog Pin,<br><b>INPUT</b><br>Mode  | Code:     | <ul style="list-style-type: none"> <li>• No need to declare <code>pinMode()</code></li> <li>• <code>analogRead(myAnalogPin);</code></li> <li>• To use an external analog reference:<br/><code>analogReference(EXTERNAL);</code></li> <li>• Remember to plug a voltage into the AREF pin to use an external analog reference.</li> <li>• <code>pinMode(myPin, INPUT_PULLUP); //optional</code></li> </ul> |
|                                      | Used for: | <ul style="list-style-type: none"> <li>• Receiving a voltage.<br/>(0→1023, maps to 0→5V, or 0→AREF)</li> </ul>                                                                                                                                                                                                                                                                                           |
| Analog Pin,<br><b>OUTPUT</b><br>Mode | Code:     | <code>pinMode(myAnalogPin, OUTPUT);</code><br><code>digitalWrite(myPin, HIGH);</code><br><code>digitalWrite(myPin, LOW);</code>                                                                                                                                                                                                                                                                          |
|                                      | Used for: | <ul style="list-style-type: none"> <li>• Same as digital pin in OUTPUT mode (you can use analog pins if you run out of digital pins).</li> <li>• Not PWM-capable.</li> </ul>                                                                                                                                                                                                                             |

## The Serial Monitor

The Arduino IDE's built-in *serial monitor* is a faster and more useful alternative to the LCD serial module. The serial monitor can send and receive data between your PC and microprocessor, in real time. Serial communications can slow down your sketches, but while you are writing your code, serial messages really help to “see” what’s going on in the microprocessor. After you are done debugging, if serial messages are not part of your final program then you can comment out all of the serial commands and leave them for future debugging. Using the serial monitor is easy. First, you need to initialize the serial monitor in the `setup()` function using the command:

```
Serial.begin(9600); //start serial monitor
```

The number “9600” is the baud rate. This means that 9600 bits are transmitted in 1 second (1 baud = 1 bit per second, or bps). A range of baud rates are possible (see Table 4-4). A baud rate of 9600 is a very common first choice. You would select a faster baud rate if serial communications are slowing down your sketch, and high speed is essential (e.g. 3D printing software is typically set to 250,000 baud, although it is more advisable to use an SD card). Writing to the serial monitor is very similar to writing to the LCD screen. The two main commands are:

```
Serial.print("Hello world"); //prints txt
Serial.println("Hello world"); //prints txt+return
```

The first command `Serial.print()` prints “Hello world” to the serial monitor. The second command `Serial.println()` also prints “Hello world”, then advances to a new line afterwards. Like the `LCD.print()` command, you can combine strings. You can print the values stored inside variables to the serial monitor by not enclosing the variable name in quotes.

- j) We will edit the previous sketch to illustrate how to use the serial monitor. The lines in bold are the new lines added:

```
// Example: Printing to the serial monitor
const byte ledPin=6; //ledPin is digital pin 6
const byte potPin=A0; //potPin is analog pin A0
int divRead=0; //for storing analog reading
```

**Table 4-4.**  
**Selectable baud rates (bps) in the Arduino IDE serial monitor.**

|       |        |
|-------|--------|
| 300   | 38400  |
| 1200  | 57600  |
| 2400  | 74880  |
| 4800  | 115200 |
| 9600  | 230400 |
| 19200 | 250000 |

```

void setup() {
 pinMode(ledPin,OUTPUT); // set ledPin as OUTPUT
 Serial.begin(9600); //start serial monitor 9600bps
 Serial.println("Voltage in divs"); // print string
}

void loop() {
 divRead=analogRead(potPin); //read divs from pot
 Serial.println(divRead); //print divRead to serial
 int y=map(divRead,0,1023,0,255); //scale to PWM
 analogWrite(ledPin,y); //write PWM signal
}

```

Write and upload this sketch, then press **Ctrl+Shift+M** (for Windows) or **⌘+Shift+M** (for MacOS) to start the serial monitor. Make sure the drop-down menu on the serial monitor window is set to the correct baud rate. Try twisting the potentiometer while watching the measurements scroll by. Is the display moving too quickly? Try adding `delay(100);` somewhere in the `loop()` function.

For more tips and tricks with `Serial.print()` and `Serial.println()` commands, see *Serial.print()*, *Serial.println()*, and *Serial.write(): Escape Sequences and Advanced Output Options* in Section 10.

## The Serial Plotter

The *serial plotter* is another useful built-in feature, which draws numbers as they are received from your device to an autoscaling graph. To start the serial plotter, first close the serial monitor, then press **Ctrl+Shift+L** (for Windows) or **⌘+Shift+L** (for MacOS). Try using the serial plotter. What do you notice about the axis scaling as the plotter runs?

## Subroutines and Functions

So far, we've written commands in three areas that exist in every sketch:

- Global Space (at the top of a sketch, and outside other functions)
- Inside the `setup()` function (runs only once)
- Inside the `loop()` function (loops continuously)

However, programming is much easier when we can write our *own* functions, so we don't have to rewrite the same commands repeatedly. This makes our code easier to review and modify. We can also build up libraries

of our own useful functions, which can speed up what we need to do considerably.

### *Properties of Functions*

Functions have access to global variables, but not local variables defined in *other* functions (e.g. variables declared in the `setup()` and `loop()` functions).

In order to be accessible to your entire sketch, functions are declared in global space, *before, between, or after* (but not inside) the `setup()` or `loop()` functions.

Any variables you declare inside a function will be *local* to that function. In other words, a local variable will only exist inside that function, while the function is running. Once the function stops running, the variable will be forgotten.<sup>3</sup> However, as you will see, you can pass variables (and values) into and out of functions easily, and any global variables are accessible within the function you write.

It is a good idea to make a function completely independent from the rest of the sketch, so you can easily copy it into other sketches, without too much hassle. Although a function has access to global variables, if you can avoid referring to them directly, you won't have to declare the same global variables in a new sketch.

**Table 4-5. Common types of functions.**

| <i>Function Type:</i> | <i>Function Returns:</i> |
|-----------------------|--------------------------|
| void function         | nothing                  |
| bool function         | a bool                   |
| byte function         | a byte                   |
| char function         | a character              |
| int function          | an integer               |
| long function         | a long integer           |
| float function        | a float                  |
| String function       | a String                 |

### *Void Functions*

The simplest type of function is a ***void function***, called “void” because it returns nothing. This is often called a “subroutine” in other programming languages. A void function is simply a different place to bundle a series of commands, so that every time you call that function, the commands inside the function are “executed”. This means the code in the function runs.

---

<sup>3</sup> If you declare a ***static variable*** inside a function without initializing it (e.g. `static int myvariablename;`), then its last stored value will be remembered next time it runs. However, it will not be accessible by other functions, or in global space.

- k) Let's try writing a void function that flashes your LED on pin 6, twice:

```
// Example: Writing a function to flash an LED
const byte ledPin=6;
```

```
void setup(){
 pinMode(ledPin,OUTPUT);
}
```

```
void loop(){
 flashLED();
}
```

This is how we *call* a void function. See how much cleaner the loop() function looks now?

```
void flashLED(){
 for(int i=0;i<2;i++){
 digitalWrite(ledPin,HIGH);
 delay(500);
 digitalWrite(ledPin,LOW);
 delay(500);
 }
 delay(2000);
}
```

We defined our function *after* the loop() function; however, as long as it is outside the setup() and loop() functions, it will be in global space.

Although this function works, it's not very portable. For instance, if it were copied into another program, you would have to make sure the byte variable ledPin was defined in global space, or the sketch wouldn't compile. Also, wouldn't it be fun to specify the number and speed of flashes, when we call the function? Try re-writing the *void function* like this:

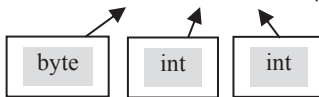
```
void flashLED(byte fPin, int fnum, int fwait){
 // flashLED flashes an LED.
 // fPin: pin# to flash (e.g. 6)
 // fnum: #flashes (e.g. 5)
 // fwait: msec delay betw flashes (e.g. 250)
 for(int i=0;i<fnum;i++){
 digitalWrite(fPin,HIGH);
 delay(fwait);
 digitalWrite(fPin,LOW);
 delay(fwait);
 }
 delay(2000);
}
```

Make sure you define the *variable types* for each input argument, and separate them with commas. All the variable types we discussed (int, float, long, String, char, boolean, etc.) can be input arguments.

We have defined *input arguments* for the function `flashLED()`, so that when you call the program, you can specify the pin number (`fPin`), the number of flashes (`fnum`), and the delay between flashes (`fwait`). Those input arguments only have meaning within the function. They are just placeholders. The type of variable you enter as an input argument must match the variable type of the input argument when you call the function, or the program won't compile.

To *call* this function, just add the values you would like to use when you run `flashLED()` in the sketch:

```
flashLED(ledPin, 5, 250);
```



This is how we *call* a void function with *input arguments*. When you call the function, make sure your argument variable types match the ones you defined in the function (or *cast* them so they match).

This function call will flash digital pin 6, 5 times, with a 250 msec delay between flashes. Try changing the input arguments to see how calling works. We can now copy this function to *any* sketch, and this *void function* will still compile and work the same way, because it doesn't access any global variables.

### *Call-by-Value vs. Call-by-Reference*

When we called the `flashLED()` void function in the line above, the input arguments were *call-by-value*. This means the function makes a local copy of the input arguments, uses those values while the function is running, then destroys the values when it is finished. Each original variable passed to the function is not affected. If you pass a number to a function (as we did above), then there is really nothing to talk about—this is the proper way to do it. However, if you pass a variable to the function and you try to *change* the contents of the variable *while the function is running*, then the original variable will never get changed, only the local copy.

If you would like to change the original variable within a function, you can either leave it out of the argument line, make it a global variable, and change it as usual, or you can do something called *call-by-reference*. To call by reference, put an “&” sign in front of the variable name when you are defining the function's input arguments. This gives the function the power to change the value inside the original variable. This idea is illustrated in the following two sketches. The sketch on the *left* tries to change the pin number within the function, but it can't, because it only changes the local

“copy” inside the function. The LED on pin 13 never lights up. The sketch on the *right* calls by reference (&fPin). It can now change the original ledPin variable, so the first time the function is called, it will flash the LED on pin 6, and from then on, it will flash the on-board LED on pin 13. *The only difference between the two sketches is the “&” sign.*

**Table 4-6. Example sketches for call-by-value (left) vs. call-by-reference (right).**

| <i>Call-by-Value:</i>                                                                                                                                                                                                                                                                                          | <i>Call-by-Reference:</i>                                                                                                                                                                                                                                                                                           |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>byte ledPin=6;  void setup(){   pinMode(ledPin,OUTPUT);   pinMode(13,OUTPUT); }  void loop(){   flashLED(ledPin); }  void flashLED(byte fPin){   for(int i=0;i&lt;10;i++){     digitalWrite(fPin,HIGH);     delay(100);     digitalWrite(fPin,LOW);     delay(100);   }   delay(2000);   fPin=13; }</pre> | <pre>byte ledPin=6;  void setup(){   pinMode(ledPin,OUTPUT);   pinMode(13,OUTPUT); }  void loop(){   flashLED(ledPin); }  void flashLED(byte &amp;fPin){   for(int i=0;i&lt;10;i++){     digitalWrite(fPin,HIGH);     delay(100);     digitalWrite(fPin,LOW);     delay(100);   }   delay(2000);   fPin=13; }</pre> |

**Note:** *Arrays* are call-by-reference by default, so you don’t need to use the “&” sign when defining an array as a function input argument.

### *Float Functions*

A *float function* works *exactly* the same as a void function, with one notable exception: a float function *returns* a float number when it is finished. To return a number from any type of function, usually the last line is “return myResult;” (where myResult is a value, expression, or variable you want returned). The following example sketch defines a function that adds two float numbers together, and returns their sum as a float number:

```
float answer=0.0;

void setup(){
 answer=addNums(2.0,1.0);
}

void loop(){
}

float addNums(float a, float b){
 return a+b;
}
```

Our simple float function.

When the program runs the return command, it leaves the function at that line, regardless of whether or not there is any code after. You can also use the return command on its own (e.g.: return;), to exit a void function early. It is similar to the break command, but is used to exit a function, not a loop.<sup>4</sup> Let's go back to our tablet weight example sketch from Section 3, and this time, write a function that calculates (and returns) the average tablet weight:

```
float tabletWeight[5]={0.352,0.314,0.387,0.343,0.308};
float avgWeight=0.0;

void setup(){
 Serial.begin(9600);
 avgWeight=getAvg(tabletWeight,5);
 Serial.print("Average weight: ");
 Serial.println(avgWeight,4);
}

void loop(){
}

float getAvg(float gWeight[], int gnum){
 float total=0.0;
 for(int i=0;i<gnum;i++){
 total=total+gWeight[i];
 }
 total=total/(float)gnum;
 return total;
} // end bracket for getAvg
```

Array doesn't need brackets here.

Function call. Store the number returned by getAvg() to avgWeight.

Print avgWeight with 4 decimals to serial monitor.

An array can be an input argument.

Local float variable, only exists inside the function getAvg()

Casting the int gnum as a float.

This is the return statement, which returns a float number (average tablet weight).

<sup>4</sup> In fact, you can even use the return; function to exit the main loop function, instead of the while(1); command mentioned in Section 3.



- l) Write and upload this sketch. What's the average tablet weight?

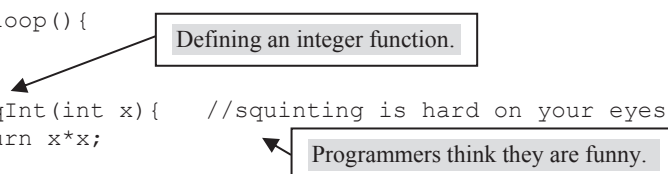
### *Integer (and other) Functions*

How do you write an integer function? You write it the same way as a float function, only you define the function with `int` instead of `float`. The following example sketch has a function that squares an integer. The function is called the same way as a float function.

```
void setup() {
 Serial.begin(9600);
 int answer=sqInt(13); // sqInt() called here.
 Serial.println(answer);
}

void loop() {
}

int sqInt(int x) { //squinting is hard on your eyes
 return x*x;
}
```



Use the same syntax for writing String and bool functions: just specify the required variable type before the function name. Functions are so useful, that after you get the hang of writing them, you won't be able to *function* without them!

### *Function DOs and DON'Ts*

You may notice that in the last sketch, the integer `answer` was declared on the same line as the function was called:

```
int answer=sqInt(13); // sqInt() called here.
```

By declaring `answer` inside the `setup()` function, its contents will be remembered as long as the `setup()` function is still running, so the next line where `answer` is printed to the serial monitor will work. Declaring variables on the fly like that may look careless, but is actually more memory efficient than declaring `answer` as a global variable, and fine as long as you don't try to access `answer` from outside the `setup()` function. You can call a function the same way inside the `loop()` function—and the local variable you declare to store the answer will be remembered for the remainder of code in the loop.

Table 4-7 summarizes some useful tips in writing successful functions. Make sure you comment your functions to remember why you wrote them,

and how they work. This will help keep track when you are looking at your code later, wondering what it was you were thinking.

**Table 4-7. DOs and DON'Ts in writing functions.**

| <i>DOs</i>                                                                                                                                                                                                        | <i>DON'Ts</i>                                                                                                                                                                                                                                  |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Come up with a system for naming function input argument variables. Notice I used the prefix “f” for input argument variables to remind me they belong locally to the flashLED function, not to global space.     | Don't name your arguments the same name as your global variable names. It makes your code confusing. Besides, if they are global variables, they don't need to be arguments.                                                                   |
| If possible, make your function independent from the rest of the sketch. You can refer to global variables, but they are better off passed into the function through arguments, so the function is more portable. | Don't refer to local function variables outside the function they are defined in.                                                                                                                                                              |
| Make sure when you call your function, your input argument types match the way you defined them in the function. If a type doesn't match, cast it (or change the argument type).                                  | Don't try to change the value of an input argument within the function, unless you <i>call-by-reference</i> , or unless it's an array. Consider making that variable global, rather than passing it to the function through an input argument. |
| Give your functions short and descriptive function names and argument variable names, to help you understand what's going on later.                                                                               |                                                                                                                                                                                                                                                |
| <i>Comment</i> your functions: describe what the function does, and what each local variable is. Give example default values in comments.                                                                         |                                                                                                                                                                                                                                                |

### #define and #ifdef Statements

One functionality in C++ programming that we won't spend time on, but you should be aware of, is the **#define** statement. This type of statement is called a *preprocessor directive*. It is only used in global space, usually at the very top of your sketch. The **#define** statement doesn't occupy any memory at all in the processor, it's just an instruction for the precompiler. For instance, some programmers like to use **#define** for defining pin numbers, instead of byte variables. To illustrate, a sketch that blinks an LED on pin 13 could be:

```

#define LEDPIN 13

void setup() {
 pinMode(LEDPIN, OUTPUT);
}

void loop() {
 digitalWrite(LEDPIN, HIGH);
 delay(500);
 digitalWrite(LEDPIN, LOW);
 delay(500);
}

```

There is *no equal sign* when using a `#define` statement, and *no semicolon* at the end of the line. You can still use comments after a `#define` statement if you like.

How is this different from defining a byte variable called `ledPin`? The difference is that the variable `LEDPIN` in this case does not exist, and its value (once defined) cannot be changed. *Before* the program is compiled, the compiler will replace all instances of `LEDPIN` with the number 13, as if you had never written the word `LEDPIN` at all. It will do this even if you have the word `LEDPIN` as part of another variable name, so be careful how you decide to use it! If you tried the following command in the middle of the sketch:

```
LEDPIN=7;
```

the compiler will interpret this as: “13=7” and produce an error. Although you can use lower case letters in `#define` statements, the convention is to use all caps, to remind us that they aren’t real variables.

The `#define` statement can also be used to define strings, short and simple one-line functions, and any other thing you might not want to repeat in a sketch. For example, the following are valid `#define` commands:

```

#define DEBUG //comment out to disable DEBUG mode
#define WARNINGMSG "Your freezer is too warm."
#define PI 3.14159 //pi to enough decimals
#define WAITASEC delay(1000) //alias for wait 1 sec
#define CIRCUMF(r) (PI*2*r) //create a function

```

You can use defined terms in other `#define` statements, as we did with `PI` in `CIRCUMF(r)`. Also notice how `(r)` is treated as a function *input argument*. Once defined, you can use these definitions in a sketch, just as you would normal variables or functions:

```

Serial.println(WARNINGMSG); //send me a warning
float circleArea=PI*100.0; //area of circle diam=10
WAITASEC; //delay 1 second
float answer=CIRCUMF(2.5); //circumference, r=2.5

```

You can use `#ifdef` statements in *anywhere else* in your sketch, based on any terms you created with `#define`:

```
#ifdef DEBUG // if DEBUG is defined (as above)
 Serial.println("Debug mode is on.");
 // Other commands could go here.
#else // else is optional here
 Serial.println("Debug mode is off.");
 // Other commands could go here.
#endif
```

These are treated as pre-compiler instructions. When you compile the sketch, these commands are handled first. This can help you customize what you would like the compiler to put together, save memory, and drastically alter how your sketch compiles just by commenting out a single `#define` statement. This can make your sketch more flexible, and take up less memory in the microprocessor. Similarly, `#ifndef` paired with `#endif` (and optionally, `#else`) will test if a term has *not* been defined. See <http://www.cplusplus.com/doc/tutorial/preprocessor/> for other great examples of preprocessor directives.

### ***General Programming Etiquette***

The overarching goal of a good sketch is clarity. The following checklist will help you polish your final sketch so that others (or perhaps you, several years later) can follow and understand what you did, and how you compiled your code.

#### ***Programming Checklist***

- Title header identifying program name, Arduino IDE version compiled, author, date, and purpose of program.
- Identify authors and sources for any libraries (and their version numbers) used (e.g. Github links).
- Provide comments detailing any specific wiring or pin allocations.
- Declare appropriate variable types (global vs. local, location, byte int float String, etc.)
- Select useful/detailed variable names.
- Provide brief descriptions of each variable defined in line comments.
- Provide example/default values of parameters in line comments.
- Provide comments for important programming lines and functions.
- Use proper syntax that compiles well and gets you the “right answer”.
- Use proper indenting/spacing for functions, `if()` statements, and loops (two spaces in for each new level).
- Give proper credit for any routines or algorithms used from other sources.

## Activity 4-1: NTC Thermistor Circuit

**Background:** Temperature control is critical in many pharmaceutical processes. For instance, a 1 °C change in temperature can result in a 10% change in the viscosity of a liquid. There are many tests (e.g. USP dissolution) where a 37°C water bath is required. Accelerated product stability testing is another important example where temperature and humidity are controlled.

A *thermistor* provides an accurate and inexpensive way of measuring and monitoring temperature. There are two main classes of thermistors: *NTC thermistors* (for “Negative Temperature Coefficient”) and *PTC thermistors* (for “Positive Temperature Coefficient”). The resistance of an NTC thermistor *decreases* as temperature rises. The resistance of a PTC thermistor *increases* as temperature rises. (Taranovich 2011)

**Table 4-8. Advantages and disadvantages of thermistors.**

| <i>Thermistor Advantages:</i>                                                                                                                                                                                                                                                                 | <i>Thermistor Drawbacks:</i>                                                                                                                                                    |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>• Very inexpensive, easy to waterproof</li> <li>• Operate over a wide range of voltages</li> <li>• Don’t require an amplifier</li> <li>• Very accurate (<math>\pm 1\%</math>, or <math>0.25^\circ\text{C}</math>)</li> <li>• Robust/durable</li> </ul> | <ul style="list-style-type: none"> <li>• Need to be individually calibrated</li> <li>• Sluggish response to temperature change</li> <li>• Narrower temperature range</li> </ul> |

Other popular ways of measuring temperature include:

- **Thermocouples** - very wide temperature range, very fast response, but they can be fragile, and require a separate module/amplifier to read. Non-linear temperature response.
- **RTD sensors** (for “Resistive Temperature Detectors”). Resistance *increases* as temperature rises. These are the most accurate and cover the widest temperature range, but are more expensive and have a slower response time. Linear temperature response.
- **ICs** (e.g. LM35)–easy to use, linear response with temperature, but narrow temperature range.
- **Infra-red beam** - very fast, but less accurate.

**Goal:** In this activity, we will be performing a 2-point calibration on a 10 k $\Omega$  NTC thermistor at room temperature (point 1), and in an ice water bath (point 2), using an ohmmeter. We will then make an Uno thermometer, that turns on an LED when the measured temperature falls below a specific value. We will then view the data on the serial monitor and plotter.

**Materials:**

- 1 x Digital Multimeter
- 1 x Breadboard
- 1 x 220Ω Resistor
- 1 x 10K (1% tol) Resistor
- 1 x 10K (5% tol) Resistor
- 1 x 10K Thermistor
- 1 x LED
- 8 x Male/Male Jumpers
- 2 x Alligator Clip Wires
- 1 x Beaker with Ice Water
- 1 x Glass Thermometer

**Procedure:***Calibrating a Thermistor*

- 1) Using an ohmmeter, measure and record the resistance of the thermistor at room temperature. Record room temperature using a glass thermometer. The resistance of the thermistor should be in the vicinity of 10 kΩ at room temperature. Standing water will provide a more accurate and stable thermistor resistance measurement.
- 2) Immerse the thermistor probe completely in an ice water bath. Leave the thermistor wires and connector hanging outside the bath. Allow the temperature to equilibrate for at least 2 minutes.
- 3) Measure and record the resistance of the thermistor while it is immersed in the ice bath. The resistance should now be noticeably higher.

*Two-Term Exponential Thermistor Equation*

The relationship between temperature and thermistor resistance is nonlinear. A popular mathematical relationship describing the shape of the temperature/resistance curve for a thermistor is the ***two-term exponential thermistor equation***: (Chen 2009, 1103-1111)

$$\frac{1}{T} = \frac{1}{T_0} + \frac{1}{B} \ln\left(\frac{R}{R_0}\right) \quad \text{or alternately,} \quad R = R_0 \exp\left[B\left(\frac{1}{T} - \frac{1}{T_0}\right)\right]$$

where:

T is the measured temperature (in Kelvin);

T<sub>0</sub> is a reference temperature (in Kelvin);

B is the temperature coefficient of the thermistor;

R is the measured resistance of the thermistor at T (in Ω);

R<sub>0</sub> is the measured resistance of the thermistor at T<sub>0</sub> (in Ω).

By measuring the resistance of the thermistor at two different temperatures, we can re-arrange the *two-term exponential thermistor equation*, to solve for B:

$$\frac{1}{B} = \frac{\frac{1}{T} - \frac{1}{T_0}}{\ln\left(\frac{R}{R_0}\right)}$$

$$B = \frac{\ln\left(\frac{R}{R_0}\right)}{\left(\frac{1}{T} - \frac{1}{T_0}\right)}$$

$$(1) \quad B = \frac{\ln\left(\frac{R}{R_0}\right)}{\left(\frac{1}{273.15} - \frac{1}{23.0+273.15}\right)}$$

$R$  = Measured resistance of thermistor at 0 °C

$R_0$  = Measured resistance of thermistor at room temperature

$T_0$  = room temperature

Download the spreadsheet *Bcoeff.xlsx* from the course website. This spreadsheet calculates the temperature coefficient using the above equation for your thermistor, based on your measured resistance values at room temperature and 0 °C. This spreadsheet will save you time, so that you can focus your efforts on coding the sketch.

After you solve for B, you can use the two-term exponential thermistor equation to convert *any* measured resistance to temperature:

$$(2) \quad T(^{\circ}\text{C}) = \left[ \frac{1}{T_0} + \frac{1}{B} \ln\left(\frac{R}{R_0}\right) \right]^{-1} - 273.15\text{K}$$

In order to read the resistance of the thermistor, we don't use an ohmmeter in a circuit (although we could!). The Arduino doesn't measure resistance, but the analog pins can measure voltage. So, we can set the thermistor up in a voltage divider, by wiring the thermistor in series with a fixed resistor of approximately equal value, then reading the *voltage* across the thermistor. The fixed resistor in this context is sometimes called the *sense resistor*.

m) Build the following circuit:

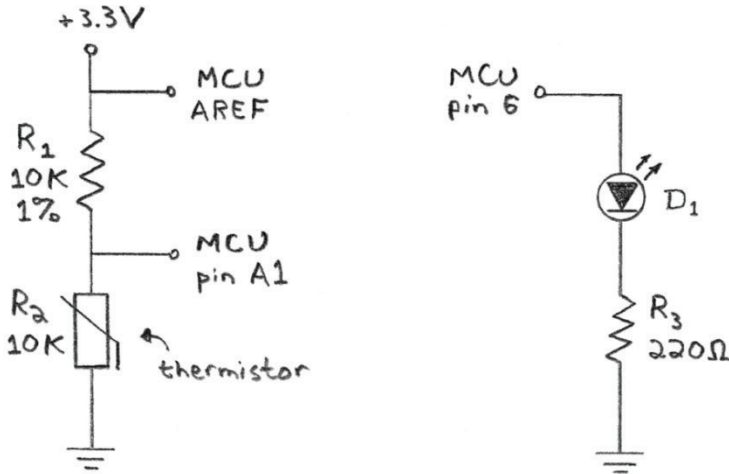


Figure 4-11. Thermistor + LED circuit. Resistor  $R_1$  is the *sense resistor*.

We will use the LED circuit on the right as an indicator light. In this circuit, analog pin A1 will measure the voltage ( $V_{out}$ ). We will need to convert the voltage to a resistance value. We now need the **voltage divider equation** for this task (see *The Voltage Divider Equation*, Section 1):

$$V_{out} = V_{in} \left( \frac{R_2}{R_1 + R_2} \right)$$

We are putting 3.3V into the divider ( $V_{in}=3.3V$ ), and we are using a very accurate 10 k $\Omega$  ( $\pm 1\%$ ) resistor ( $R_1=10K$ ). There are two benefits for supplying +3.3V to the voltage divider instead of +5V. The first benefit is that we can use the AREF pin, and read the temperature at a higher resolution (3.2 mV/div). An added benefit is that we are taking advantage of the on-board 3.3V voltage regulator on the Arduino Uno, so regardless of how we power the board (external 12V supply, or USB-powered by a laptop), the voltage supply to the thermistor circuit will be a regulated 3.3V, and less sensitive to small changes in supply voltage. Otherwise, you may find that you need to re-calibrate your circuit, because you switch power supplies on your Arduino Uno (e.g. from the USB port from your laptop to a 12V DC adapter).

Re-arranging the voltage divider equation to solve for  $R_2$ , the resistance of the thermistor:

$$(3) R_2 = \frac{V_{out}R_1}{V_{in}-V_{out}} = 10,000\Omega \left( \frac{V_{out}}{3.3V-V_{out}} \right)$$



We can then write a *float function* that converts an analog reading from pin A1 to a temperature. The function should:

- 1) Convert *divs* to *volts*; (Recall that  $\text{volts} = \text{divs} \times 3.3\text{V} / 1023.0$ , when  $\text{AREF} = 3.3\text{V}$ .)
- 2) Using equation (3), convert *volts* to a resistance value, using the rearranged voltage divider equation to solve for  $R_2$ ;
- 3) Using equation (2), convert the resistance to a temperature, using the two-term exponential thermistor equation;
- 4) Return temperature, in  $^{\circ}\text{C}$ .

Try writing this sketch on your own. Remember to connect a jumper from the 3.3V pin on the Arduino Uno to the AREF pin, and use the following command in your `setup()` function:

```
analogReference(EXTERNAL);
```

**Hint:** `log()` is the natural logarithm function in C++.

- 5) Modify your sketch to *turn on the LED* using digital pin 6, if the measured temperature is less than  $25^{\circ}\text{C}$ .
- 6) Label and store your thermistor for the next section. You will be using the same thermistor in Activity 5-1. Record your calibration data in Table 4-9.

**Table 4-9. Table for Experimental Results in Activity 4-1.**

| <i>Parameter</i>  | <i>Description</i>                                        | <i>Value</i> |
|-------------------|-----------------------------------------------------------|--------------|
| $R_0$             | Resistance of thermistor at room temperature, in $\Omega$ |              |
| $R_{\text{cold}}$ | Resistance of thermistor in ice water, in $\Omega$        |              |
| $T_0$             | Room temperature, in Kelvin                               |              |
| B                 | B coefficient of thermistor                               |              |

### Program Improvements:

- 7) Modify the sketch to report the average of five temperature readings, instead of a single reading.
- 8) Format your output to the serial monitor in .CSV format (comma separated values), e.g.:

```
Volts (V), Resistance (Ohm), Temperature (degC)
1.72, 10940.70, 24.23
1.70, 10562.24, 25.24
1.68, 10317.46, 25.91
1.66, 10117.88, 26.48
```

If you don't have enough time to finish your sketch during class, try to finish it at home.

**BONUS:** The `map()` function only works with integers. Try writing a float function that remaps a *float* number from one float range to another. Apply it to our sin curve function, for our pulsing LED. In other words, instead of:

```
x=sin(i)*127.5+127.5;
```

Write a function that can be called like this:

```
x=fmap(sin(i), -1.0, 1.0, 0.0, 255.0);
```

## Learning Objectives for Section 4

After having attended this class, the student will be able to:

- 1) Represent numbers in binary or base 10 using byte variables.
- 2) Read from and write to digital pins on the Arduino Uno, using the proper `pinMode` and syntax.
- 3) Distinguish between an analog and digital pin, in terms of functionality and limitations.
- 4) Use Pulse Width Modulation (PWM) to dim an LED.
- 5) Use the `map()` function to remap an integer from one range to another.
- 6) Independently set up a momentary button as a digital input, using an appropriate pull-down or pull-up resistor.
- 7) Based on a button schematic, predict whether or not a pin value will be in a LOW or HIGH state while the button is pushed.
- 8) Compare and toggle the states of bool variables using Boolean operators (`!`, `&`, `|`).
- 9) Know what the default range and resolution is for analog pins. Be able to change the analog reference level using the AREF pin.
- 10) Convert from *div* units (measured from an analog pin), to volts, taking an external analog reference voltage into account.
- 11) Use the serial monitor to output experimental data in .CSV format.
- 12) Become proficient at writing void, integer, and float functions to bundle repeated commands and simplify code.

- 13) Use proper function-writing etiquette to avoid common pitfalls in coding.
- 14) Calibrate a thermistor using two known temperatures. Use the two-term exponential thermistor equation to calculate and report the measured temperature.

### Section 4 - Station Content List

- 1 x Digital Multimeter
- 1 x Breadboard
- 1 x 220 $\Omega$  Resistor
- 1 x 10K Resistor (1% tol)
- 1 x 10K Resistor (5% tol)
- 1 x 10K Potentiometer
- 1 x Momentary Switch
- 1 x 10K Waterproof Thermistor
- 1 x LED
- 8 x Male/Male Jumpers
- 1 x Beaker with Ice Water
- 1 x Glass Thermometer

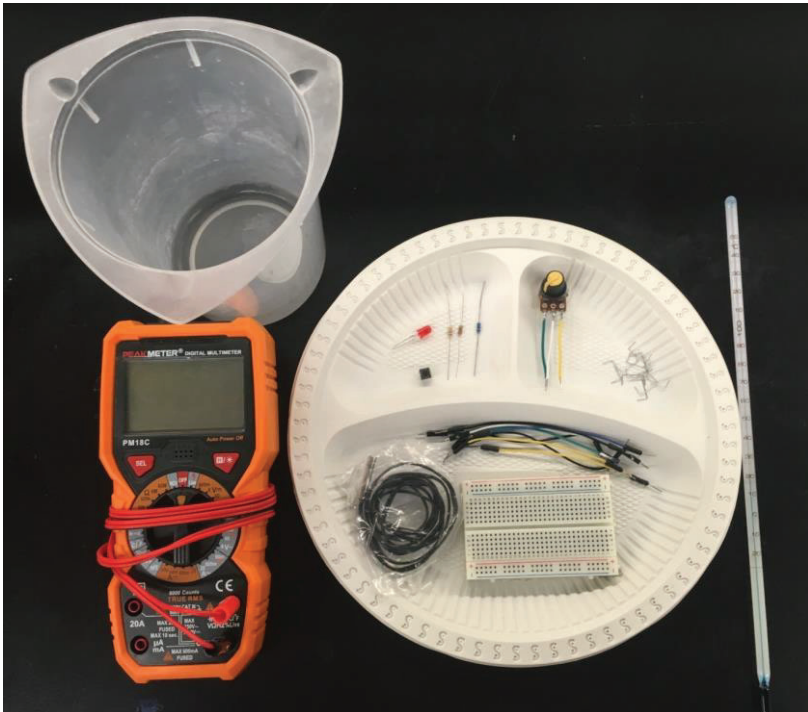


Figure 4-12. Section 4 station setup.

## SECTION 5

# SWITCHING HIGHER POWER DEVICES: RELAYS, TRANSISTORS, TRIACS

|                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |                                                                                                                                        |
|-------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------|
| What You'll Be Learning | <b>Lecture:</b> Arduino Uno power limitations. Relays, high vs. low side switching, wiring multiple power supplies. Diodes, transistors (NPN and PNP BJTs, base resistor calculation, cutoff, active, and saturation modes). MOSFETs (N-channel and P-channel). TRIACs: opto-isolation, dimming AC power.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |                                                                                                                                        |
| What You'll Be Doing    | <b>Pick any two activities:</b><br><b>Activity 5-1:</b> Building a relay-controlled hot plate temperature bath using the thermistor from Activity 4-1. Writing a simple sketch to maintain a desired temperature.<br><b>Activity 5-2:</b> Building an NPN-transistor-controlled DC motor circuit. Using PWM to control motor speed through commands on the serial monitor.<br><b>Activity 5-3:</b> Modifying the circuit in Activity 5-2 to be controlled using a MOSFET, instead of an NPN transistor.<br><b>Demo:</b> Relay-controlled USP Dissolution Apparatus.<br><b>Demo:</b> Piezo as a microphone (transistor biasing). Piezo as a speaker.<br><b>Spotlight:</b> Other Arduino platforms: LilyPad, Nano, Leonardo, Mega, PCduino. AtTiny85: LilyTiny. ESP8266: ESP-01, NodeMcu, LinkNode D1. |                                                                                                                                        |
| Files you will need     | All course files are available for download at:<br><a href="http://pb860.pbworks.com">http://pb860.pbworks.com</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | <ul style="list-style-type: none"><li>• <i>Thermostat.ino</i></li><li>• <i>Transistors.xlsx</i></li><li>• <i>GetCSV.xlsm</i></li></ul> |

### Voltage and Current Limitations of the Arduino Uno

The Arduino Uno works well for powering and interacting with logic circuits and low-voltage, low-power devices ( $\leq +5V$ ). However, every power source has its limits. When a power source can't provide enough voltage or current to run a device, the first thing you might notice is that the

device won't power up properly. An LCD screen might be dim, communications (like serial monitor commands) don't seem to happen, and the device will probably not run as intended. The power supply might get hot. Some components may even get damaged when run at a lower voltage than required. This makes it important to understand the voltage and current input requirements and output limitations for the Uno: (King 2017; Atmel Corporation 2016)

- DC current per I/O pin: **40 mA max** (limit: **200 mA** total for all I/O pins combined, not including +5V pin)
- DC output voltage, I/O pins: **5V max** (can be slightly lower if powered by USB)
- DC current for 3.3V pin: **50 mA**
- Input voltage (adapter jack): **7-12V** (although 5V is provided through USB)
- Maximum total output current: **500 mA** when USB-powered, up to **1A** when powered using an external 12V adapter
- Maximum current from +5V pin: **500 mA** when USB-powered, up to **1A** when powered using an external 12V adapter
- Maximum sink current (2 ground pins): **400 mA max** (this is the amount of current the Uno can sink, or receive)

To reach the full limits above, the Arduino Uno should be powered by a 12V DC adapter through the adapter jack, rather than through a USB cable.

If the maximum voltage and current an Arduino Uno puts out is 40 mA per digital pin, how can you control and power a device that runs on 12V, or requires over an amp to run properly? Most of the fun stuff we'd like to do requires more power than a 5V microcontroller can provide. This section discusses how to handle it without frying your microcontroller or your equipment.

The easiest and safest way to handle higher voltage is to use a relay. This strategy keeps your *logic* (MCU side—deciding when to turn on and off) separate from your *load*.

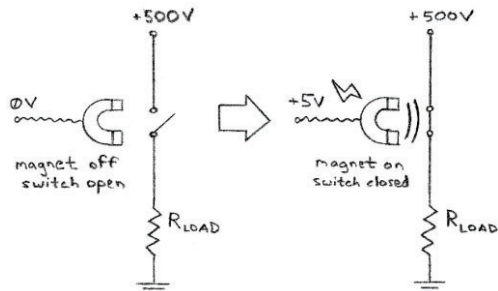


Figure 5-1. A high voltage induces a magnetic field, pulling a switch closed.

## Relays

A *relay*, put very simply, is an electronically-controlled switch. A typical electro-mechanical relay consists of a current that runs through a coil, inducing a magnetic field. A very simplified schematic is presented in Figure 5-1. A 5V signal creates a magnetic field, which then pulls a switch closed, connecting a load to +500V.

A substantial benefit of using a relay is electrical isolation. There is no part of the +500V circuit in Figure 5-1 that is electrically connected to the +5V circuit. The two systems interact only through the magnetic field.

Here is a circuit diagram symbol for a typical relay (Single Pole, Double Throw, or *SPDT*):

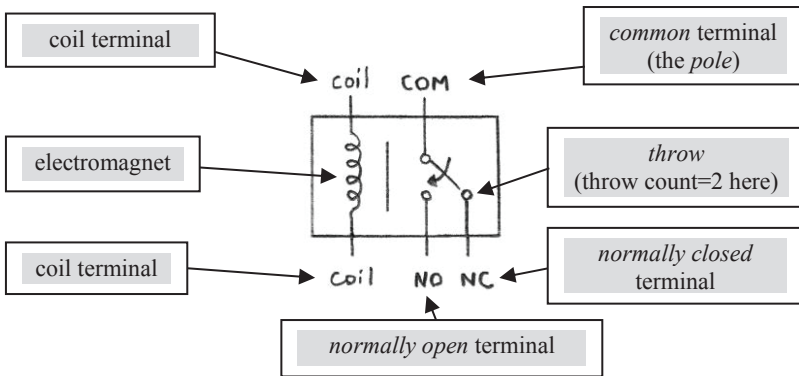


Figure 5-2. Relay circuit diagram symbol (Single Pole, Double Throw - SPDT).

The number of *poles* on a switch defines how many separate circuits the switch can control (equal to the number of *common* terminals). The *throw count* is the number of positions (or terminals) each pole can be connected to. Figure 5-3 shows two very typical configurations for relays: DPST and DPDT. DPST relays can close two independent poles with one throw each (normally open). This means it can act as an on-off switch for two separate circuits. DPDT relays can switch two independent poles, with two throws each. This means you can select whether or not turning on the relay *closes* or *opens* the switches. You would select whatever configuration your circuit requires. A SPDT relay is likely adequate for most simple applications.

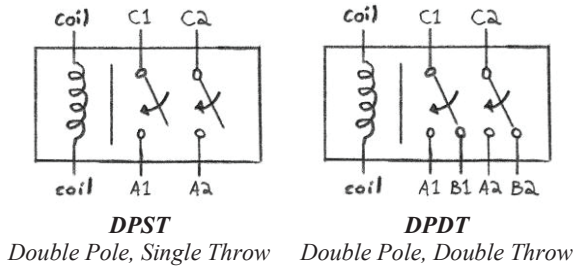


Figure 5-3. Two common types of relays: DPST and DPDT.

The relay modules we have in the lab are SPDT, capable of switching up to 10A of AC power from a wall outlet (250V AC or 125V AC). This is a considerable upgrade from the Uno's power limitations. There is more to these relay modules than an electromagnet and a switch. The relay module protects the microcontroller with an optocoupler circuit, depicted in Figure 5-4. The optocoupler circuit sends IR light to an IR-triggered transistor, which then triggers a second transistor to apply a voltage to the relay coil. Diodes and transistors will be discussed later in this section.

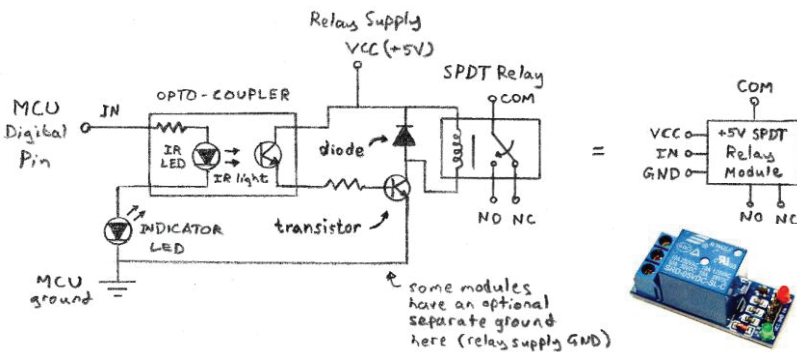


Figure 5-4. Internal circuit diagram of a SPDT relay module. An opto-coupler isolates the Arduino Uno from the relay supply. Some relay modules allow for a separate ground for the supply powering the relay, although in the lab we will power the relay module using the Arduino +5V pin. (ELECTFREAKS wiki 2015)

The way that an SPDT relay is wired will change how it works. The relay module in Figure 5-5 (left) will close the circuit (turn on) when you send +5V to the IN pin on the relay module (using the `digitalWrite()`)

command), and the circuit in Figure 5-5 (right) will open the circuit (turn off) when you send +5V to the IN pin on the relay module.

Notice that the *logic side* (MCU side) has a ground separate from the *load side* (also called the *power side*). The *logic side* is grounded to the Arduino Uno, and the *load side* is grounded to the 12V power supply ground. These two grounds are NOT connected. This way, a failure on the load side does not destroy the logic side. Keeping power and logic grounds separated also helps to reduce noise on the power side interfering with the logic.

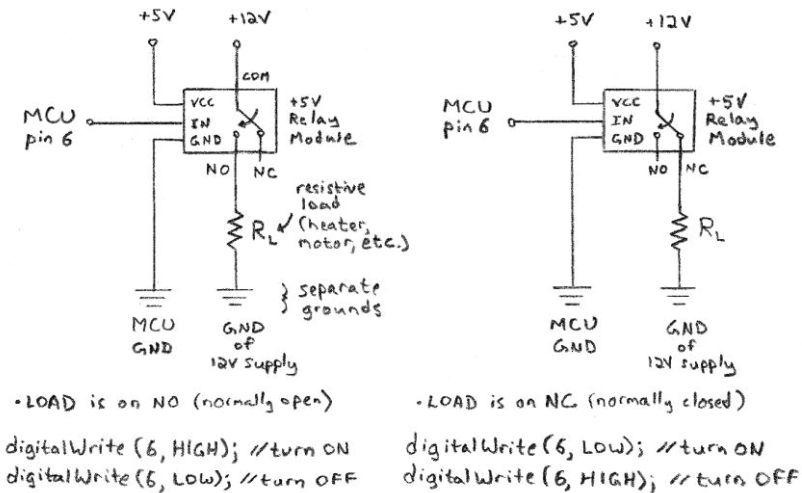


Figure 5-5. Using a SPDT relay as a *normally open* (NO) switch vs. a *normally closed* (NC) switch. The switch terminals have been sketched into the relay module boxes to help illustrate switching direction.

There are some disadvantages to relays. Firstly, relays tend to be audibly noisy. You can hear them when they switch, like the turn signal of a car, or the changing of a stoplight. When a relay switches on or off, there is a tiny spark created that can even be visible. This could be extremely dangerous in an environment with flammable vapours. The spark also causes pitting, oxide deposits, and sometimes welding on the relay's internal terminals. Ultimately this oxide builds up and the relay switch will either weld onto one terminal permanently (open or closed), or no longer be able to electrically connect with either terminal at all. Electro-mechanical relays wear out.

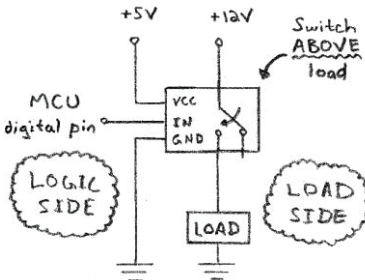


In addition to electro-mechanical relays, there are also **solid-state relays**. These relays have no moving parts. Consequently, they can switch more quickly (up to  $\sim 1$  msec on low-end devices) and noiselessly. They are less prone to wearing out. Solid state relay modules usually have the benefit of opto-isolation, so logic and power can still have separate grounds. Disadvantages of solid-state relays are their bulky sizes for higher load switches, relatively lower power ratings, and higher cost compared with electro-mechanical relays. Solid state relays are less “all-purpose” than mechanical relays. There are different models for switching AC vs. DC power. In addition, their switching functionality is limited to only single pole, single throw, and are available with a high-trigger (like PNP) *or* low-trigger (like NPN) options.

### High Side Switching vs. Low Side Switching

The two circuits in Figure 5-5 are examples of **high side switching**. This means the switch is *above* the load ( $R_L$ ). In other words, it’s on the higher voltage side. From a safety perspective, this strategy is generally a good idea, particularly with high voltage (e.g. 120V AC). When a high side switch is open (off), then there is a smaller chance that part of the circuit could touch a ground wire (or your finger) and result in a short circuit. With **low side switching**, the relay is *lower* than the load.

High Side Switching:



Low Side Switching:

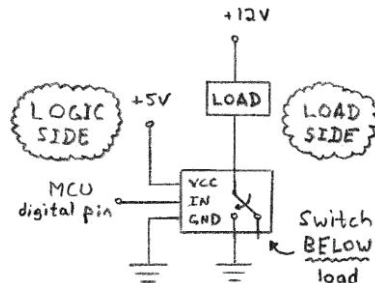


Figure 5-6. The relay module on the left is wired as a high side switch (above the load). The relay module on the right is wired as a low-side switch (below the load).

**Test your understanding:** both relays in Figure 5-6 are drawn in the **normally open, held closed** (NOHC) switch configuration. Can you draw them as **normally closed, held open** (NCHO)? Can you think of an application where **NCHO** would be a poor design decision?

### Powering a Relay with a Separate Supply

A relay on its own takes quite a bit of power to operate. You *can* power a 5V relay module using the Arduino's +5V power pin, but you might find when the relay switches, bad things happen:

- the sensor readings get spikes;
- the LCD backlight dims or flickers;
- the LCD display shows a string of garbled scrolling text;
- the MCU crashes, freezes, or restarts;
- the serial connection freezes.

You can try scattering *decoupling capacitors* around your circuit (more on that later), but a *better* fix is to power your relay module with a separate +5V power supply. The supply should share a common ground with the Arduino Uno. This means your circuit could potentially have three separate power supplies. The following diagram illustrates how you can use individual power supplies for the microcontroller, relay, and load:

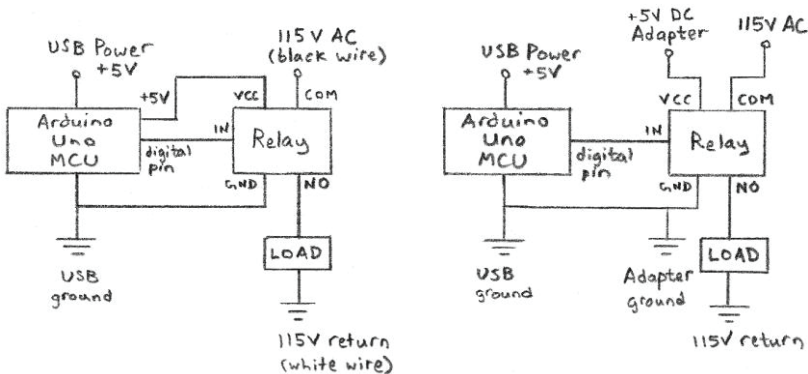


Figure 5-7. The relay module on the left is powered using the Arduino Uno, which can introduce a lot of switching noise to your measurements. The relay module on the right is powered using a separate +5V DC adapter.

**Test your understanding:** are the relays in Figure 5-7 drawn as *high-side* or *low-side* switches?

### Vin Pin: Arduino Uno

Looking for higher voltage or more current quickly? You can access your unregulated power supply from the Uno's “*Vin*” pin to power other modules.

*V<sub>in</sub>* is a special pin connected to the positive terminal of your power supply. However, using this pin might not isolate relay switching noise from your sensors, and *V<sub>in</sub>* will have the same voltage as your power adapter, which might be a problem depending on what you are trying to power. If you are powering the Arduino Uno with a 12V DC adapter, *V<sub>in</sub>* will be at +12V. If you are powering the Arduino through USB by plugging it into a laptop, *V<sub>in</sub>* will be about +5V. In a pinch, *V<sub>in</sub>* might be a quick work around if your module is current-limited by the Arduino's regulated power pins. You can also use the *V<sub>in</sub>* and *GND* pins to power the Arduino Uno, with a DC adapter that has bare wires. Just make sure you get the polarity right, or you might damage the microprocessor.

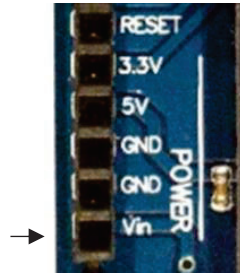


Figure 5-8. The *V<sub>in</sub>* pin gives you access to the power supply voltage used to power the Uno, before the on-board 5V voltage regulator.

## Diodes (P-N Junction, or Rectifier Diodes)

We have used the LED circuit diagram symbol many times already in this book (Figure 5-9, left).

An LED is a specialized type of diode. A diode is conceptually a one-way sign for current. Understanding how a diode works is an important step in understanding how transistors work. Diodes are also important in protecting your circuit from current travelling the wrong way and damaging components.

The circuit diagram symbol for a diode is shown in Figure 5-9 (right).

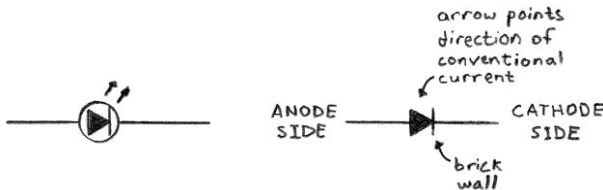


Figure 5-9. Diode symbols: LED (left), and general diode symbol (right).

How do diodes constrain the flow of electrons to one direction? The key in this interesting behaviour is in the semi-conductive nature of silicon. Pure elemental silicon is not a good conductor. However, silicon doped with

boron has the occasional space to hold an electron. This is often referred to as an electron “hole”. This makes the silicon slightly positive, which is why it is called **P-type silicon**. Similarly, silicon doped with phosphorus has an extra free electron that can be mobile, just like copper. This makes the silicon slightly negative, which is why it is called **N-type silicon**. When you put these two materials adjacent to each other, the result is a **P-N junction**, illustrated in Figure 5-10. (Scherz and Monk 2016)

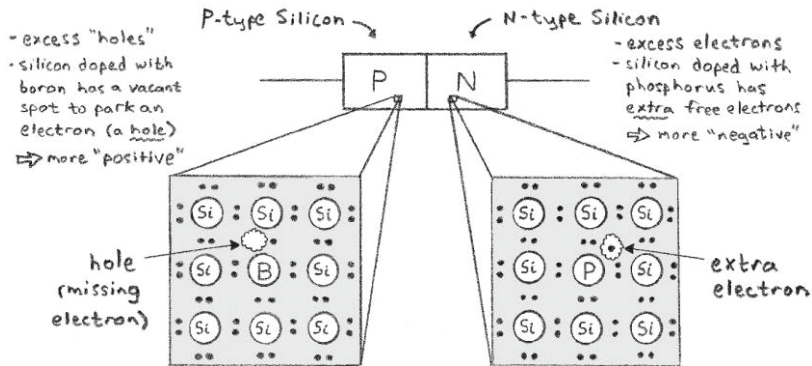


Figure 5-10. Anatomy of a diode. Silicon has 4 outer valence electrons. When doped with boron (3 outer valence electrons), a space is formed capable of temporarily accepting an electron (left). When doped with phosphorus (5 outer valence electrons), there is an extra electron, capable of flowing (right).

If conventional current flows from the anode side to the cathode side (P to N), then we say the diode is **forward biased** (keep in mind the electrons actually flow the other way):

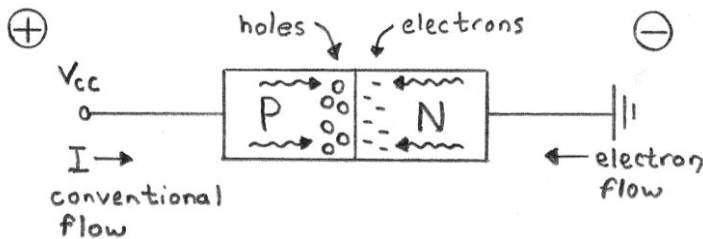


Figure 5-11. When conventional current tries to flow from anode to cathode, a diode is **forward biased**, and electrons can jump across the P-N junction.

Electrons flow toward the P-N junction from the N side, and “holes” (or spaces) migrate toward the P-N junction from the P side. The junction narrows, and electrons can flow across (see Figure 5-11).

If conventional current flows from the cathode side to the anode side (N to P), then we say the diode is *reverse-biased*:

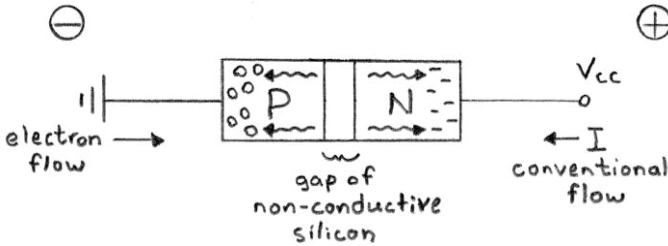


Figure 5-12. When conventional current flows from cathode to anode, a diode is said to be *reverse biased*, and electrons can no longer cross the P-N junction.

The P-N junction widens, becoming a thicker layer of non-conductive silicon with filled outer valence electron shells. Free electrons can no longer cross the P-N junction, and current can't flow (see Figure 5-12).

If the current or voltage difference gets too high in either direction, the diode will break down (overheat, or even pop), so it pays to know what sort of currents you are expecting, wire the diode the correct way as intended, and match the diode's limits according to the application. One popular diode is the 1N4007: (ON Semiconductor Corp 2018)

Maximum Forward Current: 1A

Forward Voltage Drop at  $I_F=1A$ :  $V_F=1.1V$

(there is some voltage drop across a forward-biased diode)

PRV (Peak Reverse Voltage): 1000V  
(that should be enough!)

Some other ways to describe PRV:

- DC blocking voltage
- Peak Inverse Voltage
- Breakdown Voltage

Diodes usually have a line painted on them to show which side has the *brick wall*, so you know which way to install them. A silver line on the 1N4007 marks the cathode side (N-terminal, or brick wall). Other diodes (e.g. Zener diodes) follow the same convention.

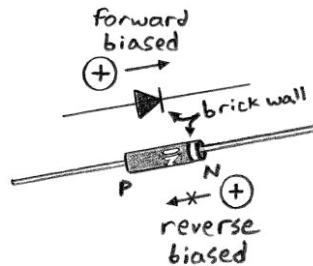


Figure 5-13. 1N4007 diode. A stripe indicates the brick wall (location of N-terminal).

## Transistors

If antibiotics are said to be among the most impactful innovations in medicine, you can think of transistors as having that magnitude of impact in electronics.

There are many different types of transistors, and like relays, they are basically electronically-controlled switches. However, rather than a magnet pulling the switch closed, a transistor can be controlled by current, or voltage. This is best illustrated with the following circuit:

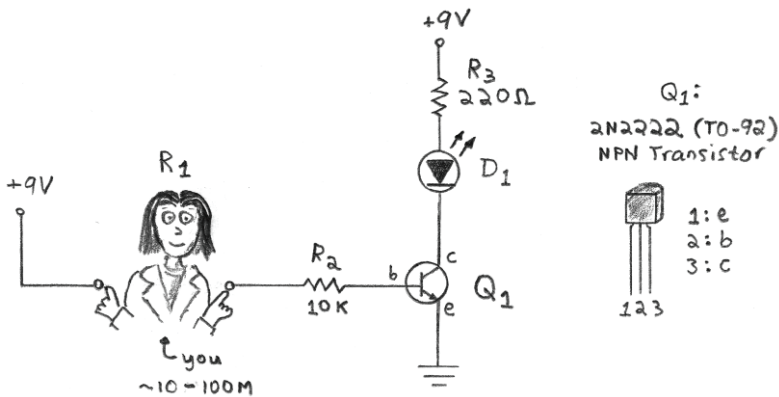


Figure 5-14. Circuit to illustrate how a transistor works.  $Q_1$  is a 2N2222 NPN transistor in a TO-92 package. (ON Semiconductor Corp 2013)

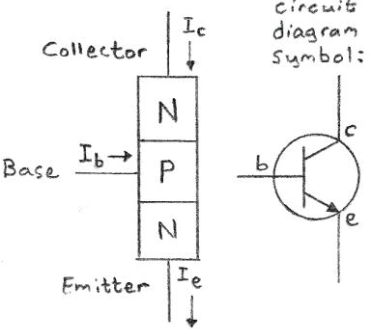
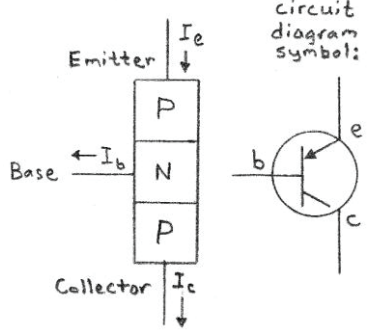
Such a small amount of current is required to “turn on” the transistor. Try turning on the LED in the above circuit (Figure 5-14) by holding the terminals, one in each bare hand. The tiny bit of current flowing through your body is enough to switch on the transistor. Try it, it’s fun! Transistors can switch much faster than relays. They can switch quickly enough to keep up with PWM frequencies. Contrast this to the high-power demands of a bulky, slow relay module, and you can see why transistors are just plain amazing. A drawback is that the logic and power sides need to share the same ground for a transistor to work, so the logic side is no longer electrically isolated from the power side of your circuit.

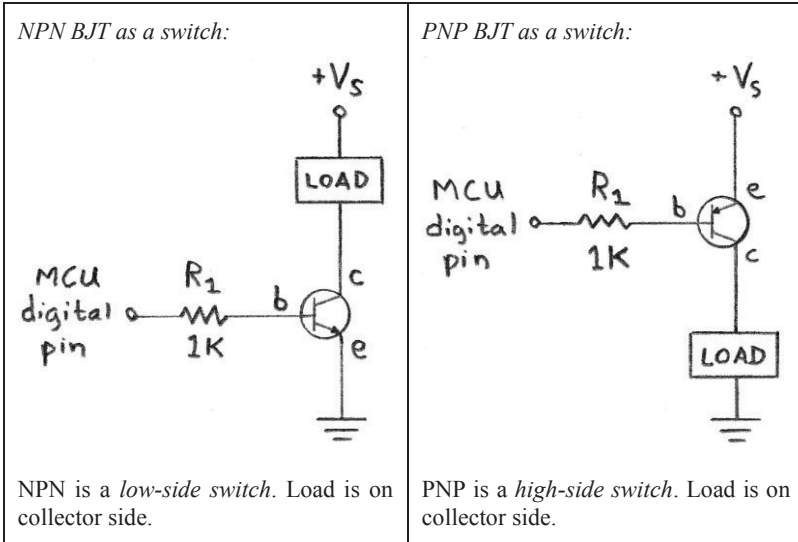
### *Bipolar Junction Transistors (BJTs)*

There are many different types of transistors. The example transistor in Figure 5-14 is a **Bipolar Junction Transistor** (BJT). BJTs are extremely

popular, perhaps because they are easy to work with, conveniently tiny, and dirt cheap. The construction of a BJT is very similar to a diode, only there are three layers of doped silicon instead of just two, giving rise to two different types of transistors: NPN and PNP (summarized in Table 5-1).

**Table 5-1. NPN and PNP Bipolar Junction Transistors (BJTs).**

| NPN Transistor                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | PNP Transistor                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|  <p style="text-align: center;"><b>“Never <u>P</u>oints <u>i</u>n<u>w</u>ards”</b></p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |  <p style="text-align: center;"><b>“<u>P</u>oints <u>i</u>n<u>w</u>ards <u>P</u>ermanently”</b></p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <p><i>Operation:</i></p> <p>When base current (<math>I_b</math>)=0: OPEN SWITCH (<i>cutoff mode</i>)</p> <ul style="list-style-type: none"> <li>• <math>V_b &lt; V_c, V_b &lt; V_e</math></li> <li>• To “shut off” the transistor, set the base terminal voltage to GND.             <ul style="list-style-type: none"> <li>• Set digital pin to LOW.</li> </ul> </li> </ul> <p>When <math>V_e &lt; V_b &lt; V_c</math>: AMPLIFIER (<i>active mode</i>)</p> <p>When <math>V_b &gt; V_c</math> and <math>V_b &gt; V_e</math>: CLOSED SWITCH (<i>saturation mode</i>)</p> <ul style="list-style-type: none"> <li>• To completely turn on the transistor, put “enough” current through the base terminal. Set digital pin to HIGH, and use a base resistor that sends enough current to turn the transistor completely on.</li> </ul> | <p><i>Operation:</i></p> <p>When <math>I_b = 0</math>: CLOSED SWITCH (<i>saturation mode</i>)</p> <ul style="list-style-type: none"> <li>• To “turn on” the transistor, set digital pin to LOW, and use a base resistor that sinks enough current to ground to turn the transistor completely on.</li> </ul> <p>When <math>V_e &gt; V_b &gt; V_c</math>: AMPLIFIER (<i>active mode</i>)</p> <p>When <math>V_b &gt; V_c</math> and <math>V_b &gt; V_e \rightarrow</math> OPEN SWITCH (<i>cutoff mode</i>)</p> <ul style="list-style-type: none"> <li>• To “shut off” the transistor, the base voltage needs to be equal to the emitter. Set the digital pin to HIGH. This will shut off the transistor if the emitter voltage is +5V.</li> </ul> |



### *NPN Transistors: Selecting a Base Resistor Value*

How do you figure out what value of base resistor to use ( $R_1$  in Table 5-1, also referred to as  $R_b$ )? Depending on the application, you can take different approaches to determining an appropriate value for the base resistor of a BJT.

#### *Method 1: Convention and Experimentation*

For an Arduino Uno, if your goal is to switch between cutoff mode and saturation mode, try using a 1K resistor. This is a typical value for a base resistor, and in general will put most transistors in saturation mode. A 1K resistor might provide a more current than you need, but it should work well. If it doesn't quite get you to saturation mode, try lower resistance values until you find one that works (and your power side turns on completely).

#### *Method 2: The 10% Current Rule*

For a better approximation, try using the 10% current method to calculate the base resistor required. (Scherz and Monk 2016) The 10% current method states that in general, most BJTs should be in saturation mode when the base current ( $I_b$ ) is 10% of the planned collector current ( $I_c$ ):



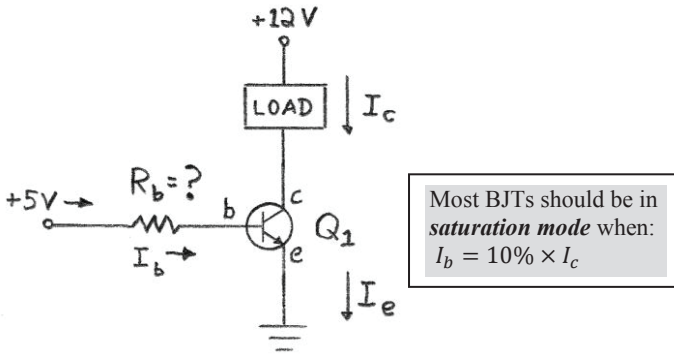


Figure 5-15. Worked example for the **10% Current Rule**.

**Question:** The load in Figure 5-15 has an operating voltage of 12V, and a measured load resistance of  $500\Omega$ . What base resistor is required to put the transistor into saturation mode, using the 10% rule?

**Answer:**

$$V = IR \rightarrow I = \frac{V}{R}$$

$$I_c = \frac{12V}{500\Omega} = 0.024A = 24mA$$

$$I_b = 10\% \times I_c = 10\% \times 0.024A = 0.0024A = 2.4mA$$

The base current needs to be 2.4 mA to get the transistor into saturation mode. Given that the microcontroller will supply this current at 5V, we can calculate the base resistor value required:

$$R_b = \frac{V}{I_b} = \frac{5V}{2.4mA} = 2.1k\Omega$$

So a base resistor of 2.1K would result in enough current to put the transistor in this circuit into saturation mode (in other words, turn the device completely on). Looking at the table of common fixed resistor values in the appendix (Table A-5), the closest fixed resistor values to our answer are 2.0K and 2.4K. We can select a value of 2.0K to be more conservative, to make sure enough base current is supplied.

### Method 3: Using the $h_{FE}$ or $\beta$

Every transistor has its own specific amplification factor. Some datasheets refer to this value as  $\beta$ , and some call it  $h_{FE}$  (for *hybrid parameter forward current gain, common emitter*). A transistor's datasheet will give you an idea of the magnitude of this parameter, but the transistor you hold in your hand might be slightly different. For instance, the ON Semiconductor® datasheet for the P2N2222A lists the following  $h_{FE}$  values:

**Table 5-2.  $h_{FE}$  values of P2N2222A (TO-92) (ON Semiconductor Corp 2013)**

| DC Current Gain                                | $h_{FE}$ Min | $h_{FE}$ Max |
|------------------------------------------------|--------------|--------------|
| $I_c = 0.1 \text{ mA}, V_{ce} = 10 \text{ V}$  | 35           | -            |
| $I_c = 1.0 \text{ mA}, V_{ce} = 10 \text{ V}$  | 50           | -            |
| $I_c = 10 \text{ mA}, V_{ce} = 10 \text{ V}$   | 75           | -            |
| $I_c = 150 \text{ mA}, V_{ce} = 10 \text{ V}$  | 100          | 300          |
| $I_c = 150 \text{ mA}, V_{ce} = 1.0 \text{ V}$ | 50           | -            |
| $I_c = 500 \text{ mA}, V_{ce} = 10 \text{ V}$  | 40           | -            |

So we can expect for a P2N2222A transistor to have a  $h_{FE}$  close to 75 (the closest table value to our example:  $I_c=24 \text{ mA}, V_{ce}=12 \text{ V}$ ). The equations for  $h_{FE}$  work similarly to the *10% Current Rule*, only this parameter converts the base current to the resulting collector current:

$$I_c = h_{FE} I_b = \beta I_b$$

If our transistor example in Method 2 is a P2N2222A, then we can do this same example with the datasheet's  $h_{FE}$  value. From before, we calculated that we would like a collector current of 0.024 A (=24 mA). Therefore:

$$I_c = h_{FE} I_b \rightarrow I_b = \frac{I_c}{h_{FE}} = \frac{24 \text{ mA}}{75} = 0.32 \text{ mA}$$

Now, we can use the same equations to calculate  $R_b$ :

$$R_b = \frac{V}{I_b} = \frac{5 \text{ V}}{0.32 \text{ mA}} = 15.6 \text{ k}\Omega$$

We can see using this method that *much* less current is actually required to get the transistor into saturation mode than Method 1 or Method 2 estimated, and the most energy efficient “answer” would be a resistor value of 15K, the closest fixed resistor value in the table of common fixed resistor values in the appendix (Table A-5, *Common Fixed Resistor and Capacitor*

Values). However, this assumes that the actual  $h_{FE}$  of the transistor matches the datasheet perfectly, which might not happen exactly. Here is where a design decision comes in. Will you select 1K, or 15K? Can you think about some design considerations that might impact your answer?

### *NPN Transistors in the Active Region*

So far, our calculations have focused on cutoff and saturation modes of transistors. One of the features of a transistor is that unlike a relay, there is a *transition region* where the transistor is half-on and half-off. This is known as the **active region** of the transistor. You can take advantage of the active region to amplify an analog signal. Finding it can be tricky. Even after working out the math properly, each individual transistor has a slightly different  $h_{FE}$  value. A transistor may need to be manually **biased** using an oscilloscope or other mechanism where you can see what's going on with the output signal. This means the base current must be adjusted to find the active region, which you can now think of as the sweet spot between **cutoff mode** and **saturation mode**. The following schematic shows a typical strategy for setting up a transistor as an (inverting) amplifier:

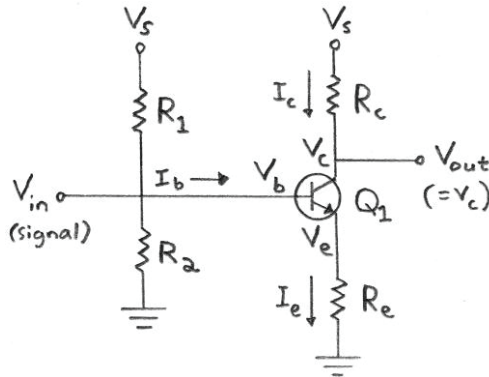


Figure 5-16. An NPN transistor configured as a common emitter amplifier.

We will trace through some of the math to derive the **gain** of this amplifier (or in other words, how much the signal is amplified).

$$V_b - V_e = 0.6 \text{ V (a standard value for transistors)}$$

$I_c$  is specified as a design parameter (also called  $I_Q$ , or **quiescent current**). It's the collector current you require, for your signal or load. Starting with the  $h_{FE}$  equation:

$$I_c = h_{FE}I_b \rightarrow I_b = \frac{I_c}{h_{FE}}$$

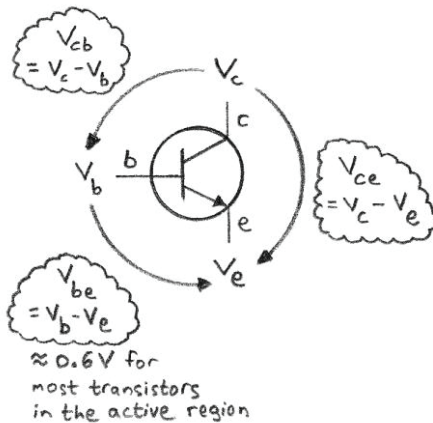
$$V_e = I_e R_e$$

**KCL:**  $-I_e + I_b + I_c = 0$  (on transistor)

$$I_e = I_b + I_c = I_b + h_{FE}I_b = I_b(1 + h_{FE})$$

$$I_e \approx h_{FE}I_b \quad (\text{if } h_{FE} \gg 1, \text{ a typical value can be } \sim 100)$$

Figure 5-17 provides a closer look at the voltages around the NPN transistor.



To calculate  $V_c$ , subtract the voltage drop across resistor  $R_c$  from  $V_s$ :

$$V_c = V_s - I_c R_c$$

For a *small* change in voltage at the base terminal,  $\Delta V_b$ , we can look at the response:

$$\Delta V_c = \Delta I_c R_c$$

And for a small fluctuation,

$$\Delta V_b \approx -\Delta V_e$$

The collector current will be approximately the same as the emitter current:

$$\Delta I_E = \frac{\Delta V_e}{R_e} \approx -\frac{\Delta V_b}{R_e} \approx \Delta I_c$$

$$\rightarrow \Delta V_c = \Delta I_c R_c = -\frac{\Delta V_b}{R_e} R_c$$

$$\rightarrow \frac{\Delta V_c}{\Delta V_b} = -\frac{R_c}{R_e} = \text{signal gain}$$

Figure 5-17. Calculating the voltage changes around the three terminals of a bipolar junction transistor used as a common emitter amplifier in Figure 5-16.

The take-home message in the derivation in Figure 5-17 is that the ratio  $R_c/R_e$  determines how much the voltage change ( $\Delta V_b$ ) is amplified. Practically speaking, you can **bias** a transistor (that means, tweak the input current to find the active region) by replacing  $R_1$  and  $R_2$  with a 50K or 100K

potentiometer set up as a voltage divider. You could build the circuit, read  $V_{out}$  with an oscilloscope, and turn the potentiometer until you can see the amplified signal. A perturbation in  $\Delta V_b$  results in a larger change in the response,  $\Delta V_c$ . Since each transistor has a slightly different  $h_{FE}$ , this approach is more effective than doing the math and committing to anticipated fixed resistor values. Select  $R_c/R_e$  so that your gain is what you need (e.g.  $R_c=10K$  and  $R_e=1K$  for a  $\sim 10X$  gain). The gain is negative, so a common emitter amplifier is called *inverting*. We will explore signal gain in more detail in Section 7.

The spreadsheet *Transistors.xlsx* is posted on the course website. It will help you with fundamental calculations concerning transistors. The biggest advantage of using a transistor to amplify a signal is speed. Transistors can easily amplify signals well into MHz, and even GHz region.

### *Darlington Pairs*

If a very high impedance signal needs to be amplified (in other words, a very low current), two transistors may be “doubled up” in what is known as a *Darlington pair*. The *output* on the collector side of the transistor is used as the *input* of the next transistor. The result is that the total  $h_{FE}$  of the system is equal to the  $h_{FE}$  of each transistor, *multiplied* together. (Scherz and Monk 2016)

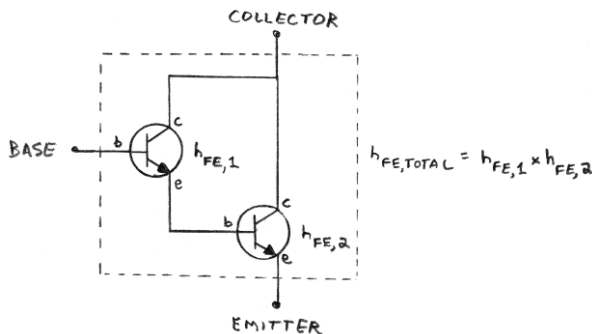


Figure 5-18. A Darlington pair of NPN transistors.

You can make your own Darlington pair using two of the same transistors, or you can purchase them pre-wired in a single package. For example, the TIP120 is an NPN Darlington pair, and the TIP125 is a PNP Darlington pair, both having an  $h_{FE}$  of  $\sim 1000$ . (ON Semiconductor Corp 2014) In the more general sense, wiring any components in series like this

(output to input) is called *daisy chaining*. There is a cost to daisy chaining transistors: the response time through a Darlington pair takes longer, since the signal needs to make its way through two transistors instead of just one. This might not be important if your signal frequency is much lower than what the Darlington pair can handle.

Interestingly, LEDs can also *detect* light. An LED produces a *very* tiny bit of current when light shines directly on it. Figure 5-19 shows an example of a Darlington pair of NPN transistors used to turn on LED  $D_2$ , when LED  $D_1$  is used as a light sensor. This circuit illustrates how the little bit of current *produced* by LED  $D_1$  can be amplified

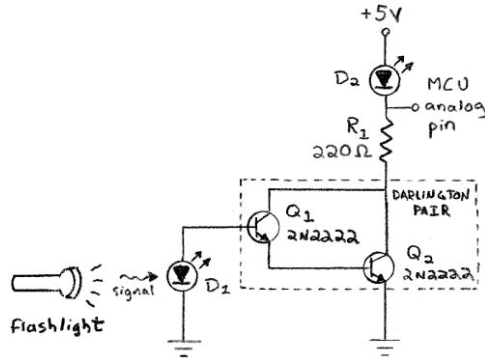


Figure 5-19. A regular LED as a light sensor.

by the combined  $h_{FE}$  value of a Darlington pair of 2N2222 transistors. LED  $D_2$  can be replaced with a 10K resistor to improve the voltage swing of the output. Alternately, a DC motor could replace  $R_1$  and  $D_2$ , for a light-triggered motor circuit.

### Current Gated vs. Voltage Gated

One of the features of bipolar junction transistors is that they are *current gated*, meaning that a little bit of current changes their state. You might ask the question, isn't there a transistor that behaves more like a relay, so that when you set a digital pin **HIGH** it turns on, and when you set the digital pin **LOW**, it turns off, without having to add a base resistor? We could just use a relay, but what if we want faster control than a relay can provide (e.g., with PWM)? Relays can't switch quickly enough.

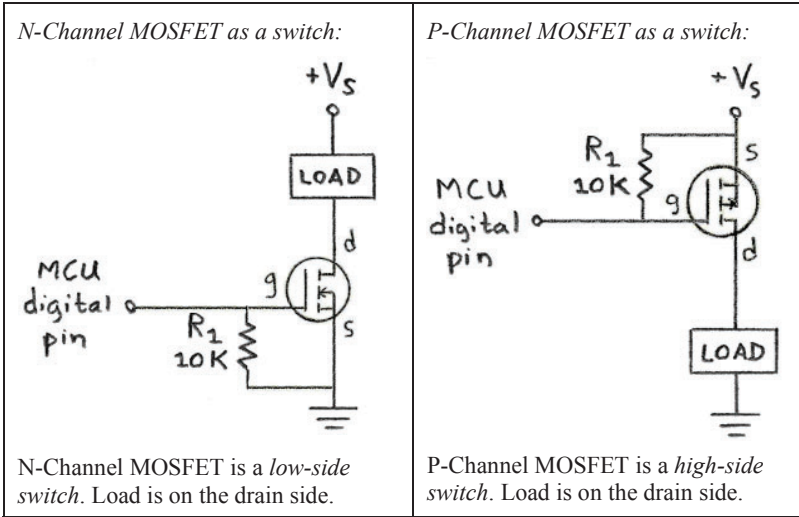
Enter the world of MOSFETs. MOSFETs are *voltage gated*, rather than current gated. They are triggered by a change in voltage. MOSFETs can handle very high voltage and current, although there are also high-power transistors available for this task.

## *MOSFETs*

A *MOSFET* (short for “Metal Oxide Semiconductor Field Effect Transistor”) is a special type of transistor. Similar to BJTs, there are N-channel and P-channel MOSFETs, summarized in Table 5-3.

**Table 5-3. N-Channel MOSFET vs. P-Channel MOSFET.**

| <i>N-Channel MOSFET</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        | <i>P-Channel MOSFET</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <ul style="list-style-type: none"> <li>• Arrow points inwards</li> <li>• Switch is normally open</li> </ul> <p><i>Operation:</i></p> <ul style="list-style-type: none"> <li>• <math>V_{GS(th)} &gt; 0V</math> (e.g. <math>V_{GS(th)}</math> for 2N7000 = 2.1V typical if <math>I_D=1\text{ mA}</math>)</li> <li>• If <math>V_G &gt; V_{GS(th)}</math>: Switch is ON (saturation mode)</li> <li>• If <math>V_G &lt; V_{GS(th)}</math>: Switch is OFF (cutoff mode)</li> <li>• Use a pull-down resistor to protect the gate from turning on if MCU pin floats, or gate wire disconnects. (ON Semiconductor Corp 2011)</li> </ul> | <ul style="list-style-type: none"> <li>• Arrow points outwards</li> <li>• Switch is normally closed</li> </ul> <p><i>Operation:</i></p> <ul style="list-style-type: none"> <li>• <math>V_{GS(th)} &lt; 0V</math> (e.g. <math>V_{GS(th)}</math> for BS250 = -1.9V if <math>I_D=1\text{ mA}</math>)</li> <li>• If <math>V_G &gt; V_S - V_{GS(th)}</math>: Switch is OFF (cutoff mode)</li> <li>• If <math>V_G &lt; V_S - V_{GS(th)}</math>: Switch ON (saturation mode)</li> <li>• Use a pull-up resistor to protect the gate from turning on if MCU pin floats, or gate wire disconnects. (Vishay Siliconix 2004)</li> </ul> |



**N-Channel MOSFET Construction**

When the gate voltage exceeds a threshold ( $V_G > V_{GS(th)}$ ), a channel of electrons lines up between the drain and source along a thin layer of silicon, allowing current to flow. This thin layer of silicon is very sensitive to electrostatic discharge (ESD), so be careful when handling MOSFETS. Extra precautions can be taken (e.g. wearing a grounding bracelet, touching a large piece of metal before touching the MOSFET, or not rubbing your feet against carpet). Figure 5-20 illustrates how N-channel MOSFETS are constructed. (Scherz and Monk 2016)

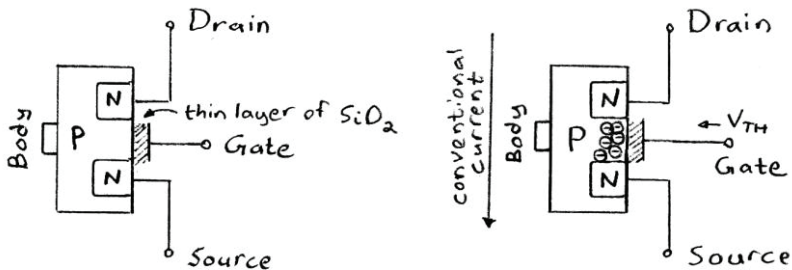


Figure 5-20. When  $V_{GS(th)}$  is applied to the gate terminal, an N-channel MOSFET allows conventional current to flow from drain to source.



### *Some properties of MOSFETs*

- MOSFETs can handle surprisingly high amounts of current, e.g. FQP30N06 (N-Channel MOSFET,  $I_D=32\text{ A}$ ).
- They are comparable in speed to BJTs (great for high speed applications).
- They have a very high input impedance. This means that they draw very little current from the logic side (potentially saving battery power).
- They have very low output impedance. This means that lots of current can flow through them on the power side.
- Unlike BJTs, MOSFETs are triggered by changes in voltage, not current.
- The body of the MOSFET is usually tied to the source, which is electrically connected to the metal tab on the MOSFET (if there is one, e.g. TO-220 package).
- You can bias a MOSFET like a transistor in the active region, but MOSFETs are not very reliable in the active range. They work better as on/off switches.

### *Using a MOSFET as a Switch*

Make sure when powered on, the MOSFET you select can provide more than enough current for your load.

If the power you are driving is  $>0.25\text{ W}$ , then use a MOSFET package capable of accepting a heat sink, and attach one (e.g. the TO-220 package). The heat sink should not be electrically connected to the source, because it could touch/short out other components. Commercially available heat sinks come with thin insulating layers to place between the chip and the heat sink (see Figure 2-18).

### *2N7000 (N-Channel MOSFET)*

**Specifications:** (ON Semiconductor Corp 2011)

Maximum Drain Current: 200 mA (continuous), 500 mA (pulsed)

Voltage between Drain (D) and Source (S):  $V_{DSS} = 60\text{ V}$  (max)

Resistance between D&S (on):  $R_{DS} = 1\text{-}2\ \Omega$  (ideally this should be small)

2N7000 (TO-92)

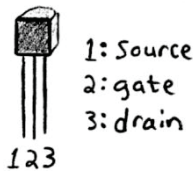


Figure 5-21. Pin-out diagram for 2N7000 N-channel MOSFET.

Gate Threshold Voltage:  $V_{GS(th)} = 2.1 \text{ V}$  (Arduino Uno can turn this on with +5V)

Thermal Resistance:  $R_{\theta JA} = 312.5 \text{ }^\circ\text{C/W}$

Turn-on and Turn-off time: 10 ns

## TRIACs

A **TRIAC** (short for “Triode for Attenuating Current”) provides a way to switch AC circuits using a DC signal. TRIACs are *current gated*, and look identical to transistors and MOSFETs (especially when produced in the same package, e.g. TO-92 or TO-220). However, they are able to handle the high voltage swings of

alternating current, and are also able to switch quickly enough to dim AC power (which as it turns out is much slower than PWM, at a pokey 50 or 60 Hz). TRIACs come with three leads: Gate, MT1, and MT2. The circuit diagram symbol for a TRIAC is provided in Figure 5-22. Figure 5-23 illustrates a very simple on/off circuit for an AC load, e.g. an incandescent microscope lamp.

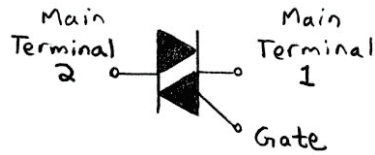


Figure 5-22. Circuit diagram symbol for a TRIAC.

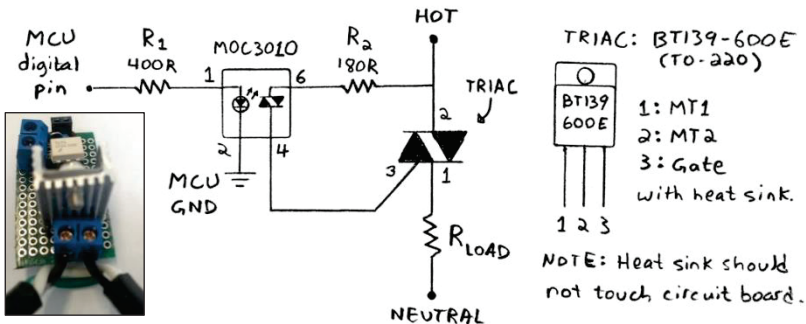


Figure 5-23. A TRIAC, controlled by microcontroller digital pin for a resistive load connected to 60 Hz, 120V AC power. This circuit can interrupt the hot wire in the middle of an extension cord, to create a high-side switch.

Notice the different notation for AC power in Figure 5-23 (**HOT** and **NEUTRAL**). For this circuit, although “NEUTRAL” is essentially the ground (or return) wire for AC, it does not need to be electrically connected

to the Arduino Uno's ground. In fact, it's much safer to electrically isolate the MCU from AC power completely. The circuit above could also work by connecting the TRIAC's pin 3 directly to the MCU through a current-limiting resistor, but then the MCU would not be electrically isolated from the dangers and noise of *mains electricity* (electricity from a wall outlet). An extra component does this job: the MOC3010. This is an opto-isolator chip that shines an IR LED (connected to pins 1 and 2) to turn on an internal switch (connected to pins 4 and 6). The MOC3010 isolates AC power from the MCU, protecting it in case the TRIAC fails. For switching an inductive load on and off like an AC motor, see the top half of the circuit diagram in Figure 5-24. For switching 240V AC power, consult the MOC3010 datasheet. (Fairchild Semiconductor Inc. 2014)

A heat sink might be required for the TRIAC, depending on how big the load current is in the above circuit. Setting this circuit up is quite easy, as it is similar to setting up a transistor with a few more needed parts. However, the shock hazard makes it dangerous to play around with in the lab, so there are no planned activities using TRIACs.

If you'd like to *dim* AC power, the circuit above won't do it. What first comes to mind is, why don't you send a PWM signal to the TRIAC to dim your light? Although that sounds like a good idea, switching the AC voltage on and off at the Arduino's PWM frequency will be out of phase with mains electricity. Even at narrow PWM widths, you might see flickering, but not dimming. To dim an AC signal, you need to detect where the voltage crosses at zero (e.g. with a zero-crossing detector like the H11AA1), then turn the TRIAC off or on, depending on your dimming strategy. An example of this implementation is provided in Figure 5-24. As with Figure 5-23, a heat sink should be used, and it should not touch the circuit board to prevent the chance of a short circuit. (Loflin 2018; Fairchild Semiconductor Inc. 2014)

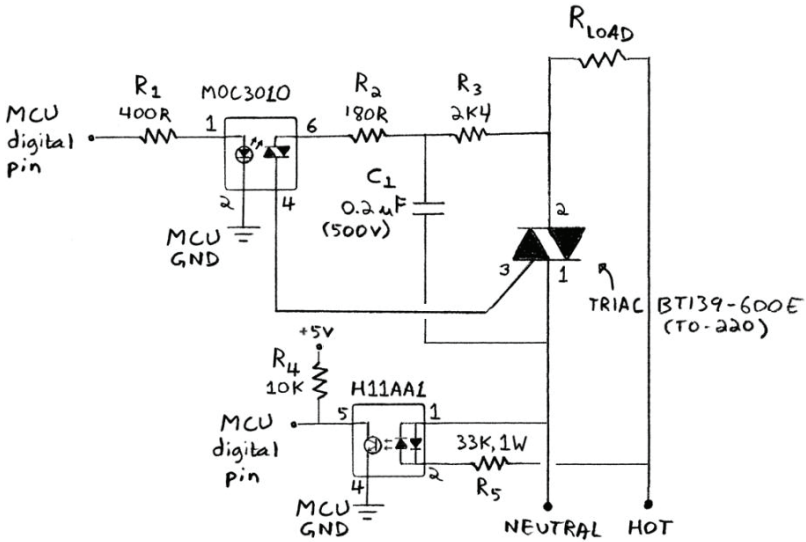


Figure 5-24. TRIAC with zero-crossing detector, for an inductive or resistive load. This circuit is a low-side AC dimming switch.

If you wait to switch the power *on* after detecting a zero crossing, it's called forward-phase dimming, and if you wait to switch the power *off* after detecting a zero crossing, it's called reverse-phase dimming. The dimming effect is proportional to how much time the circuit spends *off* in a cycle. This idea is analogous to a well-timed PWM. Forward phase and reverse phase dimming are illustrated in Figure 5-25.

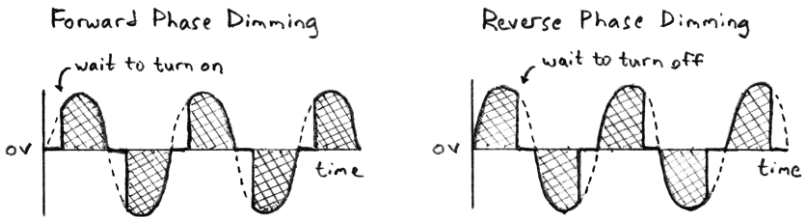


Figure 5-25. Forward phase dimming (left) and reverse phase dimming (right) with a TRIAC. The TRIAC is switched on and off strategically to chop an AC supply into narrower widths to dim a resistive load. (Coleman 2015)

Getting the timing right is crucial for being able to dim a lamp, or modulate the power on an AC device (like a heater or motor). An example sketch *triacDimmer.ino* is provided in the appendix. For an easy-to-use library, see Anson Mansfield's *TriacDimmer*, available through the Library Manager of the Arduino IDE. (Mansfield 2017)

### ***BT139-600E (TRIAC)***

The BT139-600E in Figure 5-23 is a great light-duty TRIAC, although the legs will not fit into a breadboard, nor should they—breadboards can't support mains electricity. The TO-220 package it comes in is heat-sinkable, which is a good idea for higher-current loads. The following maximum values are provided on the NXP datasheet for this component: (NXP Semiconductors 2013)

- $V_{\text{DRM}}$ : (repetitive peak off-state voltage): 600V (max)
- $I_{\text{T(RMS)}}$ : (RMS on-state current): 16A
- $I_{\text{GT}}$ : (gate trigger current): 2.5-10 mA (max 2A)

Since this circuit involves soldering and high voltage from the wall outlet, this is not a project for a beginner. Relay strategies are safer in general if dimming the load is not an essential design feature. Relays can also be pulsed slowly to attain better control.

## **Protecting your Circuit from DC Motors**

Small, low amperage DC motors provide few problems when powered directly from a microprocessor. However, larger DC motors, particularly when switched using transistors, MOSFETs, H-bridges, or relays, can generate lots of noise which can damage downstream parts, and reset your microcontroller. When power is cut from a DC motor, rotational momentum keeps it spinning for a bit. For a short while, it acts as a generator, producing voltage in the wrong direction that can damage the circuit. This can be dealt with using two common strategies: protection diodes, and capacitors.

### ***Protection Diode***

A diode can keep current from running backwards when a DC motor generates electricity. This is generally a good idea whenever you are driving any **inductive load**, like a hobby motor or fan. An inductive load converts electricity to a magnetic field, or vice-versa. *Flyback* happens when you suddenly stop an inductive load, and the magnetic field that has been

generated from electricity *back-converts* to electricity, causing a voltage spike, and a surge in the opposite (wrong) direction. Flyback can destroy electronic components. A diode wired directly across the load ties up the flyback voltage in a short loop, to dissipate harmlessly across the inductor and diode. In this context, we will call the diode a **protection diode** or **flyback diode**. It protects the transistor, and any downstream components.

Most small DC motors can run uneventfully without protection diodes for a short while, but diodes are inexpensive (pennies each!) and cheap insurance for the rest of your circuit. A protection diode has many alternative names (e.g. snubber diode, catch diode), but amounts to connecting a suitably-selected silicon diode across the terminals of an inductive load like a DC motor (Figure 5-26). You can also have another look at Figure 5-4, where a protection diode is connected across the relay coil, protecting the transistor below it.

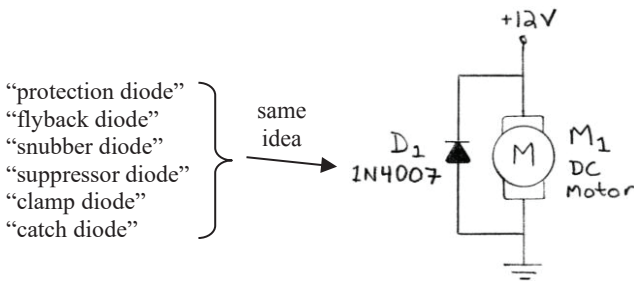


Figure 5-26. Protection diode on a DC motor. In this configuration, the DC motor can only spin in one direction.

It is *very* important not to accidentally install a protection diode backwards. Doing so creates a short circuit between high voltage and ground, causing damage (and usually, a very blown diode). Remember to keep the printed stripe on a protection diode pointed towards the *high* side of voltage.

### *Reducing DC Motor Noise with Capacitors*

Although most H-bridges have built-in protection diodes (e.g. the L293D DIP Quadruple Half-H Driver), larger hobby DC motors draw enough current when starting or stopping that can unwantedly reset the microcontroller. One practical fix to dampen voltage dips and spikes is to solder non-polarized (usually ceramic) capacitors directly across the motor terminals, as shown in Figure 5-27.

For most practical cases, a 0.1  $\mu\text{F}$  capacitor will be sufficient for this purpose. Alternately, capacitors can be soldered from each terminal to the metal housing of the DC motor, or both strategies can be used together: one 0.1  $\mu\text{F}$  capacitor connecting both terminals, and one 0.1  $\mu\text{F}$  capacitor connecting each terminal to the DC motor body. (Pololu Corporation 2015)

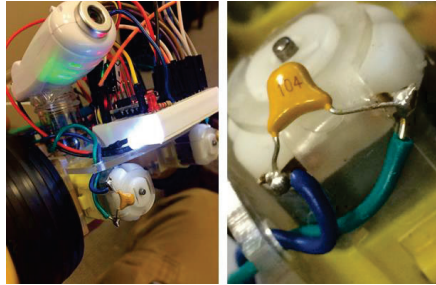


Figure 5-27. Noise-reducing capacitor on “Bob”, the laboratory video rover. These capacitors fixed his nasty resetting problem.

### Activity 5-1: Hot Plate Thermostat

**Goal:** In this activity, we are going to use a 10A relay module with your thermistor circuit (Activity 4-1). A relay takes the place of the LED light on pin 6.

#### Materials:

- Arduino Uno MCU & USB cable
- Laptop with Arduino IDE installed
- Digital Multimeter
- Breadboard
- 1 x 10K Resistor (1% tol)
- 1 x 10K Thermistor (Section 4)
- 1 x 5V, 10A Relay Module
- 1 x Hot Plate
- 3 x Male/Male Jumpers
- 3 x Male/Female Jumpers (long)
- 1 x 400 mL Beaker

#### Procedure:

Build the circuits in Figure 5-28.

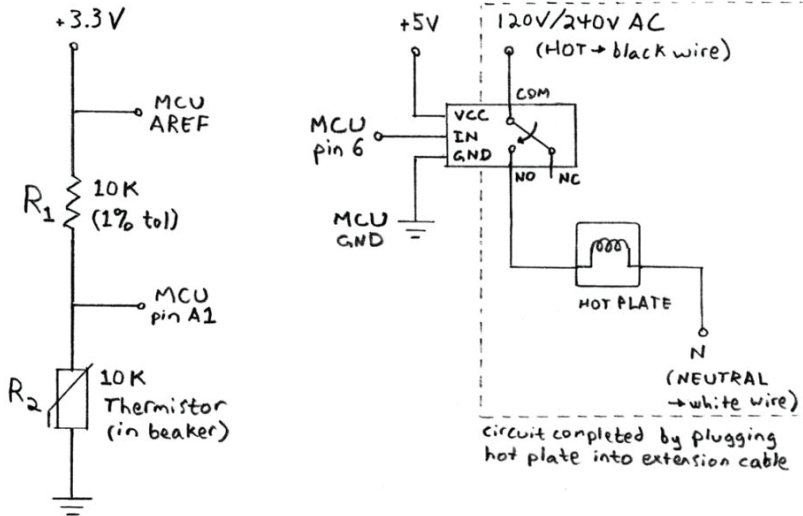


Figure 5-28. Circuit diagram for Activity 5-1: hot plate thermostat.

**Note:** The relays in class are pre-wired to an extension cord. You do not need to make relay connections to AC power for this exercise.

**Test your understanding:** The relay is wired to switch the hot wire, in the normally open terminal. Why is switching the hot wire safer than switching the neutral wire?

**Warning: AC shock hazard.** Once you plug in the relay cable to AC power, be careful not to touch any metal parts on the relay module. Also make sure that the relay module is not touching any other metal parts or bare wires.

- 1) Fill a 400 mL beaker with 150 mL of water. Set the beaker on a hot plate.
- 2) Obtain the thermistor circuit you calibrated from Activity 4-1.
- 3) Connect the circuit diagram above.
- 4) Immerse the thermistor probe tip in the water.
- 5) If you finished your sketch from last class, you can use it in this exercise. Otherwise, download the file *Thermostat.ino* from the course website, and upload the program to your Arduino Uno.
- 6) Download *GetCSV.xlsm* to acquire the data into Microsoft Excel, in real time.

**Test your understanding:** What part of the sketch turns the relay module on and off?



## Activity 5-2: Transistor as a Switch for a DC Motor

**Goal:** In this activity, we will connect the Arduino Uno to a 12V DC motor. Although the Uno supplies +5V DC, it does not supply enough current to drive the motor at top speed, so an external power source is required. A protection diode protects the transistor from DC motor flyback.

### Materials:

- Arduino Uno MCU & USB cable
- Laptop with Arduino IDE installed
- 1 x ATX Power Supply
- 1 x 12V DC Motor
- 1 x 10K Resistor
- 1 x 2N2222 Transistor
- 1 x 1N4007 Diode
- 5 x Male/Male Jumpers

### Procedure:

Build the circuit in Figure 5-29.

### Notes:

- Be careful! External power supplies like to fry microprocessors.
- The black fan wire does *not* connect to ground.

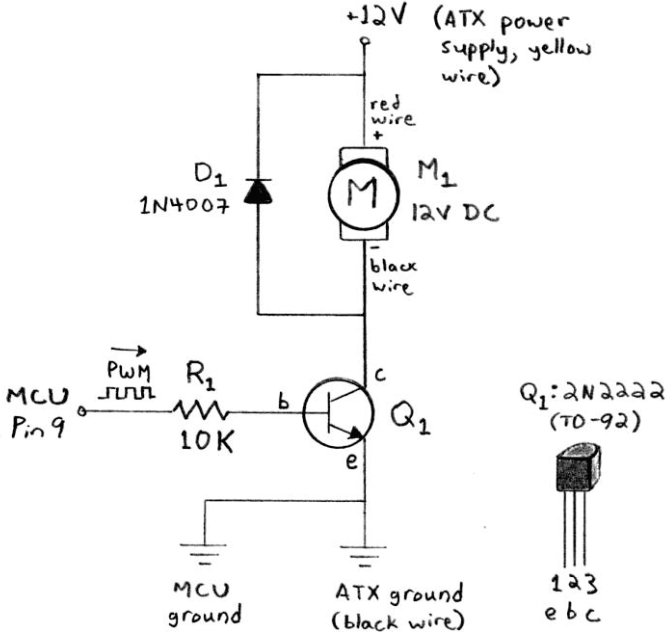


Figure 5-29. Schematic for Activity 5-2: NPN transistor-controlled DC motor.

To hook up the external power supply, connect the **yellow** wire (+12V) from the ATX floppy drive connector to the load side of the circuit, using a male/male jumper. The black wire (ground) of the ATX floppy drive connector should be connected to the emitter, AND to the Arduino Uno ground. The floppy connector is shown in Figure 5-30.

The ATX power supply has been modified by connecting the green wire on the main power connector to a black (ground) wire so that it will power on when plugged in. For a pin-out table of the main ATX power connector, see Table A-7 in the appendix.

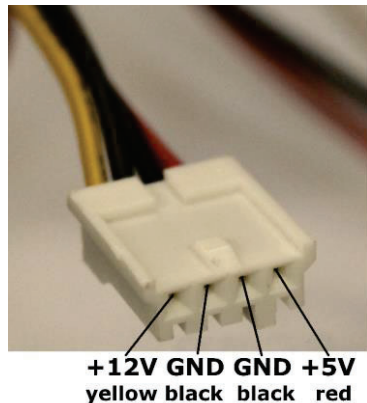


Figure 5-30. Floppy drive connector of an ATX power supply, capable of supplying up to 3 amps. ATX power supplies are salvageable from old desktop computers.

We will be using our own PWM strategy from before to control motor speed. To shake things up a bit, we will be using the serial monitor to set the speed (between 0 and 255), while the sketch is running. To accomplish this, write the following sketch:

```
//Activity 5-2: serial-controlled motor speed
const byte motorPin=9;
int motorSpeed=0;

void setup(){
 Serial.begin(9600);
 pinMode(motorPin,OUTPUT);
 Serial.println("");
 Serial.println("Enter motor speed from 0-255:> ");
}

void loop(){
 if(Serial.available()){ //if user entered smthng
 motorSpeed=Serial.parseInt(); //read&conv to int
 analogWrite(motorPin,motorSpeed); // set PWM
 }
}
```

Compile and upload this program. Test out and refine your circuit.

#### Notes:

- Start the serial monitor to test your circuit. Make sure the serial monitor is set to the correct baud rate (9600 bps).
- Make sure that "No line ending" is selected on the serial monitor pull-down menu.
- To enter the number for motor speed, type it in the status bar of the serial window, then press "Send" (not Enter) to send it to the MCU.
- In this sketch, we don't try to read something from the serial buffer unless there is data available, checking with the command `Serial.available()`. This command returns `true` if the user has entered something new, and `false` if they haven't.

#### Consider:

- Constraining the values that the user enters to the *working range* of the motor (the range of numbers that spans all motor speeds):  
`motorSpeed=constrain(motorSpeed,0,255);`

With the `constrain()` command, any number less than the lower limit (first number) will get bumped up to the lower limit, any number greater than the upper limit (second number) will get bumped down to the upper limit, and any number in between the two limits will remain the same.

- Checking that the data the user entered is a number, and returning a warning if it wasn't.
- Serial commands are also possible between two microcontrollers. Microcontrollers can talk to each other, and other devices through serial communications!

You can control motor speed using a potentiometer, read using `analogRead()`, a function (e.g. a sine wave), or simply on/off control (like the hot plate thermostat) with a setpoint, using `digitalWrite()`. Motor speed can also be adjusted based on data from a limit switch, proximity sensor, or gyroscope, which becomes important in mechanical and robotic applications.

### *Parsing Serial Data*

In the last sketch, we used the `Serial.parseInt()` command to convert the data received from the serial monitor window into an integer. There are similar functions available for converting serial data to other variable types, if you need them:

```
myStringVar = Serial.readString(); // read String
myIntVar = Serial.parseInt(); // read int
myFloatVar = Serial.parseFloat(); // read float
myByteVar = Serial.read(); // read byte
```

### **Activity 5-3: MOSFET as a Switch for a DC Motor**

**Goal:** The following circuit replaces the 2N2222 transistor in Activity 5-3 with a 2N7000 N-Channel MOSFET. The 10K pull-down resistor is insurance against a floating pin state, which could potentially cause the gate to turn on and off randomly. A protection diode protects the MOSFET from DC motor flyback.

#### **Materials:**

- Arduino Uno MCU & USB cable
- Laptop with Arduino IDE installed
- 1 x ATX Power Supply
- 1 x 12V DC Motor
- 1 x 10K Resistor
- 1 x 27000 N-Channel MOSFET
- 1 x 1N4007 Diode
- 5 x Male/Male Jumper

**Procedure:**

Build the circuit in Figure 5-31, following the same procedure as Activity 5-2.

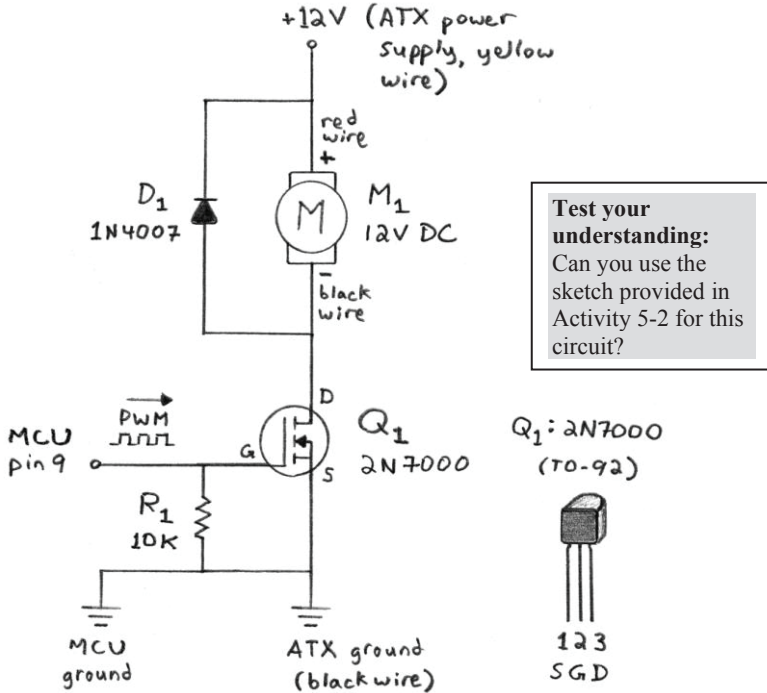


Figure 5-31. Circuit diagram for Activity 5-3: MOSFET-controlled DC motor.

## Learning Objectives for Section 5

After having attended this class, the student will be able to:

- 1) Identify the voltage and power limits of the Arduino Uno.
- 2) Match the current, voltage, and power requirements of a given load to an appropriate switching strategy: relay, transistor, MOSFET, or TRIAC, so that a suitable device can be selected for specific design requirements.
- 3) Discern high-side from low-side switching, and describe the safety implications of each.
- 4) Define and describe switch construction: number of poles, throw number, normally open, and normally closed.

- 5) Identify which switching strategies require a common ground, and which allow for separate grounds for logic and power.
- 6) Be able to illustrate, using diagrams, how diodes, relays, NPN and PNP BJTs, and MOSFETs work.
- 7) Write a simple sketch for a thermostat circuit.
- 8) Identify and draw from memory the circuit diagram symbols for the switching devices in this section.
- 9) Identify proper placement of a protection diode to protect a circuit from inductive load flyback.
- 10) List and define the three transistor modes: cutoff, active, and saturation.
- 11) Correctly label the collector, base, and emitter terminals on NPN and PNP BJTs, and the gate, drain, and source terminals on N-channel and P-channel MOSFETs.
- 12) Compare and contrast the functionalities of BJTs vs. MOSFETs.

## Section 5 - Station Content List

- Arduino Uno MCU & USB cable
- Laptop with Arduino IDE installed
- Digital Multimeter
- Breadboard
- 1 x 10K Resistor (1% tol)
- 1 x 10K Thermistor from Section 4
- 1 x 5V, 10A Relay Module
- 1 x Hot Plate
- 3 x Male/Female Jumpers (long)
- 1 x 400 mL Beaker (1/3 filled tap water)
- 1 x 1L Plastic Beaker (with ice water)
- 1 x Glass Thermometer
- 1 x ATX Power Supply
- 1 x 12V DC Motor
- 1 x 10K Resistor (5% tol)
- 1 x 2N2222 Transistor
- 1 x 27000 N-Channel MOSFET
- 1 x 1N4007 Diode
- 8 x Male/Male Jumpers

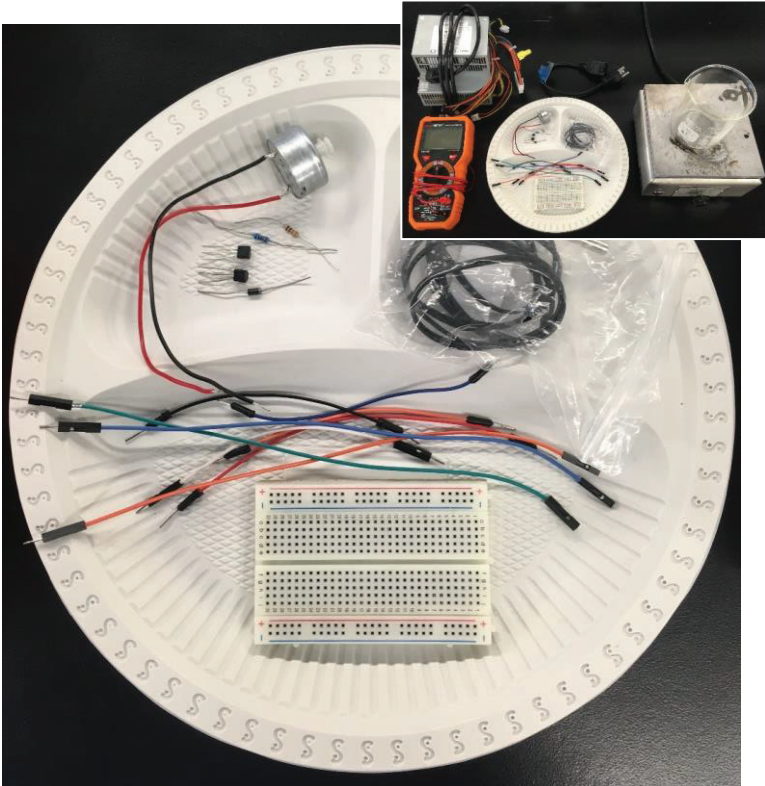


Figure 5-32. Section 5 station setup.

# SECTION 6

## PROCESS CONTROL

|                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |                                                                                                                               |
|-------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------|
| What You'll Be Learning | <p><b>Lecture:</b> Types of motors: DC motor, stepper motor, servo motor. Using an H-bridge to drive a motor (example: L298N motor driver). Introduction to process control. Open loop, feed forward, feedback. On-off controller, P+I+D controller. Interpreting system responses: undamped, underdamped, overdamped, critically damped.</p>                                                                                                                                                                                                                                                                                                                                                                                                                   |                                                                                                                               |
| What You'll Be Doing    | <p><b>Pick (Activity 6-1 and 6-3) or (Activity 6-2(a or b) and 6-4)</b></p> <p><b>Activity 6-1:</b> Using an H-bridge to control speed and direction of a regular DC motor.</p> <p><b>Activity 6-2(a):</b> Using a ULN2003 stepper motor driver to control a 28BYJ-48 stepper motor.</p> <p><b>Activity 6-2(b):</b> Using an A4988 stepper motor driver to control a Nema-17 stepper motor.</p> <p><b>Activity 6-3:</b> Setting up a simple servo motor control circuit, using a potentiometer to control servo position.</p> <p><b>Activity 6-4:</b> Tuning a PID controller (program ready) for temperature control of a circuit board.</p> <p><b>Demo 1:</b> Transistor-switched IR robot (H-bridge).</p> <p><b>Demo 2:</b> Video rover: L298N H-bridge.</p> |                                                                                                                               |
| Files you will need     | <p>All course files are available for download at:<br/> <a href="http://pb860.pbworks.com">http://pb860.pbworks.com</a></p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | <ul style="list-style-type: none"> <li>• <i>28BYJ-48.ino</i></li> <li>• <i>A4988.ino</i></li> <li>• <i>PID.ino</i></li> </ul> |

### When “Close Enough” Isn’t Close Enough

Controlling equipment sometimes requires a certain amount of precision, more than “turn this device on until this happens, and then stop”. DC motors are flexible, in that you can control their speed and direction of rotation. When more *precise* control of motion is required, stepper and servo motors have their own niches, finding utility in many scientific applications. This section focusses on the most common types of control strategies, the



principles of which extend to controlling many types of scientific and engineering systems.

## How a DC Motor Works

A DC motor is an ingenious device that works through the induction and reversal of a magnetic field using electricity. It essentially works through magnetic repulsion. When current runs through a coil, the coil becomes magnetized according to *Fleming's Left Hand Rule*, and is repulsed by fixed magnets, called stators. Early DC motors had brushes, although the majority of DC motors are now brushless. As the coil rotates, it reverses polarity because its contacts (through the brushes) connect to the other side of the commutator, so the process repeats. Figure 6-1 illustrates this cycle.

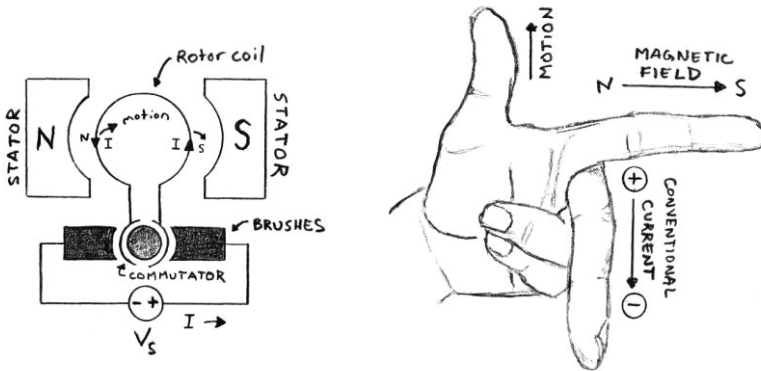


Figure 6-1. The rotor coil of a motor (left) will spin according to Fleming's left hand rule (right).

If you wire a DC motor backwards, the symmetry of the device results in the shaft turning in the opposite direction (e.g. counter-clockwise vs. clockwise). Usually, DC motors have more than one coil. There are a series of coils, resulting in smoother motion.

## Using an H-Bridge to Control Motor Speed and Direction

As mentioned above, if you reverse the power to a DC motor (switch (+) and (-) wires), it will spin in the opposite direction. If you would like a motor to be able to switch directions during its regular operation, re-wiring

it on the fly would be inconvenient. An **H-bridge** can change the direction of current through a DC motor using switches, without having to rewire it. A basic H-bridge circuit diagram is presented in Figure 6-2. You can see where it gets its name, since the wires form the shape of a letter H.

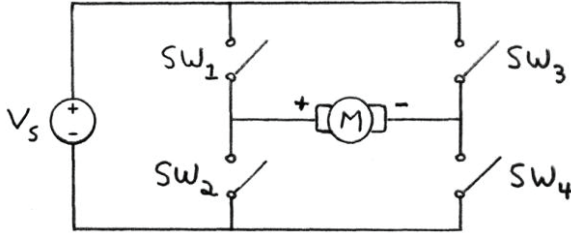


Figure 6-2. Simple H-Bridge configuration.

By opening and closing the appropriate switches, we can get a DC motor to reverse direction. Table 6-1 shows how this is done. Follow the voltage from positive to negative to see how the direction of current is reversed.

**Table 6-1. An H-bridge can switch the polarity across a motor by changing the states of four switches.**

|                                                                                                                                                                                           |                                                                                                                                                                                            |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>If SW<sub>1</sub> and SW<sub>4</sub> in this circuit are closed, and SW<sub>2</sub> and SW<sub>3</sub> are open, a regular DC motor will spin <i>clockwise</i> (shaft facing you):</p> | <p>If SW<sub>1</sub> and SW<sub>4</sub> in this circuit are open, and SW<sub>2</sub> and SW<sub>3</sub> are closed, the DC motor will run <i>counter-clockwise</i> (shaft facing you):</p> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

The orientation of the switches is key to how the H-bridge works. The switches themselves can be relays, BJTs, MOSFETs, or other types of switches. The (+) and (-) signs on the motor in the above diagrams do not represent voltage, but rather the polarity of the motor. Positive (+) is usually the DC motor's red wire, and negative (-) is the DC motor's black wire.

You can quite easily make your own H-bridge. Something to be aware of is the chance of shorting while switching. If top and bottom switches (e.g. SW<sub>1</sub> and SW<sub>2</sub>) are closed at the same time—even for a VERY short time—there will be a short circuit. This is hard on the microcontroller, and your power supply. The proper sequence of turning the switches on and off will prevent this from happening.

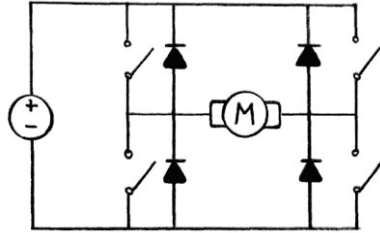


Figure 6-3. Protection diodes for an H-bridge.

You can also protect your H-bridge with protection diodes (Figure 6-3). This is important for larger DC motors.

In addition to controlling the *direction* of the DC motor by opening and closing the correct switches on an H-bridge, you can control the *speed* of the DC motor by powering it (or switching it) using a PWM signal. Higher duty cycles will spin the motor faster. This little trick of dimming an LED is made use of in so many other applications.

### ***L298N H-Bridge Motor Driver Module***

There are a variety of integrated H-bridge chips (like the L293D), and modules (like the L298N motor driver module). An H-bridge module is also called a *motor driver*, *motor drive*, or *motor controller*. The L298N motor driver has two independent H-bridges built into it, so it can control two DC motors independently, or one low-current bipolar stepper motor. It has built-in protection diodes, so you don't have to worry about flyback, and **+5V out** that can be used to power a +5V microprocessor board through its Vin pin, allowing you to power your project (MCU and motor) using a single supply.

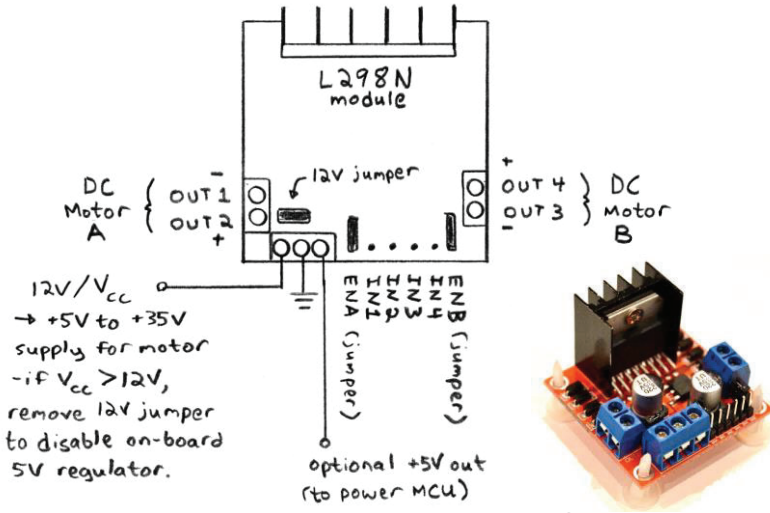


Figure 6-4. L298N H-bridge module. This module comes with an on-board regulator that you can also use to power the logic side of your circuit. Check the top and underside of the module to confirm pin and screw terminal identities, as some modules may vary.

In Figure 6-4, *IN1* and *IN2* are controls for the built-in H-bridge for DC motor A. In this way, you can use +5V signals from your microcontroller to power a motor with much higher voltage. *ENA* means “enable motor A”. If *ENA* is left floating, or at GND, *OUT1* and *OUT2* will both be LOW. *ENA* needs to be HIGH for DC motor A to spin, which is why there is a jumper for tying it to +5V for a stepper motor. If you remove the jumper on *ENA*, you can control the motor speed of DC motor A by sending a PWM signal directly to *ENA*.

**Table 6-2. L298N H-bridge control of two independent DC motors.**

| Input Pin State      | Output Pin State                   | Result                                      |
|----------------------|------------------------------------|---------------------------------------------|
| IN1: GND<br>IN2: +5V | OUT1: GND<br>OUT2: V <sub>cc</sub> | DC motor A rotates <i>clockwise</i>         |
| IN1: +5V<br>IN2: GND | OUT1: V <sub>cc</sub><br>OUT2: GND | DC motor A rotates <i>counter-clockwise</i> |
| IN3: GND<br>IN4: +5V | OUT3: GND<br>OUT4: V <sub>cc</sub> | DC motor B rotates <i>clockwise</i>         |
| IN3: +5V<br>IN4: GND | OUT3: V <sub>cc</sub><br>OUT4: GND | DC motor B rotates <i>counter-clockwise</i> |

Similarly, you can control the speed of DC motor B by sending a PWM signal to *ENB*. This makes connections easier, because you only need one PWM signal to control the speed of a DC motor.

### To Power One Regular DC Motor (Side A):

- 1) Open jumper ENA, by pulling it off the bridging pins, and sliding it back on one of the original pins (EN jumpers should be open for a regular DC motor).
- 2) Connect a PWM-capable pin (On the Uno: 3, 5, 6, 9, 10, 11) to ENA. This will control motor speed for DC Motor A in both directions.
- 3) Connect Arduino digital pins to IN1 and IN2.
- 4) To connect the power:
  - For a **5-12V** DC motor: Connect external +5 to +12V to  $V_{cc}$  (also labelled 12V on some L298N boards). Leave the +5V pin open. This pin provides a regulated +5V out from  $V_{cc}$ , that can optionally be used to power the Uno (or other 5V module).
  - For a **>12V** DC motor: Remove the 12V jumper, connect external power to  $V_{cc}$ , and don't use the on-board regulator (it can't handle more than 12V).
- 5) Connect the (-) and (+) terminals of the DC motor to OUT1 and OUT2, respectively.
- 6) Connect MCU GND to H-bridge GND. It's important if you are using separate power supplies (e.g. MCU powered using a laptop, and DC motor powered using an external power supply) that you directly connect the MCU ground to the H-bridge ground, or the H-bridge won't be able to receive the microcontroller's signals.

Repeat this method with IN3, IN4, ENB, OUT3, and OUT4 for a second DC motor.

Using the H-bridge for DC motor 1:

- For clockwise turning: Set IN1=LOW, IN2=HIGH.
- For counter-clockwise turning: Set IN1=HIGH, IN2=LOW
- Send a PWM signal to ENA to control motor speed.

Motor drivers are extremely sensitive to voltage spikes caused by wiring connections with the power turned on. Make all your connections and wiring changes with the power supplies unplugged.

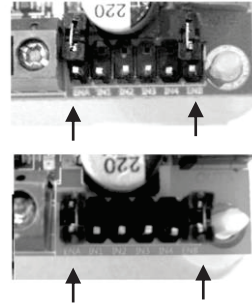


Figure 6-5. Pin jumpers in the *open* position (top), and in the *closed* position (bottom), L298N module.

## Stepper Motors

A *stepper motor* is a specialized type of DC motor that rotates much more slowly, in discrete steps. Stepper motors rotate by a small fixed angle with every DC pulse (rather than a constant applied voltage). The Nema-17 motor in Figure 6-6 has 200 steps per revolution, although other stepping angles are available in the Nema-17 package (e.g 400 steps). Stepping angle is calculated using the following formula:

$$\text{Stepping Angle} = \frac{360^\circ}{200 \text{ steps}} = 1.8^\circ/\text{step}$$

There are many kinds of stepper motors. Unlike regular DC motors, stepper motors are brushless, and have more than two wires to connect:

- Unipolar stepper motor: 5 or 6 wires
- Bipolar stepper motor: 4 wires

Regular DC hobby motors are high speed, low torque. Stepper motors are the opposite; they are very high torque and lower speed. Lower speed doesn't necessarily mean slow, as some stepper motors are capable of 1000 rpm rotation rates. Stepper motors use a lot more current than regular DC motors, but are more precise, which makes them preferred for 3D printing, CNC milling, and laser cutting. A stepper motor can spin continuously (360° range of rotation), and can also reverse direction. Figure 6-7 shows the circuit diagram symbols for unipolar and bipolar stepper motors, and how they are connected to their respective motor drivers. (Leger 2012) The motor drivers have been greatly simplified to only show the orientation of switches, which are drawn in as transistors but could be other types of switches as well. Unipolar steppers tend to have 6 wires (as shown) or 5 wires if the two  $V_{cc}$  leads are made common. A bipolar motor has 4 wires. A bipolar motor has higher torque than a comparable unipolar motor. Interestingly, there are easy ways to convert a unipolar motor to bipolar, to increase torque. (Adriaensen 2013) Despite the many switches shown for the bipolar stepper motor in Figure 6-7, symmetry results in only four microcontroller wires needed at most to control the motor. Motor driver modules can reduce the number of wires even further.

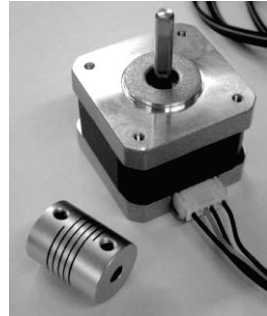


Figure 6-6. A Nema-17 bipolar stepper motor with 5mm shaft coupler.

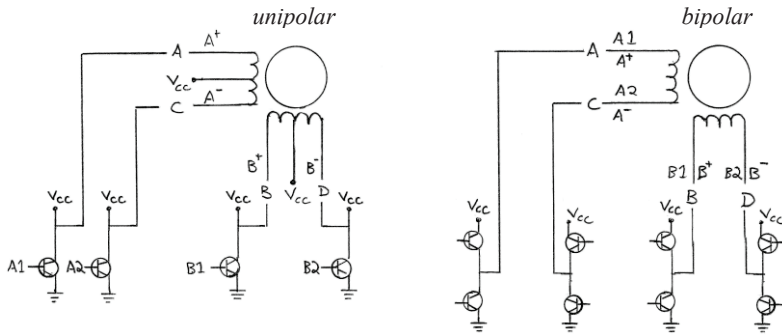


Figure 6-7. Circuit diagram symbols for unipolar (left) and bipolar (right) stepper motors. A unipolar stepper motor can be controlled by 4 switches, whereas a bipolar stepper motor requires two H-bridges to operate.

### ***28BYJ-48 Stepper Motor with ULN2003 Motor Driver***

Thanks to mass production, the 28BYJ-48 stepper motor is inexpensive and widely available through electronics and auction webstores. It is usually sold together with the ULN2003 motor driver module. This capable unipolar 5-wire stepper motor comes in 5V and 12V versions. The motor has 4 coils (or phases) and a stepping angle of  $5.625^\circ$  (or 64 steps per revolution). However, it is geared down so that 64 revolutions of the internal motor maps to one revolution of the external 5mm motor shaft. This results in a very small stepping angle of  $0.08789^\circ$ , or 4096 steps per revolution.

The 5V version of the motor can be powered with the ULN2003 motor driver directly from the Uno without an external 5V power source, although an external power supply is recommended. There are many libraries available to control this motor. The following sketch illustrates how easy it is to control the 28BYJ-48 stepper motor without a library. The sketch (*28BYJ-48.ino*) is also available for download on the course website:

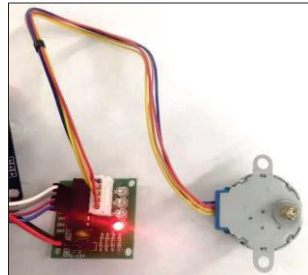


Figure 6-8. A 28BYJ-48 stepper motor, with ULN2003 driver module.

```

/* Test sketch: 28BYJ-48 Stepper with ULN2003
const int stepsPerRev=4096;
const byte IN[4]={8,9,10,11}; // define motor pin array
void setup(){
 Serial.begin(9600);
 for(int i=0;i<4;i++){
 pinMode(IN[i],OUTPUT); // set pins to output mode
 }
}

void loop(){
 Serial.println("Stepping clockwise.");
 motorStep(4096,10); // CW 4096 steps @10rpm
 delay(1000);
 Serial.println("Stepping counter-clockwise.");
 motorStep(-4096,10); // CCW 4096 steps @10 rpm
 delay(1000);
}

void motorStep(int mSteps, float rpm){
 //convert rpm to time delay:
 float t=60000.0/(rpm*stepsPerRev);
 const bool mSequence[8][4]={
 {1, 0, 0, 1}, // step 0
 {1, 0, 0, 0}, // step 1
 {1, 1, 0, 0}, // step 2
 {0, 1, 0, 0}, // step 3
 {0, 1, 1, 0}, // step 4
 {0, 0, 1, 0}, // step 5
 {0, 0, 1, 1}, // step 6
 {0, 0, 0, 1} // step 7
 };
 static int mStep; // remember last val of mStep
 for(int i=0;i<abs(mSteps);i++){ // STEP pulses
 if(mSteps>0){ // clockwise
 mStep++;
 if(mStep>7)mStep=0;
 }else{ // counter-clockwise
 mStep--;
 if(mStep<0)mStep=7;
 }
 for(int j=0;j<4;j++){
 digitalWrite(IN[j],mSequence[mStep][j]);
 }
 delay(t);
 }
}

```



You can see how the 8x4 array of boolean variables (`mSequence`) helps neatly define the pin states for the stepper motor wires. These array elements are used in the `digitalWrite()` statement to send 5V pulses to the motor in the correct sequence.

### *Nema-17 Stepper Motor with A4988 Motor Driver*

The Nema-17 stepper motor in Figure 6-8 gets its name from the dimensions (having a face plate length and width of 1.7”). This is not to be confused with a model number, since there are many different Nema-17 models, varying in operating voltage and current, holding torque, and stepping angle. The Nema-17 stepper motor has become extremely popular in 3D printing and CNC devices. The original version of this course had the Nema-17 driven by the L298N motor driver, but many of the driver modules burned out because they couldn’t handle the high current of these powerful motors. One solution is to deliver 50% duty cycles to ENA and ENB to the L298N module to limit the current, but a better solution is using a more appropriate motor driver like the Pololu A4988 (capable of driving up to 1.5A stepper motors) or DRV8825 (capable of driving up to 2A stepper motors). These motor drivers need to be “tuned” to deliver the correct amount of current to the stepper motor. Too little current will result in skipped steps, erratic steps, weaker torque, or no movement at all. Too much current will result in loud, jittery movement, overheating, and may damage the driver and motor. These motor drivers are inexpensive, but sensitive – which is why many of them now live in the “toast” bag in my lab.<sup>5</sup>

Despite the finicky setup, the A4988 motor driver offers superior control for the Nema-17, allowing for full, half, quarter, eighth, and even sixteenth steps of the motor. Moving a stepper motor by a fraction of a step is called **microstepping**. Given that many Nema-17s have 200 steps per revolution, this means that  $200 \times 16 = 3200$  steps per revolution are possible. Once the driver is properly set up, it is extremely reliable.

According to Pololu, these motor drivers are sensitive to voltage spikes, and must be connected carefully with the power off. (Pololu Corporation 2015) The schematic in Figure 6-9 can be used to wire a Nema-17 to a microprocessor with an A4988 stepper motor driver. Figure 6-9 has jumpers

---

<sup>5</sup> When a component burns out, don’t make the mistake of throwing it away. Perhaps it just needs a rest from thermal overload. At the very least, keep it for spare parts. Many components that were thought blown up have been revived from the “toast” bag, or scavenged for other projects.

connecting pins ENABLE, MS1, MS2, and MS3 to the microcontroller. These pins on the A4988 have internal pull-down resistors, so if you are not planning on microstepping the stepper motor, you can leave those connections out in your circuit and they will default to ENABLE=0, MS1=0, MS2=0, and MS3=0, for regular stepper motor operation.

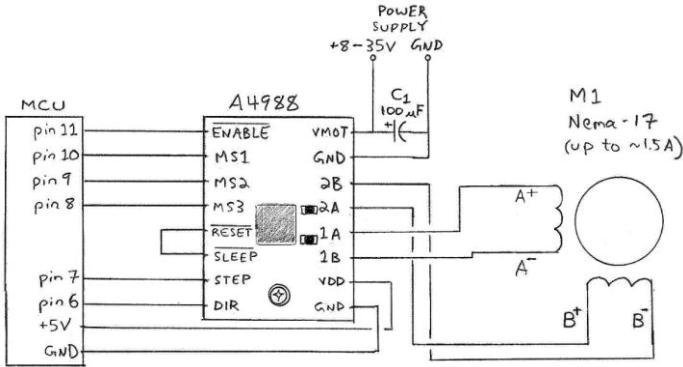


Figure 6-9. Connecting a Nema-17 to a microprocessor with an A4988 stepper motor driver. Build this circuit with the power off.

Connecting the stepper motor to the motor driver requires identifying Nema-17's four wires. Which wires map to 1A, 1B, 2A, and 2B? Unfortunately, there are no universal colour code conventions for stepper motor wires. The datasheet of your specific motor might specify the pin allocations. Many stepper motor connector cables flip positions of the middle two pins, so watch out for that when you are making your connections. The following is a suggested starting point: (Hobby CNC Australia 2015)

**Table 6-3. Suggestions for 4-Wire stepper motor colour codes.**

| Coil | Scheme 1 | Scheme 2 | Scheme 3 | Scheme 4 | Scheme 5 | Connect to A4988 pin |
|------|----------|----------|----------|----------|----------|----------------------|
| A+   | Red      | Red      | Yellow   | Red      | Yellow   | 1A                   |
| A-   | Blue     | Grey     | Blue     | Blue     | Orange   | 1B                   |
| B+   | Black    | Yellow   | Red      | Yellow   | Black    | 2A                   |
| B-   | Green    | Green    | Green    | White    | Brown    | 2B                   |

Unfortunately, finding the correct terminals for a stepper motor can be tricky if the terminals are unknown. One effective approach is to measure the resistance (or test for continuity) across *any two* of the four motor wires. If the resistance is low (e.g. less than 100Ω), then you have found a pair of terminals leading to the same coil (e.g. A+ and A-), and the other two wires are the second pair of wires leading to the second coil (e.g. B+ and B-). You can therefore narrow down your wires to two pairs, each pair controlling a separate coil. Wire the first pair to 1A and 1B, and the second pair to 2A and 2B. Put a bookmark on this procedure here, and now it's time to set the current for the A4988.

The A4988 has a tiny trim potentiometer that you can set with a small screwdriver or microspatula. Make sure all connections are firm and snug in Figure 6-9, and it's time to turn on the power to the microprocessor (logic side). With a voltmeter, measure the voltage from any ground in your circuit (or the GND pin of the A4988) to the metal screw on the A4988's trim pot (see Figure 6-10). The goal is to adjust this voltage according to the following formula: (Pololu Corporation 2015)

$$V_{ref} = 8 \times I_{max} \times R_{cs}$$

where  $V_{ref}$  is the measured voltage,  $I_{max}$  is the maximum total current the motor driver will send to the motor, and  $R_{cs}$  is the resistance of the on-board current-sense resistor (a common value is 50 mΩ, but may vary depending on the version). As with all electronics, the math will get you close, then it's up to you to taper the current down if the current is running hot, or up if you don't see any movement at all. Table 6-4 provides some common values:

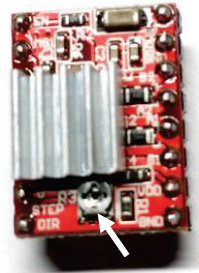


Figure 6-10. A4988 motor driver. White arrow points to trim potentiometer.

**Table 6-4.  $V_{ref}$  settings (in mV) for A4988 motor drivers.**

| $I_{max}$ (A) | $R_{cs}=50m\Omega$ | $R_{cs}=68m\Omega$ | $R_{cs}=100m\Omega$ | $R_{cs}=200m\Omega$ |
|---------------|--------------------|--------------------|---------------------|---------------------|
| 0.33          | 132                | 180                | 264                 | 528                 |
| 0.40          | 160                | 218                | 320                 | 640                 |
| 0.50          | 200                | 272                | 400                 | 800                 |
| 0.67          | 268                | 364                | 536                 | 1072                |
| 1.00          | 400                | 544                | 800                 | 1600                |
| 1.33          | 532                | 724                | 1064                | 2128                |
| 1.50          | 600                | 816                | 1200                | 2400                |
| 1.67          | 668                | 908                | 1336                | 2672                |
| 2.00          | 800                | 1088               | 1600                | 3200                |

You can find out the current requirements for your stepper motor ( $I_{\max}$ ) from the datasheet of the model number on the stepper motor body. Look up the  $V_{\text{ref}}$  setting for the required current in Table 6-4, then adjust the trim potentiometer as close as you can to this voltage. Once the reference voltage is set, it's time to turn on the power supply to the motor and test the stepper motor for movement. If you can't find the voltage or current requirements for your motor, you can start with the smallest  $V_{\text{ref}}$  and try slowly increasing it until your motor starts moving smoothly at the torque you need. The A4988 requires an 8-35V power supply, so it would not be a good choice for controlling a 5V stepper motor. Lower-voltage motor drivers like the L298N motor driver module, the L293D IC, or the A3967 EasyDriver (capable of microstepping) would be better suited to a 5V stepper motor.

There are many libraries available for the A4988 stepper motor; however, the following basic sketch will help start you off, and is also available on the course website (*A4988.ino*). The sketch should move the stepper motor forwards and backwards by 200 steps:

```
// A4988 test sketch
const int stepsPerRev=200; //change as needed for motor
byte n; // multiplier for microstepping
const byte ENA=11; // set ENA low to enable motor
const byte MS1=10; // microstepping pins
const byte MS2=9;
const byte MS3=8;
const byte STEP=7; // sends a pulse for each step
const byte DIR=6; // changes direction of stepper

void setup(){
 Serial.begin(9600);
 pinMode(ENA,OUTPUT); // set pins to output
 pinMode(MS1,OUTPUT);
 pinMode(MS2,OUTPUT);
 pinMode(MS3,OUTPUT);
 pinMode(STEP,OUTPUT);
 pinMode(DIR,OUTPUT);
 setMicroStep(0); // 0:full step mode (choose 0-4)
}

void loop(){
 Serial.println("Stepping clockwise.");
 motorStep(200,10); // clockwise 200 steps @10rpm
 delay(1000);
 Serial.println("Stepping counter-clockwise.");
 motorStep(-200,10); // CCW 200 steps @10rpm
 delay(1000);
}
```

```

void setMicroStep(byte RES){
 const bool MSvals[5][3]={
 {0, 0, 0}, // RES=0: full steps
 {1, 0, 0}, // RES=1: 1/2 steps
 {0, 1, 0}, // RES=2: 1/4 steps
 {1, 1, 0}, // RES=3: 8th steps
 {1, 1, 1} // RES=4: 16th steps
 };
 digitalWrite(MS1,MSvals[RES][0]);
 digitalWrite(MS2,MSvals[RES][1]);
 digitalWrite(MS3,MSvals[RES][2]);
 n=pow(2,RES); // calculate multiplier for steps/rev
}

void motorStep(int mSteps, float rpm){
 //convert rpm to time delay:
 float t=60000.0/(rpm*stepsPerRev*n*2.0);
 unsigned int timer=millis();
 if(mSteps<0){ // set dir (use mSteps>0 to reverse)
 digitalWrite(DIR,0); // counter-clockwise
 }else{
 digitalWrite(DIR,1); // clockwise
 }
 digitalWrite(ENA,LOW); // enable motor
 for(int i=0;i<abs(mSteps);i++){ // STEP pulses
 digitalWrite(STEP,HIGH);
 delay_(t);
 digitalWrite(STEP,LOW);
 delay_(t);
 }
 digitalWrite(ENA,HIGH); // disable motor
}

void delay_(float x){ // allows for delays <1ms
 if(x>1.0){
 delay(x);
 }else{
 delayMicroseconds(x*1000.0); //convert to usec
 }
}

```

Upload the sketch and see if the motor spins properly. You may need to exchange or reverse the wire pairs **with the power off** if you find the stepper motor doesn't spin well in both directions. Changing the wiring on a motor driver with the power on will likely damage it. If the motor is wired incorrectly, or if there is a poor connection, the motor will just vibrate. One very frustrating afternoon was spent realizing that one of the wires on a

stepper motor connector was broken, so keep the integrity of your connections in mind while troubleshooting.

Now with your motor spinning, you may find you need to adjust the current. If the motor or the motor driver is “growling” or running too hot to the touch, you can gently turn down  $V_{ref}$  on the trim pot counter-clockwise to a lower level that still results in smooth movement. If the motor doesn’t seem to be moving at all, try slowly increasing  $V_{ref}$ . Unfortunately, since these drivers are so prone to damage if miswired, trying a new motor driver might also yield better results.

If you notice your motor is spinning in the opposite direction you were anticipating but still reversing direction properly, you can either keep the wiring as is and reverse direction in your programming code, or you can reverse the stepper motor wire pairs to the motor driver (connect the A+ and A- terminals to where B+ and B- were connected, and vice-versa).

If you would like to run a Nema-17 (or other) 4-wire stepper motor without an A4988 (for instance, with another motor driver), an example sketch (*4WStepper.ino*) is provided in the appendix, which uses the Arduino IDE’s built-in stepper motor library: *Stepper.h*. This sketch for instance would appropriately control a low current 5V stepper motor salvaged from a CD-ROM drive using the L298N motor driver. Another sketch is provided that does not need an external library (*4WStepper\_noLib.ino*). In this sketch, you can see the sequence of pulses that advance the stepper motor with each step.

## Servo Motors

A servo is a geared DC motor with its own built-in microcontroller. The microcontroller performs two basic functions: it receives a control signal to control the position of its shaft, then it adjusts the shaft position and monitors it through a feedback mechanism, like an encoder wheel (also called a position sensor). By varying the duty cycle of the signal to the servo (often a 1-2 msec duty cycle over a 50 Hz frequency, or 20 msec



Figure 6-11. MG995 servo with shaft attachments.

period), the motor position will change. (Tower Pro Datasheet 2017) If a servo skips a step accidentally, or is manually twisted off course, it will compensate and return to the calibrated position as long as the signal

persists. Servos are very popular in automotive, robotic, and radio-controlled applications.

One limiting factor with a servo is the *range* of rotation. Many servos have a 180° range. Some servos have a wider range (e.g. 360°), but can only perform one rotation, and not turn continuously. A servo would be better suited as a rudder on a radio-controlled model helicopter or boat than the wheels on a model car. A servo can also be designed (or hacked) to allow for continuous rotation. However, a stepper motor is usually better suited to these applications.

The gears in a servo convert the high speed, low torque motion of a DC motor to low speed and high torque. The servo sometimes has weight as a specification (e.g. 9g for the SG-90 in Figure 6-11), since weight is a critical design parameter for most radio-controlled applications. A weight specification can also refer to a servo's working torque (e.g. the 13 kg MG946R). Protection diodes are not required for servos, as they already have built in protection for their internal electronics. Higher quality servos have metal gears which are less prone to breaking or stripping than plastic. Figure 6-11 shows the metal-gear MG995 servo, whereas the SG90 has plastic gears which can jam more easily. However, lower cost servos do not have reverse polarity protection, so take care when wiring a servo to wire it properly.

## System Control Strategies

There are many different strategies for controlling a system, or piece of equipment (also called a *plant*). We can borrow ideas from *engineering control theory* for our scientific devices. (Bellman 1964, 186-200) We will limit our discussion to single-output systems. The basic, entry-level types of control systems are *open-loop control*, *feed forward*, and *closed loop control (feedback)*.

### Open-Loop Control

The simplest form of a control strategy is *open-loop control*, which works well for predictable systems, especially for batch processes (processes that work on batches, like your dishwasher).

Chances are you've toasted bread before, so let's use control system terminology to describe how open-loop control works using your toaster to illustrate. Table 6-5 describe this process, written *without*, and *with* control system terminology.

**Table 6-5. The process of toasting bread in a toaster, described without (left) and with (right) control system terminology.**

|                                                                                                                                                                                                                                                                                                                                                                                            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ol style="list-style-type: none"> <li>1) You put toast in the toaster, then rotate the toaster timer knob to 4 minutes.</li> <li>2) The toaster knob turns on a switch, which allows 120V AC voltage to flow through the heating element.</li> <li>3) The toaster starts toasting.</li> <li>4) At 4 minutes, the knob returns back to zero, switching the heating element off.</li> </ol> | <ol style="list-style-type: none"> <li>1) User sets the <i>SETPOINT</i> on the <i>controller</i> for 4 minutes.</li> <li>2) The <i>controller</i> sends a <i>DRIVE=ON</i> signal to the <i>actuator</i>, which switches the <i>INPUT</i> stream ON, to control the <i>plant</i>.</li> <li>3) The process runs.</li> <li>4) At 4 minutes, the controller sends a <i>DRIVE=OFF</i> signal to the actuator. The actuator switches the <i>INPUT</i> stream OFF.</li> </ol> |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

If we could summarize what the controller does in this example, it would be:

FOR 4 MINUTES, DRIVE = ON. THEN, DRIVE = OFF.

We can call this our *control algorithm*.

To summarize the terminology and show a clearer picture of what’s going on, we can draw a functional block diagram of this system for our toaster example, in Figure 6-12.

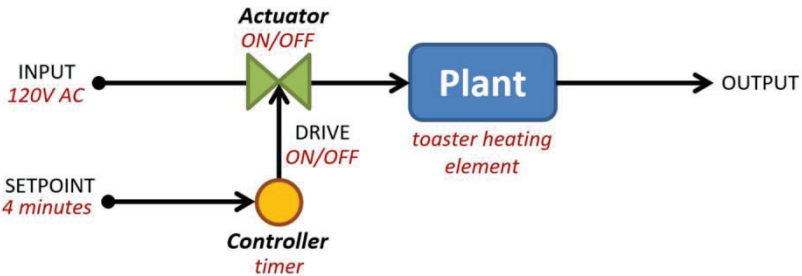


Figure 6-12. Functional block diagram of toaster control system.

How does the controller know when to start and stop? It doesn’t know at all. It’s up to the user to select a **SETPOINT**. In the design phase, the controller was calibrated by toasting bread for varying durations. Whoever designed the toaster observed for example that 7 minutes resulted in dark toast, and 1 minute resulted in light toast. Then, they printed little light and dark toast icons around the toaster knob (Figure 6-13). There is no sensor in this system, only pre-calibrated settings that are hopefully relevant to the



end user, who perhaps becomes more familiar with them after burning a few pieces of toast.

Let's use another very typical example of a control system: controlling the speed of a car. Using an open-loop control strategy, on a flat empty highway on a windless day, you could find out the relationship between how far down you press the gas pedal, and how fast the car goes. If you observe that pressing the gas pedal  $\frac{3}{4}$  of the way down results in a speed of 100 kph, then you can use the same setting next time, and "fix" the gas pedal in that position,

just like our toaster setting. Once you figure out this relationship, you don't need a sensor. You know that pushing the gas pedal  $\frac{3}{4}$  of the way down (or,  $DRIVE=75\%$ ) will get the car's speed to 100 kph on a flat surface on a windless day. Now we can estimate an open-loop gain constant,  $k_L$ , that establishes a linear relationship between the desired **SETPOINT** and the **DRIVE** signal:

$$DRIVE = k_L \times SETPOINT$$

$$75\% \times DRIVE = 100 \text{ kph}$$

$$\rightarrow k_L = 75\% \text{ DRIVE} / 100 \text{ kph} = 0.75\% \text{ DRIVE} / \text{kph}$$

$$DRIVE = (0.75\% / \text{kph}) \times SETPOINT$$

(**Note:** this is not C++ code, just the algorithm, or set of instructions)

Our open-loop gain constant might not be perfect. If it's off, we can tune it a bit, using more experiments. We have now proposed a relationship between our **SETPOINT**, and the **DRIVE** we need to get to a certain speed on a flat road. This is our control algorithm. We can devise a cruise control system with this algorithm, so that if the user wants a setpoint of 90 kph, the controller (in this case, a cruise control circuit) could send the following **DRIVE** signal to the actuator:

$$DRIVE = 0.75\% / \text{kph} \times 90 \text{ kph} = 67.5\%$$

This strategy could work well enough on a flat road. However, the control algorithm won't account for a big hill your car needs to climb. What then—what if you see a big hill the car needs to climb, coming up ahead in the distance? Our control algorithm can't adjust to the incline because it doesn't have a sensor to detect the inclined angle of the car.

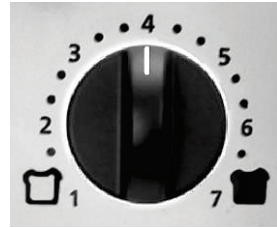


Figure 6-13. Toaster setting dial – an example of open-loop feedback control.

## Feed Forward Control

Open-loop control systems lack the ability to *measure* the process. They rely on how well the system is calibrated. If you are able to detect

disturbances on an *INPUT* stream of your plant with a sensor, you can have better control of your *OUTPUT*, because you can detect and react to a disturbance *before* it reaches your plant. If you monitor your *INPUT* stream with a *sensor*, this is called a *feed forward* control strategy.

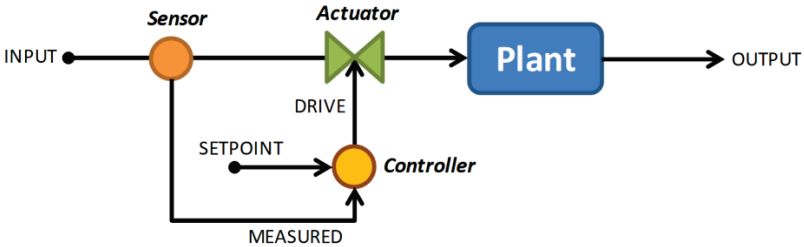


Figure 6-14. Adding a sensor to a system on an *INPUT* stream and using that information to adjust *DRIVE* is a feed-forward strategy.

In Figure 6-14, we have added a sensor to the *INPUT* stream of our plant. The user sets the *SETPOINT*. The sensor sends a *MEASURED* signal to the controller. The controller uses the *MEASURED* signal and the *SETPOINT* to make decisions about what *DRIVE* signal to send to the actuator, to control the plant. It's useful to point out that in this diagram, *SETPOINT*, *MEASURED*, and *DRIVE* are all signals. They are information, sent or received, between elements of the system.

Let's go back to our car speed example. What if we were to install a gyroscopic sensor in the car that could measure its pitch angle? We would then be considering a new input (the angle of the road). We could set up the following control algorithm:

$$\text{DRIVE} = \underbrace{(k_L \times \text{SETPOINT})}_{\text{open loop}} + \underbrace{(k_F \times \text{MEASURED})}_{\text{feed forward}}$$

The *DRIVE* signal now depends on an open loop control term, *and* a feed forward control term. Here is our functional block diagram (in Figure 6-15):

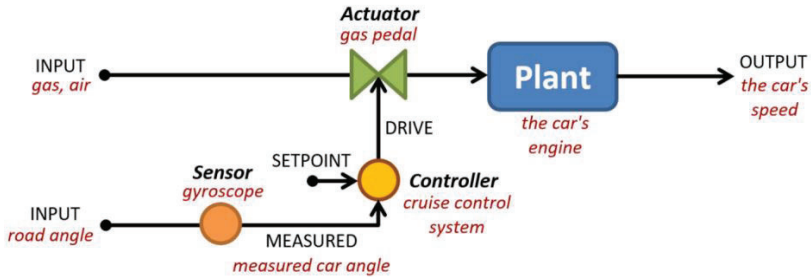


Figure 6-15. Example of a feed forward control system.

If the car is pointing uphill, the MEASURED angle of the car will be positive, and the DRIVE signal will be larger to compensate. The gas pedal will be pushed down farther, responding to the slope of the hill. If the car is pointing downhill, the MEASURED angle of the car will be negative, and the DRIVE signal will be smaller than on flat road to compensate. The gas pedal would be pushed down less, because the car would presumably be going faster downhill—it would need less power to maintain speed. We define a *feed-forward gain constant:  $kF$* , to convert the MEASURED angle to a change in DRIVE. In order to adjust the drive in the correct direction, you would want  $kF$  in this context to be positive. Solving a feed-forward model rigorously is not a straightforward task. It involves modeling the system using Laplace transforms, and explicitly solving for the resulting response curves. This approach is beyond the scope of this introductory chapter, but it is important to point out that mathematical modeling in this area is much more complicated than the simplified way it is presented here. What we *can* do is *tune* the gain constant the same way we tuned  $kL$ : with testing, until we find values that give us results that converge nicely to the correct speed.

A disadvantage of a feed-forward control strategy is that it can't possibly account for the unexpected. What if something happens to your plant that is independent of your inputs? We know that wind resistance, road surface (e.g. paved vs. dirt road), weather, wildlife, construction, and traffic should all affect speed, but it becomes very complicated to sense all of these variables and factor them into our gas pedal position in anticipation. Feed forward misses the unexpected (and the difficult to model). It also doesn't make use of the most important piece of information: the OUTPUT variable, in this case the car's current speed.

## Feedback Control

You may be familiar with the term *feedback* as something negative or corrective that your teacher or boss says about your performance. The term feedback originates in part from rocket science. Early rockets were launched into the sky, and were off course—by a lot. Considering the first rockets were actually missiles, being off course meant obliterating the wrong target. Rocket scientists realized that no matter how well they tried to aim the rockets on the launch pad (feed forward), it would be helpful to take corrective measures *while* the rocket was in flight. Despite their violent etymology, feedback systems find utility everywhere now, especially in manufacturing, scientific instrumentation, medical devices, and pharmaceutical industry.

Feedback in *control theory* means putting a sensor on the **OUTPUT** stream to measure how much the plant is off target. This information gets fed back into the controller to make adjustments. Figure 6-16 is a block diagram of a single input, single output feedback system:

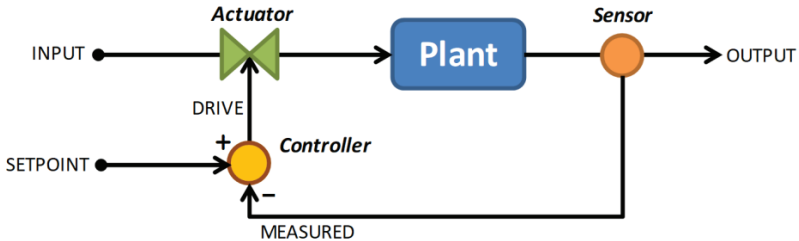


Figure 6-16. Example of a feedback control system.

In this scheme, the controller can send a DRIVE signal that takes the measured OUTPUT into account. Before we begin talking about how feedback strategies work, we need to put some practical constraints on the plant:

- 1) ***The plant should be strong enough to reach the setpoint at full power (100% DRIVE).***

A plant that is underpowered for the task will have trouble reaching the desired setpoint. In lay language, don't try to heat a stadium with a lightbulb. Your intended setpoint range will also put constraints on the plant (for instance, a -80 °C freezer will require a much stronger cooling mechanism than a 4 °C refrigerator). You can also try modifying the system so the plant

doesn't have to work as hard (e.g. by insulating the walls of a stability chamber with styrofoam).

2) *The plant should have something to work against, or be capable of exerting the opposite effect.*

If your plant *overshoots* the setpoint, the physics of the system should be able to bring the output back down. For this reason, stability chambers in pharmaceutical industry heat against cooling for more precise control.

### *On-Off Controller*

The simplest feedback strategy is one we have already used in our hot plate thermostat: the *on-off controller*. The DRIVE signal is either ON or OFF with this strategy, just like our toaster example, but the DRIVE state now depends on whether or not the MEASURED signal of the OUTPUT has reached the SETPOINT. The control algorithm for our hot plate thermostat would look like this:

```
ERROR = (SETPOINT - MEASURED)
IF ERROR > 0 THEN: DRIVE = ON
ELSE: DRIVE = OFF
```

The ERROR is how far off the MEASURED signal is from the SETPOINT. The DRIVE is now dependent on the ERROR, rather than a calibrated timer, or feed forward gain constant. You can see from the control algorithm that the DRIVE term only has two states: completely on, or completely off. For this reason, it is also called a 2-step controller, bang-bang controller, or relay-switch circuit.

One more improvement: it might be a good idea not to have your control loop be too picky. You can decide a TOLERANCE level, where the controller stops adjusting the actuator if the MEASURED signal is close enough to the SETPOINT. This avoids the system getting locked up waiting for perfection (which can happen with float variables), and can also prevent the output from overshooting. We just need to change our control algorithm to:

```
ERROR = (SETPOINT - MEASURED)
IF ERROR > TOLERANCE THEN: DRIVE = ON
ELSE: DRIVE = OFF
```

Now, the controller won't bother reacting if  $ERROR \leq TOLERANCE$ .

Temperature has a large response lag and can change slowly, so on-off controllers are often used to control temperature. Your furnace, electric stove, and fridge are likely controlled by on-off controllers.

Going back to the car's speed as an example, instead of measuring the angle of the car on a hill, our *sensor* can be the speedometer. If your car is going too slowly, you press the gas pedal all the way down. If it's going too fast, you let go of the gas pedal completely. This is a *feedback* control system. However, an on-off controller would make your passengers car sick. We need smoother control than a relay switch. What we need then is a proportional controller.

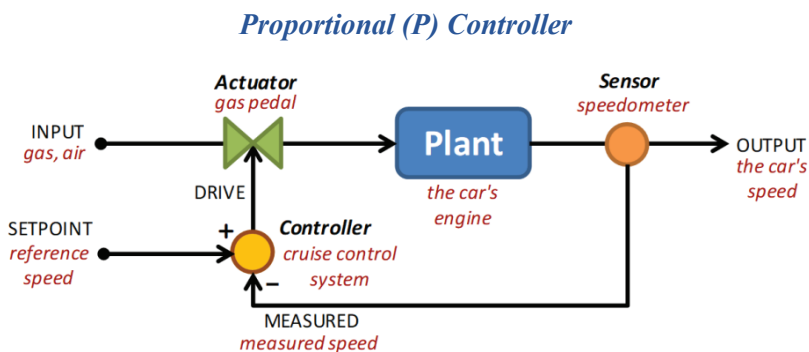


Figure 6-17. Example of a proportional feedback control system.

A car's gas pedal is capable of *graded control*. The farther down we press the gas pedal, the faster the car accelerates. We can therefore introduce a *proportional response* to our controller, rather than just an all-or-nothing *relay switch* strategy. What if the closer our measurement gets to our SETPOINT (the smaller the ERROR), the less we push on the gas pedal? This would mean that we might actually be able to converge to a constant speed, for example, a setpoint of 50 kph. We would also have full throttle where we need it—when we are far from the setpoint. Our DRIVE signal can be graded now, not just ON or OFF. DRIVE can be positive or negative, and can be dimensionless, but for now let's think of it as a percentage. DRIVE=100% means the gas pedal is floored (trying our hardest), and DRIVE=0% means completely off the gas pedal, and we have access to all values in between. We define a *proportional gain constant: k<sub>P</sub>*, to scale the magnitude of the ERROR to a DRIVE signal:

For a *proportional feedback controller*,

$$\text{DRIVE} = k_P \times (\text{SETPOINT} - \text{MEASURED}) = k_P \times \text{ERROR}$$

Proportional gain constant

What value should we give to  $k_P$ ? That entirely depends on our plant, and what we are measuring. A simplistic approach is to start with any arbitrary value for  $k_P$ , and *tune* it (try different values) until we are happy with the way the system reacts to a change in setpoint.

Let's say the car is stopped. We program  $k_P=200$  into our proportional controller and a setpoint of 50 kph, and then we observe the speed response in Figure 6-18.

### *Undamped Feedback Response*

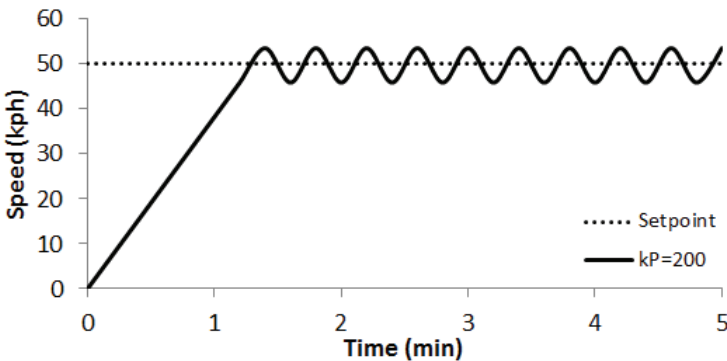


Figure 6-18. Undamped feedback response.

What's happening? The measured speed never converges to the SETPOINT. The speed oscillates between too high, and too low. This is called an *undamped response* (and a nauseating road trip). The speed never *settles*, or converges to a single value—the system is unstable. This is the same type of response as the hot-plate relay circuit in Activity 5-1.

We decide that  $k_P=200$  is way too assertive, so we try setting  $k_P=1$ . We then observe the speed response in Figure 6-19.

### Over-damped Feedback Response

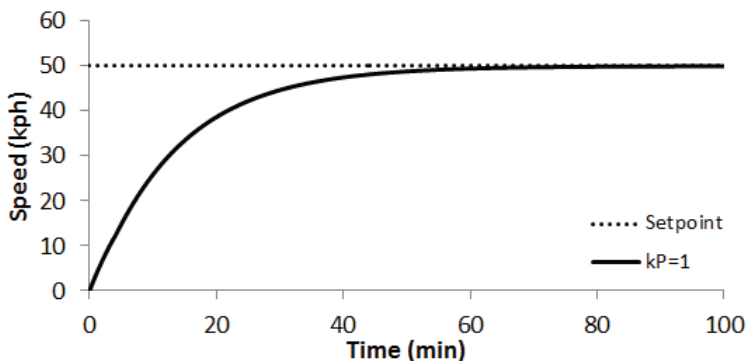


Figure 6-19. Over-damped feedback response.

Success! The speed converged to the setpoint. You might notice that it took a long time to attain the SETPOINT. In 1 hour, we would likely have reached our destination before reaching our setpoint. We call this an **over-damped response**. According to our control algorithm, every 1 kph away from the setpoint gives us 1% of DRIVE (since  $kP=1$ ). This makes our control algorithm  $DRIVE = kP \times ERROR = 1 \times 50 = 50\%$  when the car is at 0 kph initially, and as we get closer to the SETPOINT, DRIVE backs off and becomes too wimpy.

How can we fix this? Crank up the gain! We can change the value of  $kP$  to make DRIVE higher at smaller values of ERROR. What if we try  $kP=100$ ? Then, the controller will calculate  $DRIVE = kP \times ERROR$ . At the beginning when the car is at rest:

$$\begin{aligned} DRIVE &= kP \times ERROR \\ DRIVE &= kP \times (SETPOINT - MEASURED) \\ DRIVE &= 100 \times (50 - 0) \\ DRIVE &= 5000 \end{aligned}$$

The results of  $kP=100$  are illustrated in Figure 6-20.



### Under-damped Feedback Response

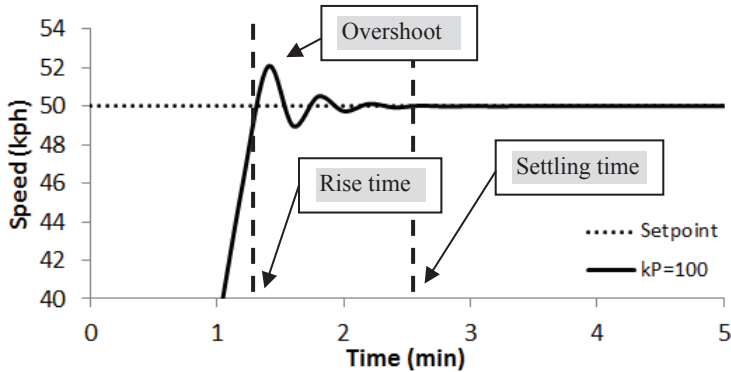


Figure 6-20. Under-damped feedback response, illustrating the concepts of rise time, overshoot, and settling time.

Since the maximum our drive signal can be is 100%, the software will constrain the response to 100%. BUT, as the speed approaches the setpoint, the drive is 100X higher, and so we can reach the setpoint *much* more quickly. We call the time it takes the system to converge to the setpoint the ***settling time***.

However, you will notice that the controller ***overshot*** the setpoint, and it took a few oscillations for the system to settle. This response is called ***under-damped***. If we increased  $k_P$  even more, then the response would eventually become ***undamped*** again, because our DRIVE would end up behaving more like the on-off controller—all or nothing.

The goal in tuning a proportional controller is to find a value for  $k_P$  that results in the fastest settling time without overshooting. This can be found experimentally—by *tuning* the value of  $k_P$  to find an optimal response:

### *Critically-damped Feedback Response*

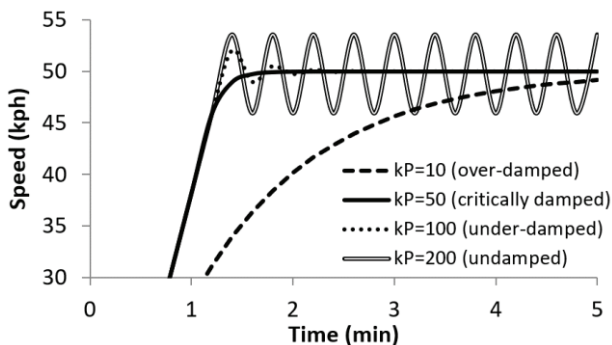


Figure 6-21. Finding a critically-damped feedback response.

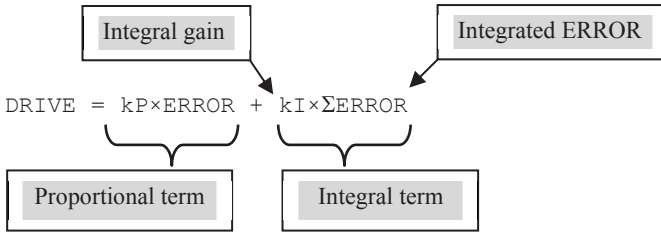
Through trial and (literally) error, we found that a  $kP$  of 50 did the best job under the circumstances. We call this particular response *critically damped*. A lower value of  $kP$  results in a longer settling time, and a higher value of  $kP$  results in under-damped behavior (oscillations).

The optimal value of  $kP$  can also change depending on what your SETPOINT is, which will be a factor when tinkering around with your system. This is especially true if the effect of the DRIVE on your plant is non-linear, which can often be the case. For instance, it is not uncommon for a DRIVE to more easily attain a lower setpoint than a higher one. Every drive and plant have limits.

The bottom line is that we can use a proportional controller (also called a *P-Controller*) for most applications, resulting in much tighter control than an on-off controller. Even if your actuator only has two states (DRIVE=0% and DRIVE=100%), you could still use a P-controller strategy by pulsing the DRIVE, more or less frequently, in proportion to how big the ERROR is. By pulsing the drive, you can attain better control than an on-off controller alone.

### *Proportional-Integral (PI) Controller*

In order to speed up settling time when critically damped proportional control isn't fast enough, or help out a system that is over-damped, we can add a term to our control strategy that *increases* the DRIVE the *longer* the measured signal is outside tolerance of the setpoint. We call this the *integral* term, because it integrates (adds up) the error the longer it is out of tolerance:



The proportional term adjusts the drive signal based on the present error term, and the integral term adds some DRIVE based on all the error signals *added up* to this point. This gives us a more assertive DRIVE signal closer to the setpoint, if the settling time is taking too long. Let's look at the same example system, with a setpoint of 50 kph, and an (over-damped)  $k_P=10$ :

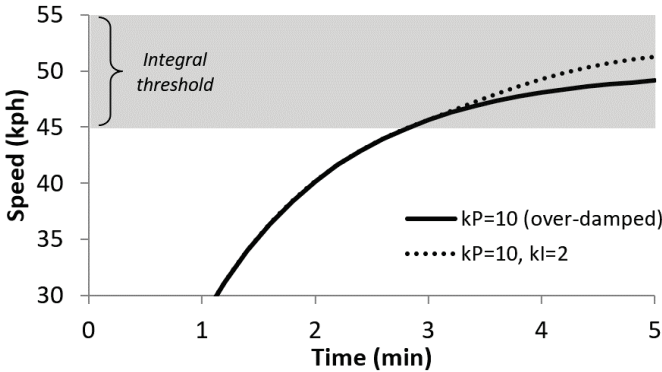


Figure 6-22. Over-damped feedback response illustrating the effects of an integral gain. This response overshoots a little, because of integral wind-up.

We need to impose some limits on the integral term, or it will spiral out of control. If we “count” this term when the measured value is far away from the setpoint, then it will really dominate the drive, and your system will become undamped. The *integral threshold* is a range where you can set the integral term to be active. When the MEASURED value is outside this range, the controller sets this term to zero to prevent it from *winding up* and becoming too large.

The following strategy works reasonably well for tuning a PI controller:

- 1) Set  $k_I=0$ , and pretend the integral term doesn't exist. Tune the proportional gain until the system is critically damped.

- 2) Define a threshold value inside which the integral term should work. This should be a wider range than your tolerance, and narrow enough to engage in the laggy region you are trying to improve.
- 3) You now need to keep track of your ERROR over time, adding it up as you go along. Start with a low value of kI and observe the response. The control algorithm can look like this:

```

ERROR = SETPOINT - MEASURED
IF ERROR < IntThresh THEN: INTEGRAL = INTEGRAL + ERROR
ELSE: INTEGRAL = 0
DRIVE = kP×ERROR + kI×INTEGRAL

```

- 4) Tune kI until you are happy with the response. Too high a kI will also result in an under-damped or undamped response. In your control algorithm, you also might want to reset the integral term after you reach the setpoint, to prevent overshoot.

### *Proportional-Integral-Derivative (PID) Controller*

Interestingly, there are philosophical ideas in feedback control. The *proportional* term reacts to the *present* error (or present deviation from setpoint). The *integral* term reacts to errors from the *past*. What's missing? The future! We can use the *derivative* of the error (or how quickly and in what direction the error is changing) to predict the *future*, and adjust the DRIVE accordingly.

Whereas the integral term helps out a system that does not attain the setpoint quickly enough, the derivative term helps out a system that reaches the setpoint *too quickly*, and overshoots it. Mathematically, the derivative term is a gain constant kD, multiplied by the derivative of the error with respect to time:

$$kD \times (dERROR/dt)$$

The gain constant kD and contribution of this term are quantitatively smaller in a tuned PID, especially when the measured value is far away from the setpoint. The derivative term is most useful in helping the system settle faster, by reducing and preventing overshoot, and dampening oscillations about the setpoint (Tim Wescott 2000, 86). If your system approaches the setpoint too quickly, then it's bound to go over it. The derivative term will compensate for this effect. The higher the rate of change of your ERROR, the more the derivative term works *against* the P+I terms. This happens on both sides of the setpoint—above and below, dampening oscillations. A colleague of mine calls this term the "bullseye factor", because it helps to reduce the settling time, and attain the setpoint more quickly. Once the system settles, the contribution of the derivative term should disappear, as

the derivative (or rate of change of ERROR with respect to time) will be zero. In a PID algorithm, a simple way of representing the derivative can be:

$$kD \times (\text{ERROR} - \text{LAST\_ERROR})$$

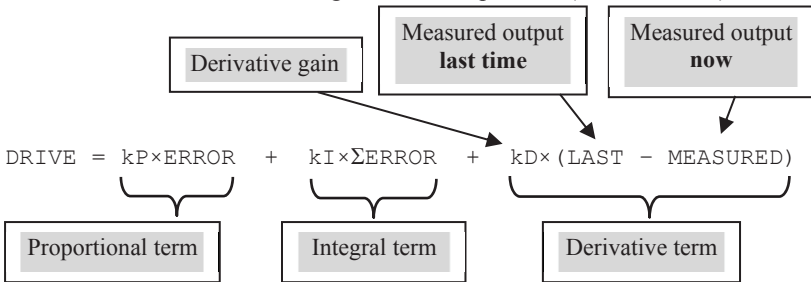
and since  $\text{ERROR} = \text{SETPOINT} - \text{MEASURED}$ , then this expression is equal to:

$$kD \times ((\text{SETPOINT} - \text{MEASURED}) - (\text{SETPOINT} - \text{LAST}))$$

$$kD \times (\text{LAST} - \text{MES})$$

Notice the difference in time ( $\Delta t$ ) is missing here, since the sampling interval for the PID algorithm is constant, and  $kD$  is dimensionless anyway.

This results in the following controller equation: (Roberts 2011)



Using a derivative to make decisions can be problematic. Derivatives are notoriously noisy, particularly if your measurements are noisy. Having a noisy derivative around the setpoint can steer the system out of control. Tuning the  $kD$  gain constant will be a balance between trying to dampen oscillations around the setpoint, and your controller producing erratic results. Smoothing or filtering the measurements could help. Techniques to smooth and filter your measurements will be discussed in Section 8. A  $kD$  value that is too large will overwhelm the P+I terms, and will ultimately amplify oscillations, resulting in an unstable (in other words, undamped) system. Derivative control is used only when needed, and when good control is not attained with P+I strategies alone.

To consider a derivative term in a PID strategy, start by following the same steps as before to tune  $kP$  and  $kI$ . After the system response is critically damped and the settling time is as fast as possible without overshooting/oscillating, if that response time is not fast enough, it's time to try adding and tuning the derivative term. Since derivatives can be noisy, keep this gain small relative to the other gains. Tune  $kP$  to speed up the overall drive response. As the system is critically damped, this will result in an underdamped system (oscillations). Start with a small value for  $kD$ . Try dampening these oscillations not by lowering  $kP$ , but by *slowly increasing*

$k_D$  to allow for a faster critically-damped response. If no value for  $k_D$  works, then try lowering  $k_P$  a bit, and repeat the process, starting with a small value for  $k_D$ . You will be working on the interplay between  $k_P$  and  $k_D$ , in order to produce the fastest settling time possible. To see the effect of the derivative term, we need to look at what's going on in the vicinity of the setpoint (Figure 6-23). The derivative term in this case turns an under-damped response into a critically damped response—not by lowering  $k_P$ , but by raising  $k_D$ . If  $k_D$  is raised too high, the system becomes undamped.

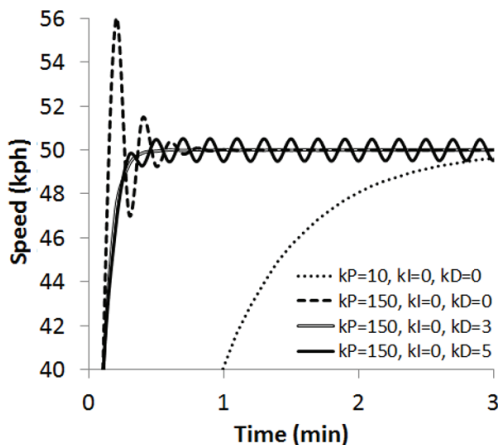


Figure 6-23. Adjusting the derivative gain to attain the setpoint.

Putting it all together, and adding an acceptable setpoint TOLERANCE, we need to do something with DRIVE once we calculate it. The algorithm in Figure 6-24 works reasonably well for a PID-controlled feedback loop. (Roberts 2011)

Programming-wise, this algorithm is fairly straightforward. You can either program this as a void function that exits when the setpoint is attained, or set it up as a function that issues the next DRIVE command and then exits, adjusting DRIVE differently the next time it is called (or not at all if MEASURED is within tolerance of the setpoint).

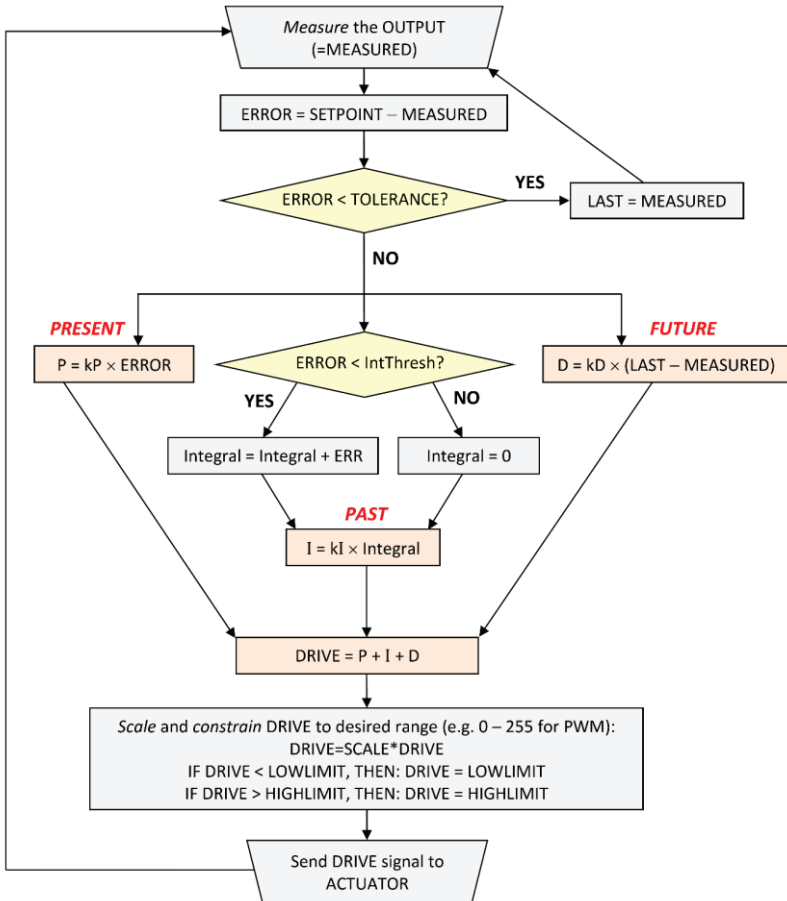


Figure 6-24. A simple PID control algorithm.

### Combining Feedback Strategies

The strategies outlined here are not mutually exclusive, they can work quite well together. Combining open-loop with feedback may yield much better control of your system. You only need to *add those terms* into the control algorithm to make use of the strategies together.

There's another matter to consider if you are planning on using a feedback strategy. If the value of DRIVE can be equal to zero when the system settles, a P-control algorithm will work well:

$$\text{DRIVE} = k_P \times \text{ERROR}$$

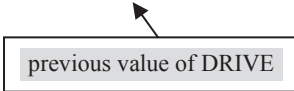
However, if our DRIVE needs to continue working to maintain a setpoint, this feedback equation calculates  $\text{DRIVE} = 0$  when  $\text{SETPOINT} = \text{MEASURED}$ , and the system will be unstable. To illustrate this problem, let's say we are controlling the speed of an airplane propeller. We would like the propeller speed to be 24,000 RPM, from a starting speed of 0 RPM. The above equation will bring it there, but once  $\text{SETPOINT} = \text{MEASURED}$ , DRIVE will be *zero*, and the controller will tell the propeller to turn off. Not good! We still need to fly. Here are two ways to handle this:

- 1) Combine an *open-loop* and/or *feed-forward* strategy with a *feedback* strategy. This will always result in a non-zero DRIVE when the system settles:

$$\text{DRIVE} = \underbrace{k_L \times \text{SETPOINT}}_{\text{open loop}} + \underbrace{k_P \times \text{ERROR}}_{\text{proportional}}$$

- 2) A quick work-around is to have the control iteratively change the previous DRIVE signal, rather than calculate an entirely new one:

$$\text{DRIVE} = \text{DRIVE} + k_P \times \text{ERROR}$$



If both methods are properly tuned, they should result in similar control. The first method will converge to  $\text{DRIVE} = k_L \times \text{SETPOINT}$  as  $\text{ERROR}$  approaches 0, and the second method will converge to a value of DRIVE that results in  $\text{ERROR} = 0$ .

Some programmable PID controllers have auto-tune features, and there are algorithms and sketches available to build this functionality into your PID controller. However, the best judge of a controller's performance is you. Does the controller perform the way you want it to? If so, then you have succeeded in your control strategy.



## Activity 6-1: L298N Motor Driver Controlling a DC Motor

**Goal:** In this exercise, you will be controlling a 5V DC motor with an L298N motor driver, powered by an ATX power supply.

### Materials:

- Arduino Nano MCU & USB cable
- Laptop with Arduino IDE installed
- 1 x ATX Power Supply
- 1 x Breadboard
- 1 x 5V DC Motor
- 1 x L298N Motor Driver
- 8 x Male/Male Jumpers
- 4 x Male/Female Jumpers
- 1 x Microspatula or Small Screwdriver

### Procedure:

Build the following circuit.

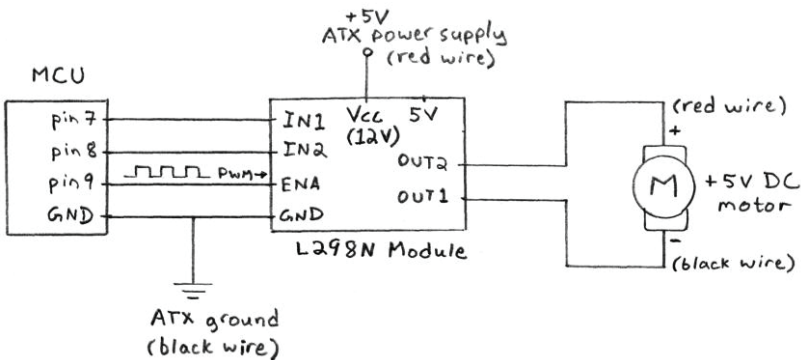


Figure 6-25. Circuit diagram for Activity 6-1.

**Note:** Make sure ENA and ENB jumpers are *open* (see Figure 6-5).

- 1) Write a void function, with an integer as an input argument (expected range: -255 to 255), that sets the DC motor speed and direction accordingly:
  - -255 to 0: counter-clockwise rotation
  - 0 to 255: clockwise rotation
  - Have the ENA, IN1, and IN2 pin numbers as input arguments to your function, so that you could use the same function to control a different motor concurrently.

- 2) Write a sketch that prompts the user for a number on the serial monitor, and uses your function to set the motor speed.

**Programming Hint:**

If speed < 0, set IN1=HIGH, IN2=LOW, then analogWrite the (-speed) to ENA.

If speed > 0, set IN1=LOW, IN2=HIGH, then analogWrite the speed to ENA.

## Activity 6-2(a): 28BYJ-48 Stepper Motor

**Goal:** In this activity, we will be wiring up a 28BYJ-48 stepper motor to a ULN2003 motor driver. For this exercise, we will control motor position with a 10K potentiometer.

**Materials:**

- Arduino Nano MCU & USB cable
- Laptop with Arduino IDE installed
- 1 x ATX Power Supply
- 1 x Breadboard
- 1 x 28BYJ-48 Stepper Motor
- 1 x ULN2003 Motor Driver
- 1 x 10K Potentiometer
- 8 x Male/Male Jumpers

**Procedure:**

- 1) Assemble the 28BYJ-48 circuit following Figure 6-26 below.

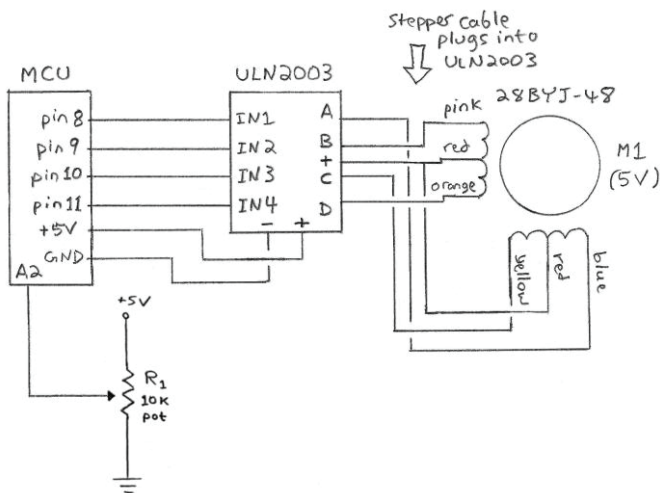


Figure 6-26. Circuit diagram for Activity 6-2(a): 28BYJ-48 stepper motor with ULN2003.

- 2) Download the sketch *28BYJ-48.ino* from the course website. Replace the loop function of the 28BYJ-48 sketch with the following code:

```
void loop(){
 static int previous; //declare previous as static int
 int val=analogRead(A2); //read pot on pin A2 (0-1023)
 motorStep(val-previous,10); //move displaced #steps
 Serial.println(val);
 previous=val; //remember last reading
}
```

**Note:** You can put any integer in the `motorStep()` command. If you exceed the `#steps/revolution` (+ or -), the motor will just keep turning the requested number of steps. We chose a potentiometer to control this motor, but we could have easily left the potentiometer out, and specified a series of movements by calling `motorStep()` in the program.

- 3) Now compile and upload the sketch. Start the serial monitor.
- 4) Try the following modifications:
- Changing the speed (fastest stable speed = ?).
  - Changing the way you call the `motorStep()` function.
  - What does declaring the integer `previous` as “static” do in the loop function? What would happen if it weren’t declared as a static variable type?

### Activity 6-2(b): Nema-17 Stepper Motor

**Goal:** In this activity, we will be wiring up a Nema-17 stepper motor to an A4988 motor driver. For this exercise, we will control motor position with a 10K potentiometer.

**Materials:**

- |                                                                                                                                                                                                                                  |                                                                                                                                                                                                                                                           |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>• Arduino Nano MCU &amp; USB cable</li> <li>• Laptop with Arduino IDE installed</li> <li>• 1 x ATX Power Supply</li> <li>• 1 x Breadboard</li> <li>• 1 x Nema-17 Stepper Motor</li> </ul> | <ul style="list-style-type: none"> <li>• 1 x A4988 Motor Driver</li> <li>• 1 x 100 <math>\mu</math>F Electrolytic Capacitor</li> <li>• 1 x 10K Potentiometer</li> <li>• 8 x Male/Male Jumpers</li> <li>• 1 x Microspatula or Small Screwdriver</li> </ul> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

**Procedure:**

- 1) Assemble the Nema-17 circuit following Figure 6-27 below. Do not power on the ATX power supply until after completing all the connections, or the motor driver may get damaged.

**Note:** Be careful to connect the electrolytic capacitor respecting the correct polarity (longer leg positive). If the capacitor is wired backwards in this circuit, it may explode.

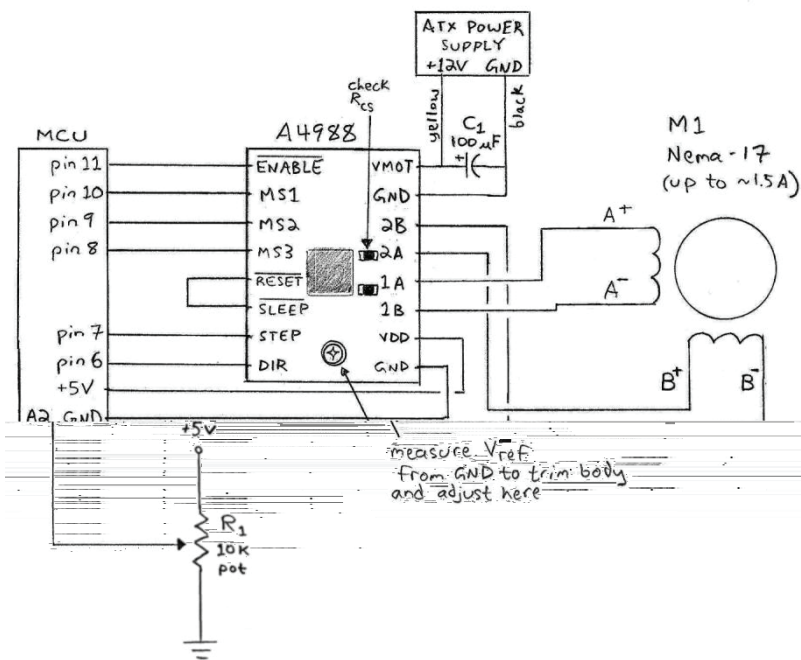


Figure 6-27. Circuit diagram for Activity 6-2(b): Nema-17 stepper motor with A4988.

- 2) Adjust the trim on the A4988 to an appropriate voltage for the current requirements of the motor (see Table 6-4, use  $R_{cs}=100\text{ m}\Omega$ ).
- 3) Download the sketch *A4988.ino* from the course website. Replace the loop function of the Nema-17 sketch with the following code:

```
void loop() {
 static int previous; //declare previous as static int
 int val=analogRead(A2); //read pot on pin A2 (0-1023)
 val=map(val,0,1023,0,200); //rescale to 1 revolution
```

```

motorStep(val-previous,10); //move displaced #steps
Serial.println(val);
previous=val; // remember last reading
}

```

**Note:** You can put any integer in the `motorStep()` command. If you exceed the `#steps/revolution` (+ or -), the motor will just keep turning the requested number of steps. We chose potentiometer control in this strategy, but we could have easily left the potentiometer out, and specified a series of movements by calling `motorStep()` in the program.

- 4) Now compile and upload the sketch. Start the serial monitor.
- 5) Adjust  $V_{ref}$  down by turning the trim potentiometer counter-clockwise if the motor is running very loud or hot.
- 6) Try the following modifications:
  - Changing the speed (fastest stable speed = ?).
  - Changing the way you call the `motorStep()` function.
  - Changing the `setMicroStep(0)`; input argument to 1, 2, 3, or 4 to try stepping the motor using to microsteps.
  - What does declaring the integer previous as “static” do in the loop function? What would happen if it weren’t declared as a static variable type?

### Activity 6-3: SG90 Servo Control

**Goal:** In this activity, we will be wiring a servo directly to our microcontroller. We will control the servo using a 10K potentiometer.

#### Materials:

- Arduino Nano MCU & USB cable
- Laptop with Arduino IDE installed
- 1 x ATX Power Supply
- 1 x Breadboard
- 1 x SG90 Servo (9g)
- 1 x 10K Potentiometer
- 8 x Male/Male Jumpers
- 1 x Burette, Burette Clamp, and Retort Stand
- 1 x Vinyl Retort Clamp (to secure the servo)

#### Procedure:

Assemble the following circuit:

Figure 6-28. Circuit diagram for Activity 6-3.

**Note:** The SG90 runs on low-enough current to be powered directly from Arduino's +5V DC power pin. A higher-power servo could be powered using an external power supply (e.g. red wire of ATX supply) instead of the Nano's 5V pin. The external power supply would need to share ground with the microcontroller.

1) Write the following sketch:

```
// control servo with potentiometer
#include <Servo.h>
Servo myServo; // create servo object called myServo
byte potPin=A2; // pin for wiper of 10k potentiometer
byte servoPin=9; // PWM signal for servo
int val=0; // to store voltage from pot wiper
byte stepsPerRev=179; //0-179 gives 180 steps (1/2 rev)

void setup(){
 Serial.begin(9600); // start the serial monitor
 myServo.attach(servoPin);
}

void loop(){
 val=analogRead(potPin); // read pot (0-1023)
 Serial.println(val); // print val
 val=map(val,0,1023,0,stepsPerRev); // rescale
 myServo.write(val); // send setpoint to servo
 delay(15);
}
```

Compile and upload this sketch, then test the servo.

**Make it meaningful:** Try attaching your servo to the spigot of a burette with some rubber bands, to create your own auto-titrator. Be careful with water and electricity. Keep your electronic components dry.

### Activity 6-4: PID Control of a 12V CPU Fan

**Goal:** The following circuit builds upon the circuit in Activity 5-2 (transistor as a switch for a DC motor). We are adding a temperature sensor, the LM35 (an IC). Normally with temperature, an on-off controller would suffice. However, this circuit will illustrate how the DRIVE signal responds to deviations from setpoint.

#### Materials:

- Arduino Nano MCU & USB cable
- Laptop with Arduino IDE installed
- 1 x ATX Power Supply
- 1 x Breadboard
- 1 x 12V DC Fan
- 1 x LM35 Temperature Sensor
- 1 x 10K Resistor
- 1 x 2N2222 Transistor
- 1 x 1N4007 Diode
- 10 x Male/Male Jumpers

#### Procedure:

Build the following circuits:

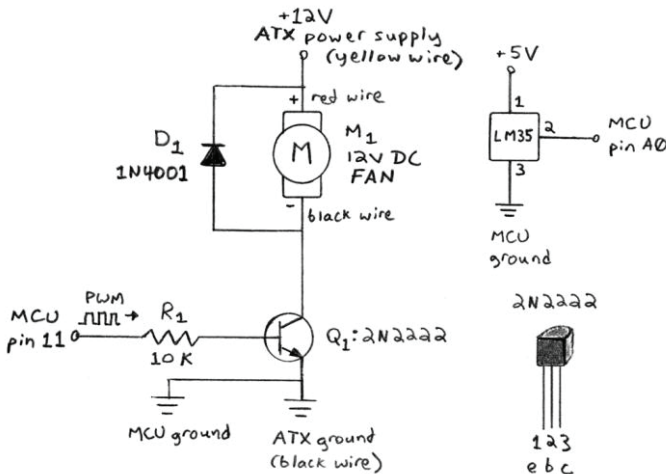


Figure 6-29. Circuit diagram for Activity 6-4.

**Notes:**

- Connect ATX ground (black wire) to Arduino ground.
- The black fan wire does *not* go to ground.

In this circuit, we are controlling the temperature of the circuit board. Our temperature sensor is an LM35 integrated circuit. Our actuator is the CPU fan, and our DRIVE signal will be controlling fan speed, to keep our plant (the breadboard) from getting too hot. We are using the Arduino Nano as a PID controller, with a SETPOINT of 21 °C.

The goal of this exercise is to tune the PID controller for the fastest settling time, without resulting in oscillations. We will be using the serial monitor to see the response. Once you assemble the circuit, download the sketch *PID.ino* from the course website (or copy it from the appendix), and read through the code.

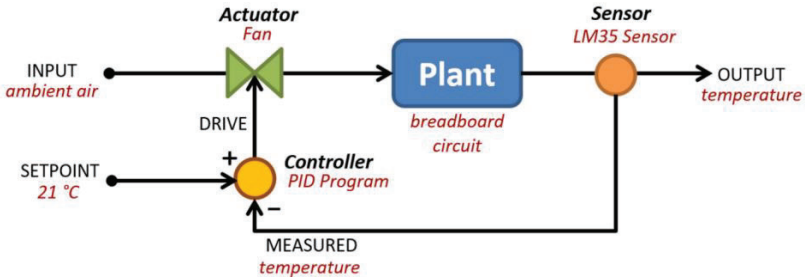


Figure 6-30. Functional block diagram for Activity 6-4.

Read over the section on how to tune a PID controller, and then adjust  $k_P$ ,  $k_I$ , and  $k_D$  to see how these terms contribute to setting DRIVE to control the temperature of a circuit-board temperature sensor. What values of  $k_P$ ,  $k_I$ , and  $k_D$  result in the best fan response, and temperature control?

## Learning Objectives for Section 6

After having attended this class, the student will be able to:

- 1) Identify the basic parts of a DC motor.
- 2) Design an appropriate control circuit for a DC motor, stepper motor, servo motor, and PC fan.
- 3) Write an appropriate sketch to control the above circuits.
- 4) Compare and contrast the strengths and weaknesses of different motor types.



- 5) Identify and categorize feedback responses to a change in setpoint or system disturbance (undamped, over-damped, under-damped, critically damped).
- 6) Draw a basic process control functional block diagram, identifying the actuator, plant, sensor, and controller.
- 7) Program and tune a PID control algorithm with satisfactory results.
- 8) Describe what the three terms in a PID controller mean, and how they are calculated.
- 9) Describe your own practical example illustrating feedforward and feedback control, using engineering control theory terminology.

## Section 6 - Station Content List, Activities 6-1 & 6-3

- Arduino Uno or Nano MCU & USB cable
- 1 x ATX Power Supply
- 1 x Breadboard (170 tie points)
- 1 x 5V DC Motor  
(optional: double-sided tape, filter paper to make a spinner)
- 1 x L298N Motor Driver
- 1 x SG90 Servo (9g)
- 1 x 10K Potentiometer
- 10 x Male/Male Jumpers
- 4 x Male/Female Jumpers
- 1 x Burette, Burette Clamp, and Retort Stand (optional- for titrator)
- 1 x Vinyl Retort Clamp (to secure the servo)
- 1 x Microspatula or Small Screwdriver

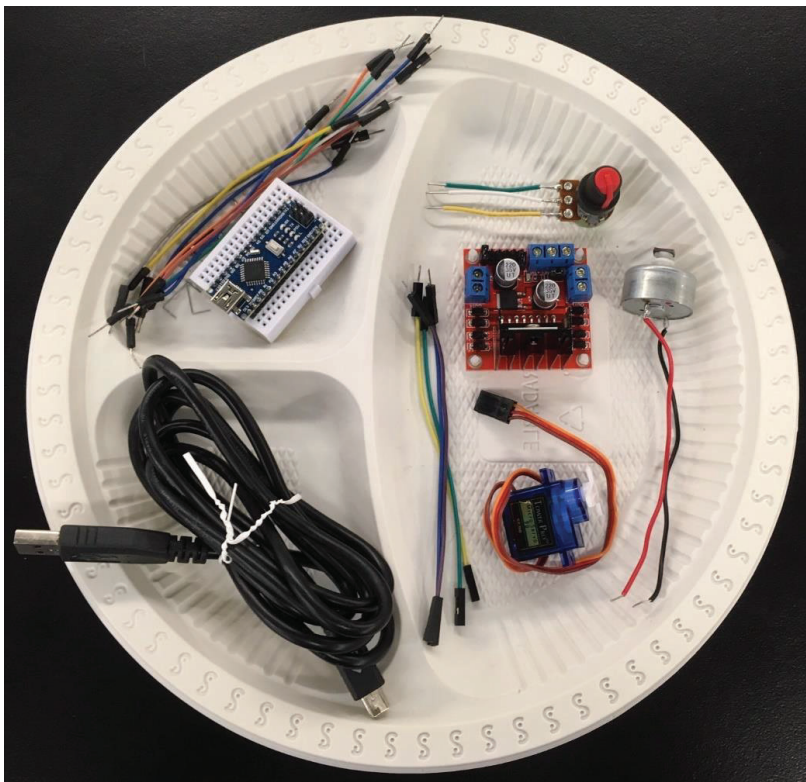


Figure 6-31. Station setup for Activities 6-1 and 6-3. Not shown: ATX power supply.

## Section 6 - Station Content List, Activities 6-2(a,b), 6-4

- Arduino Nano MCU & USB cable
- 1 x Digital Multimeter
- 1 x ATX Power Supply
- 1 x Breadboard (400 tie points)
- 1 x 28BYJ-48 Stepper Motor
- 1 x ULN2003 Motor Driver
- 1 x Nema-17 Stepper Motor
- 1 x A4988 Motor Driver
- 1 x 10K Potentiometer
- 1 x 10K Resistor
- 1 x 100  $\mu$ F Electrolytic Capacitor
- 10 x Male/Male Jumpers
- 6 x Male/Female Jumpers
- 1 x Microspatula or Small Screwdriver
- 1 x 12V DC Fan
- 1 x 2N2222 Transistor
- 1 x LM35 (Temperature sensor)
- 1 x 1N4007 Diode

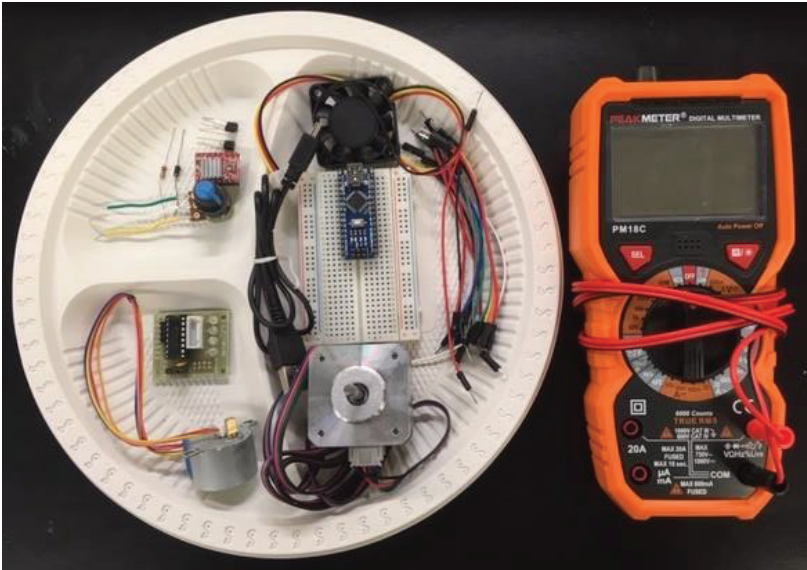


Figure 6-32. Station setup for Activities 6-2 and 6-4. Not shown: ATX power supply.

# SECTION 7

## OPERATIONAL AMPLIFIERS

|                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                                                                                                    |
|-------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| What You'll Be Learning | <b>Lecture:</b> Op-amp wiring: inverting vs. non-inverting inputs, output, power rails. Open loop: comparator. Closed loop: buffer, inverting, inverting+bias, non-inverting, non-inverting+bias, differential, summing inverting, summing non-inverting. Negative voltage. Signal attenuation. Wheatstone bridge.                                                                                                                                                                                                                                                                                          |                                                                                                    |
| What You'll Be Doing    | <b>Choose one of the following activities:</b><br><b>Activity 7-1:</b> Load cell. Amplify the signal from a load cell using an op-amp. Calibrate the load cell, and write a sketch to output measured weight to the serial monitor, with tare function. Store your device for Section 8.<br><b>Activity 7-2:</b> pH meter. Shift and gain a pH electrode from (-0.5 to +0.5V) to (0 to +3.3V) using op-amps. Calibrate a pH probe, and write a sketch to output measured pH to the serial monitor. Store your device for Section 8.<br>As this is a lengthier build, there will be no demos for this class. |                                                                                                    |
| Files you will need     | All course files are available for download at:<br><a href="http://pb860.pbworks.com">http://pb860.pbworks.com</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | <ul style="list-style-type: none"><li>• <i>scaleCal.xlsx</i></li><li>• <i>pHCal.xlsx</i></li></ul> |

### Introduction

Before digital computing was around, electronic circuits were analog. Transistors and **operational amplifiers** did much of the heavy lifting in logic circuits. An operational amplifier (or “op-amp” for short) gets its name from its originally intended purpose. Op-amps were designed to perform unit operations on one or more **signals**: addition, subtraction, multiplication, and division—of voltages. They are still used for this purpose today, although they find utility in many more applications. A **signal** can be a fixed voltage, a measurement from a probe (e.g. pH probe or thermocouple), high frequencies coming out of a guitar pickup, and other electrical signals modulate voltage. **Small building note:** If you connect your signal directly to  $V_{cc}$  or ground, it will short your signal—and it will be gone (like the off switch of a microphone).

The circuit diagram symbol for an op-amp is illustrated in Figure 7-1.

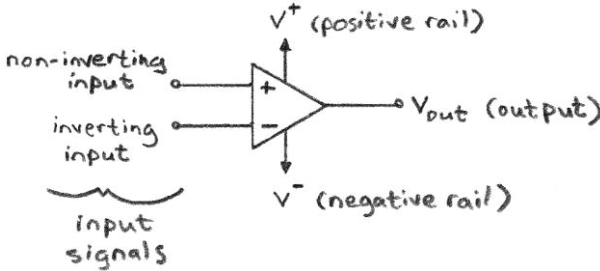


Figure 7-1. Circuit diagram symbol for an operational amplifier.

An *ideal op-amp* has the following characteristics:

- 1) No current flows into or out of the inputs;
- 2) The op-amp tries to make both inputs the same, by adjusting the output;
- 3) The *output* cannot be greater than  $V^+$ , or less than  $V^-$ , the voltages provided to the op-amp's power rails.

### Open Loop Configuration (Comparator)

In open loop configuration, the op-amp acts as a *comparator*. If the *inverting input* is lower (even 1 mV lower) than the *non-inverting input*, the op-amp will output a voltage as high as it can ( $\sim V^+$ ). Conversely, if the inverting input is higher than the non-inverting input, the op-amp will output a voltage as low as it can ( $\sim V^-$ ). The feedback response in open loop is analogous to an on-off controller, if you think of  $V_{out}$  as DRIVE. It's an all-or-nothing response. This op-amp configuration is illustrated in Figure 7-2.

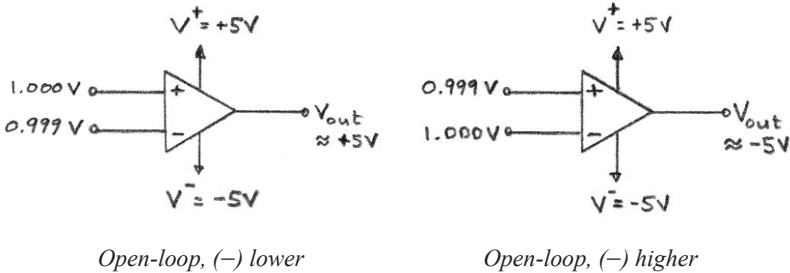


Figure 7-2. Open-loop configuration of an operational amplifier.

## Closed Loop Configuration

In open loop configuration, the op-amp “slams” the output to extremes, because it is trying to make voltages at its inputs equal, but it can’t “see” the output voltage. In closed-loop configuration, the output voltage is “fed” back into the inverting input, so  $V_{out}$  can be more sophisticated than simple on-off control. There are many closed-loop op-amp configurations. The simplest is a voltage buffer.

### Buffer

The most direct closed-loop configuration provides a feedback signal by directly connecting  $V_{out}$  to the inverting input (-). A **signal** ( $V_{in}$ ) is connected to the non-inverting input (+). This is called a **buffer**, or **voltage follower**.

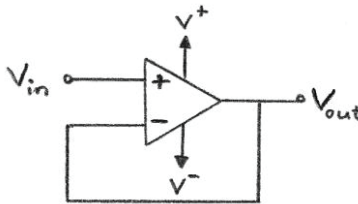


Figure 7-3. Buffer (or voltage follower) configuration of an op-amp.

Since an ideal op-amp tries to make the inverting and non-inverting terminals the same, if  $V_{out}$  is forced to equal  $V_{(-)}$ , the only way to satisfy this constraint is to have  $V_{out} = V_{in}$ . The output voltage of this op-amp will therefore be equal to the input voltage. The relationship between output and input voltages are expressed in terms of **gain**:

$$Gain = A_v = \frac{V_{out}}{V_{in}}$$

The gain of a buffer is 1 ( $A_v = (V_{out}/V_{in}) = 1$ ). If you set a sine curve as  $V_{in}$ , and looked at  $V_{in}$  vs.  $V_{out}$  on an oscilloscope, you might see the following waveforms:

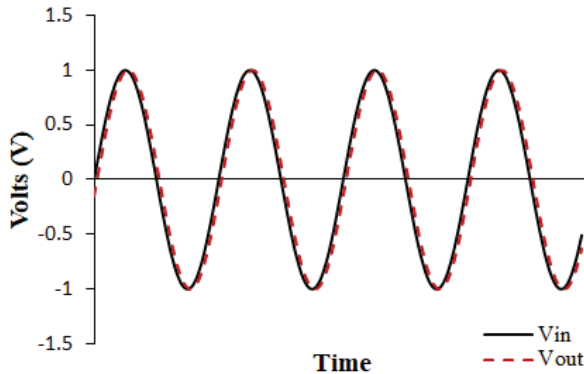


Figure 7-4. The voltage output of an ideal buffer follows the voltage input ( $V_{out}=V_{in}$ ).

### *Why would you want/need a voltage follower?*

At first glance, a voltage follower looks useless. After all, it looks like a wire can do the same thing, with respect to voltage (and it can!). However, some signals have the voltage you want, but the current is way too low. We call this a **high impedance** signal. For the purposes of DC circuits, you can think of impedance as being the same as resistance (although we use the symbol  $Z$  for impedance, not  $R$ ). Conversely, a signal with high current has low impedance.

*Buffering* a signal increases the signal's current. A buffer converts a **high impedance signal** to a **low impedance signal**, without disturbing the original circuit. We said before that an ideal op-amp has no current flowing into or out of the inputs. Real op-amps have very tiny amounts of current flowing into the inputs. For instance, the input resistance of the TL07x series amplifiers is  $10^{12} \Omega$ , meaning a 5V signal would have an input current of  $I=V/R = 5V/10^{12}\Omega = 5 \times 10^{-12}A$ . So now you can think of op-amp inputs as being very high impedance—the current is *impeded* (hindered) from going in. Once a signal is buffered, you can use it to drive a load (like a speaker). Op-amps are therefore very popular in musical instrument amplifiers. The signal coming from a guitar pickup has a high impedance, and must be buffered and amplified in order to drive a speaker and generate sound. Or, you might buffer a signal so that you can read the voltage via the Arduino Uno analog pins, or an **ADC** (analog to digital converter) module.

Another use for a buffer is to electrically isolate one part of a circuit from another. By buffering a signal, you are preventing anything

downstream of the buffer from affecting that signal. This idea becomes important later (in Section 8) when we will talk about higher order filters.

## Op-Amp Characteristics

### *Output Short-Circuit Current*

The *output short-circuit current* of an op-amp ( $I_{sc}$ ) is the maximum theoretical output current you can get if you buffer a signal. This property is important to know if you require a certain amount of current to drive a load. Op-amps are designed to have different short-circuit currents. Some op-amps are designed for specifically delivering power. For example, the OPA541 power op-amp can deliver up to 10A. (Texas Instruments Inc 2016c) Some amps are specifically designed for audio. The TL07x op-amp series is designed for audio signals, and has  $I_{sc}$  values around 40 mA. Other op-amps are designed for amplifying signals for scientific instruments (e.g. the AD623 low-cost instrumentation amplifier). The selection of an appropriate op-amp should take into consideration the output short-circuit current required for the amplified signal.

### *Gain in dB (decibels)*

Since gain numbers can get quite large, gain is typically reported in decibels, using the following formula:

$$Gain(dB) = 20 \times \log\left(\frac{V_{out}}{V_{in}}\right)$$

A buffer by definition has a gain of  $20 \times \log(1)=0$  dB. Op-amps will list the maximum DC voltage gain on their datasheets. For instance, the LM358 has a maximum gain of 100 dB, meaning:

$$\frac{V_{out}}{V_{in}} = 10^{\left(\frac{100dB}{20}\right)} = 100,000$$

### *Headroom*

Most op-amps can't quite swing their output voltages all the way to the top and bottom rails—there is a bit of a shortfall, or gap. This is called *headroom*. An op-amp's headroom can be complicated. It can depend on a whole list of factors, like the voltages applied to the top and bottom rails, the frequency of the signal going in, the load applied, and temperature. This information is found on the op-amp datasheet. For example, the TL07x



family of op-amps have a *maximum peak output voltage* vs. signal frequency response depicted in Figure 7-5.

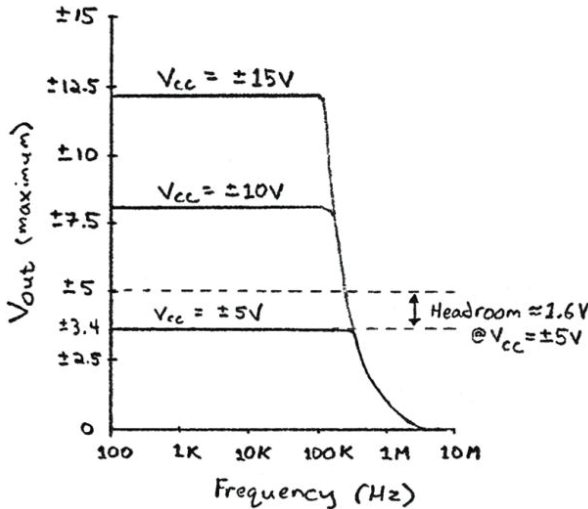


Figure 7-5. Maximum peak output voltage vs. frequency for the TL07x op-amp series ( $R_L=2k\Omega$ ,  $T=25^\circ\text{C}$ ). (Texas Instruments Inc 2017)

If you power the top rail of a TL07x series chip with +5V and the bottom rail with -5V, for a lower frequency signal (flat part of the curve), the highest voltage the TL07x chip can output will be about 3.4 V. The working range for this amp at  $V_{cc}=\pm 5$  V is -3.4V to 3.4V, and the headroom in this case is 1.6 V. If we were to use a TL07x chip as a *comparator*, we wouldn't quite get to 5V in this circuit because of this headroom (Figure 7-6, left), but we *could* if we increased the voltage to the op-amp rails (Figure 7-6, right).

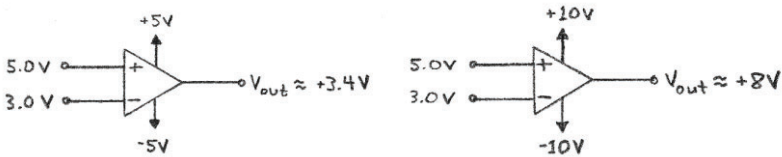


Figure 7-6. The op-amp on the right can swing higher because larger voltages are provided to the power rails.

The effect of headroom can be surprising if you aren't expecting it. Headroom can affect both rails of the amp, and is dependent on the design of the operational amplifier. An illustration of headroom is provided in Figure 7-7.

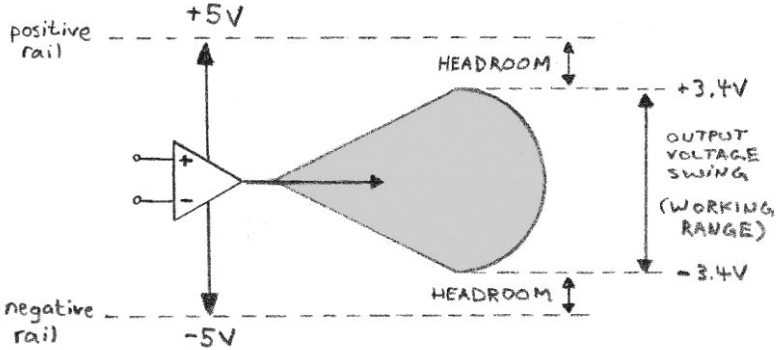


Figure 7-7. Op-amp headroom means that the output of the op-amp can't swing all the way to the power rails. This is an illustration of the TL07x series, when the op-amp is supplied with  $\pm 5\text{V}$ .

However, if you require a signal amplified over the headroom (or all the way to ground), you can always increase the voltages applied to the power rails. In particular, you need to be cautious when tying the bottom rail to ground (Figure 7-8).

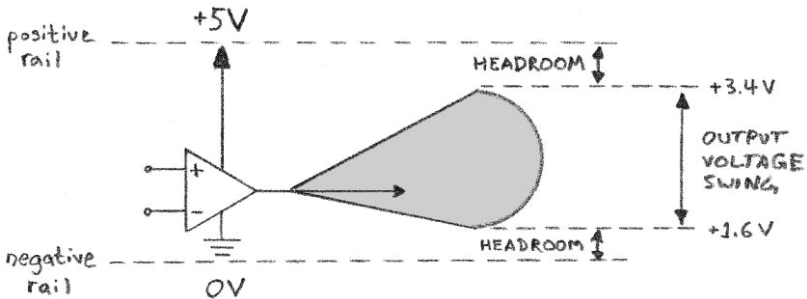


Figure 7-8. The output of a typical op-amp will not be able to swing all the way to ground, if the negative rail is connected to ground.

Particularly with scientific equipment, we are often interested in signals close to 0V—this can be related to the *sensitivity* of the instrument. In this case, a bias voltage can be added to raise the signal before amplifying, or, a negative voltage applied to the bottom rail instead. Alternately, you can purchase a rail-to-rail op-amp with essentially no headroom (Figure 7-9).

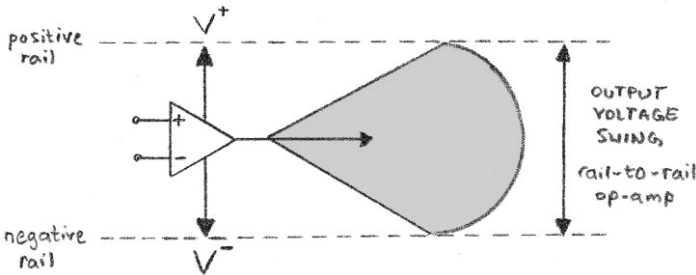


Figure 7-9. A rail-to-rail op-amp is able to swing its output within microvolts of the power rails (virtually no headroom).

There are no hard and fast rules to headroom. Different op-amps will have their own unique behaviours when they are pushed close to their supply rail limits, depending on their design and purpose. The Texas Instruments TL07x-series op-amps were designed with audio signal amplification in mind, so swinging right to the bottom rail was not as important as a wide voltage tolerance (up to 36V). (Texas Instruments Inc 2017) Some op-amps are designed to swing all the way to ground on a single supply (meaning  $V^+$  is connected to a positive voltage, and  $V^-$  is connected to ground). For instance, the dual op-amp LM358 and quad op-amp LM324 can swing all the way down to ground, and up to 3.5V when the top rail is connected to +5V and the bottom rail to ground. (Texas Instruments Inc 2015a; Texas Instruments Inc 2014a) The AD623 is a single or dual-supply low-cost instrumentation amplifier, boasting rail-to-rail output swing. (Analog Devices Inc 2016) Depending on the application, selecting the right op-amp will have a very large impact on the output signal.

### *Slew Rate*

The *slew rate* of an op-amp is how quickly it can react to a step change (an instantaneous change in voltage) at the input. A faster slew rate means that the op-amp can adjust  $V_{out}$  more quickly in response to a change in  $V_{in}$ . This is usually measured in volts per microsecond. The TL07x-series op-amps have a typical slew rate of 13 V/ $\mu$ sec, compared with the more pokey

response time ( $\sim 0.3\text{V}/\mu\text{sec}$ ) of the LM358 and LM324. (Texas Instruments Inc 2017; Texas Instruments Inc 2014a; Texas Instruments Inc 2015a) This makes the TL07x-series chips more capable of buffering and amplifying audio signals.

### Unity Gain Bandwidth

As you might imagine, the slew rate of an op-amp will limit its abilities to respond to higher frequency signals. The **unity gain bandwidth** ( $B_1$ ) of an op-amp is the highest frequency that an op-amp can amplify a signal with a gain of at least one. (Mancini 2002) A similar type of parameter is the **gain-bandwidth product** (GBW). If you are planning on amplifying a signal that has a frequency greater than the listed unity gain bandwidth or gain-bandwidth product of an op-amp, it's time to consider a faster op-amp. To be safe, take the maximum frequency signal you would like to amplify, and multiply it by 5 to get a rough estimate of the unity gain bandwidth or gain-bandwidth product required for the op-amp. The actual relationship is more complicated, but this safety margin will get you started. (Analog Devices Inc. 2009)

To provide an example, the TL07x-series op-amps have a unity gain bandwidth of 3 MHz. This means that the gain of the op-amp will be less than 1 when the frequency of the input signal is above 3 MHz. If you would like to amplify a signal, make sure it has a frequency less than  $3\text{MHz} / 5 = 600\text{ kHz}$ . The LM358 and LM324 have a listed unity gain bandwidth of 1 MHz. (Texas Instruments Inc 2017; Texas Instruments Inc 2014a; Texas Instruments Inc 2015a)

### Inverting Amplifier

The most basic configuration for using an op-amp to amplify a signal (i.e. gain  $> 0\text{ dB}$ ) is the **inverting amplifier**. The **signal** ( $V_{in}$ ) goes into the **inverting input** ( $-$ ) (Figure 7-10).

What's special about the wiring of the inverting amplifier is that the **non-inverting input** ( $+$ ) is wired to ground. The gain for an

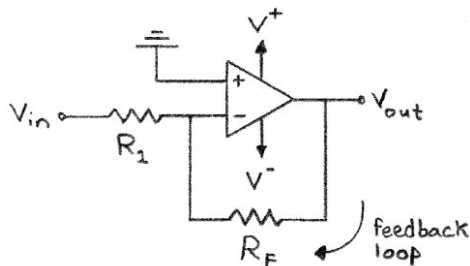


Figure 7-10. Inverting amplifier configuration of an op-amp.

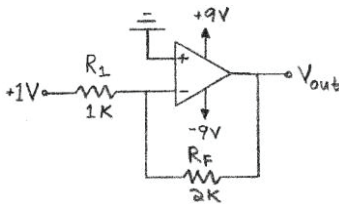
inverting amplifier is consequently negative, because the op-amp tries to make both inputs equal:

$$A_v = \frac{V_{out}}{V_{in}} = -\frac{R_F}{R_1}$$

The amplifier is called “inverting” because a positive voltage  $V_{in}$  will result in a negative  $V_{out}$ , and conversely, a negative voltage  $V_{in}$  will result in a positive  $V_{out}$ . The sign of the output is *inverted* with respect to the input. If  $R_F > R_1$ , then the signal will also be amplified.

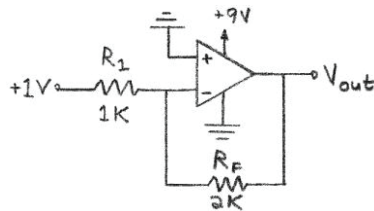
Always keep in mind that the output of any op-amp will depend on the power its rails have access to. If you connect the bottom rail to ground (a common thing to do), then the output won’t be able to swing lower than ground.

**Test your understanding** with the two scenarios in Figure 7-11. The inverting op-amp on the right is wired incorrectly:



$$A_v = \frac{V_{out}}{V_{in}} = -\frac{R_F}{R_1} = -\frac{2K}{1K} = -2$$

$$V_{out} = A_v V_{in} = -2 \times 1V = -2V$$



$$A_v = \frac{V_{out}}{V_{in}} = -\frac{R_F}{R_1} = -\frac{2K}{1K} = -2$$

$$V_{out} = A_v V_{in} = -2 \times 1V = -2V$$

However, since the op-amp can’t swing its output lower than bottom rail, then  $V_{out} = 0V$  in this configuration (ideal op-amp, no headroom).

Figure 7-11. An op-amp’s  $V_{out}$  is constrained by the voltage connected to its power rails.

For the correctly-wired op-amp (Figure 7-11, left), the gain is -2. If you set the  $V_{in}$  signal to be a sine curve swinging from -1 to +1V, and observed the  $V_{in}$  and  $V_{out}$  waveforms on an oscilloscope, you might see the waveforms in Figure 7-12.

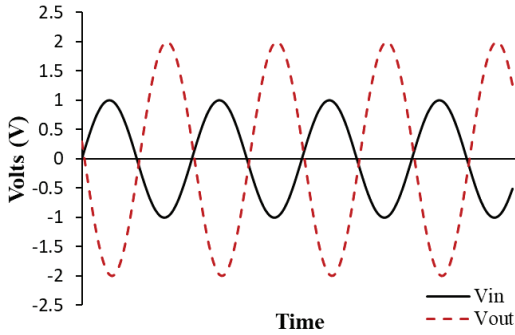


Figure 7-12. Input and output of an inverting amplifier (gain of  $-2$ ).

The  $V_{out}$  signal is *inverted* and *gained* with respect to  $V_{in}$ . This looks useful, and it is! Inverting amplifiers can be used to raise the amplitude of a signal, for instance: raising the volume of an audio signal, or the magnitude of a sensor's voltage response.

An inverting amplifier with a gain of 1 ( $R_F=R_I$ ) is called an *inverter*, illustrated in Figure 7-13.

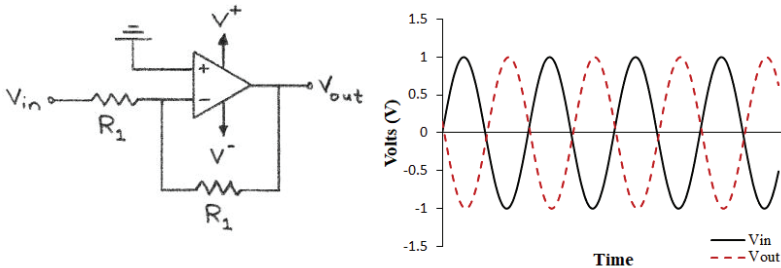


Figure 7-13. Configuration (left) and performance (right) of an inverting amplifier (unity gain).

By carefully matching resistors (check that they are as close in actual resistance values as possible) you can use this configuration to invert a negative voltage to the equivalent positive voltage (or *vice-versa*). You could use the inverter in Figure 7-13 after an inverting amplifier, in order to get the signal back to the original polarity (+ or -).

### Biasing the Output of an Inverting Amplifier

If you would like to *bias*, or shift the output voltage of an inverting amplifier by a fixed amount, one way to do this is to connect a bias voltage ( $V_b$ ) to the non-inverting input:

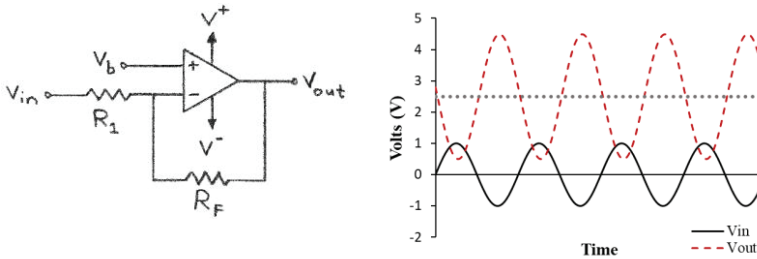


Figure 7-14. Biasing an inverting amplifier.

Let's say we would like to shift a probe's voltage range from (-1V to +1V) to (+4.5 to +0.5V), by *biasing* an inverting amplifier. We must select values of  $R_1$ ,  $R_F$ , and  $V_b$  to produce a *shift* of +2.5V and a *gain* of -2. The output signal we would like is the dashed line in the graph of Figure 7-14.

The equation to calculate  $V_{out}$  is:

$$A_v = -\frac{R_F}{R_1}$$

$$V_{out} = A_v V_{in} + (1 - A_v) V_b = -\left(\frac{R_F}{R_1}\right) V_{in} + \left(1 + \frac{R_F}{R_1}\right) V_b$$

The **bias voltage**,  $V_b$  is not simply added in this equation, because the gain acts on  $V_b$  as well. Working through this example, if  $R_1=1K$ :

$$A_v = -\frac{R_F}{R_1} = -2$$

$$\rightarrow R_F = 2 \times R_1 = 2 \times 1K = 2K$$

We know that when  $V_{in}$  is 0V, we would like  $V_{out}$  to be +2.5V. Solving for  $V_b$ :

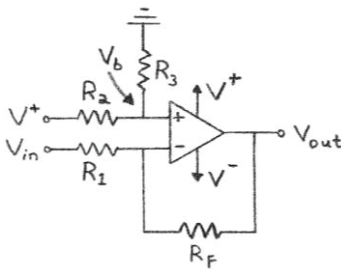
$$V_{out} = A_v V_{in} + (1 - A_v) V_b$$

$$\rightarrow V_b = \frac{V_{out} - A_v V_{in}}{1 - A_v}$$

$$V_b = \frac{2.5V - (-2 \times 0V)}{1 - (-2)} = \frac{2.5V}{3}$$

$$\rightarrow V_b = 0.833V$$

How could we produce a  $V_b$  of +0.833 V? A voltage divider from  $V^+$  will do the job, giving rise to the following op-amp configuration (and general formula):



$$V_b = V^+ \left( \frac{R_3}{R_3 + R_2} \right)$$

$$\rightarrow R_3 = \frac{R_2}{\left( \frac{V^+}{V_b} \right) - 1}$$

$$V_{out} = A_v V_{in} + (1 - A_v) V_b$$

$$V_{out} = \left( -\frac{R_F}{R_1} \right) V_{in} + \left( 1 + \frac{R_F}{R_1} \right) \left( \frac{R_3}{R_2 + R_3} \right) V^+$$

Figure 7-15. Using a voltage divider to generate a bias voltage for an inverting amplifier.

If we set  $V^+ = +5V$ , and  $R_2 = 1K$ , we can solve for  $R_3 = 0.2K$ .

By biasing our signal, we can shift it into a range that can be read by the microcontroller analog pins, which are not able to read voltages less than 0. We can now double-check the math to make sure our shift and gain work properly:

**Table 7-1. Calculating the expected op-amp output across the expected input range.**

| $V_{in,I}$ | $V_{out} = A_v V_{in} + (1 - A_v) V_b$                  |
|------------|---------------------------------------------------------|
| -1V        | $-2 \times (-1V) + (1 - (-2))0.833V = 2V + 2.5V = 4.5V$ |
| 0V         | $-2 \times (0V) + (1 - (-2))0.833V = 0V + 2.5V = 2.5V$  |
| +1V        | $-2 \times (1V) + (1 - (-2))0.833V = -2V + 2.5V = 0.5V$ |

**Test your understanding:** what would the  $V_{out}$  signal look like if we used the TL072 chip from before, with  $V^+ = +5V$ , and a headroom of 1.6V?

## Non-Inverting Amplifier

Although an inverting amplifier is useful, it can be more convenient to preserve the polarity of a signal, particularly when gaining a positive signal.



To make a non-inverting amplifier, you connect the signal ( $V_{in}$ ) to the **non-inverting input** (Figure 7-16).

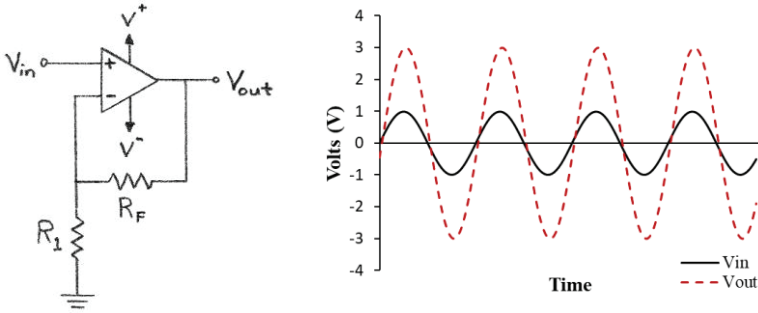


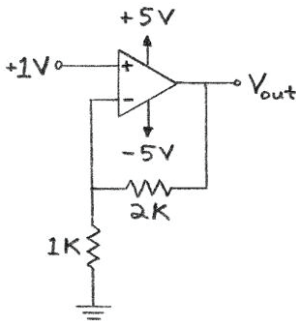
Figure 7-16. Configuration (left) and example performance (right) of a non-inverting amplifier.

Figure 7-16 (right) illustrates the performance of a non-inverting amplifier, with a gain of 3. Notice now, like the buffer configuration, that the output signal is *not* inverted with respect to the input signal.

The gain for a non-inverting amplifier is calculated using the equation:

$$A_v = \frac{V_{out}}{V_{in}} = \left( 1 + \frac{R_F}{R_1} \right)$$

A sinusoidal input is shown to illustrate how the op-amp responds to a changing signal. We can also set our signal to be a fixed voltage, and then use the formula to predict what the expected output should be (Figure 7-17).



$$A_v = \frac{V_{out}}{V_{in}} = \left( 1 + \frac{R_F}{R_1} \right) = \left( 1 + \frac{2K}{1K} \right) = 3$$

$$V_{out} = A_v V_{in} = 3 \times 1V = +3V$$

Figure 7-17. Calculating  $V_{out}$  for a non-inverting amplifier.

### *Biasing the Output of a Non-Inverting Amplifier*

If you expect your signal to be negative and you would like to shift the output into a positive range, you can also bias a non-inverting amplifier (Figure 7-18).

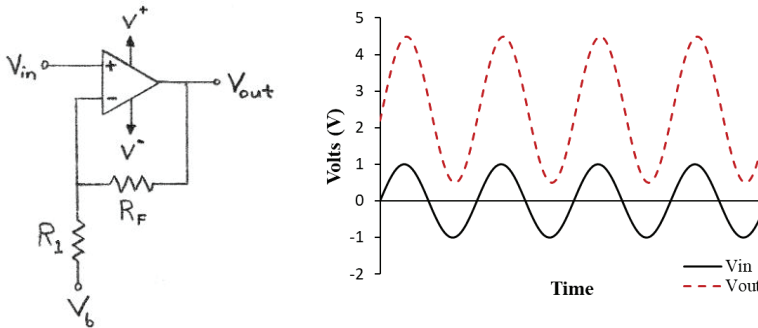


Figure 7-18. Configuration (left) and example performance (right) of a non-inverting operational amplifier, with a bias voltage.

Figure 7-18 (right) illustrates the performance of a non-inverting amplifier with a bias voltage. See if you can figure out what the values for gain and  $R_F/R_1$  are for the op-amp, and check your answer on the footnote of this page.<sup>6</sup> No peeking!

As with the biased inverting amplifier, the output voltage is not shifted by the value  $V_b$ . The actual relationship is:

$$A_v = \left(1 + \frac{R_F}{R_1}\right)$$

$$V_{out} = A_v V_{in} + (1 - A_v) V_b = \left(1 + \frac{R_F}{R_1}\right) V_{in} - \left(\frac{R_F}{R_1}\right) V_b$$

A **negative voltage** is required to shift  $V_{out}$  upwards. The shift is multiplied by  $R_F/R_1$ , which is something to keep in mind if you are planning on biasing an inverting amplifier in this configuration.

A big advantage of a non-inverting amplifier is that it keeps the output signal the right way around: in the positive region. This is especially

<sup>6</sup> From the graph,  $V_{out}$  swings from 0.5 to 4.5 (4V), whereas  $V_{in}$  swings from -1 to +1V, so the gain is  $4V/2V = 2$ .  $A_v = (R_F/R_1) + 1$ , so  $R_F/R_1 = 2 - 1 = 1$ . This means  $R_F = R_1$ . When  $V_{in} = 0$ , the gain has no effect and  $V_{out} = 2.5$ , so we can easily solve for the bias voltage:  $V_b = (V_{out} - A_v \times V_{in}) / (1 - A_v) = (2.5 - 2 \times 0) / (1 - 2) = -2.5V$ .

convenient since the Arduino Uno can't read a negative voltage with an analog pin—only a voltage from 0 to +5V. Inverting a signal doesn't have to be a problem, but it's nice to know that op-amps can handle this job without a second stage. With this configuration, you need to remember to add 1 to the ratio of  $R_F/R_1$  to calculate gain. So even if  $R_F=R_1$ , the gain is 2.

With a non-inverting amplifier, if you aren't expecting  $V_{in}$  to be negative or close to zero (which can happen), you can simplify your circuit by tying the negative rail of the op-amp ( $V^-$ ) to ground.

## Differential Amplifier

A *differential amplifier* is used for subtraction. It finds the difference between two signals. The general configuration of a differential amplifier is provided in Figure 7-19.

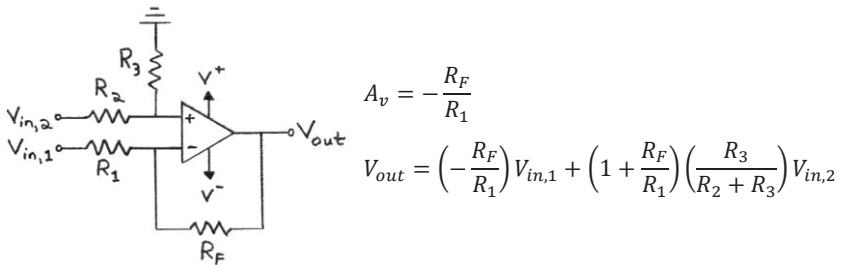


Figure 7-19. Differential amplifier configuration (left) and equations (right).

You might recognize this as an inverting amplifier with a bias—because that's exactly what it is. We can simplify this configuration by setting  $R_1 = R_2$ , and  $R_3 = R_F$ . This results in a  $V_{out}$  equation of:

$$V_{out} = \frac{R_F}{R_1}(V_{in,2} - V_{in,1})$$

To simplify this configuration further, if all resistor values in this configuration are equal, then the equation simplifies to  $V_{out}=V_{in,2}-V_{in,1}$ . This is the simplest form of a differential amplifier, as it simply subtracts the input voltages without performing any gain.

For example, the circuit in Figure 7-20 subtracts two fixed voltages.

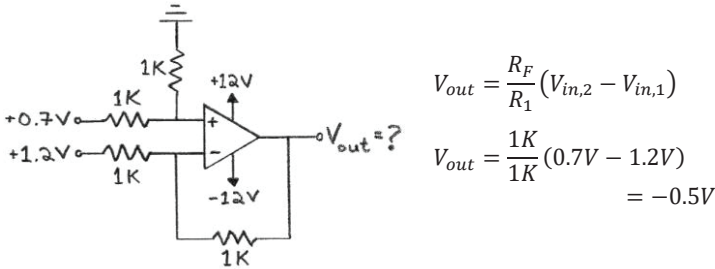


Figure 7-20. Calculating  $V_{out}$  example for a differential amplifier.

**Test your understanding:** what would the output voltage be if:

- The negative rail is tied to ground?
- $R_3$  and  $R_F$  are 2K resistors?

## Summing Amplifier (Inverting)

A summing amplifier adds voltages together. Adding signals together would be needed for example when you are mixing audio tracks together during a live performance. The generalized inverting summing amplifier op-amp configuration is shown in Figure 7-21.

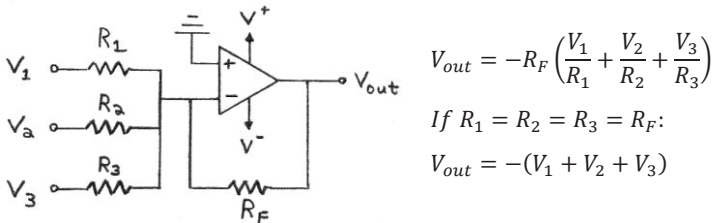
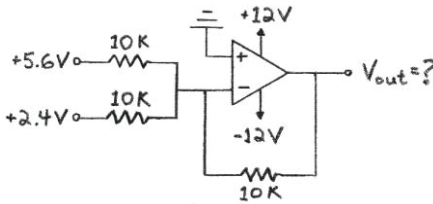


Figure 7-21. Inverting summing amplifier configuration (left) and equations (right).

Although three inputs are shown, an inverting amplifier can also have two inputs (or more than three). For this configuration, the negative rail can't be tied to ground. If it were tied to ground, you will lose a positive signal since the op-amp won't be able to output a negative voltage. Figure 7-22 shows a worked example, adding only two voltages together.



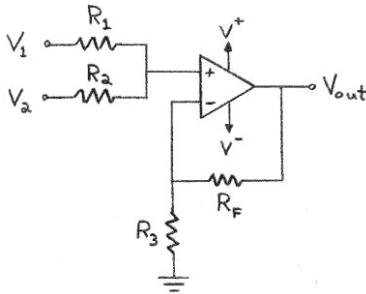
Since all of the resistors are of equal value (10K):

$$\begin{aligned} V_{out} &= -(V_1 + V_2) \\ &= -(5.6V + 2.4V) \\ &= -8.0V \end{aligned}$$

Figure 7-22. Calculating  $V_{out}$  example for an inverting summing amplifier.

### Summing Amplifier (Non-Inverting)

Although slightly more complicated in terms of doing the math, a non-inverting summing amplifier has the advantage of returning the “answer” in the correct polarity. A general non-inverting summing amplifier configuration is shown in Figure 7-23. This configuration is drawn with only two inputs, although more are possible.



$$V_{out} = \left(1 + \frac{R_F}{R_3}\right) \left[ V_1 \left(\frac{R_2}{R_1 + R_2}\right) + V_2 \left(\frac{R_1}{R_1 + R_2}\right) \right]$$

Figure 7-23. Non-inverting summing amplifier configuration (left) and equations (right).

If all the resistors are of equal value, then the equation for  $V_{out}$  simplifies to:

$$V_{out} = (1 + 1) \left[ V_1 \left(\frac{1}{2}\right) + V_2 \left(\frac{1}{2}\right) \right] = (V_1 + V_2)$$

We can use this amplifier configuration to shift and gain any signal to a convenient range (such as 0 to +3.3V, for the analog pin of your microcontroller to read).

Figure 7-24 shows a worked example, with fixed voltages as inputs. Note that because both inputs are positive, and this is a non-inverting amplifier, we can just tie the negative rail to ground. Careful when doing this though, because if the output voltage is too close to the bottom rail (which is now 0) then a non-rail-to-rail op-amp (like the TL07x series) will not be able to report signals close to the bottom rail.

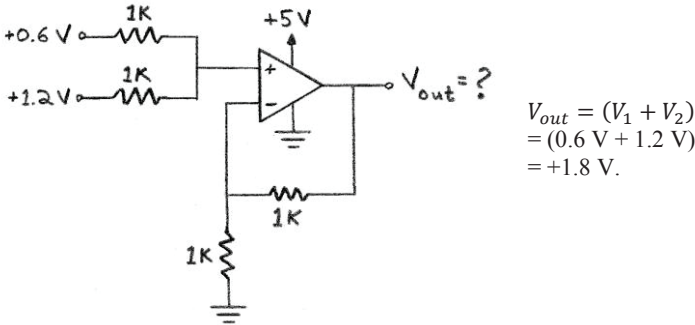


Figure 7-24. Calculating  $V_{out}$  example for a non-inverting summing amplifier.

### Summing Amplifier (Non-Inverting) Equations Solved

One of the more practical applications of a summing non-inverting amplifier is to take an anticipated signal from a sensor with a known (or expected) voltage range, *shift* it with an appropriate bias voltage ( $V_b$ ), and also *gain* it so that it spans the entire `analogRead()` range of the Arduino Uno (0-5V) without exceeding this range. This will maximize the response and resolution of the signal, and protect the microcontroller from damage. The op-amp configuration in Figure 7-25 is up to the task.

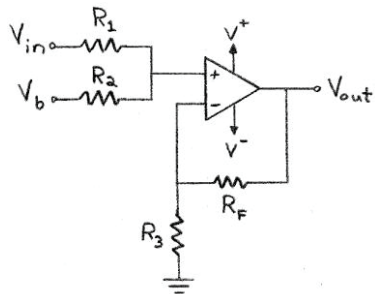


Figure 7-25. This non-inverting summing amplifier shifts and gains the signal,  $V_{in}$ .

We start with the  $V_{out}$  equation for this op-amp configuration:

$$(1) \quad V_{out} = \left(1 + \frac{R_F}{R_3}\right) \left[V_{in} \left(\frac{R_2}{R_1 + R_2}\right) + V_b \left(\frac{R_1}{R_1 + R_2}\right)\right]$$

This equation is a little more daunting, but broken into parts, it makes sense. What this equation is saying is that the output voltage is a function of the gain of the op-amp, set by the term  $\left(1 + \frac{R_F}{R_3}\right)$ , multiplied by the input voltage at the non-inverting input  $V_{in(+)}$ , which is determined by the voltage divider network at that input  $\left[V_{in} \left(\frac{R_2}{R_1+R_2}\right) + V_b \left(\frac{R_1}{R_1+R_2}\right)\right]$ . A detailed derivation of this term is provided in the appendix. We can then decide on a bias voltage, solve for the resistor values  $R_1$  and  $R_2$ , then plan the resistor values for the gain we want by solving the equation:

$$\text{Gain} = \left(1 + \frac{R_F}{R_3}\right)$$

$$(2) \quad \frac{R_F}{R_3} = \text{Gain} - 1$$

### *How do we use these formulas?*

In order to illustrate how to these equations work, we need an example. Let's say we have a glass pH electrode that gives a signal voltage proportional to the pH of a solution. A pH probe sends out a voltage signal that varies linearly with pH, from about -420 mV (at pH 14) to +420 mV (at pH 0), according to the Nernst equation. (Bard and Faulkner 2001; Omega Engineering 2018) To allow for a little wiggle room at both ends, we can widen the expected voltage range of our signal to be (-0.5V to +0.5V). We would like to *shift* this signal up so that it is not negative, and we would like to *gain* the signal so that it spans 0-3.3V. We select 3.3V (and not 5V) to allow for some headroom with the op-amp output (the TL07x series gives us about 1.6V headroom with a +5V positive rail, so 3.4V would be the highest  $V_{out}$  the op-amp can report). Recall that we can also make use of the microprocessor's AREF pin so that the analog readings range from 0-3.3V, giving us better resolution. (see Section 4: *External Analog Reference: AREF Pin*).

There are two ways this method is solved here: a longer method which explains how the signal makes its way through the op-amp, and a quicker method which jumps straight to the math. Both arrive at the same answer. If you would like less of an explanation, skip down to the quick way.

### Solving the Summing Amplifier with Bias: Longer Method

- a) The first step is to solve for the bias voltage ( $V_b$ ) that we would like to apply to our pH electrode signal. This will be responsible for shifting the signal upwards. The equation governing how the bias voltage affects the input voltage at the non-inverting terminal is:

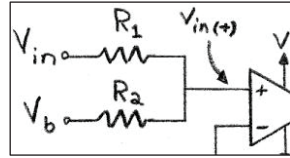


Figure 7-26. Voltage at the non-inverting input,  $V_{in(+)}$ .

$$(3) V_{in(+)} = V_{in} \left( \frac{R_2}{R_1 + R_2} \right) + V_b \left( \frac{R_1}{R_1 + R_2} \right)$$

A reliable arbitrary decision for a summing amplifier is to set  $R_1 = R_2 = 1K$ . Now we can solve for a bias voltage  $V_b$  that would make the lowest  $V_{in(+)} = 0V$ . Let's re-arrange Equation (3) to solve for  $V_b$ , substituting in  $R_2 = R_1 = 1K$ :

$$V_{in(+)} = V_{in} \left( \frac{1K}{1K + 1K} \right) + V_b \left( \frac{1K}{1K + 1K} \right)$$

$$(4) \rightarrow V_{in(+)} = \frac{V_{in} + V_b}{2}, \text{ so } V_{in(+)} \text{ is just the average of } V_{in} \text{ and } V_b.$$

Re-arranging (4):

$$(5) V_b = 2 \times V_{in(+)} - V_{in}$$

At the lowest  $V_{in}$  ( $V_{in,min} = -0.5V$ ), we would like  $V_{in(+)} = 0V$ :

$$V_b = 2 \times 0V - (-0.5V)$$

$$V_b = +0.5V$$

- b) Now we calculate what the voltage range will be at the non-inverting input using equation (4), and we know *a priori* that the range of  $V_{out}$  should be (0 to 3.3V). These values are summarized in Table 7-2. The table includes the signal midpoint,  $V_{mid}$ , so you can see where it ends up after shifting and gaining.

**Table 7-2. Gaining and shifting a pH probe signal to a convenient range for the analogRead() function (longer method).**

|                                                  | $V_{in}$<br>(pH probe) | $V_{in(+)}$<br>$= (V_{in} + V_b) / 2$ | Desired $V_{out}$ range<br>for analogRead() |
|--------------------------------------------------|------------------------|---------------------------------------|---------------------------------------------|
| at $V_{in,min}$ :                                | -0.5V                  | $V_{in(+),min} = 0V$                  | $V_{out,min} = 0V$                          |
| $V_{mid} =$<br>$(V_{in,min} + V_{in,max}) / 2$ : | 0V                     | $V_{in(+),mid} = +0.25V$              | $V_{out,mid} = +1.65V$                      |
| at $V_{in,max}$ :                                | +0.5V                  | $V_{in(+),max} = +0.5V$               | $V_{out,max} = +3.3V$                       |



In looking at Table 7-2, we can now calculate the required  $A_v$  of the op-amp. This will be the gain necessary to amplify the range of  $V_{in(+)}$  (0 to +0.5V) to the full range the Uno will read (0 to +3.3V).

Amplification will be applied *after* the voltage divider formed by  $R_1$  and  $R_2$ , so we need to use the inverting input range ( $V_{in(+),max}$ ) on the denominator of our equation for  $A_v$ . We therefore need an  $A_v$  of:

$$A_v = \frac{V_{out,max} - V_{out,min}}{V_{in(+),max} - V_{in(+),min}} = \frac{3.3V - 0V}{0.5V - 0V} = \frac{3.3V}{0.5V} = 6.6$$

We now have the design parameters we need to remap  $V_{in}$  to a great working range for our 5V microprocessor, illustrated in Figure 7-27.

The gained signal will be more sensitive to changes in pH, and there won't be any negative voltages to potentially damage the microcontroller.

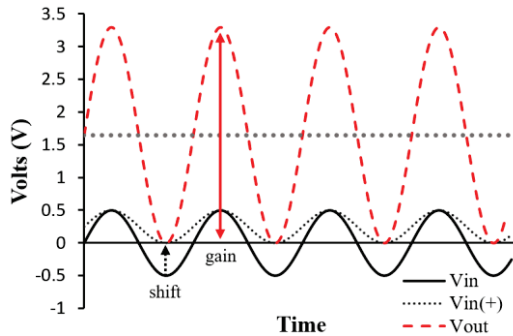


Figure 7-27. Shifted and gained pH electrode voltage signal.  $V_b$  shifts the signal positive, and after amplifying, the total shift is +1.65V.

- c) The next step is to solve the resistor values for the  $A_v$  we need, using Equation (2):

$$\frac{R_F}{R_3} = A_v - 1$$

$$\frac{R_F}{R_3} = 6.6 - 1 = 5.6$$

We can set  $R_3=1K$  (arbitrary), so:

$$R_F = 5.6 \times R_3 = 5.6 \times 1K = 5.6K$$

We have now solved the circuit completely, and we can re-draw it with our set and calculated resistor values:

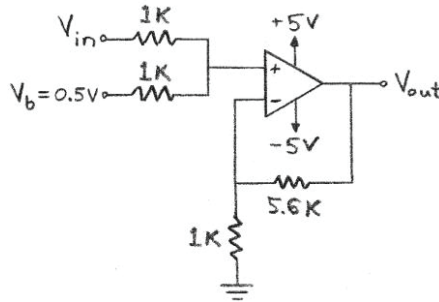


Figure 7-28. Solved non-inverting summing amplifier for pH meter.

The final equation of our op-amp will be Equation (1), with all our known values substituted in:

$$V_{out} = \left(1 + \frac{5.6K}{1K}\right) \left[ V_{in} \left( \frac{1K}{1K + 1K} \right) + 0.5V \left( \frac{1K}{1K + 1K} \right) \right]$$

$$V_{out} = 6.6 \left( \frac{V_{in} + 0.5V}{2} \right) = 3.3(V_{in} + 0.5V)$$

$$V_{out} = 3.3V_{in} + 1.65V$$

The overall signal gain of our  $V_{in}$  signal is 3.3, and the shift of that signal is 1.65V. Note that these are *different* values from  $A_v$  (our closed-loop gain of 6.6) and  $V_b$  (our bias voltage of 0.5V).

The solved op-amp configuration in Figure 7-28 could have equally worked with 10K for  $R_1$ ,  $R_2$ , and  $R_3$ , and 56K for  $R_F$ . The op-amp would have functioned just as well. So why start with 1K? A 1K resistor is a good all-purpose value when you need to make an arbitrary decision like this. Keeping things simple means you won't have to go shopping for new resistor values every time you start a project. The lab inventory has thousands of 1K resistors. They are called for in almost every project.

### ***Solving the Summing Amplifier with Bias: Quicker Method***

To speed up solving this problem, we don't need to think about what's going on at the inverting input of our amplifier. The math is already handled in the summing with bias op-amp equation. Start with a table of the known and desired ranges of  $V_{in}$  and  $V_{out}$ :

**Table 7-3. Gaining and shifting a pH probe signal to a convenient range for the analogRead() function (quicker method).**

|                   | $V_{in}$<br>(pH probe) | Desired $V_{out}$ range for<br>analogRead() |
|-------------------|------------------------|---------------------------------------------|
| at $V_{in,min}$ : | -0.5V                  | $V_{out,min} = 0V$                          |
| at $V_{in,max}$ : | +0.5V                  | $V_{out,max} = 3.3V$                        |

From Equation (1):

$$V_{out} = \left(1 + \frac{R_F}{R_3}\right) \left[ V_{in} \left( \frac{R_2}{R_1 + R_2} \right) + V_b \left( \frac{R_1}{R_1 + R_2} \right) \right]$$

Set  $R_1$ ,  $R_2$ , and  $R_3 = 1K$  (arbitrary).

Writing out our  $V_{in}$  and  $V_{out}$  values in Table 7-3 helps us write Equation (1) twice, and we can solve two unknowns with two equations.

At  $V_{in,min} = -0.5V$ ,  $V_{out,min} = 0V$ . Substituting into Equation (1):

$$0V = \left(1 + \frac{R_F}{1K}\right) \left[ -0.5V \left( \frac{1K}{1K + 1K} \right) + V_b \left( \frac{1K}{1K + 1K} \right) \right]$$

$$0V = (1 + R_F) \left[ -0.25V + \frac{V_b}{2} \right]$$

$$+0.25V = \frac{V_b}{2} \rightarrow V_b = +0.5V$$

At  $V_{in,max} = +0.5V$ ,  $V_{out,max} = 3.3V$ . Substituting into Equation (1):

$$3.3V = \left(1 + \frac{R_F}{1K}\right) \left[ +0.5V \left( \frac{1K}{1K + 1K} \right) + 0.5V \left( \frac{1K}{1K + 1K} \right) \right]$$

$$3.3V = \left(1 + \frac{R_F}{1K}\right) \times 0.5V \rightarrow R_F = 5.6K$$

We can now write the solved Equation (1), substituting all known values:

$$V_{out} = \left(1 + \frac{5.3K}{1K}\right) \left[ V_{in} \left( \frac{1K}{1K + 1K} \right) + 0.5V \left( \frac{1K}{1K + 1K} \right) \right]$$

$$V_{out} = 3.3V_{in} + 1.65V$$

The overall gain of the signal is 3.3, and the overall shift is 1.65V (see Figure 7-27).

### *Buffering $V_{in}$ and Generating $V_b$*

Even though we solved the resistor values we need, we are not quite done yet, because the pH probe voltage needs to be buffered before being

added to the bias voltage. This is because the pH probe only puts out a very tiny current, so buffering it before adding in the bias voltage will help protect the signal and lower its impedance. We also still need to generate a +0.5V bias voltage. This can be accomplished by using a voltage divider from a 3.3V supply. To electrically isolate the bias voltage from the voltage divider, it would be prudent to buffer it as well. Putting these ideas together and adding a few more details, we obtain the circuit diagram in Figure 7-29.

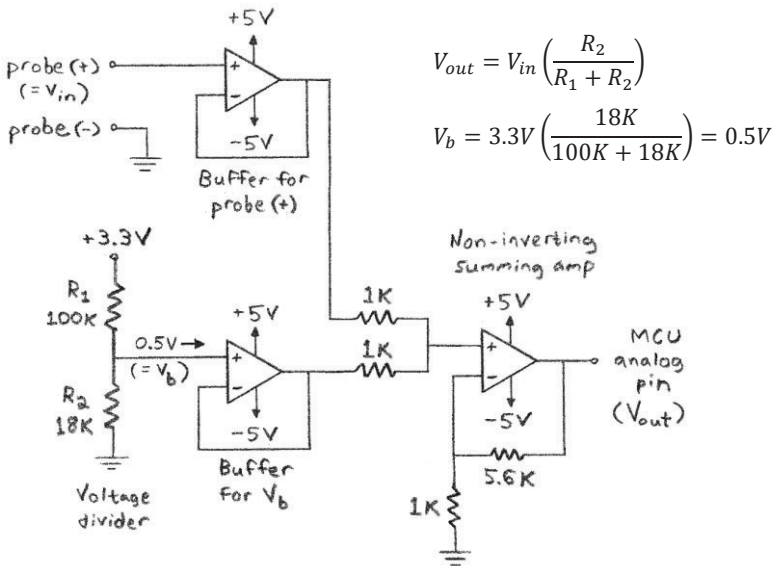


Figure 7-29. The bias voltage is generated using a voltage divider, and then buffered before adding it to the (buffered) probe voltage.

Stepping through the schematic in Figure 7-29, a bias voltage of +0.5V is generated using a voltage divider, then buffered. The pH probe signal is buffered. Then, the two signals (bias + probe) are combined, then gained in a non-inverting summing amplifier. The resulting signal  $V_{out}$  is read by a microcontroller analog pin.

A bias voltage is only needed if you are expecting negative volts out of your sensor (e.g. pH probe, or K-type thermocouple). If you get rid of  $V_b$ , this circuit simplifies to a non-inverting amplifier.

## Negative Voltage?

**Negative voltage** (e.g.  $-0.5\text{V}$ ) just means that the current runs the other way (recall that conventional current runs from higher voltage to lower voltage). If you reverse the voltmeter leads on a fresh 9V battery, the voltmeter will read  $-9\text{V}$ , instead of  $+9\text{V}$ . However, seeing a negative voltage for the first time on a circuit diagram can be a little mystifying. Where do these negative volts come from? How can you use them? What if you can't tie the  $V^-$  terminal of an op-amp to ground, because you would like it to work with a negative voltage? How do you supply negative volts, depicted in Figure 7-30? There are different approaches to answering this question.

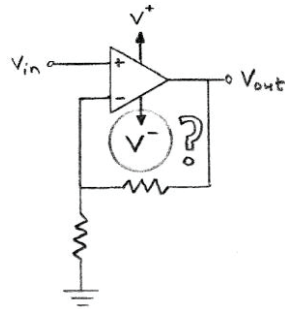


Figure 7-30. How do you supply negative volts?

### *Solution 1: Using a Virtual Ground*

The *negative* just has to be relative to what the op-amp “sees”. You can make the “ground” at a higher voltage than  $V^-$  and just keep track of what you are doing. Popular ways of doing this include a split power supply, or a voltage divider.

In both cases, the op-amp is told that ground is a higher voltage than the lowest voltage point in the system. This is often called a **virtual ground**, but really, it's just a different voltage reference point in the system. Either way, the op-amp doesn't care—and will operate with respect to whatever ground you choose. The probe should be tied to the *same virtual ground* as the op-amp, or the signal will be at the wrong voltage level. By tying the negative terminal of the probe to the virtual ground (Figure 7-31, top:  $+9\text{V}$ , Figure 7-31, bottom:  $+4.5\text{V}$ ) you are asking the probe to “float up” and work from a higher voltage level.

Some probes might not like it when you ground them to a higher voltage, but voltage is all about *relative* differences. As long as the probe doesn't get shorted against true ground, it shouldn't matter. This is one of the reasons why line workers don't get shocked when they touch power lines. As long as they don't touch ground at the same time, they can safely touch a high voltage line.

Compare this to other methods we discussed to bias operational amplifiers, and you can see that this is a simpler method to shift  $V_{out}$  higher, by telling the probe and the op-amp to work from a higher, virtual ground.

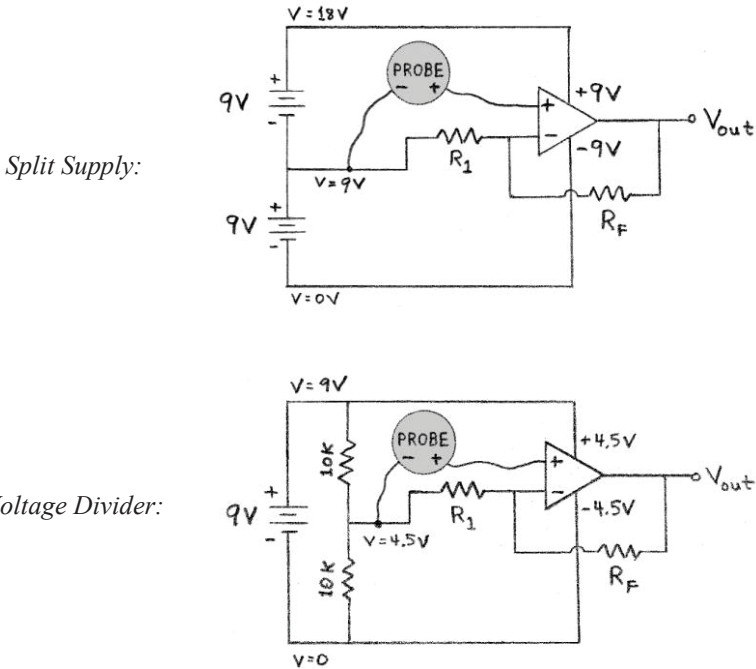


Figure 7-31. Splitting a power supply (top) and using a voltage divider (bottom) for supplying a negative voltage to the bottom rail of an op-amp.

### **Solution 2: Negative Voltage Generator**

Another practical option for providing negative volts to an op-amp without using a split supply or voltage divider is to use a voltage converter, like the ICL7660 (Figure 7-32). Wired in the configuration shown in Figure 7-32, this chip acts as a **negative voltage generator**. It generates a negative voltage supply (down to  $-10\text{V}$  DC) from a positive voltage source (up to  $+10\text{V}$  DC). It was specifically designed for supplying op-amps with negative voltage. It also has some other interesting functions—it can be used as a

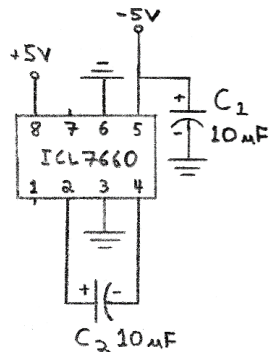


Figure 7-32. ICL7660 wired as a negative voltage generator ( $-5\text{V}$  out on pin 5).

voltage doubler, producing voltages twice as high as your power supply. (Intersil Corporation 2013)

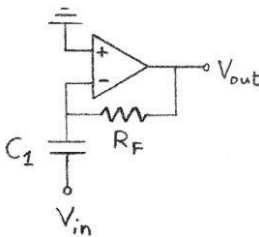
### *Solution 3: Negative Supply Line from an ATX Power Supply*

Some power supplies have negative voltage terminals or wires that you can easily access. For instance, the blue wire on an ATX power supply provides a regulated -12V DC. If you are already using an ATX power supply for your circuit, you can take advantage of this wire. With the power supply unplugged, pry or clip the wire off the connector, carefully strip it, and connect it to the bottom rail of your op-amp, making sure that you also connect the ATX power supply ground to your breadboard ground rail. See Table A-7 in the appendix for pin-out tables of the two main types of ATX power connectors: 20 pin and 24 pin.

## Op-Amps Can Do Calculus

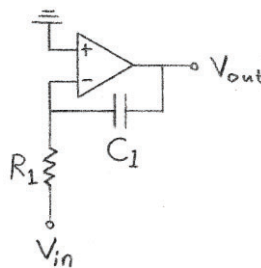
Op-amps can differentiate, and even integrate a signal. This would be useful if you need to know the rate of change of a signal, or how long a signal has been at a specific value (e.g. in a PID controller). You can also perform these functions mathematically using a microprocessor (for example, the PID control strategy presented in Section 6). (Carter and Brown 2016, 1-94)

### *Differentiator*



$$V_{out} = -R_F C_1 \frac{dV_{in}}{dt}$$

### *Integrator*



$$V_{out} = - \int_0^t \frac{V_{in}}{R_1 C_1} dt + C$$

Figure 7-33. Differentiator (left) and integrator (right) op-amp configurations.

## Signal Attenuation: Reducing the Voltage

Many probes have annoyingly low voltages that need to be gained-up using op-amps or other gain strategies. Occasionally, you may run into the opposite problem: the signal voltage range of a probe is too high to be read by the limited 0-5V range of an analog pin.

For example, let's say you have a probe with a signal that can range from 0-12V. How would you scale this down? This is called **attenuating** a signal. A simple way of handling **signal attenuation** is to use a voltage divider, followed by a buffer. This strategy is shown in Figure 7-34.

Taking the impedance of the signal into consideration, any two resistor values could be used for our example, as long as  $R_2 \approx 0.7 R_1$  (e.g.  $R_1=150\Omega$ ,  $R_2=100\Omega$ ).

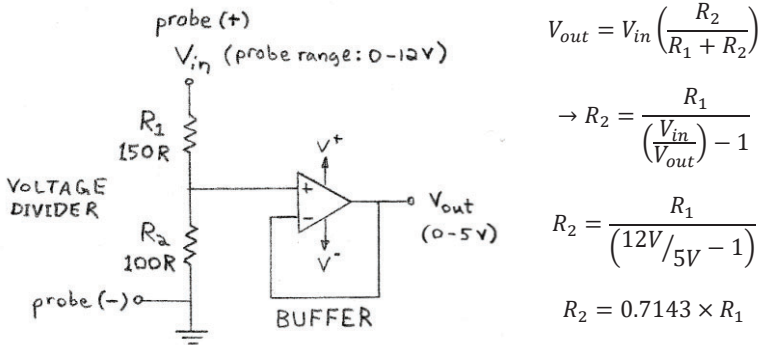


Figure 7-34. Attenuating a probe signal using a voltage divider, then buffering the output.

## Activity 7-1: Load Cell Scale

**Background:** A load cell is a strain gauge that has been designed to change resistance as it bends very slightly under pressure. A voltage is applied across it ( $V^+$ ), and the output voltage ( $\Delta V$ ) changes linearly with weight. An op-amp is required to boost the signal.

A load cell is a clever implementation of a **Wheatstone bridge** (Figure 7-35), a device that can measure very small changes in resistance. Although its structure looks like a

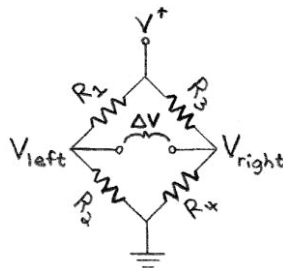


Figure 7-35. Structure of a Wheatstone bridge.



diamond, you can think of a Wheatstone bridge essentially as two voltage dividers wired in parallel. Are you seeing a common theme yet? Voltage dividers are everywhere. An advantage of a Wheatstone bridge is that it is a more accurate way to detect a change in resistance than the sense resistor for our thermistor circuit in Section 4.

The math is quite straightforward. The voltage divider equation is used twice:

$$V_{left} = V^+ \left( \frac{R_2}{R_1 + R_2} \right)$$

$$V_{right} = V^+ \left( \frac{R_4}{R_3 + R_4} \right)$$

$$\begin{aligned} \Delta V &= V_{left} - V_{right} = V^+ \left( \frac{R_2}{R_1 + R_2} \right) - V^+ \left( \frac{R_4}{R_3 + R_4} \right) \\ &= V^+ \left[ \left( \frac{R_2}{R_1 + R_2} \right) - \left( \frac{R_4}{R_3 + R_4} \right) \right] \end{aligned}$$

How does a Wheatstone bridge work? Let's say you have a purely resistive sensor taking the place of  $R_4$ . There are two basic strategies in using a Wheatstone bridge: one of them involves replacing resistor  $R_2$  with a potentiometer, then adjusting it until  $\Delta V$  is zero. Then, the resistance of the potentiometer is measured—which will be equal to  $R_4$  (the sensor). This method has the advantage of not being dependent on the value of  $V^+$  (which can be noisy). Another strategy is to directly measure  $\Delta V$ , and correlate it to the property you are sensing (in this case, the load). This is slightly easier as it involves fewer steps. We will be using the second strategy with our load cell.

Now you can see why the load cell has four wires: two are for supplying power to the Wheatstone bridge ( $V^+$  and GND), and the other two are for reading the voltage difference across the bridge ( $\Delta V$ ).

**Goal:** In Activity 7-1, we will use an op-amp to amplify the tiny change in voltage across the load cell, as weight is applied to it. A negative voltage generator (ICL7660) is incorporated to provide the op-amp with the ability to read voltages close to zero. If we tie the bottom rail to ground, the TL072 op-amp output might behave unpredictably around 0V, where we need it the most, as we will be measuring voltage changes close to zero.

**Materials:**

- 1 x Arduino Uno MCU & USB cable
- 1 x Digital Multimeter
- 1 x Breadboard
- 1 x Load Cell
- 1 x Retort Stand with Vinyl Retort Clamp
- 1 x TL072 Op-amp (DIP)
- 1 x LM358 Op-amp (DIP)
- 1 x ICL7660 Negative Voltage Generator (DIP)
- 4 x 1K Resistors
- 3 x 10K Resistors
- 2 x 100K Resistors
- 2 x 1M Resistors
- 2 x 10  $\mu$ F Electrolytic Capacitors
- 1 x Momentary Switch
- 1 x Set Calibration Weights
- 15 M/M Jumpers
- 1x Microspatula
- 1 x Highlighter

**Procedure:**

- 1) Assemble the following circuits:

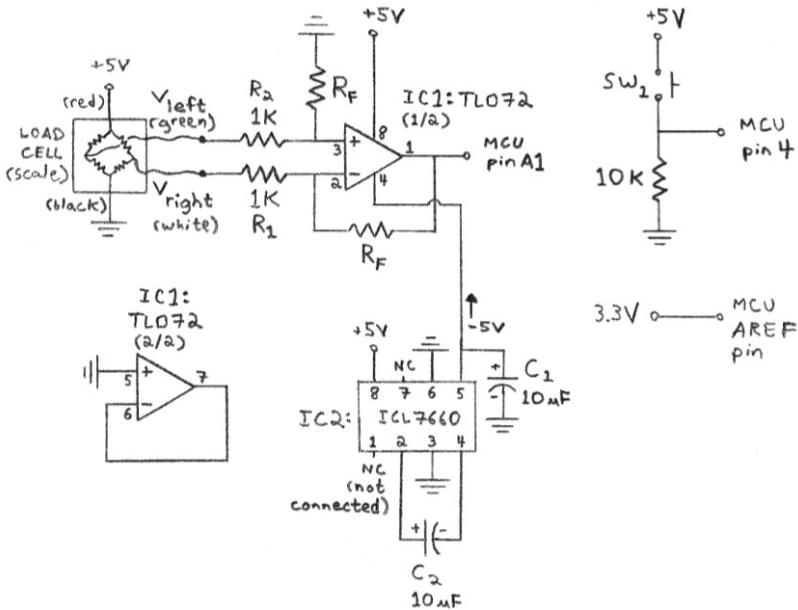


Figure 7-36. Circuit diagram for Activity 7-1 (load cell scale).

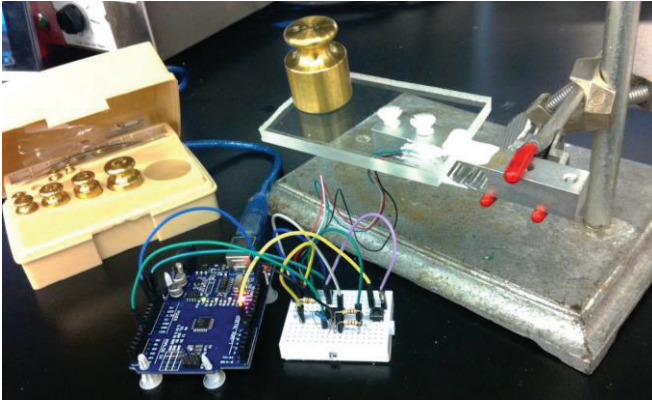


Figure 7-37. Experimental setup for Activity 7-1 (load cell scale).

- 2) Write and upload a sketch that reads the load cell amplified voltage on analog pin A1, and prints the results to the serial monitor. Remember to add the `"analogReference(EXTERNAL);"` line to the `setup()` function, and to connect +3.3V to the AREF pin.
- 3) Try different values of  $R_F$  until you get a useable signal (one that changes across the range of your calibrated weights, without maxing out the signal).
- 4) Calibrate the load cell with the weight set, then modify your sketch so that it reports measurements in grams. Weight is a linear function of voltage:

```
weight=scaleInt+(scaleSlope*voltage);
```

To help you solve for `scaleInt` and `scaleSlope`, download *scaleCal.xlsx* from the course website.

- 5) Program the momentary switch on the right of the circuit diagram as a tare button. To work this into your program, have the program subtract a global float `tareWeight` variable from the reported weight:
 

```
weight=scaleInt+(scaleSlope*voltage)-tareWeight;
```
- 6) How does op-amp type affect the output signal? Replace the TL072 in your circuit with an LM358 op-amp. The pin allocations are the same for both chips, so you will not need to rewire the circuit. Does the LM358 result in a different response of your signal?

**Note:** Are you having trouble finding your signal? Here are some focused troubleshooting tips. Also see the *Troubleshooting Guide* in the appendix for more suggestions.

- With a multimeter, check that pin 5 of the negative voltage generator is actually producing close to -5V.
- Make sure the bare resistor wires aren't touching other resistor wires, creating unintentional connections.
- Check the wiring of your op-amp for loose or wrong connections. Wiring an op-amp is tricky. Sometimes the best way to fix wiring is to re-wire from the beginning.
- The TL072 has an unused op-amp in the circuit (pins 5, 6, and 7). Consider using this op-amp as a second amplification stage (configured as a summing, non-inverting amplifier), rather than using one very large gain for a single op-amp. This will help preserve your signal and keep it from maxing out.

**Leave your circuit (connected to your Arduino Uno) assembled for the next section.**

**Test your understanding:** Which op-amp configuration is used in this activity?

## Activity 7-2: pH Meter

**Background:** A glass pH electrode probe puts out a very tiny current, too weak to be read by a regular volt meter or analog pin. The probe signal needs to be *shifted* and *gained* from (-0.5 to +0.5V) to (0 to +3.3V). This problem is well-suited to operational amplifiers, and is a common problem to solve for many different sensors.

The strategy illustrated in Figure 7-29 will be used. The TL074 is a great choice for this application since it is a quad op-amp chip (meaning it has 4 independent op-amps included in one package). A negative voltage generator (ICL7660) is incorporated to provide the op-amps with the ability to read negative voltages from the pH probe.

The circuit diagram in Figure 7-29 has been redrawn here to reflect the physical layout of the TL074 chip. This is a different circuit diagram style than the “exploded” approach used in Activity 7-1, Figure 7-36 where the two op-amps in the TL072 chip are depicted separately. Either approach is valid for a circuit diagram. The first style (exploded components) makes more visual sense in terms of understanding the process workflow of the circuits. The second style preserves the physical layout of the integrated circuit, and some find easier to build from because the connections are drawn more literally. Which style do you prefer?

**Goal:** To build a pH meter, and then calibrate it using pH standard solutions.

**Materials:**

- 1 x Arduino Uno MCU & USB cable
- 1 x Digital Multimeter
- 1 x Breadboard
- 1 x Glass pH Electrode with Stand
- 1 x Female Coaxial Connector for pH Electrode
- 1 x TL074 Op-amp (DIP)
- 1 x LM324 Op-amp (DIP)
- 1 x ICL7660 Negative Voltage Generator (DIP)
- 3 x 1K Resistors
- 1 x 5.6K Resistor
- 1 x 18K Resistor
- 1 x 100K Resistor
- 2 x 10  $\mu$ F Electrolytic Capacitors
- 15 x M/M Jumpers (2 long)
- pH Standard Solutions (pH 4, 7)
- pH Electrode Storage Solution
- Tissues
- 1 x Microspatula or Small Screwdriver
- 1 x Highlighter
- Plastic Squeeze Bottle with Distilled Water
- 1 x 250 mL Waste Beaker

**Procedure:**

- 1) Build the pH meter circuit in Figure 7-38.

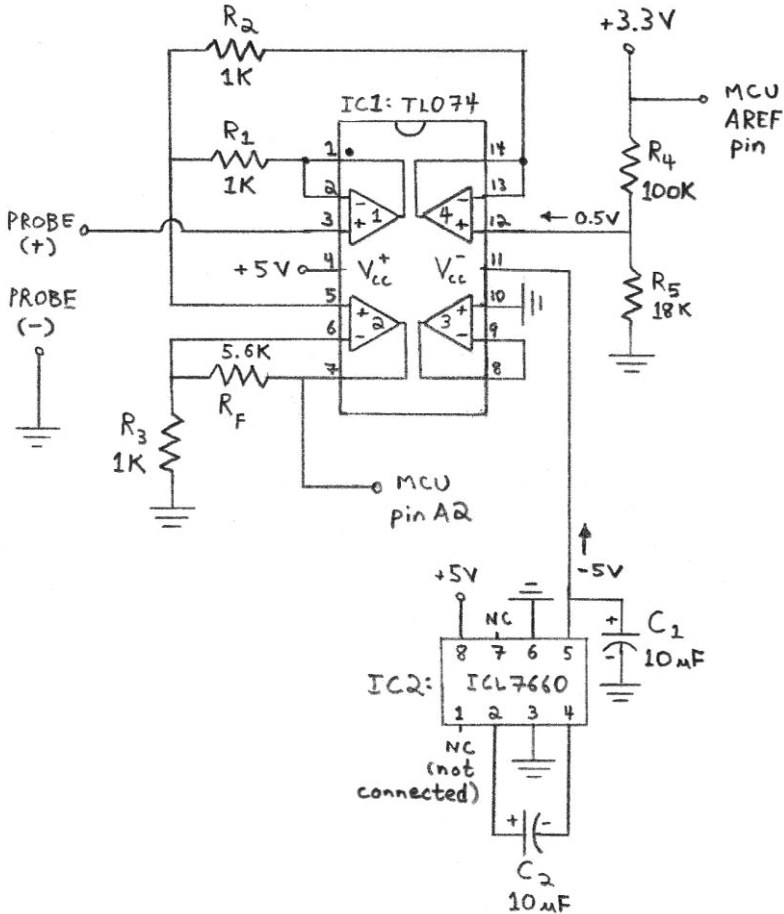


Figure 7-38. Circuit diagram for Activity 7-2 (pH meter).

**Note:** Connect the pH probe to the op-amp using the female coaxial connector.

- 2) Write and upload a sketch that reads the voltage on analog pin A2, and prints the results to the serial monitor. Remember to add the `analogReference(EXTERNAL);` line to the `setup()` function, and to connect +3.3V to the AREF pin.

- 3) Calibrate your probe with pH standards (4 and 7). You can download *pHCal.xlsx* from the course website to help calculate the pH probe slope and intercept for your program.
- 4) Edit your sketch to output the pH value to the serial monitor. The probe voltage is a linear function of pH:  
`pH=probeInt+(probeSlope*voltage) ;`
- 5) How does op-amp type affect the output signal? Replace the TL074 chip in your circuit with an LM324 op-amp. The pin allocations for this chip are the same as the TL074, so you will not need to rewire the circuit. Does the LM324 op-amp result in a different response in your signal?

**Note:** Are you having trouble finding your signal? Here are some focused troubleshooting tips. Also see the *Troubleshooting Guide* in the appendix for more suggestions.

- With a multimeter, check that pin 5 of the negative voltage generator is actually producing close to -5V.
- Make sure the bare resistor wires aren't touching other resistor wires, creating unintentional connections.
- Check the wiring of your op-amp for loose or wrong connections. Wiring an op-amp is tricky. Sometimes the best way to fix wiring is to re-wire from the beginning.

**Leave your circuit (connected to your Arduino Uno) assembled for the next section.**

**Did you know?** A thermocouple also outputs a tiny voltage from -0.5V to +0.5V. This circuit could double as a thermocouple amplifier, using the same resistor values.

## Learning Objectives for Section 7

After having attended this class, the student will be able to:

- 1) Identify the role an op-amp plays based on how it is represented in a circuit diagram.
- 2) Explain the purpose and the effect on an input signal waveform for the following devices:
  - Comparator, buffer, inverting amplifier, non-inverting amplifier, differential amplifier, summing amplifier (inverting and non-inverting)
- 3) Appropriately wire the power rails on an op-amp, taking into consideration headroom and signal characteristics.
- 4) From memory, draw a circuit diagram detailing how to provide negative volts to an op-amp, using a split supply, voltage divider, or negative voltage generator.
- 5) Given the equations relating  $V_{in}$  and  $V_{out}$ , calculate appropriate resistor values for op-amp configurations to accomplish a specific task (e.g. gain and/or shift).
- 6) Calculate appropriate resistor values to shift and gain a probe or sensor signal to 0-3.3V for the `analogRead()` function, for any DC probe.
- 7) Describe how a Wheatstone bridge works and be able to draw its structure from memory.



### Section 7 - Station Content List, Activity 7-1

- 1 x Digital Multimeter
- 1 x Breadboard
- 1 x Load Cell
- 1 x Retort Stand with Vinyl Retort Clamp
- 1 x TL072 Op-amp (DIP)
- 1 x LM358 Op-amp (DIP)
- 1 x ICL7660 Negative Voltage Generator (DIP)
- 4 x 1K Resistors
- 3 x 10K Resistors
- 2 x 100K Resistors
- 2 x 1M Resistors
- 2 x 10M Resistors
- 2 x 10  $\mu$ F Electrolytic Capacitors
- 1 x Momentary Switch
- 1 x Set Calibration Weights
- 15 M/M Jumpers (5 long)
- 1x Microspatula
- 1 x Highlighter

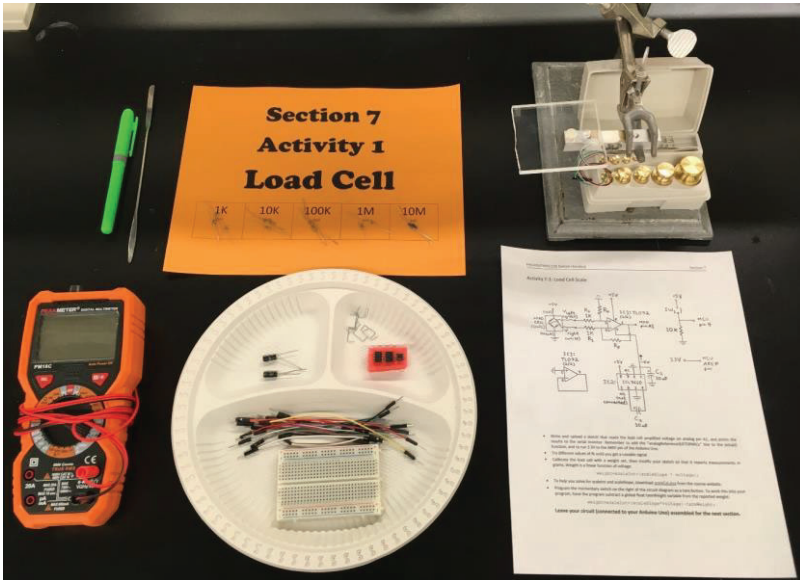


Figure 7-39. Station setup for Activity 7-1.

## Section 7 - Station Content List, Activity 7-2

- 1 x Digital Multimeter
- 1 x Breadboard
- 1 x Glass pH Electrode with Stand
- 1 x Female Coaxial Connector for pH Electrode
- 1 x TL074 Op-amp (DIP)
- 1 x LM324 Op-amp (DIP)
- 1 x ICL7660 Negative Voltage Generator (DIP)
- 3 x 1K Resistors
- 1 x 5.6K Resistor
- 1 x 18K Resistor
- 1 x 100K Resistor
- 2 x 10  $\mu$ F Electrolytic Capacitors
- 15 x M/M Jumpers (5 long)
- 2 x M/F Jumpers
- pH Standard Solutions (pH 4, 7)
- pH Electrode Storage Solution
- Tissues
- 1 x Microspatula or small screwdriver
- 1 x Highlighter
- Plastic Squeeze Bottle with Distilled Water
- 1 x 250 mL Waste Beaker

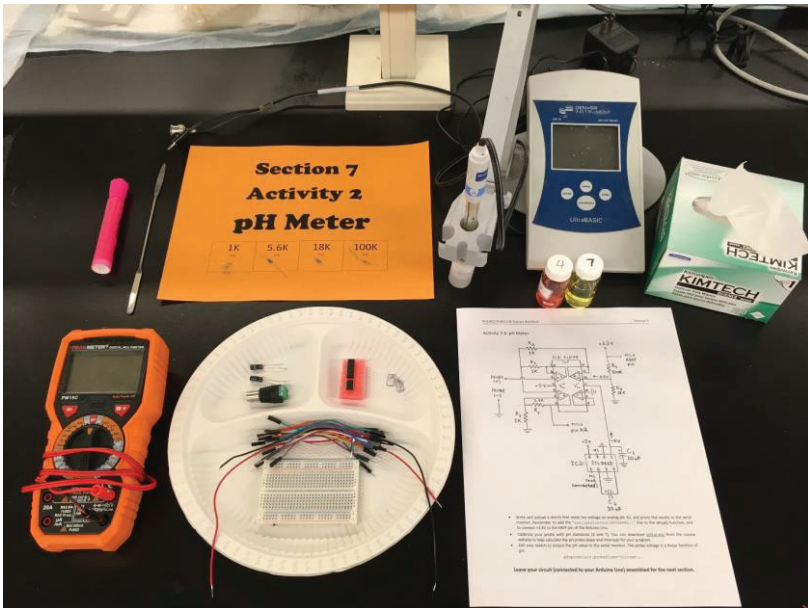


Figure 7-40. Station setup for Activity 7-2.

## SECTION 8

# DATA FILTERING, SMOOTHING, AND LOGGING

|                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |                       |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------|
| What You'll Be Learning | <b>Lecture:</b> Data filtering: high pass, low pass, band pass filters. Practical considerations of op-amps: input and output impedance. Reducing signal noise. Measuring noise amplitude. Data collection and smoothing: mean, median, mode, mean + threshold. Data logging: time functions, SD card logging. Logic shifting.                                                                                                                                                                                                                                                                                                                                                                                                        |                       |
| What You'll Be Doing    | <b>Pick any two of the following activities:</b><br><b>Activity 8-1:</b> Design and incorporate an LPF (low-pass filter) to filter incoming data from your Activity 7-x device.<br><b>Activity 8-2:</b> Incorporate a data smoothing strategy in your device sketch, using the smoothing library <i>QuickStats.h</i> .<br><b>Activity 8-3:</b> Add an SD card reader to your Activity 7-x device, and record the device output in .CSV format.<br><b>Demo 1:</b> RTC clock - better time accuracy<br><b>Demo 2:</b> Oscilloscope - interpreting signal and frequency information. Function generator output.<br><b>Demo 3:</b> RC filtering a signal (0.1 $\mu$ F and 1 K, $f_c=1592$ Hz). LPF (RC) and HPF (CR). 60 Hz notch filter. |                       |
| Files you will need     | All course files are available for download at:<br><a href="http://pb860.pbworks.com">http://pb860.pbworks.com</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    | • <i>QuickStats.h</i> |

### Data Filtering

The first thing you might notice when you assemble your first op-amp circuit successfully is, wow is that signal ever noisy! There are different approaches to dealing with noise. Specific undesired frequencies or bands of frequencies can be identified and filtered out. We will begin our discussion of data filtering by looking at first-order low-pass and high-pass filters.

### Low-Pass Filters (LPFs)

If your sensor readings look really jumpy, the first type of filter that you will be interested in is a basic **low-pass filter**. Low pass filters allow lower frequencies to pass through (hopefully your signal), but attenuate (reduce) higher frequency changes, which can be unwanted noise from your system.

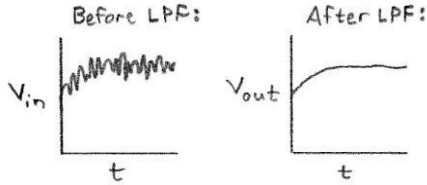


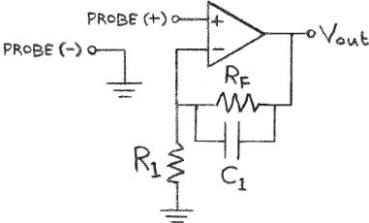
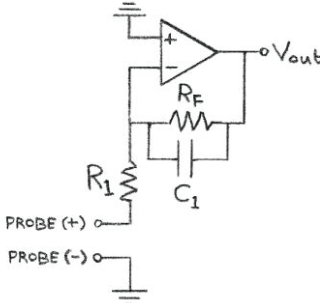
Figure 8-1. The effect of a carefully designed low-pass filter (LPF).

The **cutoff frequency**—a very important parameter for a filter—is the frequency above or below which the filter starts to “work”, filtering out unwanted changes in voltage. Some texts refer to this as “corner frequency”. With an RC filter (in Table 8-1), this frequency depends on the time constant  $\tau$  of the RC network, via the equation  $f_c = \frac{1}{2\pi RC} = \frac{1}{2\pi\tau}$ .

Low-pass filters are quite common in sensor filtering, as high frequency noise from AC power, fluorescent lights, and nearby equipment can really bury a signal in unwanted peaks.

**Table 8-1. First-order passive and active low-pass filters. (Texas Instruments Inc 2013, 1-26)**

|                                                                                                                                                                                                                                                                                                                                                                |                                                                                                                                       |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|
| <p><b>Passive Low-Pass Filter</b></p> <p>Cutoff Frequency: <math>f_c = \frac{1}{2\pi R_1 C_1}</math></p> <p>Pass-Region Gain: <math>V_{out} &lt; V_{in}</math></p> <p>More specifically:</p> <p>The capacitive reactance, <math>X_c</math>, of a capacitor, with a signal of a given frequency <math>f</math>, is: <math>X_c = \frac{1}{2\pi f C_1}</math></p> | <p><b>Active Low-Pass Filter</b></p> <p>Pass-Region Gain: 0 dB</p> <p>Cutoff Frequency: <math>f_c = \frac{1}{2\pi R_1 C_1}</math></p> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|

|                                                                                                                                                                                                                                                                       |                                                                                                                                                                                                                                                                            |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>For an LPF:</p> $V_{out} = V_{in} \times \frac{X_c}{\sqrt{R_1^2 + X_c^2}}$                                                                                                                                                                                         | <p><b>Active Low-Pass Filter:<br/>Non-Inverting, Amplifying</b></p>  <p>Pass-Region Gain: <math>1+(R_F/R_1)</math></p> <p>Cutoff Frequency: <math>f_c = \frac{1}{2\pi R_F C_1}</math></p> |
| <p><b>Active Low-Pass Filter:<br/>Inverting, Amplifying</b></p>  <p>Pass-Region Gain: <math>-(R_F/R_1)</math></p> <p>Cutoff Frequency: <math>f_c = \frac{1}{2\pi R_F C_1}</math></p> |                                                                                                                                                                                                                                                                            |

You might notice in Table 8-1 that the power rail lines on the op-amps have not been drawn. The power rails are still implied. They are left out here to reduce circuit diagram clutter. It's important to realize that the inverting amp needs a negative voltage applied to the bottom rail, but the bottom rail of the non-inverting amp can be tied to ground if a negative signal (plus the headroom needed on the positive side, close to ground) is not anticipated.  $V_{in}$  is the voltage from some probe we would like to filter, labeled PROBE (+).

The biggest difference between a passive and active LPF is that there is inevitably a bit of a voltage drop across the  $R_1$  resistor for the passive LPF, which may or may not be important. The active LPF buffers the signal (and amplifies it, if so configured) to compensate for the drop in voltage. A caveat of the inverting active RC filter is that you will notice the signal is connected to the *inverting* input. This means if this signal is positive, the filtered signal will be negative (with consequences associated—e.g. negative

power supply needed on the bottom rail of the op-amp). The non-inverting active RC filter provides filtering without inverting the signal.

These filters don't completely remove all frequencies higher than  $f_c$ . The attenuation is more pronounced the higher the frequency is. To demonstrate, Figure 8-2 shows a **Bode Magnitude Plot**, (pronounced *boh-dee*) which graphs the effect an LPF has on the gain of a signal, depending on the signal's frequency. In the "PASS REGION", the gain in decibels is zero ( $V_{in}=V_{out}$ ). However, as the signal frequency increases past  $f_c$ , the gain decreases—meaning that the output signal attenuates (lowers). This is called the "STOP REGION", where  $V_{out}<V_{in}$ .

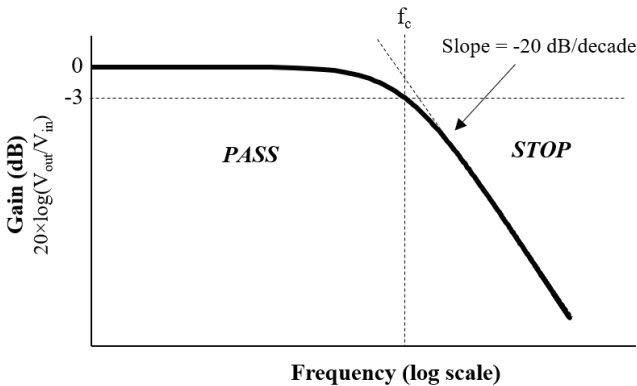


Figure 8-2. Bode Magnitude Plot for a first-order LPF.

At the cutoff frequency, the gain is -3 dB, which means that:

$$\text{Gain} = -3 \text{ dB} = 20 \times \log\left(\frac{V_{out}}{V_{in}}\right)$$

$$-\frac{3}{20} = \log\left(\frac{V_{out}}{V_{in}}\right)$$

$$\frac{V_{out}}{V_{in}} = 10^{\left(-\frac{3}{20}\right)} = 0.7079$$

The slope of response above  $f_c$  is -20 dB per decade, or -6 dB per octave. For musicians, an *octave* higher is a *doubling in frequency*. Many musical instruments are tuned to a fourth-octave A note, at 440 Hz. An octave higher is 880 Hz, which is a fifth-octave A note. In electronics, the term *octave* is also used with frequency, even though no musical instruments are involved. A *decade* is change by a multiple of ten in frequency (rather than its familiar meaning of 10 years). A decade above 440 Hz is  $10 \times 440\text{Hz} = 4,400 \text{ Hz}$

(or 4.4 kHz). The frequency range of human hearing is about 20 Hz to 20 kHz, so it would still be in the audible range (although higher than the notes on a piano keyboard). (Nagel et al. 2016)

**Example:** What is the attenuation of an unwanted frequency of 60 Hz, if the cutoff frequency ( $f_c$ ) of a low-pass filter is 6 Hz?

**Answer:** The gain of the LPF is -3 dB at the cutoff frequency (6 Hz). One decade higher, the gain will be:

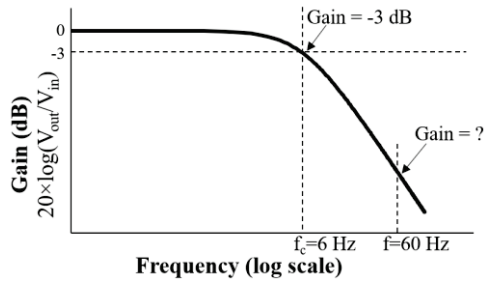


Figure 8-3. Calculation of the gain of an LPF, one decade higher than the cutoff frequency.

$$\text{Gain}(dB) = -3 \text{ dB} - \left( 1 \text{ decade} \times 20 \frac{\text{dB}}{\text{decade}} \right)$$

$$\text{Gain} = -23 = 20 \times \log \left( \frac{V_{out}}{V_{in}} \right)$$

$$\frac{V_{out}}{V_{in}} = 10^{\left(\frac{-23}{20}\right)} = 0.0708$$

The output signal would be reduced to about 7% of the original voltage. Not bad! By carefully selecting values for  $R_1$  and  $C_1$ , an LPF can be designed to filter out higher frequency noise from your signal.

### Low Pass Filter Design: Worked Practical Example

Let's say you have a very noisy signal. You determine the approximate frequency of oscillation by looking on an oscilloscope, and find that the distance between voltage spikes is 0.5 milliseconds. To convert this to a frequency:

$$f_{noise} = \frac{1}{(0.5 \text{ msec} \times 1 \text{ sec} / 1000 \text{ msec})} = 2000 \text{ Hz} = 2 \text{ kHz}$$

Your signal is pretty much a constant DC signal, and not expected to fluctuate very much. So you are considering using an LPF to filter this out.

Resistor values can be dialed in with a potentiometer. Capacitor values are harder to come by and more limited (although variable capacitors do exist). So rather than start with a resistor value and calculate the required

capacitor value, we will start with a 100 nF (= 0.1  $\mu$ F) capacitor, and find out the required resistance for our filter.

If we planned for the cutoff frequency to be equal to the frequency of noise, then we would only get a gain of -3 dB (or  $V_{out}/V_{in}=0.7$ ). For the filter to be effective, the cutoff frequency can be at least a decade below the noise. Let's opt for a cutoff frequency of 0.2 kHz, or 200 Hz. Now we can calculate the resistor value required:

$$f_c = \frac{1}{2\pi R_1 C_1}$$

$$R_1 = \frac{1}{2\pi f_c C_1} = \frac{1}{2\pi(200 \text{ Hz})(100 \times 10^{-9} \text{ F})} = 7957.75 \Omega = 7.96 \text{ k}\Omega$$

We now know we need about an 8K resistor with a 0.1  $\mu$ F capacitor to filter out our noise. We could choose a passive LPF (Figure 8-4, left), or if buffering the signal is required, an active LPF (Figure 8-4, right).

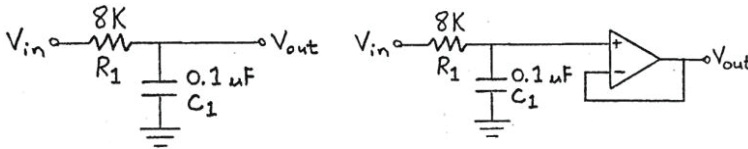


Figure 8-4. Worked example for an LPF,  $f_c=200$  Hz. Left: passive LPF, right: active LPF with unity gain.

The gain of the 2 kHz noise will be -23 dB, or  $\sim 7\%$  of the original signal amplitude. If signal amplification is required, we could also choose a non-inverting active low-pass filter with  $R_F=8\text{K}$ ,  $C_1=0.1 \mu\text{F}$ , and an  $R_1$  yielding a desired gain, recalling that  $\left(A_v = 1 + \frac{R_F}{R_1}\right)$  for a non-inverting amplifier. For  $R_1$ , we could either select the closest fixed resistor value to 8K (8.2K in *Common Fixed Resistor and Capacitor Values*), or we could use a 10K trim pot, and *dial in* a resistance of 8K. This means adjusting the 10K trim to 8K, using an ohmmeter. Then we could fine tune the resistance, monitoring  $V_{out}$  with an oscilloscope, until we are happy with the attenuation of noise.



### High-Pass Filters (HPFs)

High-pass filters let higher frequencies pass through. They act in the opposite manner as LPFs, in that they attenuate (reduce) frequencies *below*  $f_c$ . Figure 8-5 illustrates the possible effects of a high-pass filter. This style of filter is particularly useful with digital signals, which can have very high frequencies (e.g. see Figure 2-6, *filtering out low frequencies*).

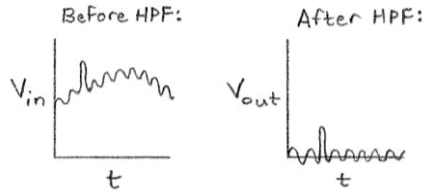
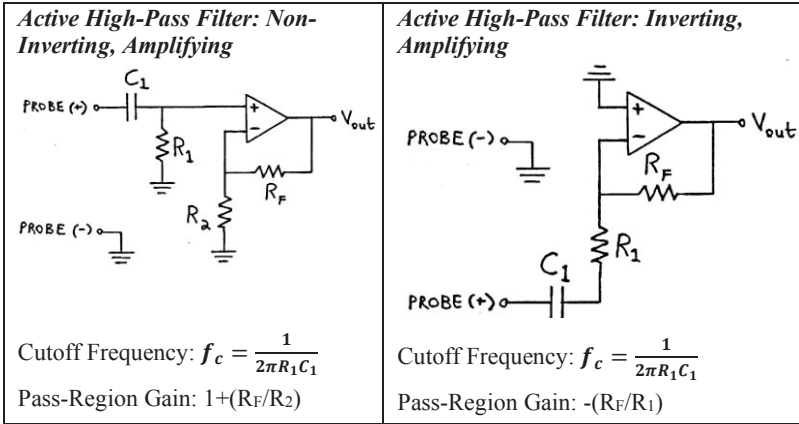


Figure 8-5. The effect of a carefully designed high-pass filter (HPF).

A high-pass filter is typically built into every speaker that has a tweeter. The high frequency sounds are sent to the tweeter, and the low frequency sounds that would otherwise damage the tweeter are filtered out. Many audio mixing boards also have a “low cut” button, which cuts out lower frequencies (e.g. thumps and footsteps) with a high-pass filter.

**Table 8-2. First-order passive and active high-pass filters. (Texas Instruments Inc 2013, 1-26)**

|                                                                                                                                                                                                                      |                                                                                                                                                                   |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><b>Passive High-Pass Filter</b></p> <p>Cutoff Frequency: <math>f_c = \frac{1}{2\pi R_1 C_1}</math></p> <p>For an HPF:</p> $X_c = \frac{1}{2\pi f C_1}$ $V_{out} = V_{in} \times \frac{R_1}{\sqrt{R_1^2 + X_c^2}}$ | <p><b>Active High-Pass Filter: Non-Inverting, Unity Gain</b></p> <p>Pass region gain: 0 dB</p> <p>Cutoff Frequency: <math>f_c = \frac{1}{2\pi R_1 C_1}</math></p> |
|                                                                                                                                                                                                                      |                                                                                                                                                                   |



The shape of the **Bode Magnitude Plot** for a first-order high-pass filter is the same as a low pass filter, but frequencies are attenuated *below*  $f_c$  (see Figure 8-6).

The equation for the cutoff frequency is the same, only now the signal is attenuated below the cutoff frequency, instead of above it (-20 dB/decade *below*  $f_c$ ).

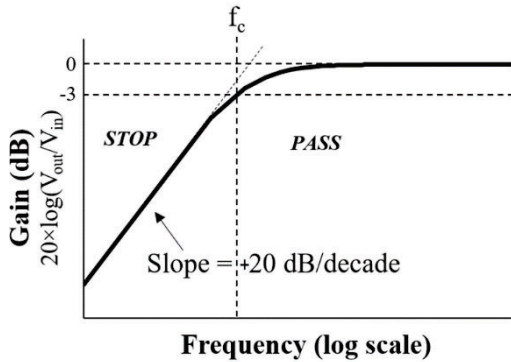


Figure 8-6. Bode Magnitude Plot for a first-order HPF.

### Inverting AC Amplifier

For some sensors (e.g. a microphone) it becomes inconvenient or undesirable to amplify a DC signal level, when the signal of interest changes at high frequencies. A carefully selected capacitor in series with a resistor will block DC levels in the signal and allow higher frequency fluctuations to pass through. Figure 8-7 shows the configuration for an ***inverting AC amplifier***. (Mancini 2002)

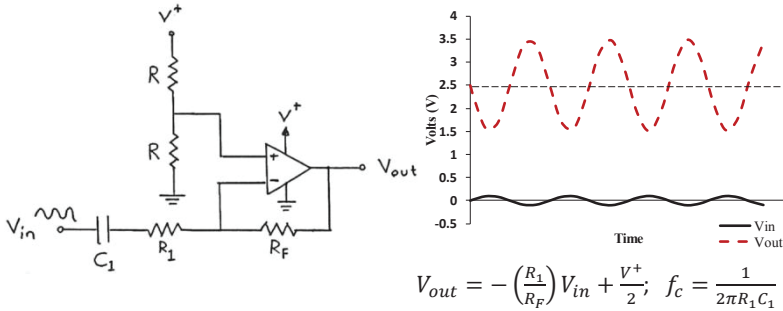


Figure 8-7. Inverting AC amplifier. Capacitor  $C_1$  filters out frequencies two decades below  $f_c$ . If  $V^+=+5V$ , the output signal will be gained by  $R_F/R_1$ , and oscillate about  $+2.5V$ .

The bias voltage applied to the non-inverting input ( $=V^+/2$ ) shifts the AC signal up to  $V^+/2$ . A benefit to this configuration is that a negative voltage supply is not needed, and the signal oscillates conveniently right in the middle of the op-amp’s working range.

The electret microphone preamplifier circuit in Figure 8-8 is a great example of the inverting AC amplifier in action. (Scherz and Monk 2016)

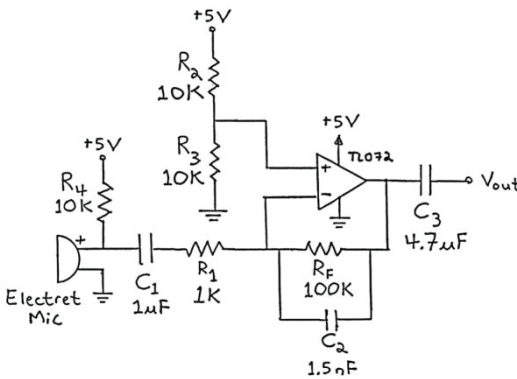


Figure 8-8. Inverting amplifier example: electret microphone preamplifier.

Capacitor  $C_2$  forms a high-pass filter with resistor  $R_F$  with a cutoff frequency of  $1/(2\pi \times 100K \times 1.5nF) = 1061$  Hz to attenuate the “squeal” from the signal, and capacitor  $C_3$  removes the final 2.5V DC level, returning it back to zero for a very useable amplified line-level audio signal. The 100K

resistor ( $R_F$ ) could be replaced by a 100K potentiometer to control the the output volume of the audio signal.

If you plan on building this circuit, note that electret microphones are polar devices. To find the positive terminal, test for continuity between a pin and the electret microphone body. The body should be connected to ground, and thus continuous (resistance=0 $\Omega$ ).

## Blocking the DC in your Signal: Charge Coupling

In the above example (Figure 8-8), capacitor  $C_3$  blocked the 2.5V DC level depicted in Figure 8-7 (right panel). In this context, it is called a **coupling capacitor**. The process of transmitting only an AC signal between stages or components of a circuit is called **charge coupling**, or **AC coupling**. A coupling capacitor is the simplest form of a high-pass filter. Figure 8-9 shows the basic configuration of a coupling capacitor.

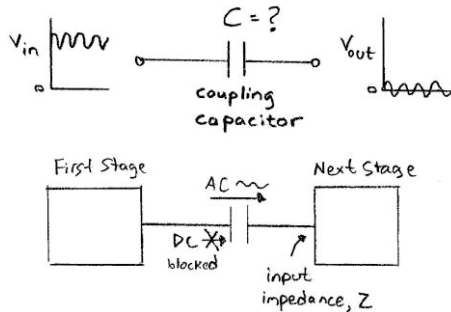


Figure 8-9. A coupling capacitor transmits AC and blocks DC.

Calculating the capacitance value of this capacitor is quite complicated,

requiring knowledge of the **input impedance** or load resistance of the next stage it will be connected to, and will depend on the frequency of the signal you would like to pass through it.

An over-simplification of this calculation is to obtain the input impedance to the next stage from the component's datasheet, or measure it (see *Measuring Input Impedance*, later in this section). Then, decide on the lowest frequency you would like to pass through to the next stage, and aim for a decade lower than that to make sure your signal will still get through. For instance, if you are dealing with an audio signal, the frequency range of human hearing is approximately 20 Hz – 20 kHz, varying with individual, age, and number of rock concerts attended. (Nagel et al. 2016) If you decide on a cutoff frequency a decade lower (2 Hz), and know that the next stage of your circuit is a guitar effects pedal with an input impedance of 10 k $\Omega$ , then you can roughly estimate the value of the capacitor required as:

$$C = \frac{1}{2\pi f_c Z} = \frac{1}{2\pi \times 2\text{Hz} \times 10,000\Omega} = 7.95 \mu\text{F} \approx 8 \mu\text{F}$$

This value should block DC voltage levels and signal frequencies under 20 Hz, while allowing higher audio frequencies to pass through. However, if this signal was going to the input stage of a non-inverting op-amp which may have an input impedance on the order of 1 M $\Omega$ , the value would change to:

$$C = \frac{1}{2\pi f_c Z} = \frac{1}{2\pi \times 20 \text{ Hz} \times 1,000,000 \Omega} = 0.0795 \mu\text{F} = 80 \text{ nF}$$

This same strategy is used with capacitor  $C_1$  in Figure 8-8, only the input impedance to the op-amp is equal to  $R_1$ . (Poole 2009) Consequently, the cutoff frequency for this coupling capacitor is calculated by re-arranging the equation:

$$f_c = \frac{1}{2\pi Z C} = \frac{1}{2\pi R_1 C_1} = \frac{1}{2\pi 1000 \Omega \times 0.000001 \text{ F}} = 159.15 \text{ Hz}$$

In Figure 8-8, coupling capacitor  $C_1$  will filter out frequencies effectively about a decade lower than  $f_c$  ( $\sim 16$  Hz).

## Higher Order Filters

### *Band-Pass Filters*

If you combine a low-pass filter with a high-pass filter, you can build a filter that only lets a narrow band of frequencies pass through. This is called a **band-pass filter**. A simple band-pass filter combines an LPF in series with an HPF. The HPF has a lower cutoff frequency than the LPF. The math is the same, you just need to do it twice. A circuit diagram and example Bode Magnitude Plot are provided in Figure 8-10.

A band-pass filter has one pass region and two stop regions. The **resonant frequency** (also called the **centre frequency**) is calculated as the geometric mean of the cutoff frequencies of the HPF and LPF:

$$f_r = \sqrt{f_{c,HPF} \times f_{c,LPF}}$$

This is the frequency that is attenuated the *least* through the filter.

The **bandwidth** is the difference between the cutoff frequencies:

$$BW_{3dB} = f_{c,LPF} - f_{c,HPF}$$

The **Q-factor** (or **quality factor**) is another interesting parameter, relating to the “pointiness”, or selectivity of the filter at the resonant frequency:

$$Q_{BP} = \frac{f_r}{BW_{3dB}}$$

A higher Q-factor means a narrower bandwidth, and consequently a more discriminating filter.

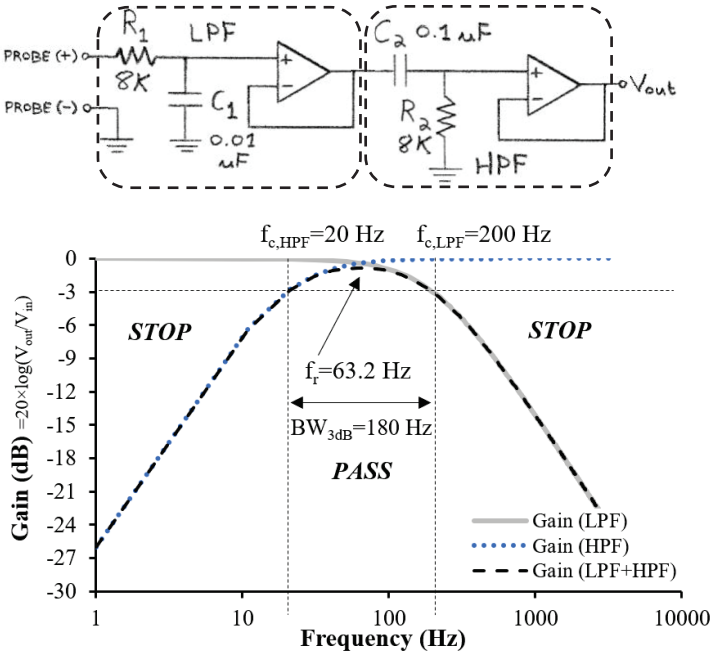


Figure 8-10. An example band-pass filter (top) and corresponding Bode Magnitude Plot (bottom).

For the example in Figure 8-10:

$$Q_{BP} = \frac{f_r}{BW_{3dB}} = \frac{\sqrt{f_{c,HPF} \times f_{c,LPF}}}{f_{c,LPF} - f_{c,HPF}} = \frac{\sqrt{20 \times 200}}{200 - 20} = \frac{63.2 \text{ Hz}}{180 \text{ Hz}} = 0.351$$

The calculated bandwidth and resonant frequency are shown in Figure 8-10.

**Test your understanding:** If you swapped the positions of the LPF and HPF in Figure 8-10, would it change the Bode Magnitude Plot?

You could use the same configuration in Figure 8-10, but with the appropriate resistor and capacitor values, plan the high-pass filter to have a higher cutoff frequency than the low-pass filter. This would result in a **stop-band filter**, the inverse of a band-pass filter. A stop-band filter attenuates frequencies *within* a narrow band, taking a “notch” out of the Bode Magnitude plot (for example, 50 or 60 Hz noise from an AC power supply).

For this reason, it is also called a **notch filter**, or **band reject filter**. However, combining an LPF and HPF to form a stop-band filter is not very efficient. A better stop-band filter design is the Twin-T notch filter (buffer optional), illustrated in Figure 8-11.

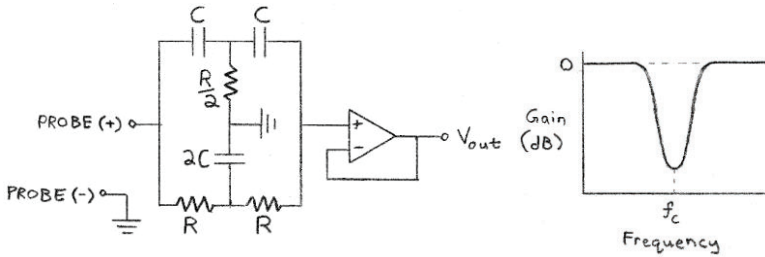


Figure 8-11. Twin-T Notch Filter (left) and Bode Magnitude Plot (right). (Carter 2006, 19-26)

Recall that two resistors with equal resistance  $R$  in parallel will result in a total resistance of  $R/2$ , and two capacitors with equal capacitance  $C$  in parallel will result in a total capacitance of  $2C$ , so you can build this circuit with four resistors of equal value, and four capacitors of equal value. Try  $R=39\text{ K}$  and  $C = 68\text{ nF}$  to filter out  $60\text{ Hz}$ , and try  $R=47\text{K}$  and  $C = 68\text{ nF}$  to filter out  $50\text{ Hz}$  (calculated by the equation  $f_c=1/(2\pi RC)$ ). These are common jobs for notch filters – to get rid of noise produced by mains electricity.

### Second Order Low-Pass and High-Pass Filters

Provided they have the same cutoff frequencies, two LPFs or HPFs in series create a second-order filter, which has twice the attenuating slope ( $40\text{ dB/decade}$ ). A second-order filter can be passive or active. For example, two passive HPFs together create a **second-order passive HPF** (Figure 8-12). Similarly, two active LPFs together can create a second-order active low-pass filter (Figure 8-13). (Karki 2002, 24.)

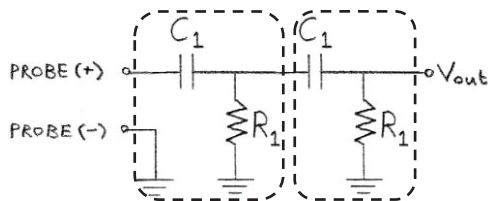


Figure 8-12. Passive second-order HPF.

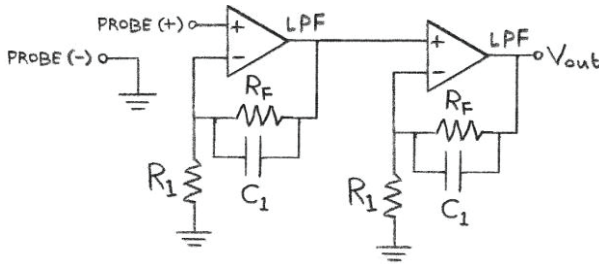


Figure 8-13. Non-inverting, amplifying second-order LPF.

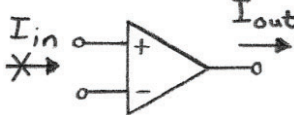
There are many other and more effective types of filters, but this is enough to get you started in the world of signal filtering.

**Test your understanding:** Could you draw a second-order, passive LPF?

## Operational Amplifiers: Practical Considerations

An *ideal op-amp* (Figure 8-14) is said to have infinite input impedance, infinite gain, and zero output impedance, as explained in Figure 8-14.

*Input impedance =  $\infty$*   
*means that essentially*  
*no current flows into*  
*the op-amp (infinite*  
*resistance,  $I_{in}=0$ )*



*Output impedance = 0*  
*means that no matter*  
*how much current is*  
*drawn ( $I_{out}$ ), the output*  
*voltage won't change*

Figure 8-14. Properties of an ideal op-amp.

So far, we have assumed the above is true, but in the physical world it isn't. Op-amps aren't ideal. There *is* a little bit of current that flows into the op-amp. There *are* limits to how much current you can draw from an op-amp output without the voltage dropping, and how much you can gain up a signal. This can have consequences to your circuit. Some of these details are provided in the datasheet of the op-amp. We need to consider these ideas in the context of our circuit.



### Impedance Considerations: Op-Amp Inputs

Let's deal with the *inputs* first. If the input impedance is not ideal, then a difference in signal impedances between inputs can unexpectedly offset/bias your output. This is bad! Since there is a bit of current going into one input, it's a good practice to try and match it as much as possible to the current going into the other input, with a comparable or appropriate value resistor called a *compensating resistor*. Figure 8-15 shows how you can balance the inputs on an inverting amplifier:

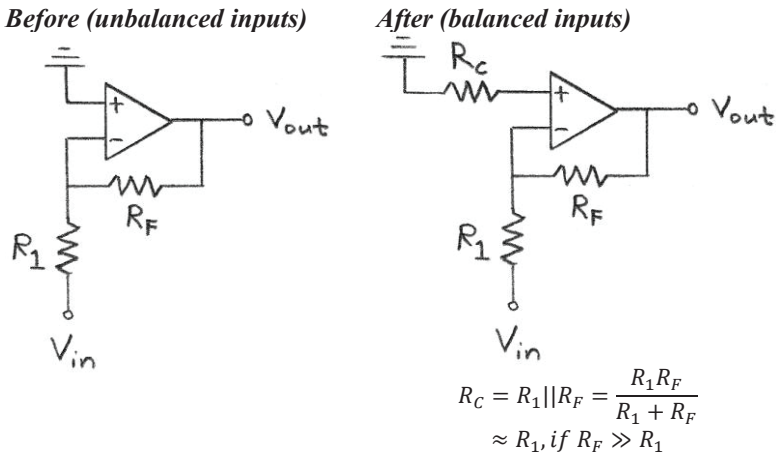


Figure 8-15. Balancing the inputs of an inverting op-amp with a compensating resistor.

The same value for  $R_c (=R_1 \parallel R_F)$  is calculated for a non-inverting amplifier (and is placed at the same spot). If the gain is high,  $R_1 \parallel R_F \approx R_1$ , as a reasonable approximation, people will use  $R_c = R_1$ .

### Impedance Considerations: Op-Amp Output

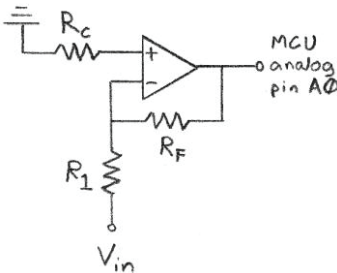
Just like the 10% rule for voltage dividers, we need to make sure our current requirements downstream of the op-amp are reasonable for what the op-amp produces. This concept is called *impedance matching*. The datasheet for the op-amp will list the maximum current it can provide (the short-circuit current, or  $I_{SC}$ ). The output impedance is a bit more complicated to think about, because it depends somewhat on the frequency of your signal *and* the gain of your op-amp. Output impedance for an op-

amp is typically somewhere between 10-500  $\Omega$ , depending on the op-amp (usually  $<50 \Omega$ ).

If you are transferring *data* (e.g. a digital signal) you want to lower your output current (i.e. increase output impedance) by putting a series resistor on the output considering the input impedance of the intended destination (where  $V_{out}$  is connected). If you don't consider this, sometimes you can damage the component receiving the data. Generally speaking, you can find out the input impedance of the next step, OR the current requirements. This information amounts to basically the same idea—providing the right amount of current from the op-amp.

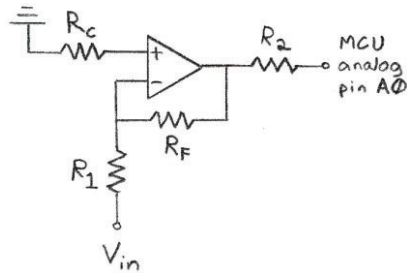
Let's say your next step after the op-amp is the MCU's analog pin A0, which has an input impedance of about 11M. You look up the output impedance for your op-amp, finding that your particular model has an output impedance of 200 $\Omega$ . As a reasonable estimate, take the geometric mean of the op-amp output impedance and input impedance of the Arduino Uno analog pin. This example is worked through in Figure 8-16.

**Before:**



Op-amp output impedance: 200  $\Omega$   
MCU input impedance:  $\sim 11M$

**After:**



$$R_2 = \sqrt{0.2K \times 11,000K} = 46.9K$$

$\rightarrow$  use a 47K resistor (closest fixed value, see Table A-5).

Figure 8-16. Matching the output impedance of an op-amp with the input impedance of the next stage (in this case, the MCU's analog pin).

If you are transferring *power* to drive a load, then make sure  $I_{sc} > I_{load}$ , or use another strategy (transistor, MOSFET, relay, or power op-amp). The **maximum power theorem** states that in order to get the maximum power transferred from a power source, then the resistance of the load should be *equal to* the Thévenin impedance of the network supplying power. With a battery, the Thévenin impedance is the internal resistance of the battery.

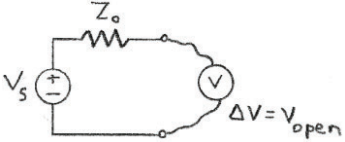
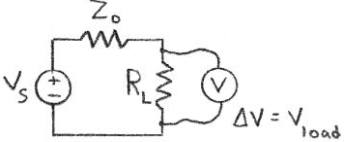
You probably haven't thought about batteries as having internal resistance, but they do. The internal resistance of a battery starts out low (usually less than 1Ω) and increases as it drains.

For example, let's say a 9V alkaline battery has an internal resistance of 2Ω. The maximum power theorem states that the load should also be 2Ω for maximum power transfer to take place. The load can certainly have a higher resistance (and the battery will last much longer), but you won't be getting maximum power transfer from your battery.

### Measuring Output Impedance

The output impedance of a circuit is equivalent to its Thévenin resistance. If you can't find a published value for the impedance of your circuit, you can also try measuring it in two steps, using Thévenin's Theorem. Table 8-3 describes a convenient method for a general circuit.

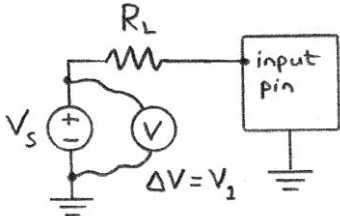
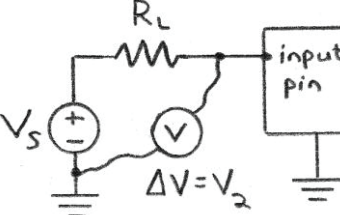
**Table 8-3. Measuring the output impedance of a signal. (Andy Collinson 2018)**

|                                                                                                                                                                                                                                                                                                                                                               |                                                                                                                                                                                                                                                                                                                                                     |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><i>Step 1:</i> Measure the open-circuit voltage with a voltmeter or oscilloscope (peak to peak for an oscillating signal, or highest voltage for a DC signal).</p>  <p>This is the Thévenin Equivalent circuit of your signal, from the point of view of the output.</p> | <p><i>Step 2:</i> Put a load resistor at the output (e.g. 1K) and measure the voltage drop across the load resistor with the same signal as Step 1.</p>  <p><i>Step 3:</i> Calculate output impedance:</p> $Z = R_L \left( \frac{V_{open}}{V_{load}} - 1 \right)$ |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### Measuring Input Impedance

If your input impedance is not published or available, it's also something you can try to measure. You can put a fixed resistor before the input pin, then measure the voltage across it. From that voltage drop, you can calculate the input impedance. Table 8-4 describes this method.


**Table 8-4. Measuring the input impedance of the input pin of a device (e.g. a microprocessor). (Andy Collinson 2018)**



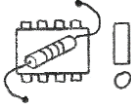
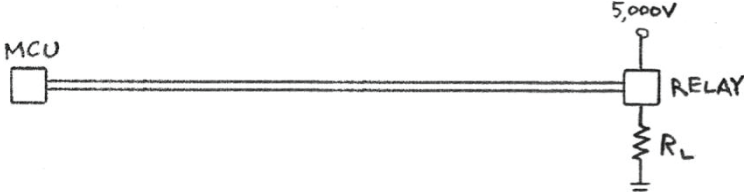
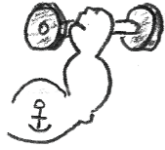
|                                                                                                                                                                                                                                                                      |                                                                                   |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|
| <p><i>Step 1:</i> <math>V_s</math> can be a generated AC signal, or a fixed DC reference signal. Connect the signal to the input pin through a load resistor (e.g. <math>1K\pm 1\%</math>) then measure the voltage before the load resistor (<math>V_1</math>).</p> |  |
| <p><i>Step 2:</i> Now measure the voltage after the load resistor (<math>V_2</math>).</p>                                                                                                                                                                            |  |
| <p><i>Step 3:</i> Calculate the input impedance: <math>Z = \frac{R_L V_1}{V_1 - V_2}</math></p>                                                                                                                                                                      |                                                                                   |

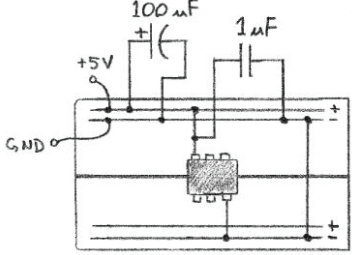
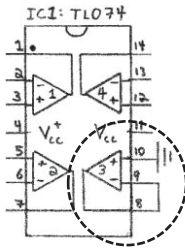
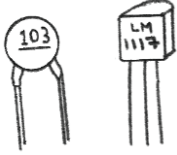

### Practical Strategies to Reduce Signal Noise

Since this is a practical course, it is important to outline some practical strategies you can use to reduce a lot of unwanted noise in your circuits. These strategies work well, without having to do any calculations whatsoever. They can perhaps save you the trouble of over-designing a circuit, and also the headache of trying to remove noise that you can prevent from ever entering your system in the first place. In no particular order, these strategies are organized in Table 8-5.

**Table 8-5. Ten practical tips for noise reduction and prevention.**

|                                                                                                                                                                                                                                                                              |                                                                                     |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
| <p>1. Make sure your circuit and all lines/wires coming out of it are away from other electrical equipment (mobile phones, balances, hot plates, pH meters, etc.) especially equipment that is plugged in. Electronics produce electromagnetic waves, which cause noise.</p> |  |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                                                                                     |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
| <p>2. Make sure your circuit has a good solid connection to ground. This applies especially to breadboards. A loose ground wire is trouble. If the only connection to your power source ground is a loose, sloppy fit, then the circuit will not be very well-behaved or stable.</p>                                                                                                                                                                                                                                                               |    |
| <p>3. Breadboard connections are just plain awful. They are meant to be temporary. Make sure all breadboard connections are snug, with leads pushed properly into holes. A soldered circuit (on a prototype board) will have less noise, and fewer problems with faulty connections.</p>                                                                                                                                                                                                                                                           |    |
| <p>4. Avoid running any wires (especially bare/uninsulated wires) over the top of an IC chip.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                  |    |
| <p>5. Position relays, solenoids, and other high-power switching devices physically as far away from your logic as practical. Switching power generates noise. Independently power these devices (using a separate supply) for better performance and less switching noise.</p>                                                                                                                                                                                   |                                                                                     |
| <p>6. Make sure your power supply is strong enough for your circuit (for both voltage and current). Most components aren't happy when they don't get enough current. Even the Arduino Uno powered through your USB port is running too lean—it will work, but not as well as it was designed to. The Uno works best with an external 9-12V power supply plugged into the DC jack, with at least 250 mA. Consider the power supply itself as a potential for noise. Older power supplies will often yield noisy results—try switching supplies.</p> |  |

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | <p>7. Every DC circuit can benefit from a 100 <math>\mu\text{F}</math> capacitor across the positive and negative rails, and a 0.1 <math>\mu\text{F}</math> or 1 <math>\mu\text{F}</math> capacitor across the <math>V_{cc}</math> to ground for <i>each</i> IC, as close to the chip's <math>V_{cc}</math> pin as physically possible. In this context, the capacitors are called <i>bypass capacitors</i> or <i>decoupling capacitors</i>. (Ross 1997)</p> |
| <p>8. Tie any unused <i>inputs</i> on an IC to ground. Sometimes the datasheet will suggest how to deal with unused pins. Generally, pins left floating can cause noise in the circuit. For unused op-amps in multi-amp chips, tie the non-inverting input to ground and have the buffer follow it. In general, leave unused <i>outputs</i> on chips floating (disconnected).</p>                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                             |
| <p>9. Consider using a separate voltage regulator to supply a reference voltage for resistive sensors. This can be the 3.3V pin on the Uno which makes use of an on-board regulator, or one that you incorporate separately (e.g. a low dropout 3.3V voltage regulator, like the LM1117-3.3). As a last resort, try connecting a 0.01 <math>\mu\text{F}</math> capacitor between a noisy probe signal and ground. This will dampen some of the noise.</p>                                                                                                 |                                                                                                                                                                                                                                                                                                                                                                             |
| <p>10. Electronic circuits pick up less noise and radio waves when they are mounted inside a case—particularly a metal one. You can try lining plastic cases with tinfoil or aluminium tape (a noise-dampening technique that likely inspired the foil hat). A bare circuit board directly open to the air can act as an antenna for noise. For the activities in this course, we haven't paid any attention to designing or using a case. However, shielding your circuit is something to keep in mind if you are working on an independent project.</p> |                                                                                                                                                                                                                                                                                                                                                                           |

Software approaches (e.g. data smoothing) are also easy and quick to implement and can be considered after carefully implementing the noise reduction strategies above. We will discuss data smoothing in more detail, later in this section.

## Measuring Noise

Noise can also be expressed in decibels, in relation to the signal. A very popular parameter you may have heard of is the **signal-to-noise ratio** (SNR). The SNR expresses the ratio of amplitudes of signal and noise, so it should hopefully be a large number. That way, your signal amplitude is much higher than the noise amplitude, and easier to discern from random fluctuations.

$$SNR = \text{Signal to Noise Ratio} = \left( \frac{A_{\text{signal}}}{A_{\text{noise}}} \right)^2$$

$$SNR_{dB} = 20 \times \log \left( \frac{A_{\text{signal}}}{A_{\text{noise}}} \right)$$

**Example:** Your car stereo advertises a signal-to-noise ratio of 100 dB. What does this mean?

**Answer:**

$$100 = 20 \times \log \left( \frac{A_{\text{signal}}}{A_{\text{noise}}} \right)$$

$$\frac{A_{\text{signal}}}{A_{\text{noise}}} = 10^{100/20} = 10^5 = 100,000$$

This means that the signal amplitude is 100,000X the noise amplitude – impressively much larger, translating to a higher audio signal quality.

One thought that may occur to you if you have a noisy system is to amplify your signal with an op-amp. However, an op-amp amplifies noise as well, using the same gain, so that may not necessarily help.

Using an op-amp *can* help increase the resolution of your reading. For instance, if your sensor range is 0 to 0.3V, the Uno analog pin (having +5V/1024 steps, or divs = 4.9 mV/div) will only be able to return 0.3V/4.9 mV/div = 61.22 divs, so you will only get 61 different numbers through the working range of the sensor—that’s not even 2 complete significant figures of precision. However, if you amplify the signal to 3.3V, you can get 3.3V/4.9 mV/div = 673.5 divs, a much more precise measurement (with continuous readings to 2 significant digits). Using an external AREF of 3.3V will increase the number of divisions in the sensor’s working range to 1024, enough for 3 significant digits. The AREF pin can also take a lower voltage, so if you know your sensor range is within 0-1V, you can try using an AREF of 1.0V, and get +1V/1024 divs = 0.977 mV/div.

If you find the analog pins constraining, you can invest in an external ADC module like the ADS1115, a tiny, inexpensive module which has 12-bit resolution in 4 channels (4096 steps). Using an external ADC isolates

your readings from everything else going on in your microcontroller and circuit.

## Data Smoothing

Even when the noise reduction strategies above are thoughtfully followed, all measurements will still have some degree of noise, which can be reduced mathematically. This is called *data smoothing*. There are many fancy mathematical filters out there. I'm going to discuss four simple and effective ones in this section that are easy enough to understand and use.

Smoothing can happen in your data acquisition sketch before it even reports a number, or you can smooth the data later, long after it has been recorded. Keep in mind though that it is better to reduce as much noise as possible in your electronic circuit first, *before* you resort to smoothing the data.

### *Mean Filter*

The most common way to smooth your data is by using a *mean filter* (or average filter), although it isn't necessarily the best choice. An average is *most* sensitive to sensor spikes. Mean filtering works by taking many readings (e.g. 100) instead of taking just one reading, and then reporting the average of all those readings. After all, it only takes an Arduino Uno ~120 microseconds to take a single analog reading, so 100 readings only takes about 12 milliseconds. It might also make sense to program a delay between readings to spread out your measurements, thus averaging over a larger time duration.

For example, let's say you have calibrated a probe so that you know the relationship between some parameter of interest, and voltage:

$$\text{some measurement} = \text{probeInt} + (\text{probeSlope} \times \text{voltage})$$

A quick mean filter for voltage on analog pin A1 can be implemented like this:

```
const float probeInt=0.16; //calibration intercept
const float probeSlope=2.51; // calibration slope
float total=0.0; // for divs
for(int i=0;i<100;i++){ // take 100 readings
 total=total+analogRead(A1); // acquire one reading
}
float answer=total/100.0; // calculate avg divs
answer=answer*5.0/1023.0; // convert divs->volts
answer=probeInt+probeSlope*answer; // volts->measure
```



If you plan on using a mean filter strategy by storing the total to an unsigned integer with a large number of readings, recall that an unsigned integer can hold a number from 0 to 65,535. This means that if every reading was 1023, you only have 64 readings before the unsigned integer maxes out and wraps around to zero, thus distorting the average. What then? You could switch to an unsigned long variable. But what if the numbers you are averaging are much larger than 1023? There is a clever, iterative approach for calculating the average of a series of readings where you don't have to generate a very large number in fear of overflowing the variable type. The variable type only needs to be able to store the largest measured reading. The numerical recipe is called calculating an *iterative mean*: (Hoffmann 2005)

$$avg_i = \begin{cases} x_i, & i = 0 \\ avg_{i-1} + \left( \frac{x_i - avg_{i-1}}{i + 1} \right), & i > 0 \end{cases}$$

where  $i$  is a counter starting at 0,  $x_i$  is the current reading, and  $avg_i$  is the updated average including reading  $x_i$ . The following sketch is similar to the previous mean filter sketch, but calculates the mean iteratively:

```
const float probeInt=0.16; //calibration intercept
const float probeSlope=2.51; // calibration slope
float avg=0.0; // for divs
for(int i=0;i<100;i++){ // take 100 readings
 avg+=(analogRead(A1)-avg)/(i+1); //iterative avg
}
float answer=avg*5.0/1023.0; //convert divs->volts
answer=probeInt+probeSlope*answer; //volts->measure
```

### *Median Filter*

A *median filter* reports the median of a series of measurements. If you sort a series of measurements in numerical order, the median is the *middle* value of this set of numbers. If you happen to have an even number of elements in this set, there is no exact middle, so the median is calculated as the average of the *middle two* values.

**Table 8-6. Calculating the median of a data set.**

|                                                                                                                                                                                               |                                                                                                                                                                                                      |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| e.g.: for an odd number of elements:<br>(1, 5, 2, 1000, 6, 1, 4)<br>First, sort the numbers:<br>(1, 1, 2, 4, 5, 6, 1000)<br>The middle value in this set is 4, so the median of the set is 4. | e.g.: for an even number of elements:<br>(4, 2, 8, 8, 6, 391, 7, 2)<br>First, sort the numbers:<br>(2, 2, 4, 6, 7, 8, 8, 391)<br>The median is the average of the two middle numbers $(6+7)/2=6.5$ . |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

You can see how selecting the median in both cases “filtered out” the big spike in each data set. Spikes happen (unfortunately) because of system noise, communication errors, grounding problems, or sometimes bad luck. Median filters are much better at filtering out spikes. The average of the above data sets would be MUCH higher than any of the other numbers, and not representative of the true measure.

### *Mode Filter*

A *mode filter* reports the *mode* of a series of measurements. The mode of a set of data is the most frequently occurring value. To find the mode of a set of data, first sort the elements in numerical order, then count how many of each unique measurement you have. Therefore, a data set might not even have a mode (if no values are repeated), or it can have more than one mode. The mode for the series: (1, 2, 3, 4, 5) is null (no mode reported). Let’s solve for the mode of the following two data sets:

**Table 8-7. Calculating the mode of a data set.**

|                                                                                                                                                                                                                                     |                                                                                                                                                   |    |    |    |       |       |    |    |    |    |    |    |                                                                                                                                                                                                                                    |    |    |    |    |    |      |    |    |    |    |    |    |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|----|----|----|-------|-------|----|----|----|----|----|----|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|----|----|----|----|------|----|----|----|----|----|----|
| e.g.: for the following measurements:<br>(1, 5, 2, 1000, 6, 1, 4)<br>Sort the numbers, and tally the frequencies:<br>(1, 1, 2, 4, 5, 6, 1000)                                                                                       | e.g.: for the following measurements:<br>(4, 2, 8, 8, 6, 391, 7, 2)<br>Sort the numbers, and tally the frequencies:<br>(2, 2, 4, 6, 7, 8, 8, 391) |    |    |    |       |       |    |    |    |    |    |    |                                                                                                                                                                                                                                    |    |    |    |    |    |      |    |    |    |    |    |    |
| <table border="1" style="width: 100%; text-align: center;"> <tr> <td>1:</td><td>2:</td><td>4:</td><td>5:</td><td>6:</td><td>1000:</td> </tr> <tr> <td>×2</td><td>×1</td><td>×1</td><td>×1</td><td>×1</td><td>×1</td> </tr> </table> | 1:                                                                                                                                                | 2: | 4: | 5: | 6:    | 1000: | ×2 | ×1 | ×1 | ×1 | ×1 | ×1 | <table border="1" style="width: 100%; text-align: center;"> <tr> <td>2:</td><td>4:</td><td>6:</td><td>7:</td><td>8:</td><td>391:</td> </tr> <tr> <td>×2</td><td>×1</td><td>×1</td><td>×1</td><td>×2</td><td>×1</td> </tr> </table> | 2: | 4: | 6: | 7: | 8: | 391: | ×2 | ×1 | ×1 | ×1 | ×2 | ×1 |
| 1:                                                                                                                                                                                                                                  | 2:                                                                                                                                                | 4: | 5: | 6: | 1000: |       |    |    |    |    |    |    |                                                                                                                                                                                                                                    |    |    |    |    |    |      |    |    |    |    |    |    |
| ×2                                                                                                                                                                                                                                  | ×1                                                                                                                                                | ×1 | ×1 | ×1 | ×1    |       |    |    |    |    |    |    |                                                                                                                                                                                                                                    |    |    |    |    |    |      |    |    |    |    |    |    |
| 2:                                                                                                                                                                                                                                  | 4:                                                                                                                                                | 6: | 7: | 8: | 391:  |       |    |    |    |    |    |    |                                                                                                                                                                                                                                    |    |    |    |    |    |      |    |    |    |    |    |    |
| ×2                                                                                                                                                                                                                                  | ×1                                                                                                                                                | ×1 | ×1 | ×2 | ×1    |       |    |    |    |    |    |    |                                                                                                                                                                                                                                    |    |    |    |    |    |      |    |    |    |    |    |    |
| The mode is the number(s) with the highest frequency count. In this case, the mode is 1.                                                                                                                                            | There are two modes in this data set: 2, and 8.                                                                                                   |    |    |    |       |       |    |    |    |    |    |    |                                                                                                                                                                                                                                    |    |    |    |    |    |      |    |    |    |    |    |    |

The mode filter did a good job at filtering out the spikes, but in the first case reported a really low number, and in the second case reported two modes. You would then need to make a decision about which of the two modes to select for your filter. Would you select the first one? The second one? An average of the two? Neither? What if no elements are repeated—what would you do then?

An advantage of reporting a mode is that it is always a measured data point that was an element of the original data set. A data purist might prefer mode over median or average.

### *Mean Filter with Threshold Rejection*

Sometimes when you are looking at the output from your sensor, common sense will tell you that there was a communication error. For instance, if you are monitoring a  $-30^{\circ}\text{C}$  freezer, and one of the readings is suddenly 0.00, or 1000, you can be sure in the context of the other readings around  $-30^{\circ}\text{C}$  that something went wrong. It pays to follow up and try to reduce the occurrences of these readings. Perhaps the communications baud rate is set too high, a thermocouple grounded against a metal object in the freezer, your probe connections are loose and would be better off soldered, or part of the circuit is sitting in a puddle and shorting out. However, sometimes data spikes happen regardless, and they can be rejected from your filtering routine using expected realistic limits.

Let's take the first example of data: (1, 5, 2, 1000, 6, 1, 4). How about we ignore all negative numbers, and any number greater than 50? We can change our mean filter routine to the following, making it less sensitive to larger spikes:

```
const float probeInt=0.16; // calibration intercept
const float probeSlope=2.51; // calibration slope
float reading=0.0; // to hold voltage reading
float total=0.0; // to calculate average
byte duds=0; // to hold # dud readings
for(int i=0;i<100;i++){ // take 100 readings
 reading=analogRead(A1)*5.0/1023.0; //get reading in V
 reading=probeInt+(reading*probeSlope); //convert
 // Is reading outside 0 and 50? If so, it's a dud.
 if(reading<0.0||reading>50.0){
 duds++; // add 1 to duds
 }else{
 total=total+reading; // include reading in avg
 }
}
total=total/(100-duds); // calculate final average
```

Note that we converted voltage to our unit of interest inside the *for* loop this time to test it, which is computationally more expensive. You could combine threshold rejection with any of the other filtering methods as well.

Occasionally with sensor readings, especially when dealing with external modules, you might obtain *NaN* as a result. This is short for “Not a Number”, or in other words, “sorry, that didn’t work.” NaN results can really throw off your math! You can filter out NaN values with the test `isnan()`. This function will return *true* if the value in a variable is NaN (*not a number*), and *false* if the value is a number. If we just wanted to filter out NaN values, we could re-write our above routine like this:

```
if(isnan(reading)){ // if reading is not a number
 duds++; // add 1 to duds
}else{
 total=total+reading;// otherwise include in avg
}
```

This code will reject any NaN values from our calculated average.

Table 8-8 provides a summary of our four filtering methods, with examples. You can see how each method returns different results. Which results do you “believe”?

**Table 8-8. Comparison of four data smoothing methods.**

| <i>Data Filter</i> | <i>Pros</i>                                                                                                           | <i>Cons</i>                                                                                                                                                                   | <i>Example 1</i><br>(1, 5, 2, 1000, 6, 1, 4) | <i>Example 2</i><br>(4, 2, 8, 8, 6, 391, 7, 2) |
|--------------------|-----------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------|------------------------------------------------|
| Mean Filter        | <ul style="list-style-type: none"> <li>• Easy to code</li> <li>• Fast</li> </ul>                                      | <ul style="list-style-type: none"> <li>• Sensitive to outliers &amp; spikes</li> </ul>                                                                                        | 145.57                                       | 53.5                                           |
| Median Filter      | <ul style="list-style-type: none"> <li>• More robust than average, less sensitive to outliers &amp; spikes</li> </ul> | <ul style="list-style-type: none"> <li>• More time consuming, computationally more steps</li> </ul>                                                                           | 4                                            | 6.5                                            |
| Mode Filter        | <ul style="list-style-type: none"> <li>• Truest to the data—returns real measurements</li> </ul>                      | <ul style="list-style-type: none"> <li>• Doesn’t always return a value, the mode can be on the extremity of your data set, and sometimes more than one mode exists</li> </ul> | 1                                            | 2, 8                                           |

|                                      |                                                                                  |                                                                                                                        |      |      |
|--------------------------------------|----------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------|------|------|
| Mean Filter with Threshold Rejection | <ul style="list-style-type: none"> <li>• Easy to code</li> <li>• Fast</li> </ul> | <ul style="list-style-type: none"> <li>• You really need to examine the sensor data to make it perform well</li> </ul> | 3.17 | 5.29 |
|--------------------------------------|----------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------|------|------|

## Data Logging

An important consideration in circuit design, particularly if you are designing scientific equipment, is how you are going to output and record your data. The first step of *data logging* is for your microcontroller to be able to report a date stamp and time stamp (the current date and time). This is especially important if you are monitoring a kinetic reaction.

The C++ environment keeps track of time, curiously, as the number of seconds that have elapsed since January 1<sup>st</sup>, 1970, at 12:00 am. This is called *epoch time*, a reference point created around the time that the Unix environment was “born”. (Op de Coul 2019) Epoch time is a large number, because it has been many seconds since the 1970s.

You might have noticed a small, caplet-shaped silver thing on the Arduino Uno board (Figure 8-17).

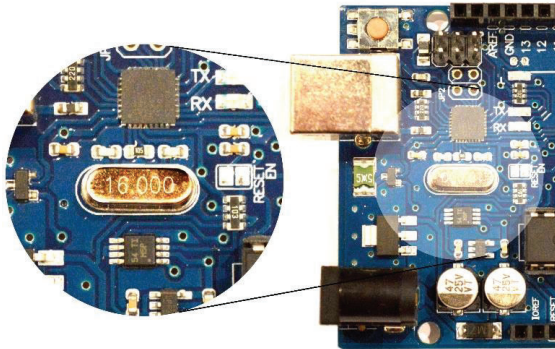


Figure 8-17. The 16 MHz crystal oscillator on the Arduino Uno, responsible for microprocessor speed.

This component is a 16 MHz piezoelectric crystal oscillator. The crystal oscillates about 16,000 times a second, and provides the microcontroller

with the pace of instructions, also called the *clock speed*. The crystal is optimized for stability and not accuracy, and so keeping track of time with the Uno has two challenges:

- 1) The processor clock isn't particularly accurate, so it may be off a few minutes per week if used as a "human" clock;
- 2) When the Arduino Uno is powered down, it will completely forget the time, and start over when powered up again.

This isn't a big problem if your experiment only lasts a day or two. However, if you are designing an Arduino clock, or a project designed to run for weeks at a time and you require an accurate time stamp, it's a good idea to hook up an *RTC* ("Real Time Clock") module (Figure 8-18). This inexpensive module houses a CR2032 lithium battery, which will power the RTC module for years. It will keep counting up epoch time even while the Arduino Uno is reset or turned off.

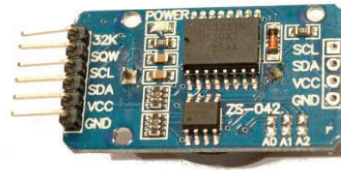


Figure 8-18. RTC module for keeping track of epoch time.

The RTC module uses a separate library, the commands of which are very similar to the *TimeLib.h* library in the next section. We will assume for this course that a minute or two per week is an acceptable error. If you would like to use one of these modules, an example sketch is provided on the course website, and libraries such as *RTCLib.h* by Adafruit are available through the Arduino IDE Library Manager.

## Arduino TimeLib.h Library

An external library written by Michael Margolis provides easy-to-use time functions for time stamping your serial output. In the Arduino IDE Library Manager, download and install the *Time.h* library by Michael Margolis. (Margolis 2014)

The following sketch illustrates how to use the Arduino *TimeLib.h* library, by setting the time, and reporting the date and time over the serial monitor. We use unsigned long variables, because *epoch time* won't be negative. There are many websites where you can find out the current epoch time. One such website is <http://www.epochconverter.com/>.

```
// Simple clock program - set the time,
// then report the time every second.
#include <TimeLib.h> // Time library (Margolis)
unsigned long t; // To store the time
```

```

unsigned long timer=0UL; // For timing log entries

void setup(){
 t=1556221236-(4*3600); // EST start time is GMT-4h
 setTime(t); // Set time with number stored in t
 Serial.begin(9600); // Start the serial monitor
 timer=millis(); // Store current time in timer
}

void loop(){
 t=now(); // Update the current time in t
 if(millis()-timer>1000){ // If 1 second has passed
 Serial.print("Date: ");
 Serial.print((String)day(t)+" /");
 Serial.print((String)month(t)+" /");
 Serial.print((String)year(t)+" ");
 Serial.print("Time: ");
 Serial.print((String)hour(t)+":");
 Serial.print((String)minute(t)+":");
 Serial.println((String)second(t));
 timer=millis(); // Reset the timer
 }
}

```

Update this sketch with the current epoch time. Compile and upload the sketch, then open the serial monitor.

- Is the time stamp correct?
- What happens when you press the Arduino Uno's reset button?

**Test your understanding:** Based on the epoch starting time in this sketch, what date and time was it written?

### *Using millis() Instead of delay()*

You will notice an important feature in this sketch: now that you can have the microprocessor keep track of the time, you can *schedule* things to happen rather than wait for them with the `delay()` function. The `delay()` function ties up the microcontroller—it can't continue to the next line in the sketch until the delay is finished. By using the `if()` statement like this:

```
if(millis()-timer>1000){ // if 1 second has passed
```

the microcontroller is free to do other things. This is *extremely* liberating, not just for logging, but for other tasks in your program. The most important command above is the `millis()` command: it works without the *TimeLib.h* library, and returns the number of milliseconds since the Arduino powered up (or since the reset button was last pressed). The *TimeLib.h* library is only

loaded to report the date and time of day. The *if* command above will work in *any* sketch. You can now replace a delay function like this:

```
delay(1000);
```

with an `if()` statement that checks how much time has passed:

```
if(millis()-timer>1000){ // if 1 second has passed
 // your code goes here
 timer=millis(); // reset the timer
}
```

Just make sure to declare the variable `timer` as an *unsigned long* variable in global space. You can program multiple timers concurrently, adding flexibility to your program. This can speed up your code immensely. Replacing `delay()` commands with timers, and turning off all serial communication (if not needed) is a really easy way of speeding up your code. Here is a worked example of a sketch that uses the `millis()` command to time an analog reading to be taken once a second.

```
//analog reading every second with millis() command
unsigned long timer=millis(); // to hold start time

void setup(){
 Serial.begin(9600);
}

void loop(){
 if(millis()-timer>1000){ // if 1 sec has passed
 Serial.println(analogRead(A0)); //take reading
 timer=millis(); // new start time
 }
 //other commands can go here without being held up
 //by a delay statement.
}
```

### *Notes about millis()*

- You don't need to include the *TimeLib.h* library to use it.
- An unsigned long variable will give you an integer up to 4,294,967,296 ( $=2^{32}$ ). This means that after 49.71 days of running, `millis()` will reset to zero. Will your program misbehave when the timer resets? A diligent programmer should keep this in mind.
- If 1 millisecond is too slow to time your events, you can also try using `micros()` to call the number of microseconds elapsed since the microprocessor turned on, and `delayMicroseconds()` to delay on the microsecond scale.



## Logging through the Serial Port

With your Arduino Uno hooked up to a PC, you can quite easily use serial communications to log sensor data. The easiest way to save data is to copy and paste it directly from the Arduino IDE serial monitor window. You can also make use of the Arduino IDE's serial monitor timestamp by ticking "show timestamp" on the control bar, meaning that time stamping happens on the PC side.

However, instead of sending data to the Arduino IDE's built-in serial monitor, you could log the serial output to a file, through the same serial port. There are free port-logging programs (e.g. RS232 Data Loggers) available online. In particular, the open-source Processing platform (available at <https://processing.org>) is an Arduino IDE-like environment that was designed specifically to handle controlling and communicating with Arduino devices once they are compiled, via the serial port. Links to port logging software in Processing and Microsoft Excel are provided on the course website.

In order to log your data effectively, you can time stamp a reading, then output the data using `Serial.println()`. For instance, a very simple program to log an analog reading to the serial port every second might look like this:

```
// Simple Port Logger
// Time since Arduino started, and analog reading

const byte sensorPin=A1; // Pin A1 for analog reading
unsigned long timer=0UL; // For timing log entries

void setup(){
}

void loop(){
 if(millis()-timer>1000){ // if 1 second has passed
 Serial.print(millis()/1000); // print #seconds
 Serial.print(","); // print comma (.CSV format)
 Serial.println(analogRead(sensorPin)); // reading
 timer=millis(); // reset the timer
 }
}
```

If you are using port logging software, the program on the PC side will receive that data, then save it to a file. The serial monitor on the Arduino IDE should be closed for this to work. Only one program can open and access a COM port at a time.

## Logging to an External microSD Card

We have already discussed .CSV format as a convenient way to output data to the serial monitor. So why bother with an SD card?

- SD cards are inexpensive, and will not significantly increase the cost of your project.
- SD cards hold a lot of data (1 GB is plenty).
- microSD cards are really tiny, and won't bulk up your project.
- SD card reader modules are inexpensive.
- A storage module means you can run your experiments without relying on a laptop or desktop computer, which might otherwise crash, power down, freeze up, or stop acquiring data because of a skype call or media program interrupting your work.

Using an SD card means that since you don't need a PC to record your data, you can set up many experiments in parallel. It's also an excellent back-up strategy for your experimental data, even if you don't plan on reading the SD card.

SD cards are also useful for storing pictures with a camera module (e.g. the OV2640 2-megapixel camera module).

### *Logic Shifters*

The SD card library is very simple to use. It works with both SD and microSD cards. In Activity 8-3, we will be setting up an SD card module, then logging data to a memory card. The microSD card modules purchased for this exercise operate on **3.3V logic**. The 5V modules are also available around the same price, but the 3.3V modules allowed me to introduce another very useful tool: a **logic shifter**. The Uno's 5V logic level would damage an SD card. A **logic shifter** converts a high voltage side (HV), in this case the 5V logic level of the Arduino Uno, down to a low voltage side (LV), in this case the 3.3V logic level of the microSD card module. The logic shifter is bi-directional: it will lower a 5V signal leaving the Arduino Uno to a 3.3V signal for the SD card module, and it will raise a 3.3V signal leaving the SD card module to a 5V signal to be read by the Arduino Uno. You can see how a logic shifter is connected in the circuit diagram in Figure 8-19.

If you didn't have access to a logic shifter, you could also reduce the 5V pin from the Arduino Uno to 3.3V by using a voltage divider (e.g.  $R_1=1K$ ,  $R_2=2K$ ). However, that will only reduce signals leaving the Arduino Uno and being received by the 3.3V device. Signals leaving the 3.3V device and going backwards up the voltage divider will not be converted to 5V through

the voltage divider. However, 3.3V is enough for a digital input on the Uno to read a HIGH signal, so if you dedicate a transmitting Arduino Uno pin (TX) to send a message to a 3.3V module, you can get away with a simple voltage divider for each TX (transmitting) pin. The RX (receiving) pins on the Arduino Uno do not require conversion from 3.3V to 5V.

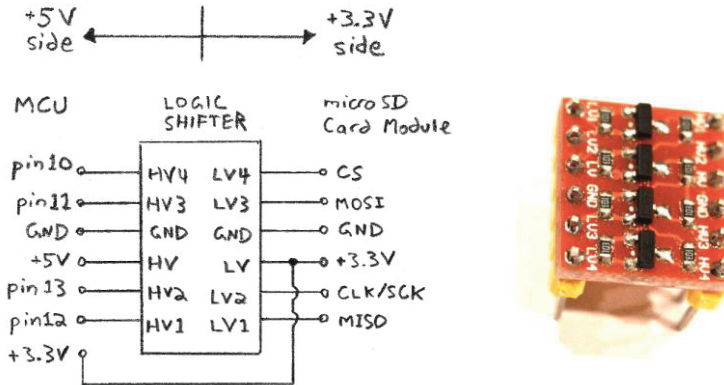


Figure 8-19. Circuit diagram (left) of a logic shifter (photo right), safely bridging a 3.3V microSD card module to the hotter 5V logic-level MCU.

One last warning about logic level: there are many online electronics tutorials showing people directly connecting 5V microprocessor boards to 3.3V modules directly, without logic shifting (e.g. an Arduino Uno to an ESP8266 wireless network card). This is generally a bad idea—the 3.3V module might work for a while, but will run much hotter than designed, and may get damaged. An inexpensive logic shifter or voltage divider can prevent this damage at a marginal cost.

## Activity 8-1: Noise Reduction

**Goal:** In Activity 8-1, we will try some basic electronic noise-reduction strategies to reduce signal noise from your Section 7 device.

### Materials:

- Stored device from Section 7
- Arduino Uno MCU & USB cable
- 1 x 100  $\mu\text{F}$  Electrolytic Capacitor
- 1 x 1  $\mu\text{F}$  Electrolytic Capacitor
- 1 x 1  $\mu\text{F}$  Electrolytic Capacitor
- 1 x 100 nF ceramic capacitor (capacitor code: 104)
- Resistor for LPF to be provided during class

### Procedure:

- 1) Retrieve your device from Section 7 (the load cell scale, or the pH meter circuit). Design and add a *low-pass filter*, to reduce high frequency noise in your signal, using a 100 nF capacitor. Is recalibration required? If so, recalibrate your device.
- 2) Use an oscilloscope to help decide a cutoff frequency, then observe noise reduction after applying your filter.
- 3) Add a 100  $\mu\text{F}$  decoupling capacitor to the power rails of your breadboard, and a 1  $\mu\text{F}$  capacitor to the  $V^+$  pin of your op-amp (see Table 8-5 for details).

## Activity 8-2: Data Smoothing

**Goal:** In Activity 8-2, we will try some data-smoothing strategies to reduce signal noise from your Section 7 device. You will be modifying your sketch to include a data smoothing filter. I have created a specific library for this exercise, called *QuickStats.h*.

### Materials:

- Stored device from Section 7
- Arduino Uno MCU & USB cable

### Procedure:

- 1) Download *QuickStats.h* from the course website (or copy it from the appendix), then place it in the same directory as the sketch for your device. Close the Arduino environment, and re-load your sketch. The library should load as a separate tab.
- 2) At the top of your device sketch in global space, add the following line, without a semicolon:

```
#include "QuickStats.h" // custom library for filtering
```

- 3) Convert the variable you store your reading into an array. For example, for the pH sketch, in global space, define a float array for your `analogRead()` values, and a float variable for the final smoothed answer:

```
float readings[100]; // array to hold analog readings
float smoothed; // to hold final filtered value
```

- 4) Now modify your reading function to fill up the array with 100 measurements. Your reading routine might look something like this:

```
for(int i=0;i<100;i++){ // take 100 readings
 readings[i]=analogRead(A2); // one measurement
 delay(10); // wait a tiny bit
}
```

The library will calculate the mean, median, or mode, depending on which one you would like to use:

```
//Uncomment the filter you would like to use:
smoothed=average(readings,100); // mean filter
//smoothed=median(readings,100); // median filter
//smoothed=mode(readings,100); // mode filter
```

This will store the average analog reading (in divs) to the float variable `smoothed`, for you to manipulate further.

- 5) Finish, compile, and run the program. Did the filter help reduce measurement noise? How can you tell if the filter is working?

You can write your own library of functions, just like *QuickStats.h*. You need only create a “.h” file, include it in the same directory as your sketch, and remember to add the `#include "whatever.h"` statement in global space of your new sketch. That way, it will be included when the sketch compiles and uploads.

**Test your understanding:** How would you program threshold rejection with any of the above data filtering strategies?

### Activity 8-3: Data Logging to an SD Card

**Goal:** To set up an SD card module as a data logger for your Section 7 device.

**Materials:**

- Stored device from Section 7
- Arduino Uno MCU & USB cable
- 3.3V 4-Channel microSD card module
- microSD card
- 1 x Breadboard
- 6 M/F jumpers
- 8 M/M jumpers

**Procedure:**

- 1) To start this exercise, format your SD card in your computer. Then, insert it into the microSD card module.
- 2) Build the circuit in Figure 8-19.
- 3) Let's make a simple sketch to test out the card module. The sketch will create a log file called "log.txt", open it, write "I was here", then close the file. This will illustrate all the commands you will need for logging a string to an SD card:

```
// Simple SD card program: open a file and write to it.
#include<SPI.h> // already installed in the Arduino IDE
#include<SD.h> // already installed in the Arduino IDE
byte CSPin=10; // pin 10 for chip select
String filename="log.txt"; //file name to create & open

void setup(){
 Serial.begin(9600); // Start the serial monitor
 if(!SD.begin(CSPin)){ // Initialize SD card
 Serial.println("Card error.");
 return;
 }
 Serial.println("Card initialized.");
}

void loop(){
 File myFile=SD.open(filename,FILE_WRITE); //open file
 if(myFile){ // if the file was opened successfully
 myFile.println("I was here."); //write msg to myFile
 myFile.close(); // close the file
 Serial.println("Data saved to SD card.");
 }else{
 Serial.println("Error writing to SD card.");
 }
}
```

```
}
```

Once the file is open, you can write to it using the commands `myFile.print()` and `myFile.println()`. These two commands are similar to `Serial.print()` and `Serial.println()`.

**Notes:**

- If the file doesn't exist on the SD card, this sketch will create it.
  - If the file already exists on the SD card, this method will open the file and append data to it, without destroying existing data.
  - Generally speaking, you should open the file *each time* you want to write to it, then close the file *each time* you are done (if this routine is inside a loop).
  - `myFile` is an object name. You can replace it with whatever name you like, and manage more than one file at a time in a sketch.
  - To give you an idea of how much data a 1 GB card can hold, I created an AC voltage logger, and ran it for a whole week, logging one reading per second (date & time stamp, + AC voltage). The resulting file was 20 MB, meaning that the SD card would run out of memory in 50 weeks, and store over 30 million log entries!
- 4) Use the code from the logging sketch in this section to add SD card data logging capabilities to your device.
- Add a timestamp (date and time) to each log line by using the *TimeLib.h* library commands (see *Arduino TimeLib.h Library* for more details).
  - Create the output in .CSV format in the log file. Each line should have the following information:  
Date, Time, Voltage, Reading
  - Run your device for a few minutes, to accumulate entries in your log file.
  - Try opening your SD card on your laptop and importing the .CSV file into Microsoft Excel.

## Learning Objectives for Section 8

After having attended this class, the student will be able to:

- 1) Visually identify circuit diagrams and Bode Magnitude Plots for first-order filters: high-pass filters, low-pass filters, band pass filters, and notch filters.
- 2) Select an appropriate RC combination to filter frequencies above or below a cutoff frequency, using  $f_c = \frac{1}{2\pi RC}$ .
- 3) Sketch a Bode Magnitude Plot for **high-pass**, **low-pass**, and **band-pass** filters, based on provided cutoff frequencies.
- 4) Calculate and interpret signal-to-noise ratio in decibels.
- 5) Convert between microprocessor clock frequency and the approximate time per clock step.
- 6) Use the *TimeLib.h* library commands to set the Arduino clock based on **epoch time**.
- 7) Generate time stamps for log entries of acquired experimental data.
- 8) Schedule timed events using `millis()`, rather than using a `delay()` to wait.
- 9) Add SD card logging functionality to any sketch.
- 10) Connect a logic shifter to communicate between modules operating at different logic levels.
- 11) Calculate the mean, median, and mode for a set of data.
- 12) Implement an appropriate data smoothing strategy, considering the smoothing method, and number of measurements.



## Section 8 - Station Content List

- Stored devices from Section 7
- 1 x microSD Card (1 GB) with microSD to SD adapter
- 1 x 4-Channel Logic Shifter Module
- 1 x microSD Card (1 GB) with microSD to SD adapter
- 1 x 100  $\mu$ F Electrolytic Capacitor
- 1 x 1  $\mu$ F Electrolytic Capacitor
- 1 x 100 nF ceramic capacitor (capacitor code: 104)
- For Activity 8-1, resistors will be provided during class.
- Digital oscilloscope (shared)
- Mini breadboard
- 15 M/M Jumpers
- 1 x Highlighter

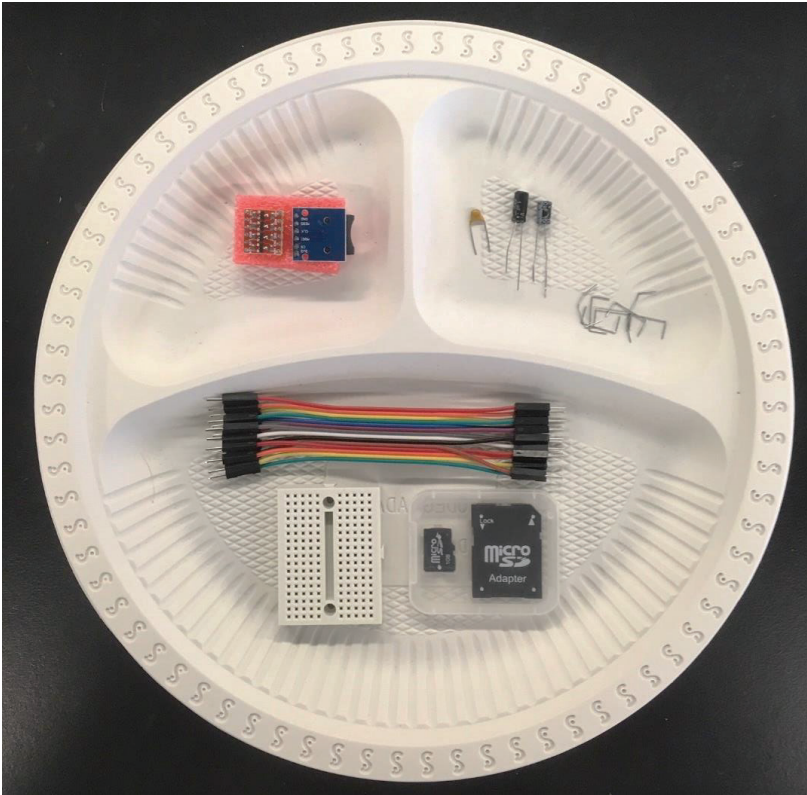


Figure 8-20. Section 8 station setup.

## SECTION 9

# DESIGN PROJECT GUIDANCE

### Introduction

This section will provide information pertaining to the design project of this course. Suggested structure of the design project evaluation is provided for the instructor, and various code snippets are provided for the student.

### Design Project Selection

A design project will help solidify concepts learned throughout this text. Design projects can be fun, intimidating, frustrating, and rewarding. Here are some independent project ideas that students have explored in this course in previous years:

**Table 9-1. Some Design Project Ideas**

|                                                                                                                                                                                                                                                                                                                                                           |                                                                                                                                                                                                                                                                                                                                                     |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Laser light scattering photometer</li><li>• Turbidity meter</li><li>• Rolling ball viscometer (hall sensor or laser break-beam)</li><li>• Tablet colour rejection circuit</li><li>• UV/Vis spectrophotometer</li><li>• Fluorimeter</li><li>• Automated microscope slide stage drive (for cell counting)</li></ul> | <ul style="list-style-type: none"><li>• Melting Point Apparatus (with video)</li><li>• Fridge/freezer monitoring system with notifications</li><li>• Flood sensor with notifications</li><li>• Tablet metal detector</li><li>• Stability chamber (heat and humidity controlled)</li><li>• Automated mouse cage door with laser break beam</li></ul> |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### *Design Project Assessment*

Although a good design never really feels “finished”, projects in this course can be evaluated using different measures. Design projects by nature have varying difficulty levels. Not all projects will resemble a completed device by the end of the design project time in class.

The following is a suggested rubric for assessment of the design projects in a small-class setting:

- 1) Design Project Proposal (10%)
- 2) In-class Project Demonstration (5-7 min/student) (5%)
- 3) Final Report (85%)

### *Design Project Proposal*

In order to provide some scaffolding for your project, and to make sure you are prepared with the resources to succeed, it will be helpful to come up with a design proposal. A proposal should provide enough detail for a launching point. You need to do some preliminary research to make sure your design is feasible, relatively safe, and that the parts you need can be easily sourced. The following elements can be present in a design proposal:

#### **1) Design Project Summary and Description (3 marks)**

- What will your device be called? Come up with a creative and unique name.
- What will your device control and/or measure?
- What pharmaceuticals principles will be explored?

#### **2) Required Equipment (2 marks)**

List specific equipment/items you think you will need. Try to be as inclusive and specific as possible. When creating your equipment list, consider:

- MCU board selection (e.g. Arduino Uno, Mega 2560, Nano, Lily, Leonardo, NodeMCU, Linknode D1, ATtiny85)
- Power supply
- Data filtering (Low-pass or high-pass filter?)
- User interface (Buttons? Matrix touch pad? Serial interface?)
- Data collection, recording, and communications (e.g. Laptop port recorder, SD card reader, internet logging (thingspeak.com), bluetooth (e.g. HC-06 4-pin serial wireless bluetooth RF transceiver module), nRF24L01, IR)
- Other equipment that will be required (e.g. breadboard, jumpers, 9V battery, connectors, etc.)

#### **3) Schedule of Planned Activities for Design Project Lab Periods (2 marks)**

In order to make full use of laboratory time, create a schedule of the specific tasks you plan on doing during each lab period. Consider:

- Design time;
- Testing Time;
- Calibrating;

- Experimenting;
- Optimizing;
- Data collection and recording;
- The number of lab periods available.

**4) Rough Equipment Schematic (2 marks)**

Do some research on your selected topic, and sketch your own rough schematic of the system you intend to create, including the components you have listed in your Required Equipment section. This will inevitably change. However, having a solid starting point will provide you with a big advantage in starting your project.

- Include any preliminary circuit drawings.

**5) References (1 mark)**

- Keep track of and cite any references you have used to help come up with your idea.
- You can build on the ideas of others. However, it is very important to properly reference any libraries, schematics, ideas, and strategies you researched from articles, datasheets, and webpages. As this is a technical report, scientific articles and datasheets are expected as references.

### *In-Class Project Demonstration*

The last half hour of the last class period is designed as an informal five-minute presentation of your design project to the rest of the class. The presentation should constitute a small portion of the design project grade. It will be a time where design ideas are shared and admired. Evaluation will be based on:

- Innovation/Creativity (1%)
- Functionality/Completion (1%)
- Presentation Skills (2%)
- Question Period (1%)

### *Final Report*

The final report brings all your design work together, presenting the project results throughout the planning, design, and testing phases. The following elements could be included in the final report:

**1) Introduction and purpose of your device (10 marks)**

Include any background, and scientific principles involved. What is the problem you are trying to solve?

**2) Summary of Design Features (15 marks)**

- How does your device work?
- Description of the main components
- Power supply strategy (e.g. USB via laptop, battery (if so, drain time), USB wall charger, 12V wall charger)
- Data filtering strategy
- Data smoothing strategy
- Data recording strategy

**3) Device Photos (5 marks)**

- A close-up (detail) photo of your circuit, and a wide-view photo of your project in action. Include an appropriate figure legend.

**4) Detailed Circuit Diagram (10 marks)**

Include a title or figure legend. Follow the conventions discussed in this text (see Figure 1-35 for more details). This diagram should be sufficiently detailed to allow someone to re-create your project without having to ask you any clarifying questions (e.g. include all resistor/capacitor values, include power rails on op-amps, label all ICs and components).

**5) Testing protocol (10 marks)**

This section requires planning, while you are developing your device. What tests did you *plan* to run? What constitutes a pass or a fail? This should be a separate section, similar to the methods section of a scientific paper.

- Even if your output is not quantitative (e.g. motor spins or not) you can collect metrics on how many times the circuit functions as expected, e.g.: Test operation of servo,  $n=10$ . Document how many times servo functioned as expected (report  $n$  passed /  $n$  total).

If your output is quantitative (e.g. a measurement) go beyond capturing and reporting the raw data only. Consider capturing:

- A calibration curve, if you are collecting sensor data
- Mean  $\pm$  %RSD of a standard (or typical) measurement
- A screenshot of the data collected

**6) Experimental Results and Analysis (15 marks)**

- Recorded results from your testing protocol
- Power consumption: Include the total measured current draw when your device is on, and the total power your device

consumes (including the microprocessor). You may combine measured and theoretical values (with proper references).

- Diagrams, tables, figures, and example calculations as required.

#### **7) Discussion of Errors and Project Refinements (5 marks)**

#### **8) Thévenin Equivalent Circuit (3 marks)**

- Pick one component of interest in your circuit, and calculate the Thévenin Equivalent circuit from the perspective of that component. Show your work.
- If your simplest circuit is a Thévenin's equivalent circuit (e.g. a DC motor powered by a digital pin), measure and report the output impedance of the digital pin in OUTPUT mode. This is the Thévenin resistance from the point of view of the load. Alternately, you can measure the input impedance of a digital pin in INPUT mode, OR the input impedance of an analog pin. Show your calculations. How does your measurement compare to a literature value for the MCU?

#### **9) References (2 marks)**

- At least five references, used in introduction section, project design, and analysis. Include references to libraries, datasheets used, scientific articles, and credit for other people's code if used.

#### **10) Appendices (10 marks)**

- Your sketch: you are expected to write the majority of your own sketch, and provide proper references of code that you used from libraries or other sources. See *General Programming Etiquette*, and *Programming Checklist* in Section 4 for more guidelines regarding your final sketch.
- A copy of your original design proposal.
- Any calculations required to design the circuit (e.g. gain, cutoff frequency, battery life, etc.)

### ***What if my design project doesn't work?***

If you are asking yourself this question, now is a good time to read the preface of this manual, and the troubleshooting guide (and flowchart) in the appendix. You will not likely finish your prototype; however, provided your idea was well considered, researched, tested, and results recorded, an incomplete device should not necessarily result in a poor evaluation. A project that isn't finished should not be confused with failure – but rather deemed “in development”. Design is all about things not working the way

you had imagined, which can be frustrating, but also very rewarding. There is great opportunity here for creativity.

It *is* expected that after the many hours of in-class designing, building, and testing, that at least part of your circuit works. If you are having extreme difficulties with your project, changing or limiting the focus may also be considered.

Another note is that you do not necessarily need to build something overly complicated. Sometimes keeping your project simple is a much better idea. The point of this project is for you to develop and test your own ideas, and incorporate key concepts discussed in this course.

## Code Snippets and Examples

Although it's difficult to forecast the sorts of routines you might need for your design project, the following example sketches will help you with some common tasks, such as programming menus, buttons, and beeps. This section will expand what you already know about microprocessing.

### *Serial Monitor Menu*

If you are planning to use the serial monitor for your project as a means of communicating with the user, you can set up a simple menu system to allow the user to select between modes and functions of your program. The following very simple sketch provides a basic routine to print a menu for the user, then have them enter a choice as a char variable. An option is also provided to enter an integer. Have a look at *Parsing Serial Data* in Section 5 for other ways of reading data from the serial monitor.

```
// Simple menu program
// Set serial monitor to "No line ending"
char choice='\0'; // initialize choice with NULL

void setup(){
 Serial.begin(9600);
 printMenu();
}

void loop(){
 if(Serial.available()){
 choice=Serial.read();
 } else if(choice=='r'){ // if user enters 'r'
 Serial.println("\nReading sample:");
 Serial.println(readSample());
 choice='\0'; //erase choice
```

```

 printMenu(); //reprint user menu
 } else if(choice=='c'){ // if user enters 'c'
 Serial.println(readSample()); //continuous reads
 } else if(choice=='s'){ // if user enters 's'
 Serial.println("\nEnter servo value (0-255):");
 while(!Serial.available()); // wait for input
 int s=Serial.parseInt();
 s=constrain(s,0,255); // constrain to limits?
 Serial.println("Setting to "+(String)s);
 // commands for setting servo value go here
 choice='\0'; //erase choice
 printMenu(); //reprint user menu
 } else if(choice!='\0'){
 Serial.println("\nInvalid option.");
 choice='\0'; //erase choice
 printMenu(); //reprint user menu
 }
}

void printMenu(){ // user menu
 Serial.println("r: Read one sample");
 Serial.println("c: Continuous reads");
 Serial.println("s: Set servo value");
 Serial.println("Enter choice: ");
}

int readSample(){ // your sample reading routine
 return analogRead(A0);
}

```

The following sketch will read a single char variable from the serial monitor (if available) and compare it to a list of possible responses using **switch case**:

```

// Menu sketch using char variable input from serial
// monitor (switch case)

void setup(){
 Serial.begin(9600);
}

void loop(){
 if(Serial.available()){
 char choice=Serial.read();
 switch(choice){
 case 'A':
 Serial.println("You selected A.");
 //more commands can go here

```



```

 break;
 case 'b': // char is case sensitive
 Serial.println("You selected b.");
 break;
 case 'c':
 Serial.println("You selected c.");
 break;
 case 10: // ignore the new line character
 break;
 default:
 Serial.println("Unknown input.");
 break;
} // end of switch
} // end if
}

```

An example of a serial monitor menu using strings is provided in *Comparing Strings*, in Section 10.

### *Using EEPROM: Memory that Doesn't Forget!*

Any data acquired while a microprocessor is on will be erased from memory after a power-down or reset. However, there is some more permanent memory inside the ATmega328 chip called **EEPROM**. EEPROM stands for Electrically Erasable Programmable Read-Only Memory. It's meant for long term storage, and will preserve the data even after the microprocessor is powered down. The ATmega328 has 1 kb of EEPROM. Writing to it and reading from it is a little more involved. Here is a sketch that writes one byte to EEPROM and then reads it back:

```

// Example: Writing one byte to EEPROM, then
// reading it back from EEPROM
#include <EEPROM.h>
byte myNumber=240;

void setup(){
 Serial.begin(9600);
 //0 is EEPROM byte number (range is 0-1023)
 EEPROM.write(0,myNumber);
 byte temp = EEPROM.read(0);
 Serial.print("EEPROM byte 0 contains: ");
 Serial.print(temp);
}

void loop(){}

```

Once this sketch is uploaded, the value 240 will remain at byte 0 in the EEPROM until it is overwritten. If you would like to store more than a single byte (for example, a slope and intercept), you can use the commands `EEPROM.put()` and `EEPROM.get()` with structures. (Mellai 2018) Structures are variable types organized in a group. See *Structures* in Section 10 for a more detailed discussion on how they are defined and used. The following sketch will write the structure called `calibration` to EEPROM memory:

```
// Example: Writing a struct to EEPROM
#include <EEPROM.h>
struct calibration{ // declare struct in global space
 float intercept; // intercept and slope are members
 float slope; // of the struct calibration
}; // Note: semicolon required here

calibration newRun; // use calibration structure for
 // newRun (to write to EEPROM)

void setup(){
 // this can go in the setup or loop function:
 newRun.intercept=0.0217; // intercept from newRun
 newRun.slope=1.4243; // slope from newRun
 EEPROM.put(0, newRun); // write newRun data @addr=0
}

void loop(){}

```

The following sketch will then read the data from EEPROM memory (for example, in the `setup()` function when the board first powers up):

```
// Example: Reading a struct from EEPROM
#include <EEPROM.h>
struct calibration { // declare struct in global space
 float intercept; // intercept and slope are members
 float slope; // of the struct calibration
}; // Note: semicolon required here

calibration oldRun; // use calibration structure for
 // oldRun (to write to EEPROM)

void setup(){
 Serial.begin(9600);
 EEPROM.get(0, oldRun); // read oldRun data @addr=0
 Serial.println(oldRun.intercept,4);
 Serial.println(oldRun.slope,4);
}

void loop(){}

```

If you put these two ideas together, we can write a sketch that loads old calibration curve data on startup, then allows for overwriting the stored data with an updated slope and intercept:

```
// Example: EEPROM to load and update calibration data
#include <EEPROM.h>
byte buttonPin=2; // for a button

struct calibration{ // declare struct in global space
 float intercept; // intercept and slope are members
 float slope; // of the struct calibration
}oldRun,newRun;

void setup(){
 Serial.begin(9600);
 pinMode(buttonPin,INPUT_PULLUP);
 Serial.println("Loading stored calibration data.");
 EEPROM.get(0, oldRun); // read oldRun at addr=0
 Serial.println("Loading stored calibration data.");
 Serial.println(oldRun.intercept,4);
 Serial.println(oldRun.slope,4);
}

void loop(){
 if(digitalRead(buttonPin)==LOW){ // if button pushed
 Serial.println("Enter new intercept: ");
 while(!Serial.available()){} //wait for data
 newRun.intercept=Serial.parseFloat();
 Serial.println("Enter new slope: ");
 while(!Serial.available()){} //wait for data
 newRun.slope=Serial.parseFloat();
 Serial.println("Saving calibration data.");
 EEPROM.put(0, newRun); // write newRun at addr=0
 oldRun.intercept=newRun.intercept; // use new data
 oldRun.slope=newRun.slope; // use new data
 }else{
 float reading=analogRead(A0);
 reading=oldRun.intercept+reading*oldRun.slope;
 Serial.println(reading);
 delay(500);
 }
}
```

*For more information: <https://www.arduino.cc/en/Reference/EEPROM>*

## Generating Beeps to Alert your User: Arduino Tone Library

A piezoelectric crystal changes its shape very quickly when a voltage is applied across it – quickly enough to generate sound. A tone library is available through the Arduino IDE that generates a signal on any digital pin, to create shrill, attention-getting tones through a piezoelectric buzzer.

An interesting aspect of piezo electric elements is that they also work in reverse. If pressure is applied directly to the element, a tiny voltage is created, and so piezoelectric crystals can also act as microphones, and pressure sensors.

The following sketch will generate one-second 440 Hz beeps on a piezoelectric buzzer (black item in Figure 9-1). For this sketch, the piezo buzzer was inserted in pin 4 (for ground) and pin 7 (for signal), as that pin spacing allows a generic piezo buzzer to be plugged directly into the Uno female pin header (see Figure 9-2).

```
// Example: Tone Function
byte piezoPin=7;
void setup(){
 //set up Pin 4 as GND:
 pinMode(4,OUTPUT);
 digitalWrite(4,LOW);
 pinMode(piezoPin,OUTPUT);
}

void loop(){
 //440 Hz tone for 1 sec:
 tone(piezoPin,440,1000);
 delay(1000);
}
```

You can experiment with different frequencies to find the loudest tone for your specific piezo. Just be mindful of your user, or there might be unintended consequences. A freezer monitoring system I developed was unplugged over one weekend, because people didn't appreciate the periodic beeping.

*For more information:*

*<https://www.arduino.cc/reference/en/language/functions/advanced-io/tone/>*

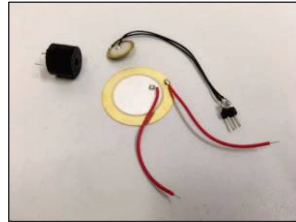


Figure 9-1. Piezoelectric elements.

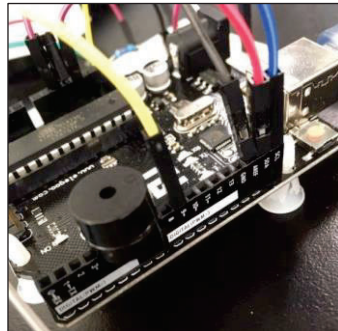


Figure 9-2. Piezoelectric buzzer plugged directly into the Arduino Uno.

### *Programming One Button with Multiple Functions*

Adding a button to a project isn't difficult, especially with the INPUT\_PULLUP option for pinMode. However, adding a second button for an infrequently used function can be circumvented by having the user press the first button in a different way. For instance, holding a button down during startup can be used to select between two modes of a device, or holding a button down during normal operation for a given duration can be used to trigger a different routine than a quick push. As long as the user is provided with appropriate instructions, This can help minimize the number of components in your design.

The following sketch uses a single button to either take an analog reading (short push), or zero the reading (long push).

```
// Two-function button sketch
const byte buttonPin = 12; // set button to pin 12
const byte sensorPin = A0; // sensor on pin A0
int blankVal=0; // to store analog reading

void setup(){
 Serial.begin(9600); // start serial monitor
 pinMode(buttonPin,INPUT_PULLUP); //input pullup mode
}

void loop(){
 Serial.println("Press: Reading. Hold: Zero.");
 while(digitalRead(buttonPin)==HIGH); // wait for push
 unsigned long buttonTimer=millis(); // start timer
 delay(1); //debounce
 bool longPush=false;
 while(digitalRead(buttonPin)==LOW){
 if((millis()-buttonTimer)>2000){ //timeout is 2 sec
 longPush=true;
 break; // stop waiting (break out of while loop)
 }
 }
 if(longPush){ // if button pushed for more than 2 sec
 Serial.println("*** ZEROING ***");
 delay(100);
 blankVal=analogRead(sensorPin); // get zero val
 Serial.println("Zeroing complete.");
 } else {
 Serial.println(analogRead(sensorPin)-blankVal);
 }
 delay(100); // debounce
}
```

See *Never Miss a Button Push Again* in Section 10 for an example sketch of attaching a button push to an ***Interrupt Service Routine***, which will even work during `delay()` statements.

## Measuring Light Intensity

Depending on the sensitivity and reading speed required, there are many different devices you can use to detect light intensity. This section provides some details on how to wire up a few popular and low-cost options for light sensors, including photoresistors (also called light dependent resistors, or LDRs), photodiodes, phototransistors, light-to-frequency converters, and ICs used to detect light.

### *Photoresistors*

If you aren't in a big hurry for a measurement (for instance, if you aren't trying to transmit data using light, or keep up with a stepper motor during a wavelength scan), photoresistors are an excellent, sensitive, and reproducible way to measure light. (Thal and Samide 2001, 1510-12) As light hits the surface of a photoconductive cell, the resistance will drop from the megaohm range all the way to the ohm range. For visible-light LDRs, the photoconductive material usually used is cadmium sulphide, absorbing light wavelengths between 400-700 nm depending on their construction. (Carolyn Mathas 2012) These components are dirt cheap, and available in different sizes and sensitivities. The simplest way to wire up a photoresistor is to wire it in series with a sense resistor to form a voltage divider, just like the thermistor in Activity 4-1. A typical circuit diagram is presented in Figure 9-4.

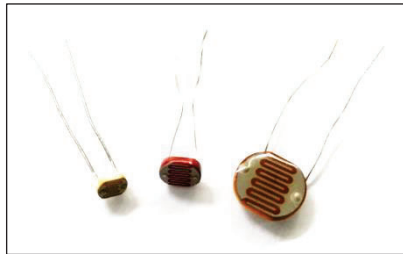
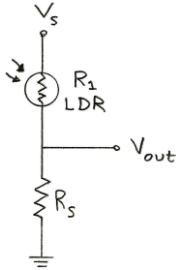


Figure 9-3. Photoresistors of various sizes (5 to 12 mm).



$$V_{out} = V_s \left( \frac{R_s}{R_1 + R_s} \right)$$

$$R_1 = R_s \left( \frac{V_s - V_{out}}{V_{out}} \right)$$

Figure 9-4. Voltage divider for a photoresistor.

A larger sense resistor ( $R_s$ ) would lead to a greater change in  $V_{out}$  as light hits the photoresistor. It may seem intimidating to look at a circuit diagram and not see a specific resistor value given, but a good value for a sense resistor really depends on the LDR selected and fiddling with the equipment (light levels, etc.) until you find an acceptable working range. Start small (e.g. 1K) and work your way up logarithmically (10K, 100K, 1M, 10M) until you find useable dark and light response values that don't saturate too early or have too small a response for the `analogRead()` function or your external ADC.

Photoresistors are used in many applications to turn on and off switches, and even provide reference signals for loads. However, if a load is added to  $V_{out}$ , it will change the balance of the voltage divider and alter the output signal. How can you isolate  $V_{out}$  from the divider? One approach is to buffer  $V_{out}$  with an op-amp. A more elegant approach is to abandon the voltage divider, and use a **transimpedance amplifier** in Figure 9-5 (left). (Scherz and Monk 2016)

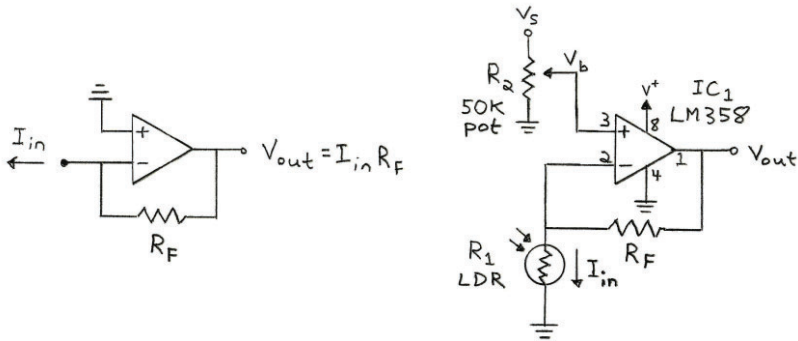


Figure 9-5. A transimpedance amplifier (left) converts current,  $I_{in}$  to voltage,  $V_{out}$  with a gain of  $R_F$ . An example (right) shows the LM358 amplifying the current through an LDR, with a bias voltage  $V_b$  to raise the signal into the output voltage swing of the amp ( $\sim 0.1\text{ V} - 3.9\text{V}$  for op-amp supplies  $V^+=+5\text{V}$  and  $V^-=\text{GND}$ ).

This op-amp configuration converts a current to a voltage, so it is also called a current-to-voltage converter. The theory is that as the photoresistor resistance value drops when light hits it, the current will increase. This current is connected to the inverting input of the op-amp, and flows through feedback resistor  $R_F$ . Consequently, the voltage drop across  $R_F = I_{LDR} \times R_F$  (linearly dependent on the current). Adding a few more details in Figure 9-5 (right), we can connect the LDR to the inverting input and tie the lower rail to ground. If the op-amp isn't rail-to-rail, we can also add a bias voltage to the non-inverting end, to boost up the signal to the working voltage output swing of the amp. This potentiometer is adjusted while the signal is close to ground (dark conditions), to bring it to the smallest non-zero output voltage. If the signal is noisy, a capacitor can be wired across  $R_F$  to form an active high-pass filter (with cutoff frequency,  $f_c = \frac{1}{2\pi R_F C}$ ). Try different  $R_F$  values until your signal swings across your measured range, testing from completely dark to light conditions, without maxing the signal out from your op-amp or ADC. A value of 1M worked reasonably well when testing this circuit with the equipment in the lab. Conveniently, the same two configurations (voltage divider, and transimpedance amplifier) can be used for many other types of light sensors.

As mentioned before, it takes some time for an LDR to equilibrate to a change in light intensity – on the order of 10-100 milliseconds. This means that if you would like to measure a change in signal that happens over a shorter time span or faster frequency, the hysteresis of the device will slow you down. I once tried to transmit a serial signal using laser light and an



LDR as light receiver, without understanding the concept of signal rise and fall time. On the receiver end, all I received was gibberish, even at the lowest baud rate.

### Photodiodes

Like a regular diode, a photodiode has a very thin layer of P-type and N-type silicon. The diode case is transparent to allow light to pass through. As light hits the N-type silicon, electrons will pass through the p-n junction, and a tiny current is *generated*, that can be measured. (Scherz and Monk 2016) Most LEDs have this property to some extent (see Figure 5-19), but *photodiodes* have been engineered to maximize this effect. The spectral range for a photodiode can be very wide (e.g. the PD638C photodiode is sensitive to light wavelengths spanning from 200-1200 nm). Black pastic casing may be used to limit the spectral range to infrared. The PD638C has a rise and fall time of 50 ns, a significant improvement over photoresistors. (Tsai and Everlight Electronics Co. 2005)



Figure 9-6. The PD638C photodiode.

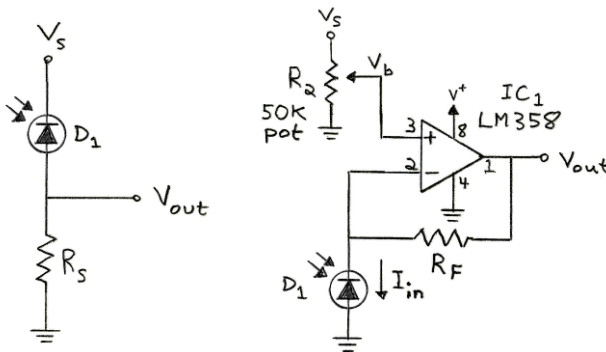


Figure 9-7. A photodiode wired in the voltage divider configuration with sense resistor (left), and as an input for a transimpedance amplifier (right).

## *Phototransistors*

A phototransistor shares some features of its BJT NPN-transistor cousin, but the base terminal is replaced with light, now acting as the transistor switch. Photons cause electrons to jump from p-type to n-type silicon, allowing electrons to flow from the emitter to the collector. The circuit diagram symbol looks more like a transistor. Although the construction is different than a visible LED, phototransistors can look extremely similar to conventional LEDs, so just like of your electronic components, make sure



Figure 9-8. The PT334-6C phototransistor (right), wrapped in electrical tape (left).

you label the storage bag well. The spectral ranges of these devices are similar to photodiodes, and the sensitivities are comparable. An example of an inexpensive phototransistor is the PT334-6C, a two-lead, 5 mm phototransistor sensitive to light wavelengths spanning 400-1100 nm, with a rise and fall time of 15  $\mu$ s under typical conditions. (Everlight Electronics Co. Ltd 2016) If you are looking to make the sensor more directional and cut out stray light from the sides, it's a good idea to wrap the phototransistor body in a small loop of black electrical tape, so that the sensor only absorbs light head-on (Figure 9-8, left).

As with the photoresistor and photodiode, the voltage divider and transimpedance amplifier can be used to wire a phototransistor for measuring light intensity (Figure 9-9).

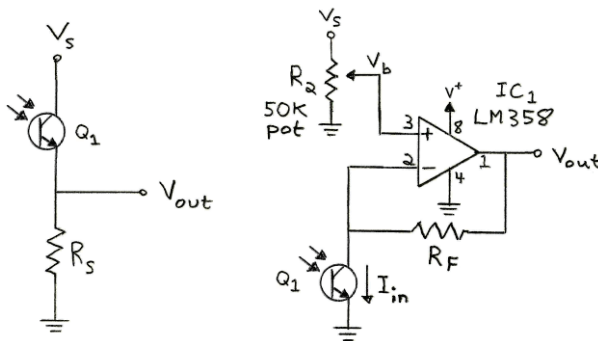


Figure 9-9. A phototransistor wired in the voltage divider configuration with sense resistor (left), and as an input for a transimpedance amplifier (right).

## *Integrated Packages*

The high demand for reliable, sensitive light sensors has brought to market some very interesting ICs that combine the above ideas in integrated packages, requiring no extra parts and less fiddling with gain values. The following are a few examples of such components.

### *Light to Frequency Converters*

The TSL235R combines a photodiode with a current-to-frequency converter on a single CMOS chip. The result is a light-to-frequency converter. This package has 3 pins (GND, Vcc, and OUT) and only requires one digital pin to be read from your microcontroller (Figure 9-10). The output signal is a square wave oscillating between Vcc and GND at a frequency linearly proportional to light intensity, ranging from the low hertz range at low light conditions to about 100 kHz in bright light. A small lens focuses light onto the photodiode.

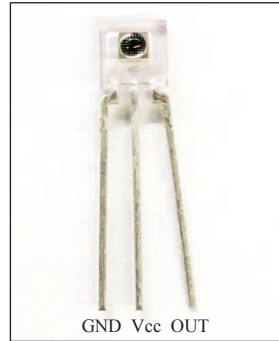


Figure 9-10. TSL235R light-to-frequency

Measuring a frequency on a digital pin is not something we have discussed so far. There are libraries available to do this, such as *FreqMeasure* by Paul Stoffregen, available through the Arduino IDE Library Manager. (Stoffregen 2015) The following sketch *TSL235R\_lightsensor.ino* works reasonably well, and is available for download on the course website. The algorithm was adapted from a sketch posted on the Arduino Playground by Rob Tillaard. (Tillaard 2011)

```

/* TSL235R Light to Frequency Converter
TSL235R - Uno:

Pin 1 - GND
Pin 2 - +5V
Pin 3 - Pin 2
*/
#define lightPin 2 // out pin of TSL235R
volatile unsigned long counter=0;

void setup(){
 Serial.begin(9600);
 pinMode(lightPin, INPUT); //use pin2 to read freq
}

```

```

void loop(){
 // read sensor with 100 msec integration time:
 unsigned long freq=readTSL235R(100);
 Serial.println(freq); // return freq in Hz
}

unsigned long readTSL235R(int t){
 //t=100 msec: can measure up to ~140 kHz
 counter=0; // reset counter
 attachInterrupt(0, countFreq, RISING); //0: pin2
 delay(t); // wait a bit to collect data
 unsigned long Hz=counter*1000/t;
 detachInterrupt(0);
 return Hz;
}

void countFreq(){ // ISR for measuring frequency
 counter++; // increase counter on rising signal
}

```

This sketch attaches an Interrupt Service Routine (ISR) to count the number of rising signals (from GND to +5V) on digital pin 2. ISRs provide a much faster and easier way of handling this task than checking a pin state through the `digitalRead()` function. For more information about how ISRs are programmed, consult *Interrupts* in Section 10.

Another way to measure the frequency of a signal using the Arduino IDE is by taking advantage of the `pulseIn()` function to measure the width of one “pulse”:

```
pulseIn(pin number, HIGH or LOW, timeout in microsec);
```

If HIGH is selected as an option, this function waits for a digital pin to go HIGH, then times how many microseconds it takes for the pin state to go LOW again. If LOW is selected as an option, the logic is reversed, and a low pulse is timed in microseconds. The timeout argument is optional, defaulting to 1 second if no value is provided. A zero is returned if the function times out before a pulse finishes. (Couto 2019) Have a look at how `pulseIn()` is called in the following sketch. A mean data smoothing filter could be invoked to help reduce measurement noise.

```

/* TSL235R Light to Frequency Converter
TSL235R - Uno:

Pin 1 - GND
Pin 2 - +5V
Pin 3 - Pin 2
*/

```

```

#define lightPin 2 // out pin of TSL235R

void setup(){
 Serial.begin(9600);
 pinMode(lightPin, INPUT);
}

void loop(){
 unsigned long freq=readFreq(lightPin);
 Serial.println(freq); // return freq in Hz
}

unsigned long readFreq(byte pin){ // freq in Hz
 unsigned long t=pulseIn(pin,HIGH);
 unsigned long Hz=1000000/(2*t);
 return Hz;
}

```

### Linear CCD

A CCD (or charge-coupled device) is essentially an array of photodiodes, oriented in a very fine matrix of pixels. Digital scanners, video cameras and still cameras make use of CCDs to capture images by focusing the image on the CCD using lenses and possibly mirrors. The TCD1304 is an example of a linear CCD that is relatively inexpensive, and useful in spectroscopic equipment. (Haffner 2017) A linear CCD is a single row of sensors which measure light intensity across one axis. Its ample 29.1 cm array offers an astounding 3648 pixels, running on a 1 MHz clock. The acquisition speed and quantity of data pose challenges to the Uno's memory and acquisition speed. (Toshiba Corporation 2001) By speeding up the `analogRead()` function and shuttling values to external SRAM memory, the Uno can acquire all 3648 pixels of data. However, a larger microprocessor (e.g. the Arduino Mega 2560) is better suited to the task. Alternately, the acquisition routine can be shortened to collect 800 pixels without exceeding the Uno's limited memory. The TCD1304's datasheet provides a recommended circuit diagram to run the chip and transmit to a microcontroller. An example sketch is available for download on the course website.



Figure 9-11. Toshiba TCD1304AP linear CCD.

A typical spectrophotometer layout is a lightsource beam, split by a monochromator into separate wavelengths. The diffracted light is reflected

or refracted through a narrow slit to select a specific wavelength. The beam travels through a cuvette in its holder, and is then detected by a light sensor. The wavelength is selected by rotating the diffraction grating with a stepper motor. The diffraction grating could perhaps be a 1000 line/mm block, a 1389 lines/mm blank DVD, or a 658 lines/mm blank CD. (Balachandran and Porter-Davis 2009) The DVD can be used as a reflective grating as is, or split into two layers and the reflective layer peeled off with tape for use as a transmission grating. A simplified schematic from the perspective of the light could be:

*light source* → *diffraction grating* → *slit* → *sample* → *detector*

However, with a linear CCD, the undiffracted light can travel through the sample, and be split *after*, directly onto the sensor. In this case, the entire spectra can be collected without the need of a stepper motor:

*light source* → *slit* → *sample* → *diffraction grating* → *detector*

This simplifies the design, as the stepper motor is no longer needed. Spectra can be acquired much faster, as the entire spectra is collected at a single time point.

Regardless of the sensor selected, wavelength can be calibrated using spectrophotometric calibration filters, or a sample with expected peaks (e.g. a holmium or didymium glass filter, or optical bandpass filters with known passband wavelength ranges). Absorbance can be calibrated and confirmed using neutral density filters. (Hellma 2008)

### **Daylight Sensors**

There are a number of light sensor modules that are calibrated to measure daylight levels, and output measured light levels as serial digital data (in lux). Daylight sensors could be used for instance to decide whether or not to switch an LCD screen to night mode, or to brighten a screen in harsh daylight for better visibility. When this course first started, I stocked up on the BHT1750 (GY-302) modules thinking that they would make good sensors for spectrophotometers. They functioned well when the incident light wasn't filtered, when the light source was pointed directly at the sensor through a cuvette. However, these sensors lacked the sensitivity required after splitting the light into separate wavelengths through a DVD disk layer or

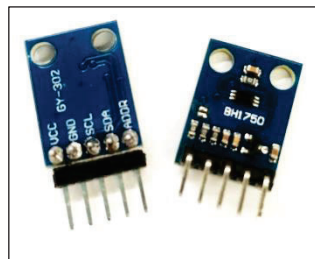


Figure 9-12. BH1750 Light Sensor module.

diffraction grating. Nonetheless, these modules are small, inexpensive, and function well at their designed tasks. Libraries are available to connect them to the Arduino Uno, e.g. *BH1750FVI* by PeterEmbedded, available through the Arduino IDE Library Manager. (PeterEmbedded 2018)

### *Laser Light Sensors*

If your design project requires a laser break-beam strategy, this small, simple laser light receiver module is calibrated to output +5V if visible laser light is detected, and GND if it is not. The response is binary, and quick. The manufacturer warns against using it on a sunny day, as the sensor can be fooled by ambient light. If the circuit is indoors, this isn't so much a problem. An on-board LED lets you know the unit is powered. An interrupt routine can be used to count duration, or perhaps a `do...while()` loop with an exit condition of the beam being broken. One student project involved using two of these sensors paired with laser diodes in a rolling-ball viscometer.

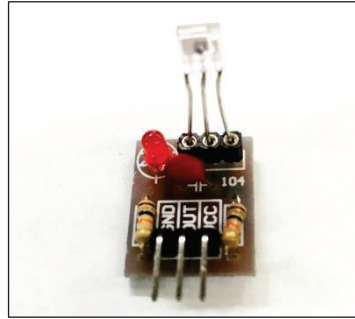


Figure 9-13. Low-cost laser light receiver.

### *Colour Sensor*

The TCS3200 module is a low-cost colour sensor. The colour sensor is an 8x8 photodiode array, with different coloured filters covering different diodes (red, green, blue, and no filter). The sensor measures and reports how much light gets through each colour filter type. This is analogous to the way our eyes detect colour, as the human eye has different cones in the retina to detect red, green, and blue light. The channels that are returned (R, G, B) are the amount of light absorbed from the sensor, and their proportions can be correlated to wavelength or perceived colour (e.g. 520 nm, yellow, pink, etc.).

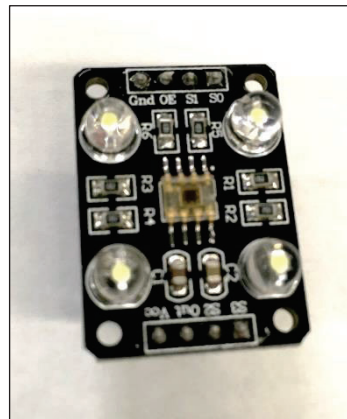


Figure 9-14. TCS3200 colour detection module.

The following sample sketch, adapted from Pamarthi Kanakaraja's forum post, provides a reading of the red, green, and blue channels of the colour sensor. Applying a mean data smoothing filter over multiple readings would help reduce signal noise. (Kanakaraja 2017)

```
/* TCS3200 color recognition sensor sketch
Color Sensor - Arduino Uno

S0 -- Pin 8
S1 -- Pin 9
OE -- GND
GND -- GND
VCC -- +5V
OUT -- Pin 10
S2 -- Pin 12
S3 -- Pin 11
*/
#define S0 8 // define pins here
#define S1 9
#define S2 12
#define S3 11
#define OUT 10
int RGB[3]={0,0,0}; // to store red, green, blue

void setup(){
 Serial.begin(9600);
 pinMode(S0,OUTPUT);
 pinMode(S1,OUTPUT);
 pinMode(S2,OUTPUT);
 pinMode(S3,OUTPUT);
 pinMode(OUT,INPUT);
 digitalWrite(S0,HIGH);
 digitalWrite(S1,HIGH);
}

void loop(){
 readColour(RGB); // read, store to RGB array
 Serial.print("R,G,B: ");
 Serial.print(RGB[0]);
 Serial.print(", ");
 Serial.print(RGB[1]);
 Serial.print(", ");
 Serial.println(RGB[2]);
 delay(500);
}
```



```

void readColour(int data[3]){
 digitalWrite(S2,LOW);
 digitalWrite(S3,LOW);
 data[0]=pulseIn(OUT,HIGH); //red
 digitalWrite(S2,HIGH);
 digitalWrite(S3,HIGH);
 data[1]=pulseIn(OUT,HIGH); //green
 digitalWrite(S2, LOW);
 data[2]=pulseIn(OUT,HIGH); //blue
}

```

## Measuring Time Duration with Interrupts

Pins 2 and 3 have a special ability on the Arduino Uno – they can also act as *external interrupts*. Interrupts can react independently from your sketch, so that very small increments in time can be measured. The following sketch could measure the time it takes a steel ball to roll between two sensors in a rolling-ball viscometer, or a perhaps a bullet to travel between two IR sensors in a ballistic chronograph. More background and information regarding interrupts and ISRs are provided in *Interrupts*, in Section 10.

```

// measuring time between two interrupt events
// variables changed in ISRs should be volatile:
volatile unsigned long time1=0UL;
volatile unsigned long time2=0UL;
unsigned long last1=0UL;
unsigned long last2=0UL;

void setup(){
 Serial.begin(9600);
 pinMode(2,INPUT_PULLUP); //this is INT0
 pinMode(3,INPUT_PULLUP); //this is INT1
 attachInterrupt(0,sensor1,FALLING); //pin 2
 attachInterrupt(1,sensor2,FALLING); //pin 3
}

void loop(){
 //a new time duration occurs when both times have
 //been updated, and time2 is later than time1.
 if(time1!=last1 && time2!=last2 && time2>time1){
 Serial.print("Timed event:");
 Serial.print(time2-time1);
 Serial.println(" usec");
 last1=time1;
 last2=time2;
 }
}

```

```
 }
 }

 void sensor1(){ // ISR for sensor 1
 time1=micros(); // get the time in microseconds
 }

 void sensor2(){ // ISR for sensor 2
 time2=micros(); // get the time in microseconds
 }
}
```

## Op-Amp Comparator with Bias Voltage: Turning an Analog Signal into a HIGH or LOW Digital Level

The sketch that measures time duration above needs the sensor to output a LOW voltage (GND) when the object of interest crosses the path of the sensor and a HIGH voltage when it doesn't. However, your sensor may not provide this convenient a signal. What if a background sensor reading gives you some middle-of-the-road signal, and only a modest change in voltage when detecting the object? A comparator op-amp could help us here. We could tie the *signal* to one input on the op-amp, and a *bias voltage* set by a potentiometer to the other input.

This strategy is commonly used by many commercially-available sensor modules marketed for the Arduino community. The modules tend to have three pins: Vcc, GND, and Vout, as well as a trim potentiometer to adjust the bias voltage. The circuit diagram in Figure 9-15 shows a typical setup. The trim potentiometer  $R_1$  is adjusted so that the comparator output is HIGH with a background reading, but that a meaningful change in the signal trips it LOW. The logic can be reversed by exchanging the inputs.

In this way, you can set a threshold value  $V_b$  for any analog voltage signal, and convert a value below this threshold to 0 to trigger an interrupt. Even if the op-amp is not rail-to-rail, the range of  $V_{out}$  should be wide enough to change the state of the digital pin. You could also use this circuit with a regular digital pin, and the `digitalRead()` function. A low-pass filter may be required before the signal if it is too noisy in relation to the signal.

An advantage of this strategy is that you obtain a quick answer if all you need is to detect the presence of something beyond a certain threshold (e.g. sound, light, carbon dioxide, etc.). However, you lose the continuous response of a sensor this way. An interesting design project used this strategy to detect if a fridge door was left open, using an LDR inside the fridge to detect lights. The LinkNode D1 microprocessor board that was

used in this project has only one analog pin, which was already being used to monitor temperature. This strategy allowed for light detection, even though there were no free analog pins.

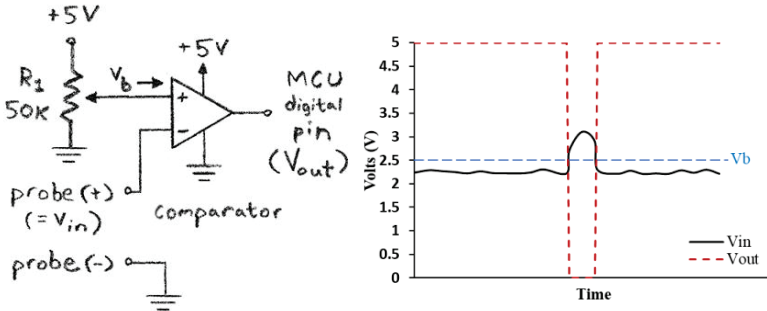


Figure 9-15. A comparator op-amp with a carefully-selected bias voltage changes a small signal response into a digital signal (HIGH or LOW).

### Matrix Keypads and LED Matrix Displays

If you need a series of buttons or LEDs for your device, you may find that you quickly run out of digital pins. One solution is to link these devices together in a matrix of rows and columns. The total number of pins is then reduced to reading a single pin for each row and column of the matrix. This section has two examples of this: a *matrix keypad*, and an *LED matrix display*.

Matrix keypads can be purchased as modules, they are salvageable from old electronics

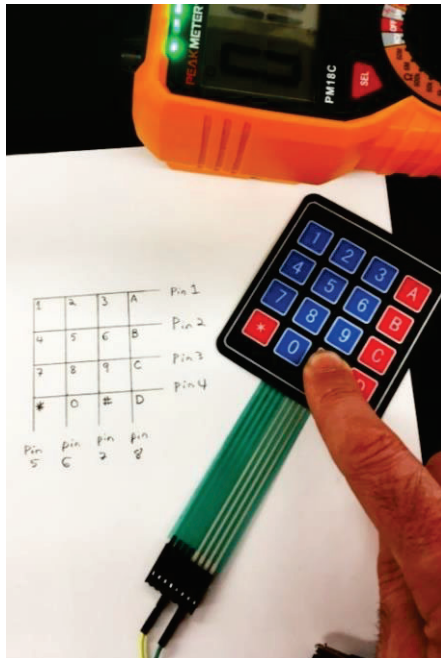


Figure 9-16. Mapping the pins of a matrix keypad to rows and columns.

such as telephones, and they can also be

built from scratch. In addition to providing the user with a better means of selecting options, they can add novelty and style to a project.

Your first task in adding a matrix keypad to your project is to look up or figure out the keypad's pin assignments for rows and columns. If the pin numbers are unknown, draw a diagram like the one in Figure 9-16, with the correct number of rows and columns for your keypad. Label the keypad buttons at each row/column intersection.

Test continuity between any two pins, pressing every button on the keypad until you find a button that completes the circuit. Write those pins numbers down on the rows and columns of the diagram. (gratefulfrog 2013) If no button push results in contact, then you have located pins on the same row or column, or perhaps found unused pins. Keep trying different pairs of pins until you have filled in all row and column pin numbers of your diagram. The process is a bit tedious, but you are finished when all pins are mapped, and you can successfully detect any button push based on your completed map.

Once you have mapped out the row and column pin numbers, it's time to write the sketch. Many libraries are available to read matrix keypads, which simplify coding. The following sketch will get you started.

```
// Matrix keypad example (4x4 matrix keypad)
#define ROWS 4 // #rows on matrix keypad
#define COLS 4 // #cols on matrix keypad
// Connections:
// Keypad row pins 1-4 to MCU pins 3-6
// Keypad col pins 5-8 to MCU pins 7-10
byte rowPins[ROWS] = {3, 4, 5, 6};
byte colPins[COLS] = {7, 8, 9, 10};

char keys[ROWS][COLS] = { //define keypad symbols
 {'1','2','3','A'},
 {'4','5','6','B'},
 {'7','8','9','C'},
 {'*','0','#','D'}
};

void setup(){
 Serial.begin(9600);
}

void loop(){
 char readKey=getKey();
 if(readKey!=0){
 Serial.println(readKey);
 delay(100); //short debounce
 }
}
```

```

 while(getKey()!=0); // wait until not pushed
 delay(100); // short debounce
 }
}

char getKey(){
 for(int i=0;i<ROWS;i++){
 pinMode(rowPins[i],INPUT_PULLUP);
 for(int j=0;j<COLS;j++){
 pinMode(colPins[j],OUTPUT);
 digitalWrite(colPins[j],LOW);
 if(digitalRead(rowPins[i])==LOW){
 return keys[i][j];
 }
 pinMode(colPins[j],INPUT);
 }
 }
 return 0; // if no key pushed, return null char.
}

```

Matrix keypads allow for more advanced user control, e.g. entering the wavelength of a spectrophotometer, or the speed of a motor. They can also be used to password protect your device.

You can organize LEDs in a matrix display in a similar manner, lighting them individually by row and column pins. The LEDs can be wired with their cathodes chained together by row, and their anodes chained together by column. This is called *common row cathode*. The alternate arrangement, connecting anodes together by row and cathodes by column is called *common row anode*.

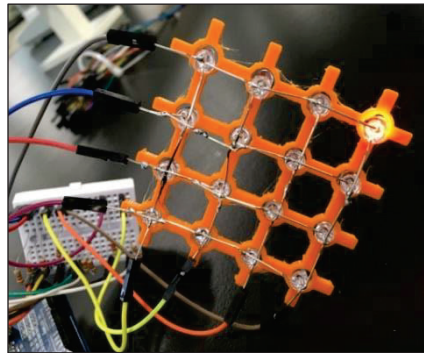


Figure 9-17. A custom LED matrix display.

For common row cathode LED matrices, Set the row of cathodes high for rows you don't want to light up, and the column of anodes low for columns you don't want to light up. Reverse this logic for the LED in the row and column you wish to light up. This will light a single LED in your matrix. Accessing individual lights and flashing them quickly enough can create the illusion that more than one LED is on continuously, through the persistence of vision effect. The LED matrix in Figure 9-17 was created by electronics enthusiast Jefferson Pun at

the Leslie Dan Faculty of Pharmacy. Prepackaged LED matrix displays are also available. The following sketch will turn on a single LED in a 2-dimensional LED matrix display (common row cathode).

```
// LED Matrix Example (4x4 LED matrix)
#define ROWS 4 //four rows (LED cathodes)
#define COLS 4 //four columns (LED anodes)
// Connections:
// LED row pins 1-4 to MCU pins 3-6
// LED col pins 5-8 to MCU pins 7-10
byte rowPins[ROWS] = {3, 4, 5, 6};
byte colPins[COLS] = {7, 8, 9, 10};

void setup(){
 writeLED(0,3); //light up LED on row 0, col 3
}

void loop(){
}

void writeLED(byte r, byte c){
 for(int i=0;i<ROWS;i++){
 pinMode(rowPins[i],OUTPUT);
 digitalWrite(rowPins[i],i==r?LOW:HIGH);
 for(int j=0;j<COLS;j++){
 pinMode(colPins[j],OUTPUT);
 digitalWrite(colPins[j],j==c?HIGH:LOW);
 }
 }
}
```

### *Charlieplexing LEDs*

Another approach to increasing the number of LEDs you can control with digital pins is called *charlieplexing*, a technique developed by Charlie Allen at Maxim. (Maxim Integrated 2003) This strategy involves wiring two LEDs in opposite directions between *every* possible pair combination of digital output pins you select for LED control. This results in  $n$  pins being able to drive  $n \times (n-1)$  LEDs. To prevent unwanted LEDs from lighting up, pins are strategically placed in INPUT mode. With an LED matrix configuration, four digital pins would only control four LEDs (no savings there). With charlieplexing, you could control up to twelve LEDs using the same number of digital pins. Table 9-2 provides pin states and an example circuit diagram using four charlieplexed pins. In order to use this technique with a +5V logic MCU, the diodes need to be able to handle being reverse

biased at 5V without getting damaged. Current-limiting resistors should be used for each digital pin (110  $\Omega$  for typical LEDs, powered using 5V pins).

**Table 9-2. Charlieplexing LEDs.**

| LED | Pin 0 | Pin 1 | Pin 2 | Pin 3 |
|-----|-------|-------|-------|-------|
| A   | 1     | 0     | INPUT | INPUT |
| B   | 0     | 1     | INPUT | INPUT |
| C   | 1     | INPUT | 0     | INPUT |
| D   | 0     | INPUT | 1     | INPUT |
| E   | 1     | INPUT | INPUT | 0     |
| F   | 0     | INPUT | INPUT | 1     |
| G   | INPUT | 1     | 0     | INPUT |
| H   | INPUT | 0     | 1     | INPUT |
| I   | INPUT | 1     | INPUT | 0     |
| J   | INPUT | 0     | INPUT | 1     |
| K   | INPUT | INPUT | 1     | 0     |
| L   | INPUT | INPUT | 0     | 1     |

The following is an example sketch for charlieplexing four digital pins.

```
// Charlieplexing example: 12 LEDs using 4 pins
byte charliePins[4]={5,4,3,2}; // use pins 5,4,3,2

void setup(){}

void loop(){
 for(int i=0;i<12;i++){
 charliePlex(i,charliePins); // light up each LED
 delay(250);
 }
}

void charliePlex(byte LED, byte Pins[4]){
 //3 states for pins: 0:LOW, 1:HIGH, 2:INPUT
 byte charlieStates[12][4]={
 {1,0,2,2}, //a (pin 0-1)
 {0,1,2,2}, //b (pin 1-0)
 {1,2,0,2}, //c (pin 0-2)
 {0,2,1,2}, //d (pin 2-0)
 {1,2,2,0}, //e (pin 0-3)
 {0,2,2,1}, //f (pin 3-0)
 {2,1,0,2}, //g (pin 1-2)
 {2,0,1,2}, //h (pin 2-1)
 {2,1,2,0}, //i (pin 1-3)
 {2,0,2,1}, //j (pin 3-1)
```

```

 {2,2,1,0}, //k (pin 2-3)
 {2,2,0,1} //l (pin 3-2)
};
for(int i=0;i<4;i++){ //set state of each pin
 if(charlieStates[LED][i]==2){
 pinMode(Pins[i],INPUT);
 }else{
 pinMode(Pins[i],OUTPUT);
 digitalWrite(Pins[i],charlieStates[LED][i]);
 }
}
}
}

```

## Need More Digital Pins?

With only 11 free digital pins on the Arduino Uno (plus 6 analog pins that can be declared as digital output pins, see *Using Analog Pins as Digital Output Pins* in Section 4), occasionally there are projects where you run out of digital pins. For example, the lab's LED clock has 37 elements to light up in order to display the time. What then? This may appear to exclude the Uno from the task; however, you can enlist the help of shift registers, which could tie up as few as three digital pins on your MCU to do this job. *Shift-out registers* provide you with more digital output pins, and *shift-in registers* provide you with more digital input pins.

### Shift-Out Registers

A *shift-out register* provides more digital output pins for your project. In order to *send* data to a shift-out register, three digital pins are required from your microcontroller: a clock pin, a latch pin, and a data pin. A shift-out register sets the state of its digital output pins (either HIGH or LOW) based on data pulsed from the microcontroller. The clock pin drums out the pace of data transfer,

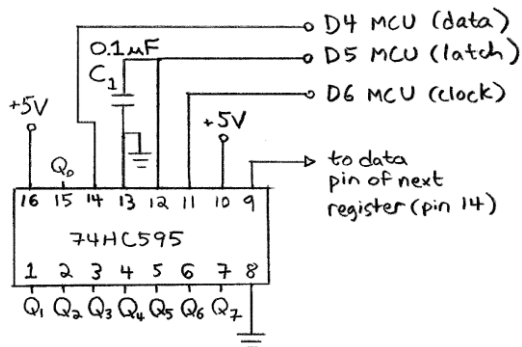


Figure 9-18. A SN74HC595 shift-out register, with digital output pins Q0 – Q7.

The clock pin drums out the pace of data transfer,



the latch pin tells the shift register when you are sending new data to it, and the data pin sends a series of high or low pulses, depending on how you would like to set the states of the shift register's digital output pins. Shift-out registers can be daisy chained, so you can add as many digital output pins as you need. The 74HC595 is a shift-out register that has eight settable digital pins. It has a wide operating voltage of 2-6V. (Texas Instruments Inc. 2015)

Figure 9-18 provides a basic connection schematic for a single 74HC595 shift-out register. The next register (optional) could be connected from pin 9 to its data pin, and share the latch and clock pins of the first shift-out register. Any of the MCU's free digital pins may be used to control the shift register. The selection of D4 to D6 in Figure 9-18 was arbitrary.

The Arduino IDE does not require an additional library to communicate with shift registers. The following example sketch will write the HIGH and LOW output states according to the byte `pinStates` to the shift-out register in Figure 9-18. (Maw and Igoe 2006)

```
// Shift-out One Bit: SN74HC595
#define dataPin1 4
#define latchPin1 5
#define clockPin1 6

byte pinStates = 0b10101010; // example byte

void setup(){
 pinMode(dataPin1, OUTPUT);
 pinMode(latchPin1, OUTPUT);
 pinMode(clockPin1, OUTPUT);
}

void loop(){
 //shift out:
 byteOut(pinStates, dataPin1, latchPin1, clockPin1);
}

void byteOut(byte Q, byte DAT, byte LCH, byte CLK){
 digitalWrite(LCH, LOW);
 shiftOut(DAT, CLK, LSBFIRST, Q);
 //use MSBFIRST to reverse order
 //(Most Significant Bit First)
 digitalWrite(LCH, HIGH);
}
```

If you would like to shift more than one byte out from daisy-chained chips, revise the `byteOut()` function: turn `Q` into an array of bytes, set the latch pin LOW, use the `shiftOut()` command *for each byte* to be shifted, then set the latch bit HIGH. The following is an example sketch for two daisy-chained shift-out registers.

```
// Shift-out Multiple Bits: SN74HC595
#define dataPin1 4
#define latchPin1 5
#define clockPin1 6
#define n 2 // # daisy-chained shift-out registers
byte pinStates[n] = {0b10101010, 0b11110000};

void setup(){
 pinMode(dataPin1,OUTPUT);
 pinMode(latchPin1,OUTPUT);
 pinMode(clockPin1,OUTPUT);
}

void loop(){
 byteOut(pinStates,n,dataPin1,latchPin1,clockPin1);
}

void byteOut(byte Q[], byte N, byte DAT, byte LCH, byte
CLK){
 digitalWrite(LCH, LOW);
 for(int i=N;i>=0;i--){
 shiftOut(DAT, CLK, LSBFIRST, Q[i]);
 }
 //use MSBFIRST to reverse bit order
 //(Most Significant Bit First)
 digitalWrite(LCH, HIGH);
}
```

### *Shift-In Registers*

A *shift-in register* provides more digital input pins for your project. In order to collect data from a shift-in register, three digital pins are required from your microcontroller: a clock pin, a load pin, and a data pin. A shift-in register reads the state of its digital input pins (either HIGH or LOW), then sends that data to your microcontroller. The clock pin drums out the pace of data transfer, the load pin tells the shift register when you want to read the pin states, and the data pin sends the information to your MCU. Shift-in registers can also be daisy chained so that you can have as many digital input pins as you need. The 74HC165 is a shift-in register that has eight

digital pins you can read the state of (so it sends 8 bits, or 1 byte of information).

Figure 9-19 provides a basic connection schematic for a single 74HC165 shift-in register. The next register (optional) could be connected from pin 10 to its data pin, and share the load and clock pins of the first shift-in register. Any of the MCU's free digital pins may be used to control the shift register. The selection of D8 to D10 was arbitrary. (Texas Instruments Inc 2015b)

The following example sketch will read the pin states from Q0 to Q7 from the shift-in register in Figure 9-19. (Alves 2015)

```
// Shift-in one bit: SN74HC165
#define data2 8
#define load2 9
#define clock2 10

byte pinStates=0b00000000; // to hold pin states

void setup(){
 Serial.begin(9600);
 pinMode(data2,INPUT);
 pinMode(load2,OUTPUT);
 pinMode(clock2,OUTPUT);
}

void loop(){
 pinStates=byteIn(data2,load2,clock2); //shift in
 Serial.println(pinStates,BIN); //print in binary
}

byte byteIn(byte DAT, byte LOAD, byte CLK){
 digitalWrite(CLK,HIGH);
 digitalWrite(LOAD,LOW);
 delayMicroseconds(5);
 digitalWrite(LOAD,HIGH);
```

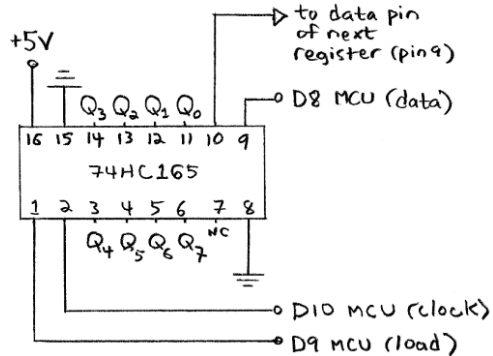


Figure 9-19. A SN74HC165 shift-in register, wired to send the pin states from digital input pins Q0 – Q7. In this configuration, pin 15 has been wired to ground, so the chip is constantly enabled, requiring one less control wire from the MCU.

```

 delayMicroseconds(5);
 byte Q=shiftIn(DAT,CLK,LSBFIRST);
 //use MSBFIRST to reverse bit order
 //(Most Significant Bit First)
 return Q;
}

```

The digital pins of a shift-in register float, just like the digital pins of an MCU in INPUT mode. Consequently, you will need to use pull-up or pull-down resistors if you plan on connecting switches to them. It is also advisable to ground any input pins you don't plan on using, as they will float randomly if not connected.

If you would like to shift more than one byte in from daisy-chained chips, revise the `byteIn()` function: turn `Q` into an array of bytes, set the load pin LOW then HIGH (as before), then use the `shiftIn()` command *for each byte* to be shifted. The following is an example sketch for two daisy-chained shift-in registers.

```

// Shift-in multiple bits: SN74HC165
#define data2 8
#define load2 9
#define clock2 10
#define n 2 //# daisy-chained shift-in registers

byte pinStates[n]; // to hold pin states

void setup(){
 Serial.begin(9600);
 pinMode(data2,INPUT);
 pinMode(load2,OUTPUT);
 pinMode(clock2,OUTPUT);
}

void loop(){
 byteIn(pinStates,n,data2,load2,clock2); //shift in
 for(int i=0;i<n;i++){
 Serial.println(pinStates[i],BIN); //print in binary
 }
}

void byteIn(byte Q[], byte N, byte DAT, byte LOAD, byte CLK){
 digitalWrite(CLK,HIGH);
 digitalWrite(LOAD,LOW);
 delayMicroseconds(5);
 digitalWrite(LOAD,HIGH);
 delayMicroseconds(5);
}

```

```
for(int i=0;i<N;i++){
 Q[i]=shiftIn(DAT,CLK,MSBFIRST);
 //use LSBFIRST to reverse order
 //(Least Significant Bit First)
}
}
```

## Bareduino – Running the ATmega328 Alone

In considering your design project, you might find that a serial connection to a computer isn't essential, that you are constrained for physical space, that you would like to extend battery life, or perhaps you would like to cut costs down. Smaller platforms, like the Arduino Nano, Pro Mini or Micro offer more compact options for controlling your devices. One option worth mentioning is that once a DIP version of the Arduino Uno is programmed, you can remove the ATmega328 chip from the board and run it separately – on a breadboard or PCB prototype board. This can potentially save some much-needed space.

How do you do this? You can just purchase an Arduino Uno (make sure it has the DIP version of the MCU, see Figure A-5, *Arduino Uno Pin-out Diagram*). Upload your sketch as you normally would, and then carefully pry the microprocessor chip off the board with a small screwdriver or microspatula. The next step is to place the microprocessor carefully on a breadboard, then hook it up to a power supply. Provide a regulated supply to pins 7 and 20 (3.3V-5V, or as low as 1.8V for the ATmega328P), and connect the supply ground to pins 8 and 22. You can use the voltage regulator from *Activity 2-2: LM317 Voltage Regulator* for this task. A few extra components are needed to run the MCU alone: a 16 MHz crystal, two 22 pF ceramic capacitors, a 10K pull-up resistor for the RESET pin, and if you like, a momentary switch to act as a reset button. A 10  $\mu$ F capacitor can be used to smooth out the power going in. The schematic can be as simple as Figure 9-20. (Navarro 2010)

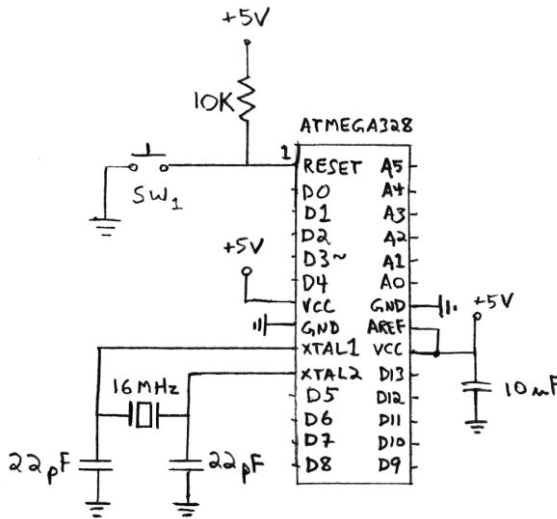


Figure 9-20 Configuring the ATmega328 chip outside the Arduino Uno. Pin numbers follow the physical layout of the DIP chip (Pin 1 is RESET).

The chip can be re-programmed by removing it from your circuit, plugging it back onto the Arduino Uno board, then uploading your sketch as usual. Alternately, you can use an external inexpensive *chip programmer* like the USBtinyISP (Figure 9-21). The USBtinyISP powers the MCU through the Vcc and GND pins, so hooking up a separate supply isn't necessary to program the MCU. To use an external programmer, make the connections provided in Table 9-3. (Willistein 2015)

**Table 9-3. Connecting the 6-pin ICSP connector from the USBtinyISP to the ATmega328, or the ATtiny85. The red cable shows which side Vcc is on.**

| USBtinyISP | ATmega328     | ATtiny85      | Pin-out, female connector of USBtinyISP |
|------------|---------------|---------------|-----------------------------------------|
| MISO       | Pin 18 (D12)  | Pin 6 (D1)    |                                         |
| SCK        | Pin 19 (D13)  | Pin 7 (D2)    |                                         |
| RST        | Pin 1 (RESET) | Pin 8 (RESET) |                                         |
| GND        | Pin 8 (GND)   | Pin 4 (GND)   |                                         |
| MOSI       | Pin 17 (D11)  | Pin 5 (D0)    |                                         |
| VCC        | Pin 7 (Vcc)   | Pin 8 (Vcc)   |                                         |

Once the chip programmer is connected, you will need to select USBtinyISP as the programmer in the Arduino IDE (Tools → Programmer → USBtinyISP) and then install the appropriate drivers for it. At the time of publication, USBtinyISP drivers were available for download at:

<https://learn.adafruit.com/usbtinyisp/drivers>

To upload your sketch, hold down the shift key while pressing the upload button, or alternately, select “Upload using programmer” from the Sketch drop-down menu.

Having a chip programmer on hand is a good idea. Sometimes when an MCU is shorted and seems to be unresponsive, it can be revived by re-burning the bootloader, using a programmer through the ICSP header of the Arduino Uno. To do this in the Arduino IDE, select the board options you’d like under “Tools→Board”, then select “Burn Bootloader”. Or, perhaps the somewhat fragile FTDI chip on the Uno dies (the chip responsible for creating a COM port through USB). If this happens, you can still upload a sketch directly to the ATmega328 chip via the Uno’s ICSP header, using a chip programmer.

You can also program other MCUs like the ATtiny85 with the USBtinyISP by connecting it to the appropriate pins. Connections between the ICSP header and the DIP version of the ATtiny85 MCU are provided in Table 9-3.

With a chip programmer, you have more options in burning the bootloader, which will give you better control over how the MCU operates. With an additional library added to the Arduino IDE Board Manager, you can even enable the ATmega328’s internal 8 MHz oscillator, so that the external crystal in Figure 9-20 is not needed. One such library is available at <https://github.com/oshlab/Breadboard-Arduino>. (Burkett 2016)

Note that the pin-out in Table 9-3 is the mirror image of the ICSP headers on the Arduino Uno (see Figure A-5, *Arduino Uno Pin-out*

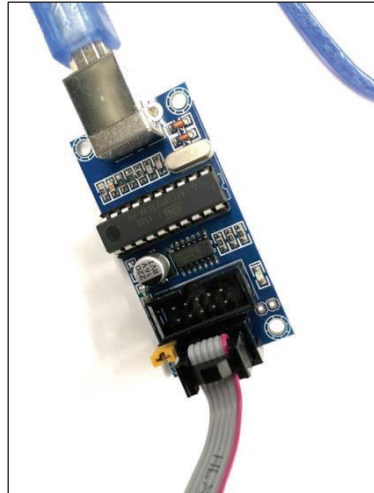


Figure 9-21. USBtinyISP chip programmer, useful for re-burning the bootloader onto an MCU, uploading a sketch through the ICSP header on the Arduino Uno, or programming an MCU directly through the appropriate pins.

*Diagram*), since it was designed to plug directly into these pins for quick bootloader burning.

## Learning Objectives for Section 9

After having completed the design project, the student will be able to:

- 1) Organize and propose the initial design concept for a novel project, selecting appropriate components and thinking through the design before starting to build it.
- 2) Propose a testing protocol to help define what a successful device or project will look like.
- 3) Plan class time to optimize use of resources and ensure that project deliverables are met.
- 4) Create a circuit diagram for your own project, using the conventions defined in this text.
- 5) Write your own sketch to control your device, and debug/troubleshoot/fine tune it depending on your results.
- 6) Practice and articulate an *elevator pitch* (3-5 minute highlight speech) for your project to communicate the core elements and principles of your device.
- 7) Document your results in a scientific technical document with enough detail for someone else to replicate your project.
- 8) Properly cite any webpages, ideas, libraries, or sketches used in your project.



# SECTION 10

## ADVANCED TOPICS IN PROGRAMMING

### Controlling MCU Registers, Interrupts and Timers

The reason this section is not covered in the lectures of this course is that none of the applications planned for the activities specifically required bitwise operations, or explicitly accessing the registers of the ATmega328. However, if you venture out into the world of other people's code, bitwise operations can be found in sketches that handle port manipulation, shift registers, LED matrix displays, and many other places. There is much more fun to be had by fiddling with an MCU's registers.

Just as shift registers hold data to be shifted in or out, an MCU has internal registers that hold data, controlling the way it works. Changing the bit values in these registers will give you access to interrupts, sleep modes, quicker reading of digital and analog pins, and some other amazing features that will be covered here. However, it is important to understand that microprocessors have very different architectures, and as such have different registers. If you write a sketch changing bit values in registers for the ATmega328, it will not work as expected with other MCUs. To start down this path, we need a quick lesson on bitwise operations and how they work.

### Bitwise Operations

If you haven't seen a bitwise operator, they look confusing, but are really quite simple. Bitwise operators function like boolean operators, but perform the logical test *on each bit* in a byte variable, separately. So far, we have thought of a byte as a string of 0's and 1's in binary that represent a base 10 integer, e.g.:

$$0b00100011 = 2^5 + 2^1 + 2^0 = 32 + 2 + 1 = 35.$$

However, we can also think of a byte as *an array of 8 separate bits*, without having to declare them as an array. We can use the byte example above to mean:

Pin 7: LOW, Pin 6: LOW, Pin 5: HIGH, Pin 4: LOW, Pin 3: LOW, Pin 2: LOW, Pin 1: HIGH, Pin 0: HIGH

where Pins 0 to 7 are *any arbitrary digital pins* for some device, and are not related to the digital pins on the Arduino Uno in any way. We use reverse order of pins here because it becomes convenient later to think about the smallest value pin being at the *end* of the byte, however it doesn't really matter for the purpose of this discussion.

To summarize, we can think of a byte as a holder of 8 independent bool variables. This is a convenient way of thinking, because it simplifies handling the information to and from your pins, which all work by shifting bytes of data in and out in this exact way. Let's define a byte variable called `myPins` with the above example:

```
byte myPins = 0b00100011;
```

We call the last bit in a byte the *least significant bit* (LSB), and the first bit the *most significant bit* (MSB). For the example byte `myPins`, the MSB is 0, and the LSB is 1. The C++ language has a way of handling each bit separately. These are called *bitwise operations*, because they can work on specific bits within a byte, even without disturbing the other bits.

It's far easier to explain bitwise operators with examples, so let's look at the first operator: the *bitwise AND*.

### *Bitwise AND (&)*

Let's say you had a byte of data stored in `myPins`, and wanted to find out if a specific bit (e.g. Pin 5) in `myPins` is set high or low. The *bitwise AND* symbol, "&" (single ampersand) could help with this. You can use the & operator with a "1" in only that bit position, and zeros everywhere else. The results from a logical AND operation will result in zeros at every other bit position, and a 1 in the 5<sup>th</sup> bit position if the state of Pin 5 is currently HIGH (or 1):

```
byte pin5state = myPins & 0b00100000;
```

What this does is set up the logical comparisons in Table 10-1 (made independently in each column):

**Table 10-1. Example bitwise AND comparison: finding out a pin state.**

| Bit             | 7            | 6            | 5            | 4            | 3            | 2            | 1            | 0            |
|-----------------|--------------|--------------|--------------|--------------|--------------|--------------|--------------|--------------|
| <i>Variable</i> | <i>Pin 7</i> | <i>Pin 6</i> | <i>Pin 5</i> | <i>Pin 4</i> | <i>Pin 3</i> | <i>Pin 2</i> | <i>Pin 1</i> | <i>Pin 0</i> |
| myPins          | 0            | 0            | 1            | 0            | 0            | 0            | 1            | 1            |
| bit mask        | 0            | 0            | 1            | 0            | 0            | 0            | 0            | 0            |
| AND result      | 0            | 0            | 1            | 0            | 0            | 0            | 0            | 0            |

This operation is called *bit masking*, because the second byte we defined, called a bit mask, sets all other positions to zero. The bit mask hides all values except for (in this case) bit 5, representing the state of Pin 5.

We now have a byte variable called `pin5state` with the “answer” to whether Pin 5 is HIGH or LOW. If we display this number as an integer, we will see the number  $2^5=32$ . If we display it as a byte, we will see `0b00100000`. However, what this number means now is “Pin 5 is HIGH”.

The serial monitor has the capability of displaying bytes instead of decimal values. The command:

```
Serial.print(pin5state, BIN);
```

will show `100000`, and the command:

```
Serial.print(pin5state, DEC);
```

or:

```
Serial.print(pin5state);
```

will print a value of 32 to the serial monitor.

Putting this logic in the context of a program, you can check the status of the Pin 5 bit using the following commands:

```
byte pin5state = myPins & 0b00100000;
if(pin5state==0b00100000){
 Serial.println("Pin 5 is HIGH.");
}else{
 Serial.println("Pin 5 is LOW.");
}
```

### *Bitwise OR (|)*

The vertical bar symbol “|”, also called “pipe”, is the *bitwise OR* operator. It can be used for setting specific bits HIGH. Let’s say that we are using `myPins` now to SET the state of the bits in `myPins`, rather than read them, and we would like to set Pin 3 HIGH without disturbing the states of the other bits. The bitwise AND operator can’t help us here. If we were to try this:

```
myPins = myPins & 0b00001000;
```

then Pin 3 would stay on only if it was already on, and all of the other pins would turn off. The bitwise OR command will do this for us:

```
myPins = myPins | 0b00001000;
```

This command makes the logical comparisons in Table 10-2:

**Table 10-2. Example bitwise OR comparison: setting a specific pin HIGH.**

| <i>Variable</i> | <i>Pin 7</i> | <i>Pin 6</i> | <i>Pin 5</i> | <i>Pin 4</i> | <i>Pin 3</i> | <i>Pin 2</i> | <i>Pin 1</i> | <i>Pin 0</i> |
|-----------------|--------------|--------------|--------------|--------------|--------------|--------------|--------------|--------------|
| myPins          | 0            | 0            | 1            | 0            | 0            | 0            | 1            | 1            |
| bit mask        | 0            | 0            | 0            | 0            | 1            | 0            | 0            | 0            |
| OR result       | 0            | 0            | 1            | 0            | 1            | 0            | 1            | 1            |

Pin 3 turned *on* without disturbing the states of the other bits.

What if we would like to turn Pin 3 OFF now without disturbing the other pin states? We would need to be clever. The OR comparison will always result in a HIGH state if the pin state is already HIGH. Can you guess the answer without reading ahead?

We will need to use a bitwise AND, but in a slightly different way. We want to turn off only Pin 3, and leave all of the other pins untouched, so the logic will look like this:

```
myPins = myPins & 0b11110111;
```

The “1”s in each position preserve the existing state of the pins with AND: LOW stays LOW, HIGH stays HIGH. The “0” for Pin 3 will change a HIGH to LOW, or leave a LOW pin state LOW. This is shown in Table 10-3.

**Table 10-3. Example bitwise AND comparison: setting a pin LOW.**

| <i>Variable</i> | <i>Pin 7</i> | <i>Pin 6</i> | <i>Pin 5</i> | <i>Pin 4</i> | <i>Pin 3</i> | <i>Pin 2</i> | <i>Pin 1</i> | <i>Pin 0</i> |
|-----------------|--------------|--------------|--------------|--------------|--------------|--------------|--------------|--------------|
| myPins          | 0            | 0            | 1            | 0            | 1            | 0            | 1            | 1            |
| bit mask        | 1            | 1            | 1            | 1            | 0            | 1            | 1            | 1            |
| AND result      | 0            | 0            | 1            | 0            | 0            | 0            | 1            | 1            |

Pin 3 turned *off* without disturbing the states of the other bits in `myPins`.

### *Bitwise NOT (~)*

The above example is a great lead-in to the *bitwise NOT* operator, “~” (or “tilde”). The NOT operator inverts the state of each bit separately. So

we could have coded the above example (turning off Pin 3 and leaving the other pin states alone) like this:

```
myPins = myPins & ~0b00001000;
```

This would result in inverting 0b00001000 to the binary number 0b11110111. In the order of operations, a bitwise NOT is performed first (before other bitwise operations), so be careful how you use it.

### *Bitwise XOR (^)*

The carrot symbol “^” is the *bitwise XOR* operator. Recall from Section 2 that XOR (exclusive OR) returns TRUE only when A and B are different (see Table 10-4).

The bitwise XOR operator can toggle (in other words, invert) the existing state of a pin, without disturbing other pin states. Using the example above starting with Pin 3 set LOW, let’s toggle the value of Pin 3 without disturbing the states of the other bits, using bitwise XOR. We would use the following expression:

```
myPins = myPins ^ 0b00001000;
```

This command makes the logical comparisons in Table 10-5:

**Table 10-5. Example bitwise XOR comparison: toggling a pin state HIGH.**

| <i>Variable</i> | <i>Pin 7</i> | <i>Pin 6</i> | <i>Pin 5</i> | <i>Pin 4</i> | <i>Pin 3</i> | <i>Pin 2</i> | <i>Pin 1</i> | <i>Pin 0</i> |
|-----------------|--------------|--------------|--------------|--------------|--------------|--------------|--------------|--------------|
| myPins          | 0            | 0            | 1            | 0            | 0            | 0            | 1            | 1            |
| bit mask        | 0            | 0            | 0            | 0            | 1            | 0            | 0            | 0            |
| XOR result      | 0            | 0            | 1            | 0            | 1            | 0            | 1            | 1            |

Pin 3 was toggled on without disturbing the other bits. If we ran the same command again, Pin 3 would be toggled HIGH, without disturbing the states of the other bits (Table 10-6).

**Table 10-4. XOR logic table.**

| <i>A</i> | <i>B</i> | <i>XOR(A,B)</i> |
|----------|----------|-----------------|
| 0        | 0        | 0               |
| 0        | 1        | 1               |
| 1        | 0        | 1               |
| 1        | 1        | 0               |

**Table 10-6. Example bitwise XOR comparison: toggling a bit state LOW.**

| <i>Variable</i> | <i>Pin 7</i> | <i>Pin 6</i> | <i>Pin 5</i> | <i>Pin 4</i> | <i>Pin 3</i> | <i>Pin 2</i> | <i>Pin 1</i> | <i>Pin 0</i> |
|-----------------|--------------|--------------|--------------|--------------|--------------|--------------|--------------|--------------|
| myPins          | 0            | 0            | 1            | 0            | 1            | 0            | 1            | 1            |
| bit mask        | 0            | 0            | 0            | 0            | 1            | 0            | 0            | 0            |
| XOR result      | 0            | 0            | 1            | 0            | 0            | 0            | 1            | 1            |

### *Shifting Bits with “<<” and “>>”*

With LEDs arranged in a matrix, you have likely seen the practicality of bit shifting, probably in a subway, airport, or store window. LED arrays are extremely popular in media and advertising. You can scroll a message quite easily across a matrix of LEDs by *bit shifting* from one position to the next. In C++, there are two shifting operators: << (left shift) and >> (right shift). You specify (with an integer number) the number of spaces you would like to shift the data. If we were to take the original byte `myPins` and shift it *two spaces to the left*:

```
myPins = myPins << 2;
Serial.print(myPins, BIN);
```

the results would be shifted left two spaces:

```
Before: 0b00100011
After: 0b10001100 (shifted left 2 spaces)
```

If we were to take the original byte `myPins` and shift it *one space to the right*:

```
myPins = myPins >> 1;
Serial.print(myPins, BIN);
```

the results would be shifted right one space:

```
Before: 0b00100011
After: 0b00010001 (shifted right 1 space)
```

What happens to the data on the end as it gets shifted off in to space? The answer is: nothing! It disappears, forever forgotten. Like your lost sock in the dryer. Bon voyage! Have a nice non-existence.

We can shift bits to simplify our code a bit as well. Taking the first example in this section (bitwise AND), we wrote the following code:

```
byte pin5state = myPins & 0b00100000;
if (pin5state==0b00100000){
 Serial.println("Pin 5 is HIGH.");
}else{
```

```

 Serial.println("Pin 5 is LOW.");
}

```

We could bit shift the value of `pin5state` so our comparison could be against a 1 or 0 in the `if()` statement, rather than the value of 32:

```

byte pin5state=myPins&0b00100000; // bit mask for pin 5
pin5state=pin5state>>5; // shift bit right by 5 spaces
if(pin5state==HIGH){
 Serial.println("pin 5 is HIGH.");
}else{
 Serial.println("pin 5 is LOW.");
}

```

**Note:** Be careful not to confuse Boolean operators with bitwise operators (Table 10-7).

**Table 10-7. Boolean operators vs. bitwise operators in C++.**

| <i>Operator</i> | <i>Boolean</i> | <i>Bitwise</i> |
|-----------------|----------------|----------------|
| AND             | &&             | &              |
| OR              |                |                |
| NOT             | !              | ~              |
| XOR             |                | ^              |

For more information: <https://playground.arduino.cc/Code/BitMath>

### *Bitwise Operators: Short Forms*

Putting all these ideas together, we can use C++ short forms for each operation. You will see these short forms in other people's sketches, because they are faster to write. Table 10-8 provides a summary chart of worked examples, coded in different short forms, on the example byte variable called `myPins` with a starting value of `0b00100011`. In the Result column, the bits affected as a result of each command are in boldface.

**Table 10-8. Short forms for common bitwise operations, with examples.**

| <i>Function</i>                                                  | <i>Syntax</i>                                                                                                                        | <i>Result</i>                                                                                                      |
|------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------|
| Check the status of pin 5, and return a 0 if LOW, and 1 if HIGH. | <pre>pin5state=myPins&gt;&gt;5&amp;1; // &gt;&gt; happens before &amp; pin5state=bitRead(myPins, 5);</pre>                           | pin5state is <b>1</b>                                                                                              |
| Set pin 3 HIGH, and set all other pins LOW.                      | <pre>myPins=1&lt;&lt;3; myPins=_BV(3);</pre>                                                                                         | myPins is<br><b>0b00001000</b>                                                                                     |
| Set pins 3, 4, and 5 HIGH, and set all other pins LOW.           | <pre>myPins=(1&lt;&lt;3)   (1&lt;&lt;4)   (1&lt;&lt;5); myPins=_BV(3)   _BV(4)   _BV(5);</pre>                                       | myPins is<br><b>0b00111000</b>                                                                                     |
| Set pin 3 LOW, and set all other pins HIGH.                      | <pre>myPins=~(1&lt;&lt;3); // parentheses are needed. myPins=~_BV(3);</pre>                                                          | myPins is<br><b>0b11110111</b>                                                                                     |
| Set pin 3 HIGH, leaving other pin states alone.                  | <pre>myPins =1&lt;&lt;3; myPins = _BV(3); bitSet(myPins, 3); bitWrite(myPins, 3, HIGH);</pre>                                        | myPins is<br><b>0b00101011</b>                                                                                     |
| Set pins 3, 4, and 5 HIGH, leaving other pin states alone.       | <pre>myPins =(1&lt;&lt;3)   (1&lt;&lt;4)   (1&lt;&lt;5); myPins = _BV(3)   _BV(4)   _BV(5);</pre>                                    | myPins is<br><b>0b00111011</b>                                                                                     |
| Set pin 3 LOW, leaving other pin states alone.                   | <pre>myPins&amp;=~(1&lt;&lt;3); // parentheses are needed. myPins&amp;=~_BV(3); bitClear(myPins, 3); bitWrite(myPins, 3, LOW);</pre> | myPins is<br><b>0b00100011</b>                                                                                     |
| Set pins 3, 4, and 5 LOW, leaving other pin states alone.        | <pre>myPins&amp;=~((1&lt;&lt;3)   (1&lt;&lt;4)   (1&lt;&lt;5)); myPins&amp;=~(_BV(3)   _BV(4)   _BV(5));</pre>                       | myPins is<br><b>0b00000011</b>                                                                                     |
| Change the state of pin 3, leaving other pin states alone.       | <pre>myPins^=1&lt;&lt;3; myPins^=_BV(3);</pre>                                                                                       | myPins is<br><b>0b00101011</b> ,<br>and toggles<br>back to<br><b>0b00100011</b> if<br>the command is<br>run again. |



|                                                                                             |                                                                                              |                                                                                   |
|---------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|
| Change the state of pins 3, 4, and 5, leaving other pin states alone.                       | <code>myPins^=(1&lt;&lt;3)   (1&lt;&lt;4)   (1&lt;&lt;5);</code>                             | myPins is 0b00011011, and toggles back to 0b00100011 if the command is run again. |
|                                                                                             | <code>myPins^=_BV(3)   _BV(4)   _BV(5);</code>                                               |                                                                                   |
| Shift the bits in myPins left by 1.                                                         | <code>myPins&lt;&lt;=1;</code>                                                               | myPins is 0b1000110. This is useful in scrolling LED matrix displays.             |
| Shift the bits in myPins right by 1.                                                        | <code>myPins&gt;&gt;=1;</code>                                                               | myPins is 0b00010001. This is useful in scrolling LED matrix displays.            |
| Combine myPins as a <i>low</i> byte, with another byte variable (HB) as a <i>high</i> byte. | <code>byte HB=0b11111111;<br/>unsigned int result=0;<br/>result=(HB&lt;&lt;8)+myPins;</code> | result is:<br>0b1111111100100011<br>(high byte combined with low byte)            |
| Extract a high byte (HB) and a low byte (LB) from a larger data type                        | <code>int x=0b1111111100100011;<br/>byte HB=x&gt;&gt;8;<br/>byte LB=x;</code>                | result is:<br>HB: 0b11111111<br>LB: 0b00100011                                    |
|                                                                                             | <code>int x=0b1111111100100011;<br/>byte HB=highByte(x);<br/>byte LB=lowByte(x);</code>      |                                                                                   |

There are more ways to accomplish the above tasks. The syntax can look confusing, but gets easier once you get the hang of it. The order of operations is important (~ then <<, >> then &, |), so if you are in doubt, use brackets to be explicit about what you would like to happen first. Be sure to leave descriptive comments in your code, to remind the future version of yourself what you meant. Bitwise operations can look very cryptic.

For more information: <http://playground.arduino.cc/Code/BitMath>

## Introduction to Port Manipulation

As you get into the inner workings of microprocessors for more complicated projects (e.g. setting up interrupts, changing clock speeds, modifying PWM frequencies, etc.), the registers in these microprocessors are set using bitwise operations. Accessing these registers is called *port manipulation*. Port manipulation needs to be coded carefully. Set a register incorrectly, and you can accidentally permanently lock yourself out of a microprocessor. Nonetheless, these commands are common in online

sketches, since programmers who commit a project to a specific microprocessor chip (e.g. the ATmega328) understand that port manipulation is faster, more flexible, and can unlock hidden abilities of that microprocessor. Sometimes, it pays to talk directly to your microprocessor by changing register values directly. Doing so cuts out all the programming shrubbery of baked-in commands like `digitalWrite()`. How would you know how to do any of this? The microprocessor's full datasheet is a great place to start. The datasheet will describe the purpose and function of each bit in each register, and recommend how to approach setting and reading these bits.

### *Worked Example: Fast Analog Read*

It's difficult to imagine 120 microseconds as being too long a duration for an analog reading. However, if analog data needs to be acquired at a faster rate, the speed of the analog reading can be controlled using the ADCSRA register of the ATmega328. Just keep in mind that a quicker analog reading will have less time to equilibrate, and may also be more noisy. We can use this example of setting the ADCSRA register of the ATmega328 chip for the general problem of "how do you set or clear register bits?" Having a look at this register will help us put the code into context, and we will better understand what is happening. We will consequently understand how to change bit values in other registers. If you would like to skip the discussion and you just need the code for a faster analog reading, jump to the end of this section for the final sketch.

The ATmega328 has many internal registers that change the way it works. Each register is usually 1 byte (or 8 bits) of information long, with each bit holding either a 0 or a 1. What makes these bits important is that they set different modes in the MCU. In fact, many of the commands we've learned so far, e.g. `digitalWrite()`, `digitalRead()`, and `analogRead()`, access the MCU-specific registers behind the scenes in their rolled-up code. The Arduino IDE simplifies this process by creating functions that are easy to use and work across the variety of supported microprocessors, so you don't have to look up and manipulate the registers yourself. It makes programming the microprocessors accessible to the hobbyist, because it simplifies coding, and allows the same sketch to work across all supported microprocessors.

Let's have a close look at the ADCSRA register, which controls how an analog reading is taken. This information comes from 21.9.2 of the ATmega328 datasheet (complete version):

**Table 10-9. Structure of the ADCSRA, the ADC control and status register (Atmel Corporation 2016)**

|            |      |      |       |      |      |       |       |       |
|------------|------|------|-------|------|------|-------|-------|-------|
| Bit        | 7    | 6    | 5     | 4    | 3    | 2     | 1     | 0     |
| ADCSRA     | ADEN | ADSC | ADATE | ADIF | ADIE | ADPS2 | ADPS1 | ADPS0 |
| Read/Write | R/W  | R/W  | R/W   | R/W  | R/W  | R/W   | R/W   | R/W   |

Here we learn that ADCSRA is an 8-bit register, and each bit it contains can be read from or written to (R/W). The datasheet then provides a table on how to set the ADC prescaler (see Table 10-10). The prescaler in this case is the MCU clock frequency (16,000 MHz) divided by the ADC input clock.

**Table 10-10. Setting the ADC Prescaler value using the ADPS0, ADPS1, and ADPS2 bits (Atmel Corporation 2016)**

| ADPS2 | ADPS1 | ADPS0 | Prescaler |
|-------|-------|-------|-----------|
| 0     | 0     | 0     | 2         |
| 0     | 0     | 1     | 2         |
| 0     | 1     | 0     | 4         |
| 0     | 1     | 1     | 8         |
| 1     | 0     | 0     | 16        |
| 1     | 0     | 1     | 32        |
| 1     | 1     | 0     | 64        |
| 1     | 1     | 1     | 128       |

The information in Table 10-10 comes from Table 21-5 of the ATmega328 datasheet.

How long should it take to acquire an analog reading if we change the prescaler value? Here is where we need to do a little math. The default prescaler set for the ADCSRA is 128 (the longest setting). The datasheet states that a normal conversion for the ADC takes 13 clock cycles, in single conversion mode. That means that at normal speed, the frequency of an analog reading would be:

$$f_{ADC} = \frac{f_{clock}}{prescaler \times 13} = \frac{16,000,000 \text{ Hz}}{128 \times 13} = 9615 \text{ Hz}$$

and that a single reading would take  $1/f_{ADC} = 0.000104 \text{ sec} = 104 \mu\text{sec}$ .

Now we get to the good part. Armed with this information, we get to speed up our reading, first by picking the prescaler value we want, and then by setting the register with the appropriate value bits. Let's choose a prescaler value of 4. That means the frequency of the ADC will be:

$$f_{ADC} = \frac{16,000,000 \text{ Hz}}{4 \times 13} = 307692 \text{ Hz}$$

and a single read should take  $1/f_{ADC} = 3.25 \mu\text{sec}$ . That's much faster! How do we change the bit values in the register?

We have already covered bitwise operations in the previous section. If you haven't had a look, now would be a good time to go back. Our goal is to change *only* the three bits ADPS2, ADPS1, and ADPS0 in register ADCSRA. We don't want to disturb the states of the other bits in this register. That could have unintended consequences. There isn't any fancy code required to access the MCU registers. Thankfully, the libraries installed for the Arduino AVR Boards Library in the Boards Manager have set up the registers as byte variables that you can access, *just like* a regular byte variable. If we would like a prescaler value of 4, we see from Table 10-10 that we need ADPS2=0, ADPS1=1, and ADPS0=0. We call writing 1 to a bit "*setting the bit*", and we call writing 0 to a bit "*clearing the bit*". Worded in another way, we need to *set bit ADPS1*, and *clear bits ADPS2 and ADPS0*, without disturbing the other bits. This can be as simple then as the following three commands:

```
ADCSRA &=~0b000000100; // clear ADPS2 only
ADCSRA |= 0b000000010; // set ADPS1 only
ADCSRA &=~0b000000001; // clear ADPS0 only
```

These methods of changing bit values are provided in Table 10-8. From Table 10-8, you can also see that the following code is equivalent:

```
ADCSRA &=~(1<<2); // clear ADPS2 only
ADCSRA |= (1<<1); // set ADPS1 only
ADCSRA &=~(1<<0); // clear ADPS0 only
```

Here is yet another way to set these bits:

```
ADCSRA &=~_BV(2); // clear ADPS2 only
ADCSRA |= _BV(1); // set ADPS1 only
ADCSRA &=~_BV(0); // clear ADPS0 only
```

These three different ways of setting bit values in the register all perform the same changes. They *clear* bits ADPS2 and ADPS0, and *set* ADPS1. This tells the MCU we would like a prescaler of 4. Not so bad! It gets even easier, because to write the above code, you would need to remember that ADPS2 is bit 2 in the register, ADPS1 is bit 1, and ADPS0 is bit 0. The Arduino AVR Boards Library also includes these individual bits as defined names, so you can use them directly in your code. Stored in these names are the positions in their respective bit registers, e.g. `Serial.print(ADPS2)`; will print "2" on the serial monitor. This means you don't have to go back to the datasheet to find out the position of the bits within their register. The following code will also work:

```
ADCSRA &=~_BV(ADPS2); // clear ADPS2 only
ADCSRA |=_BV(ADPS1); // set ADPS1 only
ADCSRA &=~_BV(ADPS0); // clear ADPS0 only
```

Our work is now done. You will find registers and their bit names referenced in other sketches, and hopefully now this process is demystified for you. People tend to use any of the methods above to set and clear bits in registers. You can also define the following two `#define` functions at the top of your sketch, which will take care of the bitwise operations for you:

```
#define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))
#define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
```

These two functions are standard methods to set or clear a single bit in a register. See <https://playground.arduino.cc/Main/AVR> for more details.

In the body of the sketch, you can use the defined functions `sbi()` and `cbi()` so that you won't be tripping through the logic. If you'd like to set ADPS1, and clear ADPS2 and APDS0 in register ADCSRA for a prescaler of 4, you can just type:

```
cbi(ADCSRA,ADPS2); // clear bit ADPS2 in ADCSRA
sbi(ADCSRA,ADPS1); // set bit ADPS1 in ADCSRA
cbi(ADCSRA,ADPS0); // clear bit ADPS0 in ADCSRA
```

or alternately:

```
cbi(ADCSRA,2); // clear bit ADPS2 in ADCSRA
sbi(ADCSRA,1); // set bit ADPS1 in ADCSRA
cbi(ADCSRA,0); // clear bit ADPS0 in ADCSRA
```

We are now ready to write our sketch. Even though we could use any of the methods above to set the register, let's test drive the `sbi()` and `cbi()` functions for fun. This code was adapted from Bruno Portaluri's Arduino Blog. It reports the average reading time of 1000 analog readings. (Portaluri 2015)

```
// Fast analog read (prescaler=4, ~3.25 usec/read)
#define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))
#define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
```

```
void setup(){
 cbi(ADCSRA,ADPS2); // clear bit ADPS2
 sbi(ADCSRA,ADPS1); // set bit ADPS1
 cbi(ADCSRA,ADPS0); // clear bit ADPS0
 Serial.begin(9600);
}
```

```
void loop(){
 unsigned int timer1, timer2;
 timer1=micros();
```

```

for(int i=0;i<1000;i++)analogRead(A0);
timer2=micros();
Serial.println(float(timer2-timer1)/1000.0);
}

```

### *Fast Digital Read and Write*

In addition to taking analog readings faster, accessing the registers directly gives us blazing read and write speeds to the digital pins. There are three *banks* of pins for the ATmega328 microprocessor:

**Table 10-11. ATmega328 pin banks. (Atmel Corporation 2016)**

| <i>Bit</i>                                                                          | <i>Bank D</i>      | <i>Bank B</i>       | <i>Bank C</i>     |
|-------------------------------------------------------------------------------------|--------------------|---------------------|-------------------|
| 7 (first)                                                                           | PD7: digital pin 7 | PB7: crystal        | -                 |
| 6                                                                                   | PD6: digital pin 6 | PB6: crystal        | PC6: reset        |
| 5                                                                                   | PD5: digital pin 5 | PB5: digital pin 13 | PC5: analog pin 5 |
| 4                                                                                   | PD4: digital pin 4 | PB4: digital pin 12 | PC4: analog pin 4 |
| 3                                                                                   | PD3: digital pin 3 | PB3: digital pin 11 | PC3: analog pin 3 |
| 2                                                                                   | PD2: digital pin 2 | PB2: digital pin 10 | PC2: analog pin 2 |
| 1                                                                                   | PD1: digital pin 1 | PB1: digital pin 9  | PC1: analog pin 1 |
| 0 (last)                                                                            | PD0: digital pin 0 | PB0: digital pin 8  | PC0: analog pin 0 |
| Bits shaded grey: do not set or change the values of these bits in their registers. |                    |                     |                   |

Each of the ATmega328's pin banks have three registers controlling them, with their bit numbering as indicated in Table 10-11. The nine registers controlling all the pin states are presented in Table 10-12.

**Table 10-12. DDR, PORT, and PIN registers for the three pin banks of the ATmega328. Bit DDD1=1 if the serial monitor is needed, otherwise it can be cleared. (Atmel Corporation 2016)**

| Bit:   | 7                | 6                | 5      | 4      | 3      | 2      | 1                 | 0                |
|--------|------------------|------------------|--------|--------|--------|--------|-------------------|------------------|
| DDRD:  | DDD7             | DDD6             | DDD5   | DDD4   | DDD3   | DDD2   | DDD1*<br>(=1, tx) | DDD0<br>(=0, rx) |
| PORTD: | PORTD7           | PORTD6           | PORTD5 | PORTD4 | PORTD3 | PORTD2 | PORTD1            | PORTD0           |
| PIND:  | PIND7            | PIND6            | PIND5  | PIND4  | PIND3  | PIND2  | PIND1             | PIND0            |
| DDRB:  | DDB7<br>(=0,xtl) | DDB6<br>(=0,xtl) | DDB5   | DDB4   | DDB3   | DDB2   | DDB1              | DDB0             |
| PORTB: | PORTB7           | PORTB6           | PORTB5 | PORTB4 | PORTB3 | PORTB2 | PORTB1            | PORTB0           |
| PINB:  | PINB7            | PINB6            | PINB5  | PINB4  | PINB3  | PINB2  | PINB1             | PINB0            |
| DDRC:  | -                | DDC6<br>(=0,rst) | DDC5   | DDC4   | DDC3   | DDC2   | DDC1              | DDC0             |
| PORTC: | -                | PORTC6           | PORTC5 | PORTC4 | PORTC3 | PORTC2 | PORTC1            | PORTC0           |
| PINC:  | -                | PINC6            | PINC5  | PINC4  | PINC3  | PINC2  | PINC1             | PINC0            |

The method for setting up and accessing a digital pin is to first set the data direction register for the appropriate pin bank (DDR), then set the PORT register of that bank to set the pin state, then look at the PIN register if you'd like to read the pin. The bank letter (D, B, or C) ends each register's name.

- 1) The DDR registers sets the data direction:
  - If a bit in the DDRx register is 0: set pin to INPUT mode.
  - If a bit in the DDRx register is 1: set pin to OUTPUT mode.
 The three DDR registers are DDRD, DDRB, and DDRC (for Bank D, B, and C, respectively).
- 2) The PORT registers are used to change the pin states:
  - If a bit in the DDR register is 1 (OUTPUT mode), then:
    - If that bit in the PORT register is 0: set pin LOW
    - If that bit in the PORT register is 1: set pin HIGH
  - If a bit in the DDR register is 0 (INPUT mode), then:
    - If that bit in the PORT register is 0: no internal pull-up resistor
    - If that bit in the PORT register is 1: INPUT\_PULLUP mode
 The three PORT registers are PORTD, PORTB, and PORTC.
- 3) The PIN registers are used to read the digital value of the pin. The three PIN registers are PIND, PINB, and PINC.

Logistically, you need to access two registers to set a digital output pin, and three registers to set and read a digital input pin. An example of which registers to access and how to set the bit values for Pin 12 is provided in

Figure 10-1. If a pin is in a different bank, replace the letter B in register and bit names with the pin bank letter.

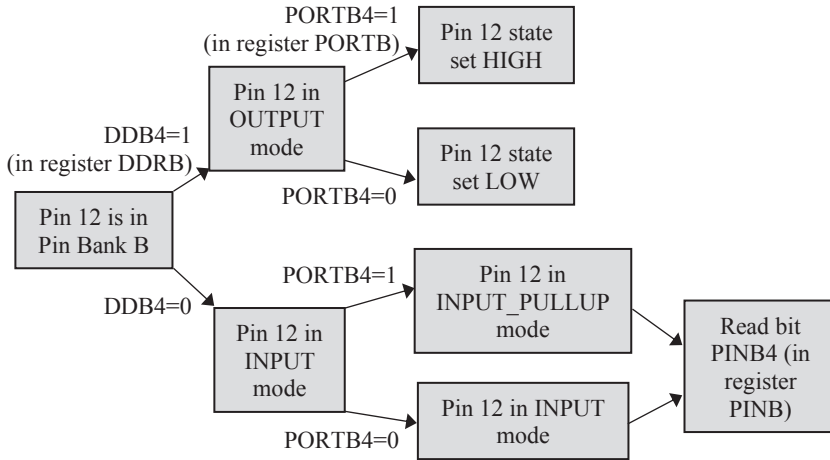


Figure 10-1. Using port registers to set and read digital pin 12 (PB4), in Bank B.

**OUTPUT Mode Example:** This is how you could set Pin 13 (in Bank B) to OUTPUT mode, then change its state to LOW. From Table 10-11, we see that Pin 13 is bit 5 (PB5). Let's try using the `_BV()` function here:

```
DDRB |= _BV(5); //Set pin 13 to OUTPUT
PORTB &= ~_BV(5); //Set pin 13 LOW
```

Alternately, we could write:

```
DDRB |= _BV(DDB5); //Set pin 13 to OUTPUT
PORTB &= ~_BV(PORTB5); //Set pin 13 LOW
```

**INPUT Mode Example:** This is how you could set Pin 4 (in Bank D) to INPUT mode, and then read the state of the pin:

Pin 4 is on Bank D, so the three registers we need to access are `DDRD`, `PORTD`, and `PIND`. From Table 10-11, we see that Pin 4 is the fourth bit in these registers. Let's try using the shifting operators here:

```
DDRD &= ~(1<<4); // set pin 4 to INPUT
PORTD &= ~(1<<4); // no internal pull-up on pin 4
byte pin4state = PIND>>4&1; //read pin 4 state
```

**INPUT\_PULLUP Mode Example:** Here is how you could set Pin 7 to INPUT\_PULLUP mode, and read the state of the pin, using port manipulation:



Pin 7 is on Bank D, so the three registers we need to access are DDRD, PORTD, and PIND. Pin 7 is the first bit in these registers. Let's try using the `cbi()` and `sbi()` commands here, and write our own little function to read a single bit from the register:

```
#define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))
#define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
#define rbi(sfr, bit) (_SFR_BYTE(sfr) >> bit & 1)
cbi(DDRD,DDD7); // set pin 7 to INPUT (clear DDD7)
sbi(PORTD,PORTD7); // set pin 7 internal pull-up
byte pin7state = rbi(PIND,PIND7); //read pin 7
```

If you don't like any of the short forms above, you can always use bit masks the way we first introduced them:

```
DDRD &= ~0b10000000; // set pin 7 to INPUT
PORTD |= 0b10000000; // set pin 7 internal pull-up
```

**Multiple Pins Example:** Now that you see how to set, change, and read a single digital pin, the real savings in port manipulation is the ability to set more than one pin at a time. If you need to set many pins in the same bank, you don't need to set them individually. For example, if you'd like to set Pins 2 to 6 to OUTPUT mode, set Pin 7 to INPUT mode, then set Pins 2-6 HIGH, you could cover those changes using the following two commands:

```
DDRD = 0b01111110; // DDD1=1(tx),DDD0=0(rx)
PORTD |= 0b01111100; // set pins 2-6 HIGH only
```

The equivalent set of commands without port manipulation would be more time and memory consuming, requiring six `pinMode()` statements and five `digitalWrite()` statements.

This approach may look like a lot of effort, but once you get the hang of it, port manipulation is much faster than a `digitalRead()` or `digitalWrite()` function. In fact, sometimes you might need to add a small delay before reading a pin, because the pin state might not have had the chance to change yet when you read the PIN register.

*For more information: <https://www.arduino.cc/en/Reference/PortManipulation>*

## Interrupts

Throughout this text, we have mainly stuck with the `digitalRead()` function to find out the state of a digital input pin. If you'd like to plan an event around a digital pin changing state (for instance, a button push), you can read a pin state inside the `loop()` function, and as long as the `loop()` doesn't take too long, then this strategy works but is analogous to the persistent "are we there yet?" question from a small child on a long car ride.

Worse off, if you have a `delay()` within the loop, `digitalRead()` won't work for you during that delay. Fortunately, microprocessors also come with pins that you can program as *interrupts*. This means that regardless of which part of the sketch your microprocessor is running, if a pin state change happens on an interrupt pin, we can use that to trigger a short, specific user-defined function *immediately*. This special function is called an *Interrupt Service Routine* (ISR).

The ATmega328 microprocessor has two *external interrupts*: one on digital pin 2 (called INT0), and one on digital pin 3 (called INT1). These two pins have this special capability. To illustrate, let's set up a momentary switch on digital pin 2, shown in Figure 10-1.

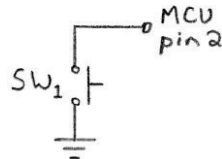


Figure 10-2. Momentary switch connected to INT0 (digital pin 2).

In the following sketch, we will set the state of pin 2 to `INPUT_PULLUP` so that pushing the switch sets pin 2 low, without requiring an external pull-up resistor. That way, the state of pin 2 will be HIGH when SW<sub>1</sub> is open (unpushed), and LOW when SW<sub>1</sub> is closed (pushed). The following sketch declares digital pin 2 as an interrupt pin, and runs the void function `myISR()` when the button is pushed:

```
// Example: Button push to trigger INT0
// variables changed in ISRs should be volatile:
volatile int counter=0;
void setup(){
 Serial.begin(9600);
 pinMode(2,INPUT_PULLUP); // internal pullup pin2
 attachInterrupt(0,myISR,FALLING) ;
 //for the ATmega328: 0=INT0(pin 2), 1=INT1(pin 3)
}
void loop(){
 Serial.print("Interrupted ");
 Serial.print(counter);
 Serial.println(" times so far.");
 delay(1000); // ISRs can run during delays
}

void myISR(){ // Interrupt Service Routine (ISR)
 counter++; // increase the counter
 // more commands can go here
}
```

After the interrupt INT0 is *attached* to the *Interrupt Service Routine* (ISR), whatever you write inside that routine (in this case the one called

myISR) will be executed when pin 2 *falls* from HIGH to LOW (when SW<sub>1</sub> is pushed).

### *Internal (Pin Change) Interrupts*

In addition to the two external interrupts on digital pins 2 and 3, the ATmega328 has internal interrupts, also called *pin change interrupts* on other digital pins (see PCINT pins in Figure A-5, *Arduino Uno Pin-out Diagram*). This feature is more easily accessible by using external libraries such as Mike Schwager's *EnableInterrupt* library, installable through the Arduino IDE Library Manager. (Schwager 2018) We can modify the above sketch to work on any PCINT pin on the Arduino (including the analog pins), making use of the *EnableInterrupt* library:

```
// Example: Pin change interrupt (button on Pin 7)
#include <EnableInterrupt.h>
#define buttonPin 7 // digital pin 7 to read button
// variables changed in ISRs should be volatile:
volatile int counter=0;

void setup(){
 Serial.begin(9600);
 pinMode(buttonPin, INPUT_PULLUP);
 enableInterrupt(buttonPin, myISR, FALLING);
}

void loop(){
 Serial.print("Interrupted ");
 Serial.print(counter);
 Serial.println(" times so far.");
 delay(1000); // ISRs can run during delays
}

void myISR(){ // Interrupt Service Routine (ISR)
 counter++; // increase the counter
 // more commands can go here
}
```

If you try running this sketch, you may notice that the interrupt triggers more than once per button push because it is picking up all the switching noise (bouncing). Since `delay()` won't work inside the ISR, it can be difficult to debounce a pin change with interrupts. The next example sketch provides a way of debouncing with interrupts.

Note that a limitation of a pin change interrupt is that it cannot wake the microprocessor from sleep mode.

### *Never Miss a Button Push Again*

A carefully-crafted ISR means that you never have to worry about missing a button push from your user. The following sketch attaches an ISR to Pin 3. The ISR is triggered when a FALLING state is detected on Pin 3, when the button is pushed. The ISR code sets `buttonState` low. Inside the loop, the value of `buttonState` is tested: if 0, then state of the on-board LED on Pin 13 is toggled with the `bool` variable `ledState`, there is a debouncing delay, then `buttonState` is reset to 1. This means that only the first falling signal will trigger the onboard LED to toggle its state.

There's no `digitalRead()` function needed. Inside the loop, you just check whatever value happens to be in the volatile variable `buttonState`:

```
// Example: Debouncing with interrupts
const byte buttonPin=3; //connect button to pin 3
volatile bool buttonState=1; //0:pushed,1:not pushed
const byte ledPin=13; //for on-board LED
bool ledState=LOW; //for led state

void setup(){
 pinMode(buttonPin, INPUT_PULLUP);
 pinMode(ledPin, OUTPUT); // set ledPin to output
 // attach buttonISR to pin 3, falling:
 attachInterrupt(1,buttonISR,FALLING);
}

void loop(){
 if(buttonState==0){ // if button was pushed
 ledState=!ledState; // toggle led state
 digitalWrite(ledPin,ledState); //write led state
 delay(500); // debounce
 buttonState=1; // reset buttonState
 }
}

void buttonISR(){ // ISR for monitoring button push
 buttonState=0; // set buttonState to 0 (pushed)
}
```

An ISR need not be triggered only when a pin state falls from high to low. The four options of triggering an Arduino Uno ISR are summarized in Table 10-13. Some microprocessors also have a HIGH option.

**Table 10-13. Trigger options for interrupt service routines.**

| <i>Interrupt Trigger Option in <code>attachInterrupt()</code></i> | <i>ISR will run when...</i>                                              |
|-------------------------------------------------------------------|--------------------------------------------------------------------------|
| LOW                                                               | the interrupt pin state is LOW.                                          |
| HIGH (setting not available for the ATmega328)                    | the interrupt pin state is HIGH.                                         |
| CHANGE                                                            | the interrupt pin state changes (from LOW to HIGH, or from HIGH to LOW). |
| FALLING                                                           | the interrupt pin state changes (falls) from HIGH to LOW.                |
| RISING                                                            | the interrupt pin state changes (rises) from LOW to HIGH.                |

A big advantage of using an interrupt is the blazing response time. The microprocessor reacts as quickly as possible to the pin state change, without having to repeatedly use the `digitalRead()` command in a loop, continuously asking an input pin for its state. For example, you could use an interrupt to trigger taking a photo of a bullet in mid-air, or to time a bullet travelling through a calibrated distance, to calculate its velocity (See Section 9, *Measuring Time Duration with Interrupts*).

The interrupt routine will run regardless of any `delay()` or other commands running in the loop. So their triggering is... bulletproof!

### *Rules for Writing an Interrupt Service Routine*

There are a few recommendations and restrictions on how to write an ISR: (Gammon 2012)

- The ISR is a special kind of function that must be `void`. It shouldn't return a value, and you can't pass values into the ISR as arguments.
- No serial commands should go inside the ISR, unless there is zero chance of the ISR running on top of itself (this happens when the ISR is triggered before the last ISR has finished running).
- Functions that rely on timers won't work inside the ISR, so don't use commands like `delay()`. The `delayMicroseconds()` command will work inside an ISR. The `millis()` command won't increment inside the ISR, so don't try to call it more than once.
- All variables you would like to change within the ISR should be declared in global space, and have the word `volatile` in front of their declaration (e.g. `volatile int counter=0;`). The

command `volatile` protects the variable from being removed during compiling, so that it can be accessed by the ISR.

- The ISR function should be as short as possible.

External interrupts can be used to wake a microprocessor up from sleep mode. This is something you need to take into consideration if you plan on making your circuit battery powered, because putting a microprocessor in sleep mode will prolong your battery life considerably (see Section 10, *Sleep Mode* for more details).

You can attach the ATmega328's two external interrupts to different ISRs, each with different triggers, and have them working simultaneously. The microprocessor will queue them if they are triggered too close together. If you would like to turn off an interrupt in the middle of a sketch so the ISR will no longer be triggered, you can use the `detachInterrupt(pin)` command. For example, the command to detach INT0 would be:

```
detachInterrupt(0);
```

There are two more interrupt commands you can use, which operate on all interrupts:

```
cli(); // disables all attached interrupts
```

This command prevents all interrupts from triggering. You want to do this before running code that you are worried might get interrupted. It is like a do-not-disturb mode for interrupts. To undo this quiet period, use the command:

```
sei(); // re-enables all attached interrupts
```

We will use both these commands in the next section. In the Arduino IDE language, the following two commands are equivalent: `interrupts()` and `noInterrupts()`.

*For more information: <https://www.arduino.cc/en/Reference/AttachInterrupt>*

## Customized Frequencies for PWM

Digital pins 3, 9, 10, and 11 on the Arduino Uno are able to generate PWM frequencies at about 490 Hz by default, and pins 5 and 6 generate PWM frequencies at about 980 Hz. (Mellai 2017b) These frequencies can be customized, by altering the settings of the timers that control them. Microprocessors have built-in timers that can be manually set, and can work in different modes. The ATmega328 microprocessor has three timers: Timer 0, Timer 1, and Timer 2. **Timer 0** controls pins 5 and 6, along with the Uno's time-related functions, e.g. `delay()`. **Timer 1** controls pins 9 and 10. **Timer 2** controls pins 3 and 11. It's well worth discussing the structure of these

timers in relation to how they work, because that will help you set and control them to do your bidding. We will deal with each timer separately. Example code will be provided for how to set each timer.

### *Timer 0*

Timer 0 is an 8-bit timer which functions as a counter, that counts from 0 to 255. This timer swings into action when you ask the Uno to set a PWM signal on pins 5 or 6 (or both). In **fast PWM mode**, you can adjust the frequency of the PWM signal by changing the settings of Timer 0. Use the following code in your sketch to set a custom frequency on Pin 5:

```
//Fast PWM on Pin 5: (e.g. 390 kHz)
//Formula: frequency=fclk/((OCR0A+1)*N)
pinMode(5, OUTPUT);
TCCR0A=_BV(COM0A1)|_BV(COM0B1)|_BV(WGM01)|_BV(WGM00);
TCCR0B=_BV(WGM02); //fast PWM mode(WGM02,WGM01,WGM00)
//uncomment for your desired prescaler:
TCCR0B|=_BV(CS00); //N=1
//TCCR0B|=_BV(CS01); //N=8
//TCCR0B|=_BV(CS01)|_BV(CS00); //N=64
//TCCR0B|=_BV(CS02); //N=256
//TCCR0B|=_BV(CS02)|_BV(CS00); //N=1024
OCR0A=40; //counter limit: 255
OCR0B=20; //pin 5 duty cycle=OCR0B/OCR0A (lim:OCR0A)
```

The commands in the above code set the *Timer Counter/Control Registers* that control Timer 0 (TCCR0A and TCCR0B), as well as the *output compare registers* OCR0A and OCR0B, which tell Timer 0 when to reset. For a more in-depth discussion about the structure of these registers, have a look at the ATmega328 datasheet. (Atmel Corporation 2016) The prescaler makes every increment of the timer increase after a *multiple* of ticks from the processor's clock instead of after each tick, which makes the timer count up more slowly. Think about a prescaler of 2 like telling the second hand on your watch to tick once every *two seconds*, instead of once per second. This means your watch would now take two hours to tell you that one hour elapsed. The Uno's microprocessor speed is 16 MHz, which works out to 62.5 nanoseconds per tick. With a prescaler of 1, when Timer 0 counts to 10, 625 nanoseconds have passed. However, if you set the prescaler of Timer 0 to 8, then  $8 \times 10 \times 62.5$  nanoseconds have passed, or 5 microseconds. This allows you to use Timer 0 over a longer range before it resets (when it gets to 255, it will automatically reset back to 0, since it is only an 8-bit timer).

Setting OCR0A in the above code will determine the frequency of the PWM cycle, according to the formula:

$$f = \frac{f_{clk}}{N \times (OCR0A + 1)}$$

where  $f_{clk}=16,000,000$  Hz and N is the prescaler you selected by uncommenting the appropriate line in the above code. The value of OCR0A should not exceed 255, the maximum value of Timer 0.

The duty cycle for this signal will be defined by OCR0B and calculated by the formula: duty cycle=OCR0B/OCR0A. The value of OCR0B should not exceed the value of OCR0A. The example code above creates a signal on Pin 5 with a frequency of 390 kHz and a duty cycle of 50%. Here is a table of example frequencies you can generate by changing OCR0A:

**Table 10-14. Fast mode PWM frequencies in Hz for Pin 5 (Timer 0).**

| OCR0A | N=1       | N=8       | N=64    | N=256  | N=1024 |
|-------|-----------|-----------|---------|--------|--------|
| 1     | 8,000,000 | 1,000,000 | 125,000 | 31,250 | 7,813  |
| 5     | 2,666,667 | 333,333   | 41,667  | 10,417 | 2,604  |
| 10    | 1,454,545 | 181,818   | 22,727  | 5,682  | 1,420  |
| 20    | 761,905   | 95,238    | 11,905  | 2,976  | 744    |
| 40    | 390,244   | 48,780    | 6,098   | 1,524  | 381    |
| 60    | 262,295   | 32,787    | 4,098   | 1,025  | 256    |
| 80    | 197,531   | 24,691    | 3,086   | 772    | 193    |
| 100   | 158,416   | 19,802    | 2,475   | 619    | 155    |
| 120   | 132,231   | 16,529    | 2,066   | 517    | 129    |
| 140   | 113,475   | 14,184    | 1,773   | 443    | 111    |
| 160   | 99,379    | 12,422    | 1,553   | 388    | 97     |
| 180   | 88,398    | 11,050    | 1,381   | 345    | 86     |
| 200   | 79,602    | 9,950     | 1,244   | 311    | 78     |
| 255   | 62,500    | 7,813     | 977     | 244    | 61     |

From Table 10-14, you can see that a PWM frequency can be generated anywhere from 8 MHz all the way down to 61 Hz (about once per second). For a Fast PWM signal on Pin 6 only, here is example code:

```
//Fast PWM on Pin 6: (e.g. 727 kHz)
//Formula: frequency=fclk/((OCR0A+1)*2*N)
//Duty cycle fixed at 50% in this mode.
pinMode(6,OUTPUT);
TCCR0A=_BV(COM0A0)|_BV(WGM01)|_BV(WGM00);
TCCR0B=_BV(WGM02); //fast PWM mode(WGM02,WGM01,WGM00)
//uncomment for your desired prescaler:
TCCR0B|=_BV(CS00); //N=1
//TCCR0B|=_BV(CS01); //N=8
```



```
//TCCR0B|=_BV(CS01)|_BV(CS00); //N=64
//TCCR0B|=_BV(CS02); //N=256
//TCCR0B|=_BV(CS02)|_BV(CS00); //N=1024
OCR0A=10; // counter limit:255
```

This code generates a 727 kHz square wave on Pin 6 with a 50% duty cycle. The OCR0A in this mode sets the frequency according to Table 10-15:

**Table 10-15. Fast mode PWM frequencies in Hz for Pin 6 (Timer 0).**

| OCR0A | N=1       | N=8     | N=64   | N=256  | N=1024 |
|-------|-----------|---------|--------|--------|--------|
| 1     | 4,000,000 | 500,000 | 62,500 | 15,625 | 3,906  |
| 5     | 1,333,333 | 166,667 | 20,833 | 5,208  | 1,302  |
| 10    | 727,273   | 90,909  | 11,364 | 2,841  | 710    |
| 20    | 380,952   | 47,619  | 5,952  | 1,488  | 372    |
| 40    | 195,122   | 24,390  | 3,049  | 762    | 191    |
| 60    | 131,148   | 16,393  | 2,049  | 512    | 128    |
| 80    | 98,765    | 12,346  | 1,543  | 386    | 96     |
| 100   | 79,208    | 9,901   | 1,238  | 309    | 77     |
| 120   | 66,116    | 8,264   | 1,033  | 258    | 65     |
| 140   | 56,738    | 7,092   | 887    | 222    | 55     |
| 160   | 49,689    | 6,211   | 776    | 194    | 49     |
| 180   | 44,199    | 5,525   | 691    | 173    | 43     |
| 200   | 39,801    | 4,975   | 622    | 155    | 39     |
| 255   | 31,250    | 3,906   | 488    | 122    | 31     |

If you wish to generate fast PWM signals for both Pins 5 and 6 at the same time, the following code could be used:

```
//Fast PWM on both Pins 5 and 6 (e.g. 5,6: 62.5 kHz)
//In this mode, timer 0 always counts to 255.
//Formula: frequency=fclk/(256*N). This formula has
//more limited frequency settings.
pinMode(5,OUTPUT);
pinMode(6,OUTPUT);
TCCR0A=_BV(COM0A1)|_BV(COM0A0)|_BV(COM0B1)|_BV(COM0B0)|
_BV(WGM01)|_BV(WGM00); // inverting mode, fast PWM
//uncomment for your desired prescaler:
TCCR0B=_BV(CS00); // N=1 (62.5 kHz)
//TCCR0B=_BV(CS01); // N=8 (7.8 kHz)
//TCCR0B=_BV(CS01)|_BV(CS00); // N=64 (976 Hz)
//TCCR0B=_BV(CS02); // N=256 (244 Hz)
//TCCR0B=_BV(CS02)|_BV(CS00); // N=1024 (61 Hz)
OCR0A=50; //pin 6 duty cycle=(255-OCR0A)/255 (lim:255)
OCR0B=128; //pin 5 duty cycle=(255-OCR0B)/255 (lim:255)
```

When both pins are used simultaneously, the counter will run to the end then reset, and the only way to change the frequency is by changing the prescaler. The duty cycles of Pins 5 and 6 are set using OCR0B and OCR0A, respectively. The example code generates a 62.5 kHz signal on Pin 6 with a duty cycle of 80%, and a 62.5 kHz signal on Pin 5 with a duty cycle of 50%.

A disadvantage of using Timer 0 is that the Arduino's time functions (e.g. `delay()`) will be affected, and time measurements will return wonky results.

### *Timer 1*

Timer 1 controls PWM timing on Pins 9 and 10. It is unrelated to the Uno IDE's timer functions, so you can safely tinker with its settings without the worry of impacting commands like `delay()` in your sketch. It does get called by the servo library, so using Timer 1 may conflict with servo control. Timer 1 is a 16-bit timer, meaning that it can count as high as 65535 (resulting in much slower programmable frequencies possible). It is controlled with registers TCCR1A and TCCR1B to set the mode and prescaler values. The following example code sets Pin 10 to 390 KHz with a 25% duty cycle using Timer 1's comparison registers, OCR1A and OCR1B:

```
//Fast PWM on Pin 10: (e.g. 390kHz, 25%duty)
//Formula: frequency=fclk/((OCR1A+1)*N)
pinMode(10, OUTPUT);
TCCR1A=_BV(COM1A1)|_BV(COM1B1)|_BV(WGM11)|_BV(WGM10);
//fast PWM, 10bit resolution
TCCR1B=_BV(WGM13)|_BV(WGM12); // CTC mode
//uncomment for your desired prescaler:
TCCR1B|=_BV(CS10); //N=1
//TCCR1B|=_BV(CS11); //N=8
//TCCR1B|=_BV(CS11)|_BV(CS10); //N=64
//TCCR1B|=_BV(CS12); //N=256
//TCCR1B|=_BV(CS12)|_BV(CS10); //N=1024
OCR1A=40; //counter limit: 65535
OCR1B=10; //pin 10 duty cycle=OCR1B/OCR1A (lim:OCR1A)
```

The following table illustrates the range of frequencies possible when setting different prescaler values for Timer 1 with OCR1A.

**Table 10-16. Fast mode PWM frequencies in Hz for Pin 10 (Timer 1).**

| OCR1A | N=1         | N=8         | N=64      | N=256    | N=1024  |
|-------|-------------|-------------|-----------|----------|---------|
| 1     | 8,000,000.0 | 1,000,000.0 | 125,000.0 | 31,250.0 | 7,812.5 |
| 10    | 1,454,545.5 | 181,818.2   | 22,727.3  | 5,681.8  | 1,420.5 |
| 50    | 313,725.5   | 39,215.7    | 4,902.0   | 1,225.5  | 306.4   |
| 100   | 158,415.8   | 19,802.0    | 2,475.2   | 618.8    | 154.7   |
| 500   | 31,936.1    | 3,992.0     | 499.0     | 124.8    | 31.2    |
| 1000  | 15,984.0    | 1,998.0     | 249.8     | 62.4     | 15.6    |
| 5000  | 3,199.4     | 399.9       | 50.0      | 12.5     | 3.1     |
| 10000 | 1,599.8     | 200.0       | 25.0      | 6.2      | 1.6     |
| 50000 | 320.0       | 40.0        | 5.0       | 1.2      | 0.3     |
| 65535 | 244.1       | 30.5        | 3.8       | 1.0      | 0.2     |

The following code sets a 24.4 kHz PWM signal on Pin 9:

```
//Fast PWM on Pin 9: (e.g. 24.4 kHz)
//Formula: frequency=fclk/((OCR1A+1)*2*N)
//Duty cycle fixed at 50% in this mode.
pinMode(9, OUTPUT);
TCCR1A=_BV(COM1A0) | _BV(WGM11) | _BV(WGM10);
TCCR1B=_BV(WGM13) | _BV(WGM12); //fast PWM mode
//uncomment for your desired prescaler:
//TCCR1B|=_BV(CS10); //N=1
TCCR1B|=_BV(CS11); //N=8
//TCCR1B|=_BV(CS11) | _BV(CS10); //N=64
//TCCR1B|=_BV(CS12); //N=256
//TCCR1B|=_BV(CS12) | _BV(CS10); //N=1024
OCR1A=40; // counter limit: 65535
```

Like Pin 6, the duty cycle in this mode is limited to 50%. Setting N and OCR1A will change the frequency of the signal according to Table 10-17.

**Table 10-17. Fast mode PWM frequencies in Hz for Pin 9 (Timer 1).**

| OCR1A | N=1         | N=8       | N=64     | N=256    | N=1024  |
|-------|-------------|-----------|----------|----------|---------|
| 1     | 4,000,000.0 | 500,000.0 | 62,500.0 | 15,625.0 | 3,906.3 |
| 10    | 727,272.7   | 90,909.1  | 11,363.6 | 2,840.9  | 710.2   |
| 50    | 156,862.7   | 19,607.8  | 2,451.0  | 612.7    | 153.2   |
| 100   | 79,207.9    | 9,901.0   | 1,237.6  | 309.4    | 77.4    |
| 500   | 15,968.1    | 1,996.0   | 249.5    | 62.4     | 15.6    |
| 1000  | 7,992.0     | 999.0     | 124.9    | 31.2     | 7.8     |
| 5000  | 1,599.7     | 200.0     | 25.0     | 6.2      | 1.6     |
| 10000 | 799.9       | 100.0     | 12.5     | 3.1      | 0.8     |
| 50000 | 160.0       | 20.0      | 2.5      | 0.6      | 0.2     |
| 65535 | 122.1       | 15.3      | 1.9      | 0.5      | 0.1     |

Generating a signal on both Pins 9 and 10 simultaneously will limit the range of frequencies selectable, as only the prescaler value will change the output frequency. Both pins will have the same frequency in this mode. However, by setting OCR1A and OCR1B, you can set the duty cycle of Pins 9 and 10, respectively. The following code would generate a signal on Pin 9 with a frequency of 15.6 kHz and a duty cycle of 50%, and a signal on Pin 10 with the same frequency (15.6 kHz) and a duty cycle of about 10%.

```
//Fast PWM on Pins 9 and 10: (e.g. 15.6 kHz)
//Formula: frequency=fclk/(N*1024). This formula has
//more limited frequency settings.
pinMode(9, OUTPUT);
pinMode(10, OUTPUT);
TCCR1A=_BV(COM1A1)|_BV(COM1B1)|_BV(WGM11)|_BV(WGM10);
TCCR1B=_BV(WGM12); //Fast PWM mode
//uncomment for your desired prescaler:
TCCR1B|=_BV(CS10); //N=1 (freq=15625 Hz)
//TCCR1B|=_BV(CS11); //N=8 (freq=1953 Hz)
//TCCR1B|=_BV(CS11)|_BV(CS10); //N=64 (freq=244 Hz)
//TCCR1B|=_BV(CS12); //N=256 (freq=61 Hz)
//TCCR1B|=_BV(CS12)|_BV(CS10); //N=1024 (freq=15 Hz)
OCR1A=512; //pin 9 duty cycle=OCR1A/1024 (lim:1024)
OCR1B=102; //pin 10 duty cycle=OCR1B/1024 (lim:1024)
```

## *Timer 2*

Timer 2 controls PWM timing on Pins 3 and 11. It is also unrelated to the Uno IDE's timer functions, so it will not impact time functions in your sketch. It does however control the `tone()` function for generating piezo beeps. If you use Timer 2, it will conflict with generating tones. Timer 2 is an 8-bit timer, which means that it can count as high as 255. It is controlled by setting the registers TCCR2A and TCCR2B to set the timer mode and prescaler values, respectively. The following example code generates a 390 kHz PWM signal on Pin 3, with a duty cycle of 50% using Timer 2's comparison registers, OCR2A and OCR2B:

```
//Fast PWM on Pin 3: (e.g. 390 kHz)
//Formula: frequency=fclk/((OCR2A+1)*N)
pinMode(3, OUTPUT);
TCCR2A=_BV(COM2A1)|_BV(COM2B1)|_BV(WGM21)|_BV(WGM20);
TCCR2B=_BV(WGM22); // waveform generation mode
//uncomment for your desired prescaler:
TCCR2B|=_BV(CS20); //N=1
//TCCR2B|=_BV(CS21); //N=8
//TCCR2B|=_BV(CS21)|_BV(CS20); //N=32
//TCCR2B|=_BV(CS22); //N=64
```

```
//TCCR2B|=_BV(CS22)|_BV(CS20); //N=128
//TCCR2B|=_BV(CS22)|_BV(CS21); //N=256
//TCCR2B|=_BV(CS22)|_BV(CS21)|_BV(CS20); //N=1024
OCR2A=40; //counter limit: 255
OCR2B=20; //pin 3 duty cycle=OCR2B/OCR2A (lim:OCR2A)
```

The duty cycle is set by the ratio OCR2B/OCR2A. Changing the prescaler and values of OCR2A result in the following ranges of output frequencies:

**Table 10-18. Fast mode PWM frequencies in Hz for Pin 3 (Timer 2).**

| OCR2A | N=1       | N=8       | N=32    | N=64    | N=128  | N=256  | N=1024 |
|-------|-----------|-----------|---------|---------|--------|--------|--------|
| 1     | 8,000,000 | 1,000,000 | 250,000 | 125,000 | 62,500 | 31,250 | 7,813  |
| 5     | 2,666,667 | 333,333   | 83,333  | 41,667  | 20,833 | 10,417 | 2,604  |
| 10    | 1,454,545 | 181,818   | 45,455  | 22,727  | 11,364 | 5,682  | 1,420  |
| 20    | 761,905   | 95,238    | 23,810  | 11,905  | 5,952  | 2,976  | 744    |
| 40    | 390,244   | 48,780    | 12,195  | 6,098   | 3,049  | 1,524  | 381    |
| 60    | 262,295   | 32,787    | 8,197   | 4,098   | 2,049  | 1,025  | 256    |
| 80    | 197,531   | 24,691    | 6,173   | 3,086   | 1,543  | 772    | 193    |
| 100   | 158,416   | 19,802    | 4,950   | 2,475   | 1,238  | 619    | 155    |
| 120   | 132,231   | 16,529    | 4,132   | 2,066   | 1,033  | 517    | 129    |
| 140   | 113,475   | 14,184    | 3,546   | 1,773   | 887    | 443    | 111    |
| 160   | 99,379    | 12,422    | 3,106   | 1,553   | 776    | 388    | 97     |
| 180   | 88,398    | 11,050    | 2,762   | 1,381   | 691    | 345    | 86     |
| 200   | 79,602    | 9,950     | 2,488   | 1,244   | 622    | 311    | 78     |
| 255   | 62,500    | 7,813     | 1,953   | 977     | 488    | 244    | 61     |

The following example code could be used to generate a signal only on Pin 11. This mode is limited to a duty cycle of 50%.

```
//Fast PWM on Pin 11: (e.g. 22.7 kHz)
//Formula: frequency=fclk/((OCR2A+1)*2*N)
//Duty cycle fixed at 50% in this mode.
pinMode(11,OUTPUT);
TCCR2A=_BV(COM2A0)|_BV(WGM21)|_BV(WGM20);
TCCR2B=_BV(WGM22); //fast PWM mode
//uncomment for your desired prescaler:
//TCCR2B|_BV(CS20); //N=1
//TCCR2B|=_BV(CS21); //N=8
TCCR2B|=_BV(CS21)|_BV(CS20); //N=32
//TCCR2B|=_BV(CS22); //N=64
//TCCR2B|=_BV(CS20); //N=128
//TCCR2B|=_BV(CS21); //N=256
//TCCR2B|=_BV(CS21)|_BV(CS20); //N=1024
OCR2A=10; // counter limit: 255
```

The frequency is set using OCR2A. The example code above generates a signal on Pin 11 with a frequency of 22.7 kHz and a duty cycle of 50%.

Changing the prescaler and values of OCR2A result in the following ranges of output frequencies:

**Table 10-19. Fast mode PWM frequencies in Hz for Pin 11 (Timer 2).**

| OCR2A | N=1       | N=8     | N=32    | N=64   | N=128  | N=256  | N=1024 |
|-------|-----------|---------|---------|--------|--------|--------|--------|
| 1     | 4,000,000 | 500,000 | 125,000 | 62,500 | 31,250 | 15,625 | 3,906  |
| 5     | 1,333,333 | 166,667 | 41,667  | 20,833 | 10,417 | 5,208  | 1,302  |
| 10    | 727,273   | 90,909  | 22,727  | 11,364 | 5,682  | 2,841  | 710    |
| 20    | 380,952   | 47,619  | 11,905  | 5,952  | 2,976  | 1,488  | 372    |
| 40    | 195,122   | 24,390  | 6,098   | 3,049  | 1,524  | 762    | 191    |
| 60    | 131,148   | 16,393  | 4,098   | 2,049  | 1,025  | 512    | 128    |
| 80    | 98,765    | 12,346  | 3,086   | 1,543  | 772    | 386    | 96     |
| 100   | 79,208    | 9,901   | 2,475   | 1,238  | 619    | 309    | 77     |
| 120   | 66,116    | 8,264   | 2,066   | 1,033  | 517    | 258    | 65     |
| 140   | 56,738    | 7,092   | 1,773   | 887    | 443    | 222    | 55     |
| 160   | 49,689    | 6,211   | 1,553   | 776    | 388    | 194    | 49     |
| 180   | 44,199    | 5,525   | 1,381   | 691    | 345    | 173    | 43     |
| 200   | 39,801    | 4,975   | 1,244   | 622    | 311    | 155    | 39     |
| 255   | 31,250    | 3,906   | 977     | 488    | 244    | 122    | 31     |

Lastly, an example sketch for generating 7.8 kHz PWM signals on both Pins 3 and 11 using Timer 2 is provided here (both at 50% duty cycle):

```
//Fast PWM on Pins 3 and 11: (e.g. 7.8 kHz)
//Formula: frequency=fclk/(256*N). This formula has
//more limited frequency settings.
pinMode(3, OUTPUT);
pinMode(11, OUTPUT);
TCCR2A= _BV(COM2A1) | _BV(COM2A0) | _BV(COM2B1) | _BV(COM2B0) |
_BV(WGM21) | _BV(WGM20); //fast PWM mode
//uncomment for your desired prescaler:
//TCCR2B= _BV(CS20); //N=1 (62.5 kHz)
TCCR2B= _BV(CS21); //N=8 (7.8 kHz)
//TCCR2B= _BV(CS21) | _BV(CS20); //N=32 (1.953 kHz)
//TCCR2B= _BV(CS22); // N=64 (976.6 Hz)
//TCCR2B= _BV(CS22) | _BV(CS20); //N=128 (488.3 Hz)
//TCCR2B= _BV(CS22) | _BV(CS21); //N=256 (244.1 Hz)
//TCCR2B= _BV(CS22) | _BV(CS21) | _BV(CS20); //N=1024 (61 Hz)
OCR2A=128; //pin 11 duty cycle=(255-OCR2A)/255 (lim:255)
OCR2B=128; //pin 3 duty cycle=(255-OCR2B)/255 (lim:255)
```

Timer 0, Timer 1, and Timer 2 are very flexible in terms of their output frequencies and duty cycles. Being able to manually control them can be helpful in many projects. You can also consider the 555 timer for this task (See *Using a 555 Timer as an External Clock* in the appendix).

*For more information: <https://playground.arduino.cc/Code/FastPWM>*

## Timing your Interrupt Service Routines with CTC Mode

Now that you know how to set prescaler values and output compare registers for the ATmega328's three timers, you can use these timers not only to generate specific PWM frequencies, but amazingly you can use them to run interrupt service routines at a regular time intervals, regardless of whatever else is going on in your sketch (e.g. a `delay()` command inside the `loop()` function). You can do this by taking advantage of *CTC mode* (Clear Timer on Compare). You just need to follow the *Rules for Writing an Interrupt Service Routine*. Let's convert a simple sketch that reads an analog pin every second, from a `delay()` statement to a timed-interrupt strategy. Our starting sketch is:

```
// analog read once per second (with delays)
int reading=0; // to store analog reading
void setup(){
 Serial.begin(9600);
}

void loop(){
 reading=analogRead(A0);
 Serial.println(reading);
 delay(1000);
}
```

The microprocessor spends most of the time in this sketch locked in the `delay()` statement waiting, and unable to run other commands. Instead, let's try programming Timer 1 to schedule an `analogRead()` every second (a sampling frequency of 1 Hz):

```
// analog read once per second (with Timer 1)
// variables changed in ISRs should be volatile:
volatile int reading=0; // to store analog reading
void setup(){
 Serial.begin(9600);
 //To set Timer 1 interrupt at 1Hz:
 cli(); //disable interrupts
 TCCR1A=0; //clear timer control register A
 TCCR1B=0; //clear timer control register B
 TCNT1=0; //clear timer counter 1
 TCCR1B=_BV(WGM12); //CTC mode
 //uncomment for your desired prescaler:
 //TCCR1B|=_BV(CS10); //N=1
 //TCCR1B|=_BV(CS11); //N=8
 //TCCR1B|=_BV(CS11)|_BV(CS10); //N=64
 TCCR1B|=_BV(CS12); //N=256
 //TCCR1B|=_BV(CS12)|_BV(CS10); //N=1024
```

```

TIMSK1|=_BV(OCIE1A); //enable timer compare interrupt
OCR1A=62499; // OCR1A=(fclk/(N*frequency))-1
sei(); //enable interrupts
}
void loop(){
 Serial.println(reading);
}

ISR(TIMER1_COMPA_vect){ // Timer 1 interrupt routine
 reading=analogRead(A0); // ISR commands go in here.
}

```

In this sketch, the volatile integer `reading` is updated using the ISR attached to Timer 1. (RobotFreak 2011) This frees up the microprocessor for other commands instead of being stuck in a `delay()` statement. The value needed for OCR1A is calculated using the formula:

$$frequency = \frac{f_{clk}}{N \times (OCRxA + 1)}$$

or, rearranged:

$$OCRxA = \frac{f_{clk}}{N \times frequency} - 1$$

To set other frequencies using Timer 1, use the above equation, or consult the chart for OCR1A and prescaler values in Table 10-16. The slowest programmable frequency for Timer 1 is 0.2 Hz. If you would like a routine to run less often (every *n* seconds), see *TimedISR\_N.ino* in the appendix. This sketch waits a multiple of 1-second cycles before running your required code. If readings are infrequent, scheduling the analog reading using the `millis()` function is a better approach (see *Using millis() Instead of delay()* in Section 8).

Since Timer 2 is an 8-bit timer, the largest prescaler value ( $N=1024$ ) combined with the largest value possible for OCR2A (255) yields a lowest possible frequency of 61 Hz, so Timer 2 would not be a good choice for the above sketch. Here is an example sketch using Timer 2 that reads an analog pin at 443 Hz, and lights up the onboard LED if the analog reading is greater than 500:

```

// Reading an analog pin using Timer 2 (at 443 Hz)
volatile int reading=0; // to store analog reading
byte ledPin=13; // on-board LED
void setup(){
 pinMode(ledPin,OUTPUT);
 //To set Timer 2 interrupt at 443Hz:
 cli(); //stop interrupts
 TCCR2A=0; //clear timer control register A

```



```

TCCR2B=0; //clear timer control register B
TCNT2=0; //set counter to 0
TCCR2A=_BV(WGM21); //CTC mode
//uncomment for your desired prescaler:
//TCCR2B=_BV(CS20); //N=1
//TCCR2B=_BV(CS21); //N=8
//TCCR2B=_BV(CS21)|_BV(CS20); //N=32
//TCCR2B=_BV(CS22); //N=64
//TCCR2B=_BV(CS22)|_BV(CS20); //N=128
TCCR2B=_BV(CS22)|_BV(CS21); //N=256
//TCCR2B=_BV(CS22)|_BV(CS21)|_BV(CS20); //N=1024
OCR2A=140; //OCR1A=(fclk/(N*frequency))-1
TIMSK2|=_BV(OCIE2A); //enable timer compare
//interrupt
sei(); //enable interrupts
}

void loop(){
// main loop is free to do other things
// Pin A1 will be read at 443 Hz
}
ISR(TIMER2_COMPA_vect){ // Timer 2 interrupt routine
reading=analogRead(A1);
if(reading>500){
digitalWrite(ledPin,HIGH);
}else{
digitalWrite(ledPin,LOW);
}
}
}

```

To set other frequencies using Timer 2, use the equation:

$$OCR2A = \frac{f_{clk}}{N \times frequency} - 1$$

or consult the chart for OCR2A and prescaler values listed in Table 10-18.

Timer 0 is also an 8-bit timer, but has a larger prescaler available (N=1024), so the slowest frequency that can be set with Timer 0 is:

$$frequency = \frac{16,000,000}{1024 \times (255 + 1)} = 61 \text{ Hz}$$

Let's write a simple sketch to set a timed ISR using Timer 0 that will `digitalRead()` a button pin at a sampling frequency of 61 Hz, and change the state of the onboard LED pin accordingly.

```

// Reading a digital pin using Timer 0
byte ledPin=13; // on-board LED
byte buttonPin=2; // digital button on Pin 2
volatile bool buttonState=HIGH; // to hold button state

```

```

void setup() {
 pinMode(ledPin, OUTPUT);
 pinMode(buttonPin, INPUT_PULLUP);
 //To set Timer 0 interrupt at 61Hz:
 cli(); //stop interrupts
 TCCR0A=0; //clear timer control register A
 TCCR0B=0; //clear timer control register B
 TCNT0=0; //set counter to 0
 TCCR0A=_BV(WGM01); // CTC mode
 //uncomment for your desired prescaler:
 //TCCR0B=_BV(CS00); //N=1
 //TCCR0B=_BV(CS01); //N=8
 //TCCR0B=_BV(CS01)|_BV(CS00); //N=64
 //TCCR0B=_BV(CS02); //N=256
 TCCR0B=_BV(CS02)|_BV(CS00); //N=1024
 OCR0A=255; // OCR0A=(fclk/(N*frequency))-1
 TIMSK0|=_BV(OCIE0A); //enable timer compare interrupt
 sei(); //enable interrupts
}

void loop(){
 //main loop is free to do other things
 //button Pin will be read at 61 Hz (every ~16 msec)
}

ISR(TIMER0_COMPA_vect){ // Timer 0 interrupt routine
 buttonState=digitalRead(buttonPin); // read buttonPin
 digitalWrite(ledPin,buttonState); // write to LEDpin
}

```

To set other frequencies using Timer 0, use the equation:

$$OCR0A = \frac{f_{clk}}{N \times frequency} - 1$$

or consult the chart for OCR0A values listed in Table 10-14. Recall that Timer 0 controls time functions like `delay()`, so these will not work as expected in the rest of your sketch if attached to an ISR. In addition, other functions that use these timers like `analogWrite()` will not work properly. If you must use these functions, disable your interrupts with `cli()` before running them, and re-enable with `sei()` after they are finished running.

*For more information:*

<https://www.instructables.com/id/Arduino-Timer-Interrupts/>

## Sleep Mode

Microprocessors burn up lots of energy just waiting for input. This energy can be conserved by using a special ability of the MCU to go into *sleep mode*. If you are running a circuit on battery power, sleep mode is essential. Even if your circuit is powered using a wall adapter, saving power is less wasteful, and kinder to the environment.

Microprocessors have their own specific set of commands to enter sleep mode. We will cover two ways of waking up from sleep mode: 1) by a pin state change attached to an external interrupt, and 2) by a watchdog timer routine.

### *Wake on Pin Change*

The easiest way to access sleep mode through an interrupt is by using the *AVR library*, already installed in the Arduino IDE. (Boellmann 2016) The following sketch illustrates how you can attach the external interrupt INT0 (Pin 2), and put the microprocessor to sleep until a button wired to Pin 2 is pushed. After waking up, the microprocessor can run a set of commands, then go back to sleep again, thus extending battery life. (Macsimski 2006)

```
// Example: Wake on button push
#include <avr/sleep.h>
#include <avr/power.h>
#include <avr/interrupt.h>
byte buttonPin=2;

void setup(){
 Serial.begin(9600);
 pinMode(buttonPin, INPUT_PULLUP);
}

void loop(){
 Serial.println("Sleeping until button push.");
 sleep(); // run the sleep routine
 Serial.println("Ready to do your bidding!");
 delay(1000);
}

void sleep(){
 // Two options for ATmega328: 0 (Pin 2) or 1 (Pin 3)
 attachInterrupt(0, onWake, FALLING);
 delay(100); // allow serial commands to finish
 set_sleep_mode(SLEEP_MODE_PWR_DOWN);
 /* set_sleep_mode() options:
```

```

 SLEEP_MODE_IDLE (least power savings)
 SLEEP_MODE_ADC
 SLEEP_MODE_PWR_SAVE
 SLEEP_MODE_STANDBY
 SLEEP_MODE_PWR_DOWN (most power savings) */
 sleep_enable();
 sleep_mode(); // MCU will sleep here
 sleep_disable(); // MCU will wake up here
 detachInterrupt(0); // detach INT0 while awake
}

void onWake(){ // ISR
 // Commands could go here. If MCU just needs to
 // wake up, leave this empty.
}

```

With the on-board voltage regulator, FTDI chip, and all the other components of the Arduino Uno running, putting the MCU to sleep doesn't save a much power unless you move off the Uno platform and use the microprocessor alone (see *Bareduino – Running the ATmega328 Alone*). With the ATmega328p running off the Uno platform, current consumption can drop as low as microAmps.

### *Wake on Timeout of Watchdog Timer*

A different strategy for waking from sleep mode is to set up a **watchdog timer** to periodically wake up, run something, and then go back to sleep. Picture your neighbour's dog waking you up every hour or so throughout the night with his annoying barking. The following sketch sets a watchdog timer to sleep for one second, wake up, then go back to sleep. In this way, multiple sleep cycles can be scheduled for as many seconds as you like. Effectively, the MCU will only be awake for a few milliseconds per second, saving precious battery life.

The following example sketch to set up a watchdog timer for low power mode was adapted from Jean Rabault's sketch, posted on GitHub.com. (Rabault 2016) The watchdog timer is set using the WDTCSR register (Watchdog Timer Control Register).

```

// Example: Wake after 5 watchdog sleep cycles
// (1second/cycle)
#include <avr/sleep.h>
#include <avr/power.h>
#include <avr/interrupt.h>
#include <avr/wdt.h>

void setup(){
 Serial.begin(9600);
 configure_wdt(); // configure watchdog timer
}

void loop(){
 Serial.println("Sleeping for 5 seconds.");
 sleep(5); // sleep for 5 seconds
 Serial.println("Ready to do your bidding!");
}
ISR(WDT_vect){ // define the watchdog ISR
 wdt_reset(); // reset watchdog timer
}

void configure_wdt(){
 cli(); // clear (disable) interrupts
 MCUSR=0; // the following are ATmega328-specific cmds
 WDTCSR |= 0b00011000; // set WDCE and WDE high
 //uncomment for your desired timeout:
 //WDTCSR = 0b01000000 | 0b0000000; //16 msec timeout
 //WDTCSR = 0b01000000 | 0b000100; //0.25 sec
 //WDTCSR = 0b01000000 | 0b000101; //0.5 sec
 WDTCSR = 0b01000000 | 0b000110; //1 sec
 //WDTCSR = 0b01000000 | 0b000111; //2 sec
 //WDTCSR = 0b01000000 | 0b100000; //4 sec
 //WDTCSR = 0b01000000 | 0b100001; //8 sec
 //From: Table 8-2, Watchdog Timer Prescale Select,
 //ATmega328 datasheet
 sei(); // set (re-enable) interrupts
}

void sleep(unsigned long n){
 delay(100); // wait for serial commands to finish
 set_sleep_mode(SLEEP_MODE_PWR_DOWN); //max power down
 power_adc_disable(); // turn off adc
 for(int i=0;i<n;i++){
 sleep_mode(); // go to sleep here
 sleep_disable(); // wake up here
 }
 power_all_enable();
}

```

## Resetting the MCU

Most MCUs have a dedicated pin for the RESET button. Having a way to reboot the MCU is important in case it locks up on the user. Many electronic devices have a hard-reset button somewhere, perhaps a small hole where you push in a paperclip or safety pin. The Arduino Uno has a reset button located at the corner of the board (see Figure A-5, *Arduino Uno Pin-out Diagram*). When you press that button, the microcontroller restarts. What if you would like to allow your sketch to reset the microcontroller? This section describes different ways to reset the MCU.

### *Reset with a Watchdog Timer*

The watchdog timer can be used to reset the microcontroller, so that if the sketch hangs or gets stuck somewhere, you get another shot at regaining control of the system. This could act as an important fail-safe mechanism. In the following example, the watchdog timer is started in the loop function with the `wdt_enable(WDTO_1S)` command. When the time elapsed since this command was executed exceeds the option specified inside the brackets, the MCU will reset. You can reset the watchdog timer at any time in your sketch with the `wdt_reset()` command. You can deactivate the timer with `wdt_disable()` if you would like to prevent a reset during an expectedly long function. (Rabault 2016)

```
/* Example: Watchdog Timer for MCU Reset
 * Adapted from:
https://github.com/jerabaul29/ArduinoUseWatchdog/blob/master/ArduinoCode/SimpleWatchdog.ino
 * wdt_enable() option:
 * -----
 * WDTO_15MS 15 msec
 * WDTO_30MS 30 msec
 * WDTO_60MS 60 msec
 * WDTO_120MS 120 msec
 * WDTO_250MS 250 msec
 * WDTO_500MS 500 msec
 * WDTO_1S 1 sec
 * WDTO_2S 2 sec
 * WDTO_4S 4 sec
 * WDTO_8S 8 sec */
#include <avr/wdt.h>

void setup() {
 Serial.begin(9600);
```

```

 Serial.println("I have been reset.");
 wdt_enable(WDTO_1S); // set 1sec watchdog reset timer
}

void loop(){
 delay(2000); // if loop takes >1sec, MCU will reset
 wdt_reset(); // reset timer (make last line of loop)
}

```

You can use the watchdog timer then to intentionally reset the MCU in a stand-alone function, to be executed whenever you like:

```

// Example: Watchdog reset on demand
#include <avr/wdt.h>
void setup(){
 Serial.begin(9600);
 Serial.println("I have been reset.");
}

void loop(){
 for(int i=0;i<1000;i++){
 Serial.println(i);
 delay(100);
 if(i==17)resetMCU(); // reset where I'd like it to
 }
}

void resetMCU(){
 wdt_enable(WDTO_15MS); // enable watchdog timer
 while(1); //intentionally max out timer
}

```

### *Hard Wiring a Digital Pin to the RESET Pin*

Another way of hard-resetting the microcontroller within in a sketch is to connect a jumper from any digital pin to the Uno's RES pin, and set the digital pin HIGH in OUTPUT mode. When you would like to reset the MCU, set the digital pin LOW with `digitalWrite()`. This is electronically equivalent to pushing the reset button with your finger.

```

// Example: MCU Hard reset
// Connect Pin 7 to RES pin
#define resetPin 7

void setup() {
 Serial.begin(9600);
 Serial.println("I have been reset.");
}

```

```

 //set HIGH first (otherwise will reset MCU)
 digitalWrite(resetPin,HIGH);
 pinMode(resetPin,OUTPUT);
}

void loop() {
 delay(5000); // wait 5 seconds
 digitalWrite(resetPin,LOW); // reset the MCU!
}

```

## Advanced Formatting and Variable Type Conversions

### *Secrets of Serial.print()*

`Serial.print()` and `Serial.println()` have some interesting baked-in options you might need or like to take advantage of. For one thing, they allow the use the C++ escape sequences in the middle of a string. Table 10-20 has some useful escape sequences:

**Table 10-20. Some C++ escape sequences you can use in strings and char variables.**

| <i>Escape Sequence</i> | <i>Description</i>                                                                                                |
|------------------------|-------------------------------------------------------------------------------------------------------------------|
| <code>\'</code>        | single quote                                                                                                      |
| <code>\"</code>        | double quote                                                                                                      |
| <code>\\</code>        | backslash                                                                                                         |
| <code>\n</code>        | line feed - new line                                                                                              |
| <code>\r</code>        | carriage return                                                                                                   |
| <code>\t</code>        | horizontal tab                                                                                                    |
| <code>\v</code>        | vertical tab                                                                                                      |
| <code>\xnn</code>      | arbitrary hexadecimal value (from UTF-8 table)                                                                    |
| <code>\unn</code>      | universal character name (arbitrary Unicode value); may result in different characters depending on your platform |

For more information: <http://en.cppreference.com/w/cpp/language/escape>

For example, `\t` will insert a tab in your output, and `\n` will insert a new line:

```
Serial.print("COL1\tCOL2\tCOL3\n");
```

Spaces are not needed between the escape characters and other text. To *remove* escape characters and spaces from a string, you can use the `.trim()` command:

```
String myString="Test of \tescape characters\n\r";
myString.trim(); // removes escape characters
```



```
Serial.print(myString);
```

If you would like to send a number to the serial monitor in binary format, you can use the following explicit command:

```
byte myPin=B00001101;
Serial.print(myPin,BIN); // prints in binary format
```

This will result in a “1101” being sent to the serial monitor, instead of the number 13. The leading zeros are not printed. The following examples summarize different output formats:

**Table 10-21. Custom output formats for Serial.print() and Serial.println().**

| <i>Format</i>                                         | <i>Syntax</i>                                                                                                       | <i>Example Output</i> |
|-------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------|-----------------------|
| Decimal (default # decimals = 2)                      | <code>byte myPin=13;</code><br><code>Serial.print(myPin,DEC);</code><br><code>Serial.print(myPin); // (same)</code> | 13                    |
| Decimal (specify # decimals)–for float variables only | <code>float myFloat=13.f;</code><br><code>Serial.print(myFloat,3);</code>                                           | 13.000                |
| Binary                                                | <code>byte myPin=B00001101;</code><br><code>Serial.print(myPin,BIN);</code>                                         | 1101                  |
| Hexadecimal                                           | <code>byte myPin=0xD;</code><br><code>Serial.print(myPin,HEX);</code>                                               | D                     |
| Octal (base 8)                                        | <code>byte myPin=015; //leading 0</code><br><code>Serial.print(myPin,OCT);</code>                                   | 15                    |

For more information: <https://www.arduino.cc/en/Serial/Print>

**Table 10-22. Additional commands that can help you manipulate Strings.**

| <i>Command Description</i>                            | <i>Syntax</i>                                                                                                                                           | <i>Example Output</i> |
|-------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------|
| <code>.length()</code><br>Find the length of a string | <code>String myString="Hello ";</code><br><code>int len=myString.length();</code><br><code>Serial.println(len);</code>                                  | 7                     |
| <code>.trim()</code><br>Trim spaces from a string     | <code>String myString="Hello ";</code><br><code>myString.trim();</code><br><code>int len=myString.length();</code><br><code>Serial.println(len);</code> | 5                     |

|                                                                                                                    |                                                                                                                                                                                                                                  |           |
|--------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------|
| <code>.indexOf()</code><br>Find the first instance of a character in a string (index numbering starts at 0)        | <pre>String myString="Hello "; int idx=myString.indexOf('l'); Serial.println(idx);</pre>                                                                                                                                         | 2         |
| <code>.lastIndexOf()</code><br>Find the last instance of a character in a string                                   | <pre>String myString="Hello "; int idx=myString.lastIndexOf('l'); Serial.println(idx);</pre>                                                                                                                                     | 3         |
| <code>.toUpperCase()</code><br>Change string to all upper case letters                                             | <pre>String myString="Hello "; myString.toUpperCase(); Serial.println(myString);</pre>                                                                                                                                           | HELLO     |
| <code>.toLowerCase()</code><br>Change string to all lower case letters                                             | <pre>String myString="Hello "; myString.toLowerCase(); Serial.println(myString);</pre>                                                                                                                                           | hello     |
| <code>.replace()</code><br>Replace all instances of one substring with another                                     | <pre>String myString="Hello "; myString.replace("ell","idee h"); Serial.println(myString);</pre>                                                                                                                                 | Hidee ho  |
| <code>.remove()</code><br>Remove parts of a string by index number                                                 | <pre>//Remove from index# to end: String myString1="Hello "; myString1.remove(3); Serial.println(myString1); //Remove from index1 to 2, incl: String myString2="Hello "; myString2.remove(1,3); Serial.println(myString2);</pre> | Hel<br>Ho |
| <code>.substring()</code><br>Create a substring from a string by index number (doesn't change the original string) | <pre>//Substring from index# to end: String myString="Hello "; mySubstr1=myString.substring(3); Serial.println(mySubstr1); //Substr from index1 to 2, incl: mySubstr2=myString.substring(1,3); Serial.println(mySubstr2);</pre>  | lo<br>el  |

For more information:

<https://www.arduino.cc/reference/en/language/variables/data-types/stringobject/>

### *Additional String Conversion Commands*

**Table 10-23. Functions to convert other variable types to Strings.**

| <i>Command Description</i>                                           | <i>Syntax</i>                                                                                                                 | <i>Example Output</i> |
|----------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------|-----------------------|
| String(,DEC)<br>Convert a decimal number to string                   | byte myByte=170;<br>String myString=String(myByte,DEC);<br>//or: String myString=String(myByte);<br>Serial.println(myString); | 170                   |
| String(,BIN)<br>Convert a binary number to string                    | byte myByte=0b10101010;<br>String myString=String(myByte,BIN);<br>Serial.println(myString);                                   | 10101010              |
| String(,HEX)<br>Convert a binary number to string                    | byte myByte=170;<br>String myString=String(myByte,HEX);<br>Serial.println(myString);                                          | aa                    |
| String(,#)<br>Convert to a string, rounded to a specified # decimals | float myFloat=3.14159;<br>String myString=String(myFloat,3);<br>Serial.println(myString);                                     | 3.142                 |

*For more information:*

*<https://www.arduino.cc/reference/en/language/variables/data-types/stringobject/>*

### *Comparing Strings*

Strings can be compared logically, just like other variable types. If you would like to test if two strings are equal, you can directly use a logical operator:

```
// Example: Comparing two strings
void setup(){
 Serial.begin(9600);
 String myString1="Hello";
 String myString2="Hello";
 if(myString1==myString2){
 Serial.println("Strings are equal.");
 }else{
 Serial.println("Strings are not equal.");
 }
}
```

```
void loop() {}
```

The following sketch will read a String from the serial monitor and then compare it to a list of possible responses using *if...then...else if*:

```
// Example: Serial menu with String as input
// Note: Set serial monitor to "No line ending"

void setup(){
 Serial.begin(9600);
 Serial.println("Enter command:");
}

void loop(){
 if(Serial.available()){
 String cmd=Serial.readString();
 if(cmd=="read"){
 Serial.println("Take a reading.");
 //more commands can go here
 } else if(cmd=="write"){
 Serial.println("Write to disk.");
 } else if(cmd=="quit"){
 Serial.println("Quit.");
 while(1); // stop program
 } else {
 Serial.println("Invalid command.");
 } // end if
 Serial.println("Enter command:"); // ask again
 }
}
```

### *Arrays of Strings and Arrays of Char Arrays*

Whereas a char variable holds a single character (0-255), and a char array holds a series of characters, you may wish to create an array of different strings of text. You can create an *array of Strings* as follows:

```
// Example: Array of Strings
String myStrings[]{"text 1", "text 2", "text 3"};

void setup(){
 Serial.begin(9600);
}

void loop(){
 for (int i=0;i<3;i++){
 Serial.println(myStrings[i]);
 delay(1000);
 }
}
```

```

 }
}

```

Alternately, you can create a series of char arrays, like this:

```

// Example: Series of char arrays (pointers)
char* myChars[]{"text 1", "text 2", "text 3"};

void setup(){
 Serial.begin(9600);
}

void loop(){
 for (int i=0;i<3;i++){
 Serial.println(myChars[i]);
 delay(1000);
 }
}

```

The asterisk after char (“char\*”) creates an array of *pointers*, rather than an array of chars. Pointers are useful in C++ when you want to point to a location of a variable rather than refer to the variable itself.

### *Using Special Characters*

Frequently, we need to display special characters in scientific applications, such as a degree sign (e.g. °C), a plus or minus sign (e.g. 5.0 ± 3.2), or a special unit (e.g. μM). The `Serial.write()` command prints a single character to the serial monitor. The following commands illustrate how to use `Serial.write()`:

```

Serial.print("37");
Serial.write(177); // 177 is decimal UTF8 for ±
Serial.print("2.0");
Serial.write(176); // 176 is decimal UTF8 for °
Serial.print("C, 100");
Serial.write(181); // 181 is decimal UTF8 for μ
Serial.print("M"); // will print "37±2°C, 100μM"

```

The `Serial.write()` command only prints one character at a time. However, you can also try copying the extended character directly into a String for printing:

```

String myMessage= "37±2°C";
Serial.println(myMessage);

```

You can use a special character directly in the middle of a char array:

```

char myMessage[]="37±2°C";
Serial.println(myMessage);

```

You can use the unicode escape sequence `\x(hex code)` to define a special character as well:

```
Serial.print("37");
Serial.print("\xB1"); //B1:HEX for ± (UTF8 table)
Serial.print("2");
Serial.print("\xB0"); //B0:HEX for ° (UTF8 table)
Serial.print("C, 100");
Serial.print("\xB5"); //B5:HEX for µ (UTF8 table)
Serial.print("M"); //will print "37±2°C, 100µM"
```

Some external devices (e.g. LCD modules) can use character codes other than UTF-8 (e.g. ASCII extended characters). This can cause confusion when a character doesn't display as expected. For instance, the 16x2 I<sup>2</sup>C LCD screens we use in class need the following command to print a degree sign:

```
lcd.write(223); // prints degree sign on serial LCD
```

The *UTF-8 and ASCII Tables* in Table A-2 of the appendix list all of the control, regular, and extended characters available to print to the serial monitor or other device, such as an SD card or LCD screen.

### *Char Arrays: Advanced Functions*

Most libraries manipulate strings of text as char arrays rather than Strings, which can be difficult to manage at first. Here are enough functions to get you started.

#### *Length of Char Arrays*

The function `sizeof()` returns the *number of bytes* occupied by a variable or variable type. If an array is used as an input argument, it returns the number of bytes occupied by the array. If a char array is used as an input argument, since a single char variable is one byte, `sizeof()` will return the length of text stored in a char array, including the terminal null character (ASCII 0). For example, the following code:

```
char myChars[]="example text";
Serial.print(sizeof(myChars));
```

would print the number 13 (the length of the string of text + the null character).

*For more information:*

<https://www.arduino.cc/reference/en/language/variables/utilities/sizeof/>

The function `strlen()` returns the length of the text stored in a char array, not including the null character. So the following code:

```
char myChars[]="example text";
Serial.print(strlen(myChars));
```

would print the number 12 (the length of the text only).

*For more information:*

<http://www.cplusplus.com/reference/cstring/strlen/?kw=strlen>

Keep in mind that commands designed for char arrays do not work on Strings. See Table 10-22 for the appropriate String functions.

### *Copying, Concatenating, and Comparing Char Arrays*

The function `strcpy()` copies one char array to another (make sure destination array size is large enough):

```
char myChars1[]="example text";
char myChars2[sizeof(myChars1)]; //destination array
strcpy(myChars2,myChars1); // copy myChars1->myChars2
Serial.println(myChars1); // prints "example text"
Serial.println(myChars2); // prints "example text"
```

The function `strcat()` concatenates two char arrays together:

```
char myChars1[]="text1";
char myChars2[]="text2";
char bothChars[50]; // big enough for both arrays
bothChars[0]=0; //erase bothChars (ASCII 0=null)
strcat(bothChars,myChars1); // myChars1 to bothChars
strcat(bothChars,myChars2); // myChars2 to bothChars
Serial.println(myChars1); // prints "text1"
Serial.println(myChars2); // prints "text2"
Serial.println(bothChars); // prints "text1text2"
```

The function `strcmp()` compares two char variables or char arrays. A zero is returned if the char arrays are equal. The following code provides an example of comparing char arrays:

```
char myChars1[]="text1";
char myChars2[]="text1";
if(strcmp(myChars1, myChars2)==0){
 Serial.println("char arrays are equal");
}else{
 Serial.println("char arrays are not equal");
}
```

*For more information:* <http://www.cplusplus.com/reference/cstring/>

### *Single Char Analysis*

The following commands in Table 10-24 test whether the character stored inside a *char* variable is a certain type (e.g. upper case, numeric, etc.).

These commands are useful when receiving serial data from a device. Each command returns either a 1 (for *true*) or a 0 (for *false*).

**Table 10-24. Commands that test the contents of a char variable.**

| <b>Command</b><br><i>Is a character...</i>                                   | <b>Syntax</b><br><i>(the following examples would all return 1)</i>                                                                                                          |
|------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>isAlpha()</code><br>... a letter? (alphabetical)                       | <code>char myChar='H';</code><br><code>bool result=isAlpha(myChar);</code>                                                                                                   |
| <code>isAlphaNumeric()</code><br>... a letter or a number?                   | <code>char myChar='3';</code><br><code>bool result=isAlphaNumeric(myChar);</code>                                                                                            |
| <code>isAscii()</code><br>... an ASCII character?                            | <code>char myChar='3';</code><br><code>bool result=isAscii(myChar);</code>                                                                                                   |
| <code>isWhitespace()</code><br>... white space?                              | <code>char myChar=' ';</code><br><code>bool result=isWhitespace(myChar);</code>                                                                                              |
| <code>isControl()</code><br>...a control character?                          | <code>char myChar=0x8; //0x8 is backspace</code><br><code>bool result=isControl(myChar);</code>                                                                              |
| <code>isDigit()</code> ...<br>a numerical digit?                             | <code>char myChar='3';</code><br><code>bool result=isDigit(myChar);</code>                                                                                                   |
| <code>isPrintable()</code><br>...a printable character?                      | <code>char myChar=' ';</code><br><code>bool result=isPrintable(myChar);</code>                                                                                               |
| <code>isGraph()</code> ...<br>a printable character with<br>no whitespace?   | <code>char myChar='3';</code><br><code>bool result=isGraph(myChar);</code>                                                                                                   |
| <code>isPunct()</code> ...<br>punctuation?                                   | <code>char myChar=',';</code><br><code>bool result=isPunct(myChar);</code>                                                                                                   |
| <code>isSpace()</code><br>...a space?                                        | <code>char myChar=' ';</code><br><code>bool result=isSpace(myChar);</code>                                                                                                   |
| <code>isUpperCase()</code> ...<br>upper case?                                | <code>char myChar='H';</code><br><code>bool result=isUpperCase(myChar);</code>                                                                                               |
| <code>isLowerCase()</code><br>...lower case?                                 | <code>char myChar='H';</code><br><code>bool result=isLowerCase(myChar);</code>                                                                                               |
| <code>isHexadecimalDigit()</code><br>...a hexadecimal digit?<br>(0-9 or A-F) | <code>char myChar='F';</code><br><code>bool</code><br><code>result=isHexadecimalDigit(myChar);</code><br><code>//bool and boolean are</code><br><code>interchangeable</code> |

For more information: <https://www.arduino.cc/en/Tutorial/CharacterAnalysis>

## Structures

A more organized way of bundling data together is to create a structure. You can group more than one variable type together in a structure, and you



can think of this grouping as a higher order array. A structure isn't a variable in and of itself, it is just a specification of how variables are grouped together. Once you declare a structure, you can create *objects* that inherit its variable types and reserve real memory space. The following example illustrates how you can define and use a structure:

```
struct myKids { // declare structure in global space
 String firstname;
 int age; // these variables are members of myKids
 char gender;
};
// Note: this semicolon is required here
myKids child1; // Define object child1 to use myKids
 // struct, and have the same members

// in local space (setup or loop function):
child1.firstname="Aidan";
child1.age=13;
child1.gender='M';
Serial.print(child1.firstname); // prints Aidan
```

As you can see, different variables are grouped together under the name “child1” having the structure of myKids. Members of child1 are accessed with a “.” after it (e.g. child1.firstname). Structures are amazing because you can create more than one child having the same structure:

```
struct myKids { // declare struct in global space
 String firstname;
 int age;
 char gender;
};

myKids child1; // declare child1 in global space
myKids child2; // declare child2 in global space

// in local space (setup or loop function):
child1.firstname="Aidan"; // data for child1
child1.age=13;
child1.gender='M';

child2.firstname="Remy"; // data for child2
child2.age=11;
child2.gender='M';
```

Here, child1 and child2 are separate objects with the same structure as myKids, each existing as different spaces in memory. In practice, you may not need to create more than one object for your struct, but the capability is there. Members of structures can also be arrays of variables, making structures extremely flexible.

An alternate syntax to create a structure is to declare all the object names right after the structure has been declared. This code will work the same as the code above:

```
struct myKids {
 String firstname;
 int age;
 char gender;
} child1, child2; //declare objects before semicolon

child1.firstname="Aidan";
child1.age=13;
child1.gender='M';

child2.firstname="Remy";
child2.age=11;
child2.gender='M';
```

To turn this structure into an *array of objects* instead of declaring child1 and child2 with different object names, the structure could be defined like this:

```
struct myKids {
 String firstname;
 int age;
 char gender;
} child[2];
// declare an array of objects having the structure
// of myKids (object array size:2)

child[0].firstname="Aidan";
child[0].age=13;
child[0].gender='M';

child[1].firstname="Remy";
child[1].age=11;
child[1].gender='M';
```

You can pass structured arrays to and from functions, just like other variable types. This can *tremendously* simplify your code. Think of it: all of your function arguments can be passed to your function using one organized structure, instead of one-by-one as separate arguments:

```
// For this function to compile, the struct myKids
// should be declared in global space.
void printChild(myKids kidx){ // struct as input arg
 Serial.println(kidx.firstname);
 Serial.println(kidx.age);
 Serial.println(kidx.gender);
```

```

}
// in local space:
printChild(child[0]); // printing info for child1
printChild(child[1]); // printing info for child2

```

As with other function input arguments, if you would like the function to be able to change the value stored in the member of an object while inside the function, use the call-by-reference sign “&”:

```

void printChild (myKids &kidx){ // call-by-reference
 kidx.firstname="Bob";//set kidx firstname to Bob
 Serial.println(kidx.firstname);
 Serial.println(kidx.age);
 Serial.println(kidx.gender);
}

```

*For more information: <http://www.cplusplus.com/doc/tutorial/structures/>*

## Unions

A **union** in C++ allows you to share the same memory space between two different variable types, or structures, or a mix of the two. This becomes important if you are looking for an easy way to convert data between variable types quickly. The simplest example sketch of a union is one declared to break up an integer (16 bits, or 2 bytes) into 2 separate bytes of data, a high byte and a low byte. This is easily solved with bit shifting (see Table 10-8), but to illustrate the solution using unions, you could write the following:

```

//Example: Union between Integer and Byte Array
union myUnion{ //declare union in global space
 int myInt; // integer to be shared with array
 byte myBytes[2]; //occupies same memory as myInt
}myData; //create a new union instance called myData

void setup(){
 Serial.begin(9600);
 myData.myInt=0b1100000000000111; // example data
 Serial.print("High byte:");
 Serial.println(myData.myBytes[1],BIN);
 Serial.print("Low byte:");
 Serial.println(myData.myBytes[0],BIN);
}

void loop(){}

```

This sketch will print the high byte 11000000 and then the low byte 111 (without the leading zeros) to the serial monitor. You can also share structures inside unions, so we can re-write this code to be a bit more snazzy:

```
//Example: Union between Integer and Struct
struct myStruct{ // declare a structure to share
 byte low; // to store low byte
 byte high; // to store high byte
};

union myUnion{ //declare union in global space
 int myInt; // integer to be shared with myBytes
 myStruct myBytes; //occupies same memory as myInt
}myData; //create a new union instance called myData

void setup(){
 Serial.begin(9600);
 myData.myInt=0b1100000000000111;
 Serial.print("High byte:");
 Serial.println(myData.myBytes.high,BIN);
 Serial.print("Low byte:");
 Serial.println(myData.myBytes.low,BIN);
}

void loop(){}

```

For more information: [http://www.cplusplus.com/doc/tutorial/other\\_data\\_types/](http://www.cplusplus.com/doc/tutorial/other_data_types/)

## Increment Operators as Array Index Values

Occasionally in C++, you will see programmers use increment operators (`i++` or `++i`) as array index values. This is a clever and quick way of saving a line of code, or eliminating a `for()` loop. However, there is a subtle difference when using these two operators. Used as an array index value, the command `i++` *first* uses the present value `i` as the array index and *then* increments the value of `i` by 1:

```
int i=0;
int myArray[2]={0,0};
myArray[i++]=3; // equivalent to: myArray[0]=3; i=i+1;
myArray[i]=4; // equivalent to: myArray[1]=4;

```

After this code is run, `myArray[0]` will have the value 3, and `myArray[1]` will have the value 4.

In contrast, the command `++i`, used as an array index value, *first* increments the value of `i` by 1 and then uses the result as an array index:

```
int i=0;
int myArray[2]={0,0};
myArray[++i]=3; // equivalent to: i=i+1; myArray[1]=3;
myArray[i]=4; // equivalent to: myArray[1]=4;

```

After this code is run, `myArray[0]` will have the value 0, and `myArray[1]` will have the value 4.

# APPENDIX

## Troubleshooting Guide

You found the troubleshooting guide. That's great! Take a deep breath. Circuits almost never work on your first try. This is normal. Sometimes the problem is with the hardware, sometimes it is with the system you are measuring, sometimes there is a bug in the program, sometimes it is your understanding of the system that is causing the trouble, and sometimes it's just rotten luck.

There are three philosophical concepts that help me troubleshoot:

- 9) **Imagination.** Being able to imagine what is wrong and devising a way to test that idea will help you immensely with problem solving. Even imagining an incorrect cause of your challenges can help you stumble upon the actual reason.
- 10) **Optimism.** Believing that a circuit or system will work is extremely important to the success of your project. If you don't believe you can solve a problem, you will be more prone to give up easily. Picture your project working. Imagine how good you will feel when that finally happens. Understand that it usually takes a lot of work to get a system functioning, and that's ok. Don't discourage yourself with negative thoughts, like you aren't smart enough, or the system will never work (see *Troubleshooting Flowchart* in the appendix).
- 11) **Resilience.** Being stubborn is an asset for problem solving. You can't solve a problem if you give up. If you find you hit a dead end and you are frustrated, take a break. I have solved many of my own electronics problems in unexpected places, when I wasn't even working on the problem – on the subway, in the shower, or bumping into a retired radio engineer in Arches Provincial Park on the picturesque Newfoundland coastline.

Table A-1, formed while working through my own electronics projects, will guide you through some basic questions to help you narrow down the problem. Figures A-1 and A-2 will help you reframe and refocus your problem.

**Table A-1. Troubleshooting questions and suggested follow-up actions.**

| <i>Ask yourself:</i>                                              | <i>Follow-Up Actions:</i>                                                                                                                                                                                                                                                                                                                                                                                                                             |
|-------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Is it plugged in?                                                 | Check that the power rails on your breadboard have voltage, and that the power supply is working at the correct voltage level.                                                                                                                                                                                                                                                                                                                        |
| Are your batteries still good?                                    | If your circuit is battery-powered, measure the battery voltage. Replace with fresh batteries if low.                                                                                                                                                                                                                                                                                                                                                 |
| Is anything smoking, melting, or is there a strong burning smell? | Disconnect power immediately. Check for accidental shorts in your circuit, and check that the Vcc and GND pins for any IC chips were wired correctly.                                                                                                                                                                                                                                                                                                 |
| Feel any chips in your circuit. Are they hot?                     | Disconnect power. Double-check that chips are not installed upside-down. This is a very common mistake to make.                                                                                                                                                                                                                                                                                                                                       |
| Are your modules/circuits properly grounded to your MCU?          | <ul style="list-style-type: none"> <li>• When there is more than one power supply in your circuit, and you are not using a relay, double-check that your supplies share a common ground.</li> <li>• You can't send a signal to a module or component without also connecting its ground pin to your MCU. Otherwise, that's just a dead end in the circuit.</li> </ul>                                                                                 |
| Is your power supply current limited?                             | It may be your power supply can't keep up with the demand of the circuit, and the components are running lean. Try a larger supply, or perhaps separate supplies if you think your circuit is current-limited.                                                                                                                                                                                                                                        |
| Are you supplying the correct voltage to a module?                | A 3.3V module doesn't run well for long on 5V. Even the data pins should be at 3.3V. Use a logic shifter or voltage divider.                                                                                                                                                                                                                                                                                                                          |
| Are all of your components correct?                               | Use the component that the circuit diagram or datasheet calls for, rather than improvising something "close enough". If you can't find a component, seek it out—don't substitute. Easily confused: 100R resistor with 100K resistor. <ul style="list-style-type: none"> <li>• Check/measure all capacitor and resistor values.</li> <li>• Check codes on DIP chips – these can be faint, tiny, and easily misread (e.g. LM555 vs. LMC555).</li> </ul> |
| Are polar devices wired respecting their correct polarities?      | Check that all polar devices (e.g. LEDs, diodes, capacitors, microphones, speakers, etc.) are wired the correct way (usually: longer leg positive). These devices will not work when wired backwards.                                                                                                                                                                                                                                                 |
| Are your chips wired correctly?                                   | Check proper pin numbers, directly from the component datasheet (not your notes).                                                                                                                                                                                                                                                                                                                                                                     |

|                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|-------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                                         | <p>Don't ever assume you can remember how to wire transistors/MOSFETS or op-amps from memory. Switch terminals are very easy to confuse.</p> <p>Microcontroller female header holes are very close together. Check for offsets in your connections.</p>                                                                                                                                                                                                                                                                                                                                                                                                             |
| Is your chip, module, or cable a dud, or is a chip in thermal overload? | <ul style="list-style-type: none"> <li>• Improperly wiring a component can fry it. Or, it may just be a dud.</li> <li>• Try swapping out your chip/module/cable with another identical chip/module/cable. However, if you have just shorted out a chip, swapping it out with a new one in the same circuit <i>without</i> adjustments will likely just result in one more blown chip.</li> <li>• Many chips and voltage regulators have built-in thermal overload protection, meaning that if they are run too hot, they will automatically shut down until they cool off again. Before you throw a part away, let it cool off first, and test it again.</li> </ul> |
| Are there any visible shorts in the system?                             | <p>e.g. bare resistor or capacitor leads touching other bare leads. Check for and remove shorts.</p> <p>If there is a short in the system, your microcontroller's built-in LED might go dim. Cut power to microcontroller immediately—shorts can damage the board. Check the microcontroller integrity by uploading a blank sketch. If it uploads, the board is likely still ok.</p>                                                                                                                                                                                                                                                                                |
| Are your jumpers and breadboard in good shape?                          | <p>Breadboard wires are very thin, and wear out. It could be that all your components <i>appear</i> to be wired correctly, but one faulty wire connection can throw your whole design. Breadboard connections are also notoriously finicky. Test connections with an ohmmeter or continuity tester. When rebuilding a circuit, try using different jumpers, on a different area of a breadboard. Throw away jumpers that are detected as faulty.</p>                                                                                                                                                                                                                |
| Can you narrow down/locate the problem?                                 | <p>With a voltmeter and/or oscilloscope, test different output voltages of your circuit <i>starting upstream</i> (from the power supply) and checking component by component. This will help you isolate where the issue is. You can easily measure your power rails, bias voltages, negative voltages, chip outputs, etc. to quickly confirm which wires and components are working and which of them are failing.</p>                                                                                                                                                                                                                                             |
| Is there a bug in your sketch?                                          | <p>Even a perfectly wired system might not work because of the program. Common errors that will lead to circuits not working:</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |



|  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|--|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|  | <ul style="list-style-type: none"> <li>• Forgetting to declare <code>pinMode (#, HIGH);</code></li> <li>• Forgetting to connect AREF pin to a voltage if using an external analog reference.</li> <li>• Low memory warnings can lead to microprocessor instability (see <i>Tips to Optimize Sketch Memory</i>).</li> <li>• Are you having trouble uploading your program? Check that your port number is correct, and that your serial monitor is closed. Try resetting the Arduino (push reset button, or unplug/replug into a different USB port). Restart the Arduino IDE or reboot your computer if your COM ports are giving you trouble, or if your program won't compile even when there aren't any program errors. Sometimes COM port connections crash, and won't reset without a restart or reboot. Problems uploading might also mean a short circuit, or a dead microcontroller.</li> <li>• Even a successfully compiled program can have math errors (e.g. check formulas, units, brackets in the wrong places, etc.). Hand-check your code with test values at each step to make sure your math is correct.</li> <li>• Do array values seem to mysteriously change throughout your program, even though your sketch doesn't ask them to change? Double check the array dimensions in the declaration statement, and the index values you are using to access the array. A common mistake is referring to element <code>arr[n]</code> when the size of <code>arr</code> is <code>n</code> (the last element of <code>arr</code> here is <code>n-1</code>).</li> </ul> |
|  | <p>By this point, you hopefully have isolated the problem. If not...</p> <ul style="list-style-type: none"> <li>• Try rebuilding the system from scratch. It's easier to start over and build your circuit carefully, than to find a tiny error in a nest of jumper wires.</li> <li>• Ask for help from your instructor, another classmate, or a newsgroup.</li> <li>• Check your understanding and assumptions of the circuit and the system. The system may be more complicated than you realize, and what you have built is actually <i>working</i> the way you built it, just not in the way you imagined it should. Often when troubleshooting, we are correcting our understanding of the system as much as we are correcting the system.</li> <li>• Try a different design, strategy, or angle of attack. Sometimes other people's circuits just don't work well. Consider buying a more suitable or user-friendly component or module. Money can be a fantastic problem solver.</li> <li>• Try taking a break. Sometimes going back to a problem later with a clear head can work wonders. Try waiting an hour, a day, a week, or even a month. This will also allow for time to collect replies from your desperate newsgroup posts.</li> </ul>                                                                                                                                                                                                                                                                                                                           |

### Troubleshooting Flowchart

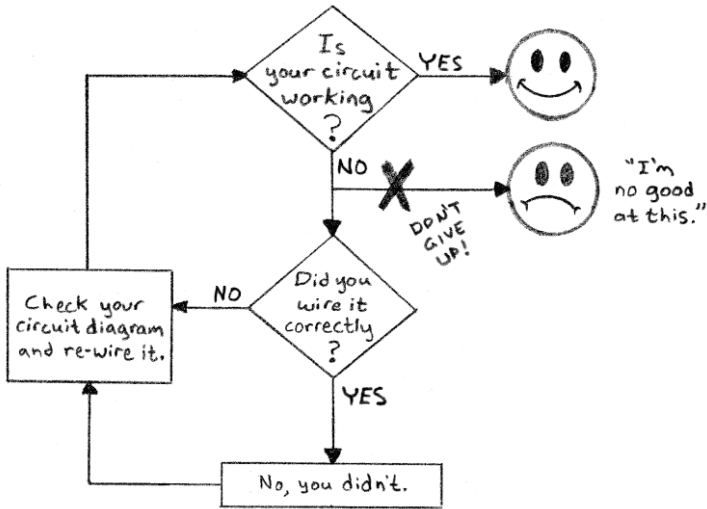


Figure A-1. Troubleshooting flowchart: developing resilience.

### Troubleshooting Zones

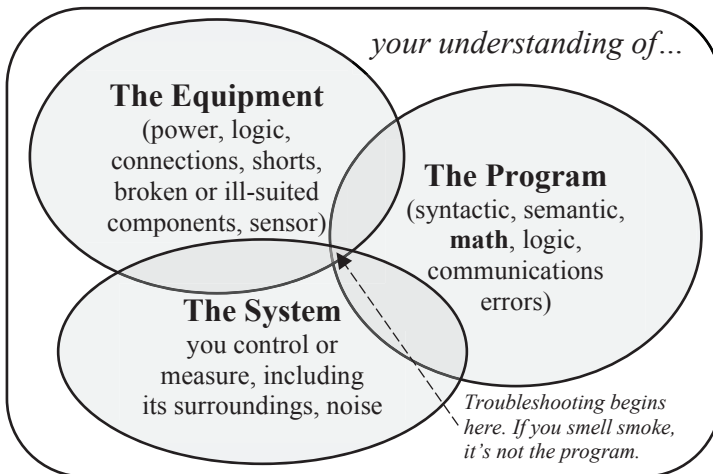


Figure A-2. Potential areas to consider troubleshooting. Many problems are multifaceted, involving more than one area.

*UTF-8 and ASCII Tables*

**Table A-2. UTF-8 character table, with decimal, hexadecimal, and binary codes. The ASCII column is the corresponding ASCII character resulting from the same code.**

| <i>UTF8</i> | <i>Dec</i> | <i>Hex</i> | <i>Binary</i> | <i>ASCII</i> |
|-------------|------------|------------|---------------|--------------|
| NUL         | 0          | 0x0        | 0b0           | NUL          |
| SOH         | 1          | 0x1        | 0b1           | SOH          |
| STX         | 2          | 0x2        | 0b10          | STX          |
| ETX         | 3          | 0x3        | 0b11          | ETX          |
| EOT         | 4          | 0x4        | 0b100         | EOT          |
| ENQ         | 5          | 0x5        | 0b101         | ENQ          |
| ACK         | 6          | 0x6        | 0b110         | ACK          |
| BEL         | 7          | 0x7        | 0b111         | BEL          |
| BS          | 8          | 0x8        | 0b1000        | BS           |
| HT          | 9          | 0x9        | 0b1001        | HT           |
| LF          | 10         | 0xA        | 0b1010        | LF           |
| VT          | 11         | 0xB        | 0b1011        | VT           |
| FF          | 12         | 0xC        | 0b1100        | FF           |
| CR          | 13         | 0xD        | 0b1101        | CR           |
| SO          | 14         | 0xE        | 0b1110        | SO           |
| SI          | 15         | 0xF        | 0b1111        | SI           |
| DLE         | 16         | 0x10       | 0b10000       | DLE          |
| DC1         | 17         | 0x11       | 0b10001       | DC1          |
| DC2         | 18         | 0x12       | 0b10010       | DC2          |
| DC3         | 19         | 0x13       | 0b10011       | DC3          |
| DC4         | 20         | 0x14       | 0b10100       | DC4          |
| NAK         | 21         | 0x15       | 0b10101       | NAK          |
| SYN         | 22         | 0x16       | 0b10110       | SYN          |
| ETB         | 23         | 0x17       | 0b10111       | ETB          |
| CAN         | 24         | 0x18       | 0b11000       | CAN          |
| EM          | 25         | 0x19       | 0b11001       | EM           |
| SUB         | 26         | 0x1A       | 0b11010       | SUB          |
| ESC         | 27         | 0x1B       | 0b11011       | ESC          |
| FS          | 28         | 0x1C       | 0b11100       | FS           |
| GS          | 29         | 0x1D       | 0b11101       | GS           |
| RS          | 30         | 0x1E       | 0b11110       | RS           |
| US          | 31         | 0x1F       | 0b11111       | US           |

| <i>UTF8</i> | <i>Dec</i> | <i>Hex</i> | <i>Binary</i> | <i>ASCII</i> |
|-------------|------------|------------|---------------|--------------|
| space       | 32         | 0x20       | 0b100000      | space        |
| !           | 33         | 0x21       | 0b100001      | !            |
| ,           | 34         | 0x22       | 0b100010      | ,            |
| #           | 35         | 0x23       | 0b100011      | #            |
| \$          | 36         | 0x24       | 0b100100      | \$           |
| %           | 37         | 0x25       | 0b100101      | %            |
| &           | 38         | 0x26       | 0b100110      | &            |
| '           | 39         | 0x27       | 0b100111      | '            |
| (           | 40         | 0x28       | 0b101000      | (            |
| )           | 41         | 0x29       | 0b101001      | )            |
| *           | 42         | 0x2A       | 0b101010      | *            |
| +           | 43         | 0x2B       | 0b101011      | +            |
| ,           | 44         | 0x2C       | 0b101100      | ,            |
| -           | 45         | 0x2D       | 0b101101      | -            |
| .           | 46         | 0x2E       | 0b101110      | .            |
| /           | 47         | 0x2F       | 0b101111      | /            |
| 0           | 48         | 0x30       | 0b110000      | 0            |
| 1           | 49         | 0x31       | 0b110001      | 1            |
| 2           | 50         | 0x32       | 0b110010      | 2            |
| 3           | 51         | 0x33       | 0b110011      | 3            |
| 4           | 52         | 0x34       | 0b110100      | 4            |
| 5           | 53         | 0x35       | 0b110101      | 5            |
| 6           | 54         | 0x36       | 0b110110      | 6            |
| 7           | 55         | 0x37       | 0b110111      | 7            |
| 8           | 56         | 0x38       | 0b111000      | 8            |
| 9           | 57         | 0x39       | 0b111001      | 9            |
| :           | 58         | 0x3A       | 0b111010      | :            |
| ;           | 59         | 0x3B       | 0b111011      | ;            |
| <           | 60         | 0x3C       | 0b111100      | <            |
| =           | 61         | 0x3D       | 0b111101      | =            |
| >           | 62         | 0x3E       | 0b111110      | >            |
| ?           | 63         | 0x3F       | 0b111111      | ?            |

Characters in Table A-2 that are shaded dark grey are non-printable control characters. These characters (and others) may not render properly on the

Arduino IDE serial monitor, and may vary between programs, devices, and operating systems.

**Table A-2. UTF-8 and ASCII character table (continued).**

| <i>UTF8</i> | <i>Dec</i> | <i>Hex</i> | <i>Binary</i> | <i>ASCII</i> | <i>UTF8</i> | <i>Dec</i> | <i>Hex</i> | <i>Binary</i> | <i>ASCII</i> |
|-------------|------------|------------|---------------|--------------|-------------|------------|------------|---------------|--------------|
| @           | 64         | 0x40       | 0b1000000     | @            | `           | 96         | 0x60       | 0b1100000     | `            |
| A           | 65         | 0x41       | 0b1000001     | A            | a           | 97         | 0x61       | 0b1100001     | a            |
| B           | 66         | 0x42       | 0b1000010     | B            | b           | 98         | 0x62       | 0b1100010     | b            |
| C           | 67         | 0x43       | 0b1000011     | C            | c           | 99         | 0x63       | 0b1100011     | c            |
| D           | 68         | 0x44       | 0b1000100     | D            | d           | 100        | 0x64       | 0b1100100     | d            |
| E           | 69         | 0x45       | 0b1000101     | E            | e           | 101        | 0x65       | 0b1100101     | e            |
| F           | 70         | 0x46       | 0b1000110     | F            | f           | 102        | 0x66       | 0b1100110     | f            |
| G           | 71         | 0x47       | 0b1000111     | G            | g           | 103        | 0x67       | 0b1100111     | g            |
| H           | 72         | 0x48       | 0b1001000     | H            | h           | 104        | 0x68       | 0b1101000     | h            |
| I           | 73         | 0x49       | 0b1001001     | I            | i           | 105        | 0x69       | 0b1101001     | i            |
| J           | 74         | 0x4A       | 0b1001010     | J            | j           | 106        | 0x6A       | 0b1101010     | j            |
| K           | 75         | 0x4B       | 0b1001011     | K            | k           | 107        | 0x6B       | 0b1101011     | k            |
| L           | 76         | 0x4C       | 0b1001100     | L            | l           | 108        | 0x6C       | 0b1101100     | l            |
| M           | 77         | 0x4D       | 0b1001101     | M            | m           | 109        | 0x6D       | 0b1101101     | m            |
| N           | 78         | 0x4E       | 0b1001110     | N            | n           | 110        | 0x6E       | 0b1101110     | n            |
| O           | 79         | 0x4F       | 0b1001111     | O            | o           | 111        | 0x6F       | 0b1101111     | o            |
| P           | 80         | 0x50       | 0b1010000     | P            | p           | 112        | 0x70       | 0b1110000     | p            |
| Q           | 81         | 0x51       | 0b1010001     | Q            | q           | 113        | 0x71       | 0b1110001     | q            |
| R           | 82         | 0x52       | 0b1010010     | R            | r           | 114        | 0x72       | 0b1110010     | r            |
| S           | 83         | 0x53       | 0b1010011     | S            | s           | 115        | 0x73       | 0b1110011     | s            |
| T           | 84         | 0x54       | 0b1010100     | T            | t           | 116        | 0x74       | 0b1110100     | t            |
| U           | 85         | 0x55       | 0b1010101     | U            | u           | 117        | 0x75       | 0b1110101     | u            |
| V           | 86         | 0x56       | 0b1010110     | V            | v           | 118        | 0x76       | 0b1110110     | v            |
| W           | 87         | 0x57       | 0b1010111     | W            | w           | 119        | 0x77       | 0b1110111     | w            |
| X           | 88         | 0x58       | 0b1011000     | X            | x           | 120        | 0x78       | 0b1111000     | x            |
| Y           | 89         | 0x59       | 0b1011001     | Y            | y           | 121        | 0x79       | 0b1111001     | y            |
| Z           | 90         | 0x5A       | 0b1011010     | Z            | z           | 122        | 0x7A       | 0b1111010     | z            |
| [           | 91         | 0x5B       | 0b1011011     | [            | {           | 123        | 0x7B       | 0b1111011     | {            |
| \           | 92         | 0x5C       | 0b1011100     | \            |             | 124        | 0x7C       | 0b1111100     |              |
| ]           | 93         | 0x5D       | 0b1011101     | ]            | }           | 125        | 0x7D       | 0b1111101     | }            |
| ^           | 94         | 0x5E       | 0b1011110     | ^            | ~           | 126        | 0x7E       | 0b1111110     | ~            |
| _           | 95         | 0x5F       | 0b1011111     | _            | DEL         | 127        | 0x7F       | 0b1111111     | DEL          |

**Table A-2. UTF-8 and ASCII character table (continued) – extended character set.**

| UTF8 | Dec | Hex  | Binary     | ASCII |
|------|-----|------|------------|-------|
| €    | 128 | 0x80 | 0b10000000 | Ç     |
| ü    | 129 | 0x81 | 0b10000001 | ü     |
| ,    | 130 | 0x82 | 0b10000010 | é     |
| f    | 131 | 0x83 | 0b10000011 | â     |
| „    | 132 | 0x84 | 0b10000100 | ä     |
| …    | 133 | 0x85 | 0b10000101 | à     |
| †    | 134 | 0x86 | 0b10000110 | â     |
| ‡    | 135 | 0x87 | 0b10000111 | ç     |
| ^    | 136 | 0x88 | 0b10001000 | ê     |
| ‰    | 137 | 0x89 | 0b10001001 | ë     |
| Š    | 138 | 0x8A | 0b10001010 | è     |
| ‹    | 139 | 0x8B | 0b10001011 | ï     |
| Œ    | 140 | 0x8C | 0b10001100 | î     |
| ì    | 141 | 0x8D | 0b10001101 | ì     |
| Ž    | 142 | 0x8E | 0b10001110 | Ä     |
| Å    | 143 | 0x8F | 0b10001111 | Å     |
| É    | 144 | 0x90 | 0b10010000 | É     |
| ‘    | 145 | 0x91 | 0b10010001 | æ     |
| ’    | 146 | 0x92 | 0b10010010 | Æ     |
| “    | 147 | 0x93 | 0b10010011 | ô     |
| ”    | 148 | 0x94 | 0b10010100 | ö     |
| •    | 149 | 0x95 | 0b10010101 | ò     |
| —    | 150 | 0x96 | 0b10010110 | û     |
| —    | 151 | 0x97 | 0b10010111 | ù     |
| ~    | 152 | 0x98 | 0b10011000 | ÿ     |
| ™    | 153 | 0x99 | 0b10011001 | Ö     |
| š    | 154 | 0x9A | 0b10011010 | Ü     |
| ›    | 155 | 0x9B | 0b10011011 | ø     |
| œ    | 156 | 0x9C | 0b10011100 | £     |
| Ø    | 157 | 0x9D | 0b10011101 | Ø     |
| ž    | 158 | 0x9E | 0b10011110 | ×     |
| ÿ    | 159 | 0x9F | 0b10011111 | f     |

| UTF8 | Dec | Hex  | Binary     | ASCII |
|------|-----|------|------------|-------|
| á    | 160 | 0xA0 | 0b10100000 | á     |
| í    | 161 | 0xA1 | 0b10100001 | í     |
| ç    | 162 | 0xA2 | 0b10100010 | ó     |
| £    | 163 | 0xA3 | 0b10100011 | ú     |
| ¤    | 164 | 0xA4 | 0b10100100 | ñ     |
| ¥    | 165 | 0xA5 | 0b10100101 | Ñ     |
| ¦    | 166 | 0xA6 | 0b10100110 | ª     |
| §    | 167 | 0xA7 | 0b10100111 | º     |
| ¨    | 168 | 0xA8 | 0b10101000 | ¿     |
| ©    | 169 | 0xA9 | 0b10101001 | ®     |
| ª    | 170 | 0xAA | 0b10101010 | –     |
| «    | 171 | 0xAB | 0b10101011 | ½     |
| ¬    | 172 | 0xAC | 0b10101100 | ¼     |
|      | 173 | 0xAD | 0b10101101 | ı     |
| ®    | 174 | 0xAE | 0b10101110 | «     |
| –    | 175 | 0xAF | 0b10101111 | »     |
| °    | 176 | 0xB0 | 0b10110000 | ◻     |
| ±    | 177 | 0xB1 | 0b10110001 | ◻     |
| ²    | 178 | 0xB2 | 0b10110010 | ◻     |
| ³    | 179 | 0xB3 | 0b10110011 |       |
| ´    | 180 | 0xB4 | 0b10110100 | ┌     |
| µ    | 181 | 0xB5 | 0b10110101 | Á     |
| ¶    | 182 | 0xB6 | 0b10110110 | Â     |
| ·    | 183 | 0xB7 | 0b10110111 | À     |
| ¸    | 184 | 0xB8 | 0b10111000 | ©     |
| ¹    | 185 | 0xB9 | 0b10111001 | ¶     |
| º    | 186 | 0xBA | 0b10111010 | ¶     |
| »    | 187 | 0xBB | 0b10111011 | ¶     |
| ¼    | 188 | 0xBC | 0b10111100 | ¶     |
| ½    | 189 | 0xBD | 0b10111101 | ç     |
| ¾    | 190 | 0xBE | 0b10111110 | ¥     |
| ¿    | 191 | 0xBF | 0b10111111 | ┐     |

**Table A-2. UTF-8 and ASCII character table (continued) – extended character set.**

| UTF8 | Dec | Hex  | Binary     | ASCII | UTF8 | Dec | Hex  | Binary     | ASCII |
|------|-----|------|------------|-------|------|-----|------|------------|-------|
| À    | 192 | 0xC0 | 0b11000000 | ┌     | à    | 224 | 0xE0 | 0b11100000 | Ó     |
| Á    | 193 | 0xC1 | 0b11000001 | └     | á    | 225 | 0xE1 | 0b11100001 | β     |
| Â    | 194 | 0xC2 | 0b11000010 | ┌└    | â    | 226 | 0xE2 | 0b11100010 | Ô     |
| Ã    | 195 | 0xC3 | 0b11000011 | └┌    | ã    | 227 | 0xE3 | 0b11100011 | Ò     |
| Ä    | 196 | 0xC4 | 0b11000100 | ┌┐    | ä    | 228 | 0xE4 | 0b11100100 | ö     |
| Å    | 197 | 0xC5 | 0b11000101 | └┑    | å    | 229 | 0xE5 | 0b11100101 | Ï     |
| Æ    | 198 | 0xC6 | 0b11000110 | æ     | æ    | 230 | 0xE6 | 0b11100110 | μ     |
| Ç    | 199 | 0xC7 | 0b11000111 | Ç     | ç    | 231 | 0xE7 | 0b11100111 | þ     |
| È    | 200 | 0xC8 | 0b11001000 | È     | è    | 232 | 0xE8 | 0b11101000 | ƒ     |
| É    | 201 | 0xC9 | 0b11001001 | É     | é    | 233 | 0xE9 | 0b11101001 | Ú     |
| Ê    | 202 | 0xCA | 0b11001010 | Ê     | ê    | 234 | 0xEA | 0b11101010 | Û     |
| Ë    | 203 | 0xCB | 0b11001011 | Ë     | ë    | 235 | 0xEB | 0b11101011 | Ü     |
| Ì    | 204 | 0xCC | 0b11001100 | Ì     | ì    | 236 | 0xEC | 0b11101100 | ý     |
| Í    | 205 | 0xCD | 0b11001101 | Í     | í    | 237 | 0xED | 0b11101101 | Ý     |
| Î    | 206 | 0xCE | 0b11001110 | Î     | î    | 238 | 0xEE | 0b11101110 | ˆ     |
| Ï    | 207 | 0xCF | 0b11001111 | Ï     | ï    | 239 | 0xEF | 0b11101111 | ˙     |
| Ð    | 208 | 0xD0 | 0b11010000 | ð     | ð    | 240 | 0xF0 | 0b11110000 | ˘     |
| Ñ    | 209 | 0xD1 | 0b11010001 | Ñ     | ñ    | 241 | 0xF1 | 0b11110001 | ±     |
| Ò    | 210 | 0xD2 | 0b11010010 | Ò     | ò    | 242 | 0xF2 | 0b11110010 | ≡     |
| Ó    | 211 | 0xD3 | 0b11010011 | Ó     | ó    | 243 | 0xF3 | 0b11110011 | ¾     |
| Ô    | 212 | 0xD4 | 0b11010100 | Ô     | ô    | 244 | 0xF4 | 0b11110100 | ¶     |
| Õ    | 213 | 0xD5 | 0b11010101 | Õ     | õ    | 245 | 0xF5 | 0b11110101 | §     |
| Ö    | 214 | 0xD6 | 0b11010110 | Ö     | ö    | 246 | 0xF6 | 0b11110110 | ÷     |
| ×    | 215 | 0xD7 | 0b11010111 | ×     | ÷    | 247 | 0xF7 | 0b11110111 | ˚     |
| Ø    | 216 | 0xD8 | 0b11011000 | Ø     | ø    | 248 | 0xF8 | 0b11111000 | °     |
| Ù    | 217 | 0xD9 | 0b11011001 | Ù     | ù    | 249 | 0xF9 | 0b11111001 | ˙˙    |
| Ú    | 218 | 0xDA | 0b11011010 | Ú     | ú    | 250 | 0xFA | 0b11111010 | •     |
| Û    | 219 | 0xDB | 0b11011011 | Û     | û    | 251 | 0xFB | 0b11111011 | ₁     |
| Ü    | 220 | 0xDC | 0b11011100 | Ü     | ü    | 252 | 0xFC | 0b11111100 | ₃     |
| Ý    | 221 | 0xDD | 0b11011101 | Ý     | ý    | 253 | 0xFD | 0b11111101 | ₂     |
| Þ    | 222 | 0xDE | 0b11011110 | Þ     | þ    | 254 | 0xFE | 0b11111110 | ■     |
| ß    | 223 | 0xDF | 0b11011111 | ß     | ÿ    | 255 | 0xFF | 0b11111111 | nb    |

The UTF-8 codes of Table A-2 were generated using the following sketch:

```
// Generate UTF-8 table
void setup(){
 Serial.begin(9600);
 for(byte i=0;i<256;i++){
```

```

 Serial.write(i);
 Serial.print(", "+(String)i+", 0x");
 Serial.print(i,HEX);
 Serial.print(", 0b");
 Serial.print(i,BIN);
 Serial.println(""); // new line
 }
}

void loop(){}

```

*ASCII codes were obtained from: <https://theasciicode.com.ar/ascii-codes.txt> (PlaneTa MarTes 2008)*

*For more information: <https://playground.arduino.cc/Code/UTF-8>*

## Tips to Optimize Sketch Memory

On a desktop PC, there is less incentive to code efficiently. Processing speed is blazing fast, and memory is ample. On a small MCU, this is not the case. A poorly written sketch will result in noticeably sluggish performance. When they compile, most sketch commands are saved to *flash memory*, and variables you declare are saved to *SRAM*, both of which have relatively low limits. The ATmega328 chip has 32K of flash memory for sketch space, and 2K of SRAM for your variables. When you compile a sketch, the compiler messages at the bottom of the Arduino IDE will let you know how much memory you have used up:

```

Sketch uses 4294 bytes (13%) of program storage
space. Maximum is 32256 bytes.
Global variables use 1852 bytes (90%) of dynamic
memory, leaving 196 bytes for local variables.
Maximum is 2048 bytes.
Low memory available, stability problems may occur.

```

The compiler will give you a warning like the one above if you get too close to memory limits. Long sketches can compile down to surprisingly small sizes; however, if you start to combine libraries (e.g. an LCD serial library with a network card library) you may approach the 32K flash/2K SRAM limits very quickly. You might be staring at the compiler window, wondering how to make all your code fit onto the chip. It becomes important then to optimize your code, to reduce the memory requirements of your sketch. Here are some quick strategies to help you trim down your memory requirements.

### 1) *Be frugal with declaring variable types.*

Use the following chart to help you decide what type of variable to declare. If you don't need the range of a larger variable, then don't use it. Some programmers stay away from float variables at all costs.

**Table A-3. ATmega328 variable sizes, and the ranges of values they can store.**

| <i>Variable type declaration</i> | <i>Bits</i> | <i>Number range</i>                                       |
|----------------------------------|-------------|-----------------------------------------------------------|
| bool myVar=0;                    | 1           | 0 or 1, FALSE or TRUE, LOW or HIGH                        |
| char myVar=0;                    | 8           | -128 to 127                                               |
| unsigned char myVar=0;           | 8           | 0 to +255 (same as byte)                                  |
| byte myVar=0;                    | 8           | 0 to +255                                                 |
| int myVar=0;                     | 16          | -32768 to +32767                                          |
| unsigned int myVar=0U;           | 16          | 0 to +65535                                               |
| long myVar=0L;                   | 32          | -2,147,483,648 to +2,147,483,647                          |
| unsigned long myVar=0UL;         | 32          | 0 to +4,294,967,295                                       |
| float myVar=0.f;                 | 32          | $-3.4028235 \times 10^{38}$ to $3.4028235 \times 10^{38}$ |

For more information: <https://learn.sparkfun.com/tutorials/data-types-in-arduino>

**String vs. char[]:** The programming community favours *arrays of char variables* over Strings because of their efficiency in memory. For example,

```
char myMessage[6]="hello";//char array, txt length+1
char myMessage[]="hello"; //let compiler decide length
```

is preferred to:

```
String myMessage="hello";
```

Once you have declared the above variables, you could use the command:

```
Serial.print(myMessage);
```

for either variable type. Recall that *char arrays* need to be one element larger than the text length, to save room for the **null character** (ASCII code 0). This signals the end of the string to the compiler, and is added automatically. That's why we declared an array length of [6] for our five-character "hello" example.

As mentioned in Section 3, to make matters slightly confusing, many programmers use the term "string" (with a lower case "s") and "character array" interchangeably, both referring to *char arrays* (despite the existence of *String* variables). This is something to be aware of when you are reviewing other people's sketches.



For more information:

<https://hackingmajenkoblog.wordpress.com/2016/02/04/the-evils-of-arduino-strings/>

## 2) Reduce the number of variables – especially global ones.

Global variables take up space immediately. Functions can use memory slightly more efficiently, because the variable is declared at the start of the function and then destroyed after the function finishes, so this will not appear as part of the overall memory tally when you compile. If you have declared a global variable that only needs to be local, then move its declaration inside the function that uses it. Save global space for the variables that really need to be global.

While you are at it, reducing the number of variables can also make a difference in your code. Consider the following code:

```
int divs=analogRead(A1); // take analog reading
float volts=divs*3.3/1023.0; // div->volt
float weight=scaleInt+scaleCal*volts; // volt->mg
```

Do three different variables really need to be declared here? The following code would still work:

```
float weight=analogRead(A1); // take analog reading
weight=weight*3.3/1023.0; // div->volt
weight=scaleInt+weight*scaleCal; // volt->mg
```

## 3) Use #define statements instead of constants.

The Arduino community seems to have a love/hate relationship with #define statements. As mentioned in Section 4, #define statements are a way of replacing a variable with a value. Of the following two statements:

```
byte ledPin=13;
#define ledPin 13
```

the first command uses one byte of SRAM, whereas the second command uses the more-ample flash memory instead.

For more information: <https://www.arduino.cc/en/Tutorial/Memory>

## 4) Optimize Serial.print() commands.

Serial.print() commands are wonderful. They are a brilliant and convenient way of debugging your program. However, they are memory hogs, and slow your code down.

If you *do* need serial commands, the *text* that is uploaded in a command like this:

```
Serial.print("Pharmacy Rocks!");
```

takes up a lot of space in SRAM. One way of saving memory is by sending the string inside the brackets to flash memory by using the following function:

```
Serial.print(F("Pharmacy Rocks!"));
```

This only works for strings, and not variables. The following code *would not compile* because the second line tries to load a variable into flash memory:

```
Serial.print(F("Temperature: "));
Serial.print(F(myTemp));
```

Shortening the text to be printed inside Serial.print() commands will also help free up memory. Once you have finished your sketch and confirmed that it works, comment out all unnecessary serial commands, abbreviate the remainder, and send any strings to flash memory using the F() function. This will recover much needed memory and speed up your sketch.

### 5) Use *PROGMEM* to store arrays of constant values to flash memory.

If you aren't planning on changing the contents of an array, then you can store the array in flash memory rather than SRAM, using the PROGMEM command. This is great for example when you are displaying graphics to an LCD, and using an array to store the individual bytes for pixels. PROGMEM works for all data types, but is really only worth using for arrays. Here are two arrays of bytes defined without and with PROGMEM:

```
// saves array to SRAM (wasteful):
const byte myBitmap[] = {
0x1F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0x00, 0x1F, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x3F,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF
};
```

```
// saves array to FLASH (better):
const byte myBitmap[] PROGMEM = {
0x1F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0x00, 0x1F, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x3F,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF
};
```

```
};
```

## **6) Use External Memory.**

There are inexpensive DIP chips available that can provide additional memory for your project. Two such chips are the 23LC1024 (SPI serial SRAM with SDI and Sqi interface), and 25LC1024 (SPI bus serial EEPROM). The 23LC1024 chip has 1024 Mbits of memory (or 128K). This means a single chip could hold 64,000 integers, or 32,000 float values, freeing up your microprocessor for more sketch room. (Microchip Technology Inc 2015) The biggest difference between these two types of memory is that data stored in SRAM will be erased when the chip is powered down, whereas EEPROM will still hold the data. However, EEPROM has a limit to the number of times you can write to the chip before becoming unreliable (1 million erase/write cycles). (Microchip Technology Inc 2008) This may sound like a large number, but if you are taking readings every few milliseconds, the number of erase/write cycles can add up quickly.

You can find libraries to support both these chips. A library *SRAMsimple* for the 23LC1024 is available at <https://github.com/dndubins/SRAMsimple>. (Dubins 2018)

## **7) Use a microprocessor with more memory, or use more than one microprocessor.**

Your project might be too ambitious for the Arduino Uno, even after optimizing space. You can then consider perhaps using two microprocessors and giving them different jobs, or alternately, you can use a microprocessor with more memory. Table A-4 summarizes the memory limits of a few different microprocessor development boards and MCUs. For a more current list of microprocessor boards in the Arduino line, see <https://store.arduino.cc/usa/arduino/boards-modules>.

**Table A-4. Some microprocessors and their memory limits. (Arduino.cc 2018; Stör et al. 2017; Paul Stoffregen 2019)**

| <i>Microprocessor Board</i>                                       | <i>Processor Chip (MCU)</i>                                     | <i>Flash Memory</i> | <i>SRAM Memory</i> | <i>EEPROM Memory</i>   |
|-------------------------------------------------------------------|-----------------------------------------------------------------|---------------------|--------------------|------------------------|
| Lily Tiny, Gemma                                                  | ATtiny85                                                        | 8 kb                | 512 b              | 512 b                  |
| Arduino Uno, Nano, Mini                                           | ATmega328                                                       | 32 kb               | 2 kb               | 1 kb                   |
| Arduino Leonardo, Arduino Yun, Arduino Micro, LilyPad, Teensy 2.0 | ATmega32u4                                                      | 32 kb               | 2.5 kb             | 1 kb                   |
| Arduino Mega, Teensyduino (v1)                                    | ATmega2560                                                      | 256 kb              | 8 kb               | 4 kb                   |
| ESP-01, NodeMCU, LinkNode D1                                      | ESP8266                                                         | 1 MB                | 32 kb              | 4 kb                   |
| Arduino MKR Zero, MKR 1000, MKR Wifi 1010                         | SAMD21 Cortex-M0+ (3.3V)                                        | 256 kb              | 32 kb              | Up to 16 kb (emulated) |
| Arduino Due                                                       | SAM3X8E ARM Cortex-M3 CPU (3.3V)                                | 512 kb              | 100 kb             | -                      |
| Teensy ++ 2.0                                                     | AT90USB1286<br>8 bit AVR<br>16 MHz (5V)                         | 128 kb              | 8 kb               | 4 kb                   |
| Teensy 3.0                                                        | MK20DX128<br>32 bit ARM Cortex-M4<br>48 MHz (3.3V)              | 128 kb              | 16 kb              | 2 kb                   |
| Teensy 3.1, Teensy 3.2                                            | MK20DX256<br>32 bit ARM Cortex-M4<br>72 MHz (3.3V, 5V tolerant) | 256 kb              | 64 kb              | 2 kb                   |
| Teensy 3.5                                                        | MK64FX512<br>32 bit, 120 MHz Cortex-M4F (3.3V, 5V tolerant)     | 1 Mb                | 256 kb             | Emulated               |

|            |                                                      |       |        |          |
|------------|------------------------------------------------------|-------|--------|----------|
| Teensy 3.6 | MK66FX1M0<br>32 bit, 180 MHz<br>Cortex-M4F<br>(3.3V) | 2 Mb  | 256 kb | Emulated |
| Teensy LC  | 48 MHz Cortex-<br>M0+ (3.3V)                         | 62 kb | 8 kb   | Emulated |

It may seem daunting to learn how to use a new microprocessor board, but it isn't as difficult as you might imagine. Platforms like the ESP8266 and the Teensy series have their own add-on software and libraries which can allow you to program them through the Arduino IDE, using the same or similar commands and libraries. For setting up the Arduino IDE to program ESP8266 boards like the NodeMCU and LinkNode D1, visit <https://github.com/esp8266/Arduino>. For setting up the Arduino IDE to program Teensy devices, visit <https://www.pjrc.com/teensy/tutorial.html>. For setting up the Arduino IDE to program the ATtiny45 or ATtiny85 MCU, visit <https://github.com/damellis/attiny>. Sometimes, the biggest transition to another board is figuring out where the digital and analog pins are located, and which are PWM-enabled. However, compatibility issues can arise with using Arduino libraries with non-supported microprocessors.

### 8) *Streamline your strategy.*

Review your sketch. Are there routines that can be shortened or simplified? Have a look at the scope of your project, and what you are trying to do. Is it too complicated? Is every component of your project essential? Your project may be too ambitious for the Arduino Uno. For instance, artificial intelligence routines are painfully slow for the Uno, and take up lots of memory.

## Using a 555 Timer as an External Clock

In *Customized Frequencies for PWM* of Section 10, we focused on how to set the internal timers of the ATmega328 to generate a desired frequency or duty cycle. Occasionally, if you run out of pins, or perhaps want an external clock, it is helpful to know how to generate your own signal. The NE555 or LM555 chips can generate square waves up to about 400 kHz, and the CMOS version (LMC555) can generate frequencies as high as 3-4 MHz.

The following circuit diagram configures the 555 timer as a 50% duty cycle oscillator in its *astable mode*, with a square wave output on pin 3. This

output can serve as a stable clock signal. If a 10K trim potentiometer is used for  $R_2$ , commonly available capacitors can be used to span a wide frequency range from ~120 Hz to 3 MHz. (Texas Instruments Inc 2016b)

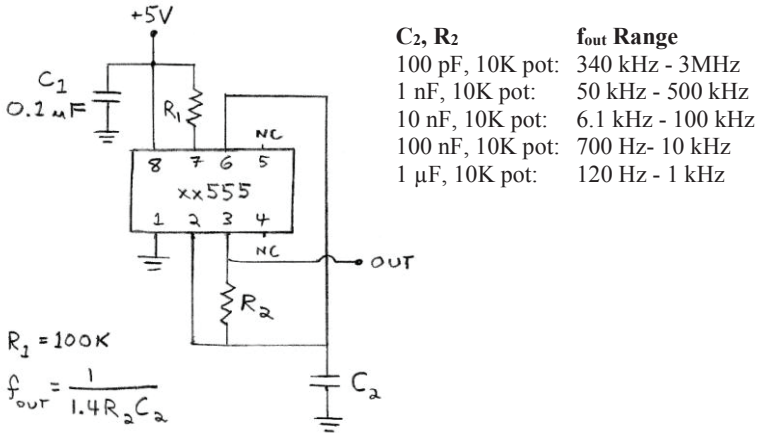


Figure A-3. A 50% duty cycle square wave generator configuration for the 555 timer (LMC555 CMOS version required above ~400 kHz). Ranges for  $f_{out}$  were obtained experimentally using a LMC555 timer and monolithic capacitors.

## Common Fixed Resistor and Capacitor Values

It's easier to build if you plan on using resistor values that are not difficult to source. The following tables, generated from the laboratory inventory, list some common fixed resistor and capacitor values.

**Table A-5. Common fixed resistor values.**

|              |             |              |                |               |                |                |
|--------------|-------------|--------------|----------------|---------------|----------------|----------------|
| 0 $\Omega$   | 10 $\Omega$ | 100 $\Omega$ | 1 k $\Omega$   | 10 k $\Omega$ | 100k $\Omega$  | 1 M $\Omega$   |
| 1 $\Omega$   | 15 $\Omega$ | 110 $\Omega$ | 1.5 k $\Omega$ | 15 k $\Omega$ | 150 k $\Omega$ | 1.5 M $\Omega$ |
| 2.2 $\Omega$ | 22 $\Omega$ | 120 $\Omega$ | 2k $\Omega$    | 18 k $\Omega$ | 180 k $\Omega$ | 2 M $\Omega$   |
| 4.7 $\Omega$ | 27 $\Omega$ | 150 $\Omega$ | 2.4 k $\Omega$ | 22 k $\Omega$ | 200 k $\Omega$ | 3.3 M $\Omega$ |
| 5.6 $\Omega$ | 33 $\Omega$ | 180 $\Omega$ | 2.7 k $\Omega$ | 33 k $\Omega$ | 220 k $\Omega$ | 4.7 M $\Omega$ |
| 7.5 $\Omega$ | 39 $\Omega$ | 200 $\Omega$ | 3 k $\Omega$   | 39 k $\Omega$ | 330 k $\Omega$ | 5 M $\Omega$   |
| 8.2 $\Omega$ | 47 $\Omega$ | 220 $\Omega$ | 3.3 k $\Omega$ | 47 k $\Omega$ | 470 k $\Omega$ | 5.1 M $\Omega$ |
|              | 56 $\Omega$ | 240 $\Omega$ | 3.9 k $\Omega$ | 56 k $\Omega$ | 560 k $\Omega$ | 5.6 M $\Omega$ |
|              | 68 $\Omega$ | 270 $\Omega$ | 4.7 k $\Omega$ | 68 k $\Omega$ | 680 k $\Omega$ | 10 M $\Omega$  |
|              | 75 $\Omega$ | 300 $\Omega$ | 5k $\Omega$    | 75 k $\Omega$ | 820 k $\Omega$ | 12 M $\Omega$  |
|              | 82 $\Omega$ | 330 $\Omega$ | 5.6 k $\Omega$ | 82 k $\Omega$ |                |                |
|              |             | 390 $\Omega$ | 6.8 k $\Omega$ |               |                |                |
|              |             | 400 $\Omega$ | 7.4 k $\Omega$ |               |                |                |
|              |             | 470 $\Omega$ | 8.2 k $\Omega$ |               |                |                |
|              |             | 500 $\Omega$ |                |               |                |                |
|              |             | 680 $\Omega$ |                |               |                |                |
|              |             | 820 $\Omega$ |                |               |                |                |

**Table A-6. Common fixed capacitor values.**

|      |       |        |        |         |             |              |
|------|-------|--------|--------|---------|-------------|--------------|
| 1 pF | 10 pF | 100 pF | 1 nF   | 10 nF   | 1 $\mu$ F   | 100 $\mu$ F  |
| 2 pF | 15 pF | 120 pF | 1.5 nF | 15 nF   | 2.2 $\mu$ F | 150 $\mu$ F  |
| 3 pF | 18 pF | 140 pF | 2 nF   | 20 nF   | 4.7 $\mu$ F | 220 $\mu$ F  |
| 4 pF | 20 pF | 150 pF | 2.2 nF | 22 nF   | 10 $\mu$ F  | 470 $\mu$ F  |
| 5 pF | 22 pF | 180 pF | 3.3 nF | 33 nF   | 22 $\mu$ F  | 1000 $\mu$ F |
| 6 pF | 27 pF | 220 pF | 4.7 nF | 37.7 nF | 47 $\mu$ F  |              |
| 7 pF | 30 pF | 270 pF | 5 nF   | 47 nF   |             |              |
| 8 pF | 33 pF | 300 pF | 6.8 nF | 68 nF   |             |              |
| 9 pF | 40 pF | 330 pF |        | 100 nF  |             |              |
|      | 47 pF | 470 pF |        | 220 nF  |             |              |
|      | 50 pF | 560 pF |        |         |             |              |
|      | 56 pF | 680 pF |        |         |             |              |
|      | 68 pF | 820 pF |        |         |             |              |
|      | 82 pF |        |        |         |             |              |

## .ino Files

### *triacDimmer.ino (Section 5)*

```
// Example of forward-phase dimming (Figure 5-24, 5-25)
// Set up a 10K pot as a voltage divider on pin A0
byte crossPin=2; // Uno Pin 2 -- H11AA1 Pin 5
byte triacPin=3; // Uno Pin 3 -- 400R -- MOC3010 Pin 1
int dur=0; // duration for dimming

void setup(){
 pinMode(crossPin,INPUT_PULLUP);
 pinMode(triacPin,OUTPUT);
 Serial.begin(9600);
}

void loop(){
 triacON(analogRead(A0)); // use 10K pot on A0
 Serial.println(dur);
}

void triacISR(){
 delayMicroseconds(dur);
 digitalWrite(triacPin,HIGH); // turn on TRIAC
 delayMicroseconds(10);
 digitalWrite(triacPin,LOW); // turn off TRIAC
}

void triacON(int drive){ // drive range: 0 to 1023
 int tMIN=1300; //min delay for triac (expt1, ~350)
 int tMAX=7660; //max delay for triac (expt1, ~8100)
 dur=map(drive,0,1023,tMIN,tMAX); // scale dur
 attachInterrupt(0,triacISR,RISING); // ISR to pin 2
}

void triacOFF(){ // use this to shut triac off
 detachInterrupt(0);
 digitalWrite(triacPin,LOW);
}
```

### *Thermostat.ino (Section 5)*

```
/* Thermostat.ino: Simple thermostat program (no sensor data
 * averaging)
 * Connections:
 * +3.3V--10K resistor--pin A1--10K thermistor--GND
```



```

* 3.3V to AREF pin
* Pin 6 to relay module
* (or Pin 6 to LED--220R resistor--GND to test)
*/

byte tempPin=A1; // Declaring the Analog input to be A1
 // of Arduino board.
byte relayPin=6; // For an LED (or relay)
float tempC=0.0; // For holding Celcius temp (floating
 // for decimal points precision)
float R = 0.0; // Variable for reading the measured
 // resistance of the thermistor
float R0 = 11930; // Resistance (Ohms) of thermistor at
 // room temperature
float T0 = 273.15 + 22.5; //Room temperature in Kelvin
float R1 = 10000.0; // Resistance (Ohms) of sense resistor
 // in voltage divider (should be ~10000
 // Ohms, measure for better accuracy)
float volts = 0.0; // Variable to read in voltage (to be
 // converted to resistance)
float B = 3672.433; // B Coefficient of Thermistor. Enter B
 // coefficient here.
float setTemp = 37.0; // Setpoint temperature for thermostat

void setup(){
 analogReference(EXTERNAL); // Use external (3.3V) AREF.
 // Make sure 3.3V connected to AREF pin
 // or or analogRead won't work!
 Serial.begin(9600); // Open serial port, set to 9600 bps
 Serial.println("Volts(V), Resistance(Ohm), Temperature(C)");
 // Set up column titles
 pinMode(relayPin, OUTPUT); // Set up relayPin as output to
 // send a digital signal to the relay
 // module.
}

void loop(){
 readTemp(); // Read the temperature (void fn below)
 // Now ouput the results to the serial monitor:
 Serial.print(volts); // Print Volts
 Serial.print(","); // Print a comma for CSV file
 Serial.print(R); // Print Resistance
 Serial.print(","); // Print a comma for CSV file
 Serial.println(tempC,2); // Print Temperature and new line
 if(tempC < setTemp){ // if measured temp less than set temp
 digitalWrite(relayPin, HIGH); // then turn on relay
 } else {
 digitalWrite(relayPin, LOW); // otherwise turn off relay
 }
 delay(1000); // Wait 1sec before taking next measure
}

void readTemp(){

```

```

volts = analogRead(tempPin) * 3.3 / 1023.0; // Take sensor
// reading from tempPin in divs (Scale
// 0-1023), and convert to volts
R = volts*R1/(3.3-volts); // Convert voltage to resistance
tempC = (1/T0)+((1/B)*log(R/R0)); // Use the two-term
// exponential thermistor equation to
// calculate temperature
tempC = 1.0/tempC; // invert the answer
tempC = tempC - 273.15; // Convert from Kelvin to Celsius
}

```

### ***4WStepper.ino: 4-Wire Stepper Control (Section 6)***

```

// Generic 4-Wire Stepper Motor Control
#include <Stepper.h> // Arduino IDE built-in library
const byte stepsPerRev = 200; // change as needed for motor
// initialize the stepper library on pins 8 through 11:
Stepper myStepper(stepsPerRev, 8, 9, 10, 11);

void setup(){
 myStepper.setSpeed(60); // set speed to 60 rpm
 Serial.begin(9600);
}

void loop(){
 myStepper.step(200); // clockwise one rotation
 delay(1000);
 myStepper.step(-200); // counter-clockwise one rotation
 delay(1000);
}

```

### ***4WStepper\_noLib: 4-Wire Stepper Control (no library required)***

```

/* 4-Wire Generic Stepper (capable of full and half-steps)
 * see: http://www.hurst-motors.com/Technical_Help.html
 * for full and half-step sequences, and more info
 *
 * Motor Driver to MCU:
 * IN1 to D8
 * IN2 to D9
 * IN3 to D10
 * IN4 to D11
 * +5V to MCU +5V (or external supply Vcc)
 * GND to MCU GND (and external supply GND if used)
 */

const int stepsPerRev=20; // change as needed for motor
const byte IN[4]={8,9,10,11}; // define motor pins as array

void setup(){
 Serial.begin(9600);
}

```

```

 for(int i=0;i<4;i++){
 pinMode(IN[i],OUTPUT);
 }
}

void loop(){
 Serial.println("Stepping clockwise in full steps.");
 motorStep(200,10); // clockwise 200 steps @10rpm
 delay(1000);
 Serial.println("Stepping counter-clockwise in full steps.");
 motorStep(-200,10); // counter-clockwise 200 steps @10rpm
 delay(1000);
 Serial.println("Stepping clockwise in half steps.");
 motorStepHalf(200,10); // CW 200 half steps @10rpm
 delay(1000);
 Serial.println("Stepping counter-clockwise in half steps.");
 motorStepHalf(-200,10); // CCW 200 half steps @10rpm
 delay(1000);
}

void motorStep(int mSteps, float rpm){
 //convert rpm to time delay:
 float t=60000.0/(rpm*stepsPerRev);
 const bool mSequence[4][4]={
 {1, 0, 0, 1}, // step 0
 {1, 0, 1, 0}, // step 1
 {0, 1, 1, 0}, // step 2
 {0, 1, 0, 1} // step 3
 };
 static int mStep; // remember last val of mStep
 for(int i=0;i<abs(mSteps);i++){ // STEP pulses
 if(mSteps>0){ // clockwise
 mStep++;
 if(mStep>3)mStep=0;
 }else{ // counter-clockwise
 mStep--;
 if(mStep<0)mStep=3;
 }
 for(int j=0;j<4;j++){
 digitalWrite(IN[j],mSequence[mStep][j]);
 }
 delay_(t);
 }
}

void motorStepHalf(int mSteps, float rpm){
 float t=60000.0/(rpm*stepsPerRev*2.0);
 const bool mSequence[8][4]={ // for motor sequence
 {0, 1, 0, 1}, // step 0
 {0, 0, 0, 1}, // step 1
 {1, 0, 0, 1}, // step 2
 {1, 0, 0, 0}, // step 3
 {1, 0, 1, 0}, // step 4
 };
}

```

```

 {0, 0, 1, 0}, // step 5
 {0, 1, 1, 0}, // step 6
 {0, 1, 0, 0} // step 7
};
static int mStep; // remember last val of mStep
for(int i=0;i<abs(mSteps);i++){ // STEP pulses
 if(mSteps>0){ // clockwise
 mStep++;
 if(mStep>7)mStep=0;
 }else{ // counter-clockwise
 mStep--;
 if(mStep<0)mStep=7;
 }
 for(int j=0;j<4;j++){
 digitalWrite(IN[j],mSequence[mStep][j]);
 }
 delay_(t);
}
}

void delay_(float x){ // allows for delays <1ms
 if(x>1.0){
 delay(x);
 }else{
 delayMicroseconds(x*1000.0); //convert to usec
 }
}
}

```

### ***PID.ino (Section 6)***

```

/*
PID.ino: Control a drive element using a PID strategy
Connections:
+5V to LM35 Pin 1
MCU Pin A0 to LM35 Pin 2
GND to LM35 Pin 3
MCU Pin 11 to the base of an NPN transistor
DC fan red wire to ATX +12V (yellow wire)
DC fan black wire to the collector of an NPN transistor
*/

byte DRIVEPin = 11; // Digital pin for PWM fan control
byte MESPin = A0; // Analog pin A0 for temperature
// measurement
float SETPOINT = 21.0; // Setpoint temperature: what you would
// like temperature to be
float MEASURED = 0.0; // Variable to store measured
// temperature

void setup(){
 Serial.begin(9600); // Open serial port, set to 9600 bps
 pinMode(DRIVEPin, OUTPUT); // Generally not needed for PWM
}

```

```

void loop(){
 myPID(1.0, 0.0, 0.0); // call PID control here, entering
 // values for kP, kI, and kD (other
 // options available in PID subroutine)
 delay(300); // optional delay statement
}

void myPID(float kP, float kI, float kD){
 /*
 PID Algorithm:
 MEASURED: Measured value for feedback
 MESPIn: pin to measure
 SETPOINT: value you would like the system to converge to
 DRIVEpin: pin to send drive signal
 TOLERANCE: Acceptable range from SETPOINT (adjust the system
until Error is within TOLERANCE).
 kP: Proportional gain; kI: Integrator gain; kD: Derivative
Gain
 IntThresh: Only run integrator if Error < IntThresh (if
control is almost done). Make this smallish, but greater than
TOLERANCE (otherwise it will never engage).
 ScaleFactor: factor to scale P+I+D to a response (then
after, impose 0-255 limits). A negative scale factor here will
deactivate DRIVE if MEASURED > SETPOINT.
 */
 float TOLERANCE = 1.0; // tolerance of control (in this
 // case, 1 means that PID won't do
 // anything if measured temperature
 // is within 1 degC of the setpoint)
 float IntThresh=5.0; // narrow region of integral term
 // (proximity to SET)
 float ScaleFactor=-1.0; // Use this to change direction and
 // rescale DRIVE if necessary

 float Error = 0.0;
 float Integral = 0.0; // make this a global variable if not
 // using the do-while loop

 float P = 0.0;
 float I = 0.0;
 float D = 0.0;
 float LAST = 0.0; // Make this a global variable if you
 // plan on using this routine to
 // continuously check (keep track of
 // last globally)

 long DRIVE = 0.0; // Nothing quite like a long drive.
do { // You can get rid of the do..while
 // loop if you want the PID routine to
 // adjust the drive ONCE each function
 // call, i.e. not keep going until you
 // reach the SETPOINT.
 // This might be important if your
 // sketch needs to do other things!
 MEASURED = analogRead(MESPIn) * 500.0/1023.0; // convert

```

```

// from divs to degC
Error = SETPOINT - MEASURED;
if (Error < IntThresh){ // prevent integral wind-up by
 // only engaging it close to SETPOINT.
 Integral = Integral + Error; // add to Error Integral
} else {
 Integral=0.0; // zero Integral if out of bounds
}
P = Error*kP; // calculate proportional term
I = Integral*kI; // integral term
D = (LAST-MEASURED)*kD; // derivative term
DRIVE = P + I + D; // Total DRIVE = P+I+D
DRIVE = (long)(DRIVE*ScaleFactor); // scale DRIVE to be in
// the range 0-255
DRIVE = constrain(DRIVE,0,255); // make sure DRIVE is an
// integer between 0 and 255. An offset
// can be used to get the DRIVE moving,
// e.g. constrain(68+DRIVE,0,255).
Serial.print("SETPOINT: "); // info to serial monitor
Serial.print(SETPOINT);
Serial.print(", MEASURED: ");
Serial.print(MEASURED);
Serial.print(", P: ");
Serial.print(P);
Serial.print(", I: ");
Serial.print(I);
Serial.print(", D: ");
Serial.print(D);
Serial.print(", DRIVE: ");
Serial.println(DRIVE);
// replace all Serial.print commands
// with the following command to
// observe the response on the serial
// plotter:
//Serial.println((String)MEASURED+", "+(String)SETPOINT);
LAST = MEASURED; // save current value for next time
analogWrite(DRIVEPin, DRIVE); // send DRIVE as PWM signal
} while (abs(SETPOINT-MEASURED)>TOLERANCE); // End of DO
// loop condition. do..while will
// always run at least once (while is
// tested at the end of the loop).
// Comment out if removing do loop.
Serial.println("SETPOINT within TOLERANCE.");
}

```

### ***QuickStats.h (Section 8)***

```

// QuickStats.h Library for Data Smoothing
// For a standalone version of this library, visit:
// https://github.com/dndubins/QuickStats

#include <math.h>

```

```

//#define DEBUG // uncomment for debugging messages

float average(float samples[],int m){
 float total1=0.0;
 for(int i=0;i<m;i++){
 total1=total1+samples[i];
 }
 return total1/(float)m;
}

float g_average(float samples[],int m){
 float total1=0.0;
 for(int i=0;i<m;i++){
 total1=total1+log(samples[i]);
 }
 return exp(total1/(float)m);
}

float stdev(float samples[],int m){
 float avg=0.0;
 float total2=0.0;
 avg=average(samples,m);
 for(int i=0;i<m;i++){
 total2 = total2 + pow(samples[i] - avg,2);
 }
 return sqrt(total2/((float)(m-1)));
}

float CV(float samples[],int m){ //Coefficient of variation
 float avg=0.0;
 float sd=0.0;
 avg=average(samples,m);
 sd=stdev(samples,m);
 return 100.0*sd/avg;
}

void bubbleSort(float A[],int len){
 unsigned long newn;
 unsigned long n=len;
 float temp=0.0;
 do {
 newn=1;
 for(int p=1;p<len;p++){
 if(A[p-1]>A[p]){
 temp=A[p]; //swap places in array
 A[p]=A[p-1];
 A[p-1]=temp;
 newn=p;
 } //end if
 } //end for
 n=newn;
 } while(n>1);
}

```

```

float median(float samples[],int m){ //calculate the median
//First bubble sort the values:
//(algorithm from https://en.wikipedia.org/wiki/Bubble_sort)
float sorted[m]; //Define and initialize sorted array.
float temp=0.0; //Temporary float for swapping elements
#ifdef DEBUG
 Serial.println("Before:");
 for(int j=0;j<m;j++){
 Serial.println(samples[j]);
 }
#endif
for(int i=0;i<m;i++){
 sorted[i]=samples[i];
}
bubbleSort(sorted,m); // Sort the values
#ifdef DEBUG
 Serial.println("After:");
 for(int i=0;i<m;i++){
 Serial.println(sorted[i]);
 }
#endif
if (bitRead(m,0)){ //If the last bit of a number is 1,
//it's odd. This is equivalent to "TRUE". Also use if m%2!=0.
 return sorted[m/2]; //If the number of data points is odd,
//return middle number.
} else {
 return (sorted[(m/2)-1]+sorted[m/2])/2.0; //If the number
//of data points is even, return avg of the middle two
//numbers.
}
}

float mode(float samples[],int m){ //calculate the mode.
//First bubble sort the values:
float sorted[m]; //Temporary array to sort values.
float temp=0; //Temporary float for swapping elements
float unique[m]; //Temporary array to store unique values
int uniquect[m]; //Temporary array to store unique counts
#ifdef DEBUG
 Serial.println("Before:");
 for(int i=0;i<m;i++){
 Serial.println(samples[i]);
 }
#endif
for(int i=0;i<m;i++){
 sorted[i]=samples[i];
}
bubbleSort(sorted,m); // Sort the values
#ifdef DEBUG
 Serial.println("Sorted:");
 for(int i=0;i<m;i++){
 Serial.println(sorted[i]);
 }
}

```



```

#endif
// Now count # times each unique number appears in sorted
// array.
unique[0]=sorted[0];
uniquect[0]=1;
int p=0; // counter for # unique numbers
int maxp=0;
int maxidx=0;
for(int i=1;i<m;i++){
 if(abs(sorted[i]-sorted[i-1])<0.0001){ // tolerance for
//two readings being the same. Set as needed.
 uniquect[p]++; //if same number again, add to count
 if(uniquect[p]>maxp){
 maxp=uniquect[p];
 maxidx=p;
 }
 } else {
 p++;
 unique[p]=sorted[i];
 uniquect[p]=1;
 }
}
}
#ifdef DEBUG
 for(int i=0;i<p+1;i++){
 Serial.println("Num: " + (String)unique[i] + " Count: "
+ (String)uniquect[i]);
 }
#endif
if (maxp>1){
 return unique[maxidx]; //If there is more than one mode,
//return the lowest one.
} else {
 return 0.0; //If there is no mode, return a zero.
}
}

```

### *TimedISR\_N.ino (Section 10)*

```

// Timed ISR to run a routine every n seconds (using Timer 1)
// variables changed in ISRs should be volatile:
unsigned long n=60; // run routine once every minute (60sec)
volatile unsigned long cycles=0; // to hold #cycles
volatile int reading=0; // to store analog reading

void setup(){
 Serial.begin(9600);
 //To set Timer 1 interrupt at 1Hz:
 cli(); //stop interrupts
 TCCR1A=0; // clear timer control register A
 TCCR1B=0; // clear timer control register B
 TCNT1=0; // clear timer counter 1
 TCCR1B=_BV(WGM12); // CTC mode

```

```

TCCR1B|=_BV(CS12); // prescaler=256
TIMSK1|=_BV(OCIE1A); // enable timer compare interrupt
OCR1A=62499; // OCR1A=(fclk/(N*frequency))-1
sei(); //enable interrupts
}

void loop(){
}

ISR(TIMER1_COMP_vect){ // Timer 1 interrupt routine
cycles++; // increment counter
if(cycles>(n-1)){
// Your short routine can go here
reading=analogRead(A0); // only read every n seconds
Serial.println(reading); //comment out for final sketch
cycles=0; // reset cycles
}
}

```

### Derivation for $V_{in(+)}$ (Section 7)

Figure 7-26 showed a resistor network at the non-inverting input of an op-amp. This derivation is provided to show how  $V_{in(+)}$  is calculated.

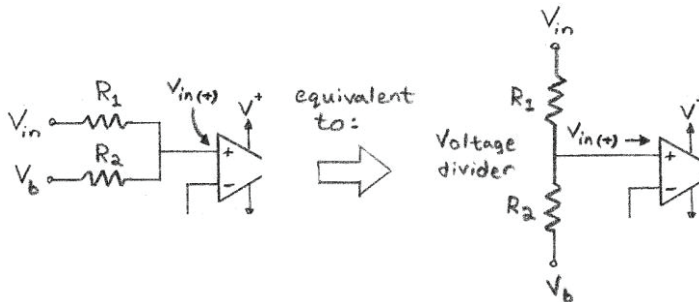


Figure A-4. Voltage divider network at an op-amp input: solving for the input voltage.

The input voltage at the op-amp terminal, in this case,  $V_{in(+)}$ , can be solved the same way as we solved the voltage divider equation. First, assume that virtually no current flows into the op-amp (ideal op-amp assumption). Now, the current flowing from  $V_{in}$  to  $V_b$  will be:

$$I = \frac{\Delta V}{R_{total}} = \frac{V_{in} - V_b}{R_1 + R_2}$$

The voltage difference across resistor  $R_2$  will be:

$$V_2 = IR_2 = \left( \frac{V_{in} - V_b}{R_1 + R_2} \right) R_2$$

$$V_2 = IR_2 = V_{in} \left( \frac{R_2}{R_1 + R_2} \right) - V_b \left( \frac{R_2}{R_1 + R_2} \right)$$

The voltage  $V_{in(+)}$  is then equal to  $V_b + V_2$ : (KVL)

$$V_{in(+)} = V_b + V_2 = V_b + V_{in} \left( \frac{R_2}{R_1 + R_2} \right) - V_b \left( \frac{R_2}{R_1 + R_2} \right)$$

$$V_{in(+)} = V_b + V_{in} \left( \frac{R_2}{R_1 + R_2} \right) - V_b \left( \frac{R_2}{R_1 + R_2} \right)$$

$$V_{in(+)} = V_{in} \left( \frac{R_2}{R_1 + R_2} \right) - V_b \left( \frac{R_2}{R_1 + R_2} - 1 \right)$$

$$V_{in(+)} = V_{in} \left( \frac{R_2}{R_1 + R_2} \right) - V_b \left( \frac{R_2}{R_1 + R_2} - \frac{R_1 + R_2}{R_1 + R_2} \right)$$

$$V_{in(+)} = V_{in} \left( \frac{R_2}{R_1 + R_2} \right) + V_b \left( \frac{R_1}{R_1 + R_2} \right)$$

This is the second term in the non-inverting summing amplifier:

$$V_{out} = \text{Gain} \times V_{in(+)}$$

$$V_{out} = \left( 1 + \frac{R_F}{R_3} \right) \left[ V_{in} \left( \frac{R_2}{R_1 + R_2} \right) + V_b \left( \frac{R_1}{R_1 + R_2} \right) \right]$$

**Table A-7. Pin-out of 20-pin and 24-pin versions of the ATX main power connector. (Fisher 2019)**

| Pin | Colour | Description                      |
|-----|--------|----------------------------------|
| 1   | Orange | +3.3V                            |
| 2   | Orange | +3.3V                            |
| 3   | Black  | GND                              |
| 4   | Red    | +5V                              |
| 5   | Black  | GND                              |
| 6   | Red    | +5V                              |
| 7   | Black  | GND                              |
| 8   | Grey   | +5V when power good (diagnostic) |
| 9   | Purple | +5V standby                      |
| 10  | Yellow | +12V                             |
| 11  | Orange | +3.3V                            |
| 12  | Blue   | -12V                             |
| 13  | Black  | GND                              |
| 14  | Green  | Connect to GND to turn on        |
| 15  | Black  | GND                              |
| 16  | Black  | GND                              |
| 17  | Black  | GND                              |
| 18  | White  | -5V                              |
| 19  | Red    | +5V                              |
| 20  | Red    | +5V                              |

| Pin | Colour | Description                      |
|-----|--------|----------------------------------|
| 1   | Orange | +3.3V                            |
| 2   | Orange | +3.3V                            |
| 3   | Black  | GND                              |
| 4   | Red    | +5V                              |
| 5   | Black  | GND                              |
| 6   | Red    | +5V                              |
| 7   | Black  | GND                              |
| 8   | Grey   | +5V when power good (diagnostic) |
| 9   | Purple | +5V standby                      |
| 10  | Yellow | +12V                             |
| 11  | Yellow | +12V                             |
| 12  | Orange | +3.3V                            |
| 13  | Orange | +3.3V                            |
| 14  | Blue   | -12V                             |
| 15  | Black  | GND                              |
| 16  | Green  | Connect to GND to turn on        |
| 17  | Black  | GND                              |
| 18  | Black  | GND                              |
| 19  | Black  | GND                              |
| 20  | White  | -5V (optional)                   |
| 21  | Red    | +5V                              |
| 22  | Red    | +5V                              |
| 23  | Red    | +5V                              |
| 24  | Black  | GND                              |

20-Pin Numbering:

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |

clip

24-Pin Numbering:

|    |    |    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |

## Arduino Uno Pin-out Diagram

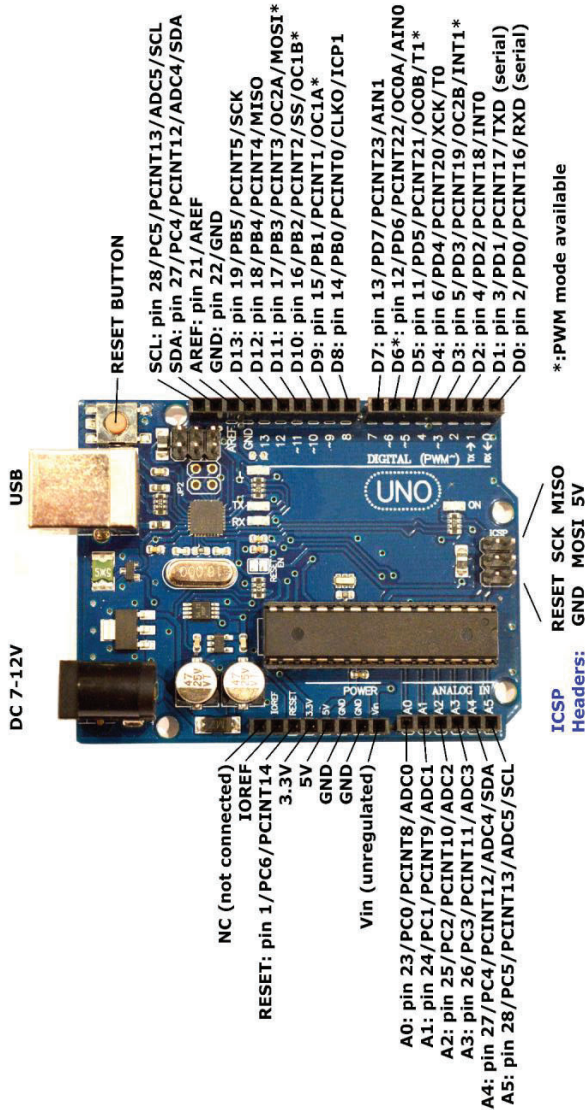


Figure A-5. Pin-out diagram for a generic Arduino Uno r3. Pin numbers (2 to 28) correspond to the pin numbers on the ATmega328 chip. (Atmel Corporation 2016)

### ATmega328 Pin-out Diagram

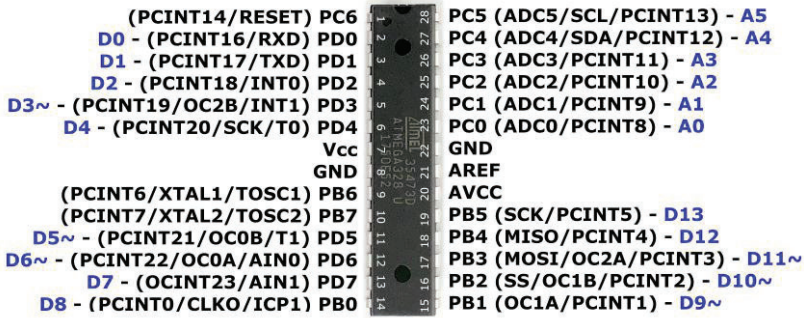


Figure A-6. Pin-out diagram for the ATmega328 28-pin (DIP). Pin labels for the Arduino IDE are indicated in blue (~ indicates PWM-capable). (Atmel Corporation 2016)

### ATtiny85 Pin-out Diagram

Figure A-7. Pin-out diagram for the ATtiny85 8-pin (DIP version). Pin labels for the Arduino IDE are indicated in blue (~ indicates PWM-capable). (Atmel Corporation 2013)

### Ohm’s Law Equation Table

Table A-8. Ohm’s Law equations, re-arranged for each term.

|            |                   |                          |                     |
|------------|-------------------|--------------------------|---------------------|
| Voltage    | $V = IR$          | $V = \sqrt{PR}$          | $V = \frac{P}{I}$   |
| Resistance | $R = \frac{V}{I}$ | $R = \frac{P}{I^2}$      | $R = \frac{V^2}{P}$ |
| Current    | $I = \frac{V}{R}$ | $I = \sqrt{\frac{P}{R}}$ | $I = \frac{P}{V}$   |
| Power      | $P = VI$          | $P = I^2R$               | $P = \frac{V^2}{R}$ |

## List of Circuit Diagram Symbols

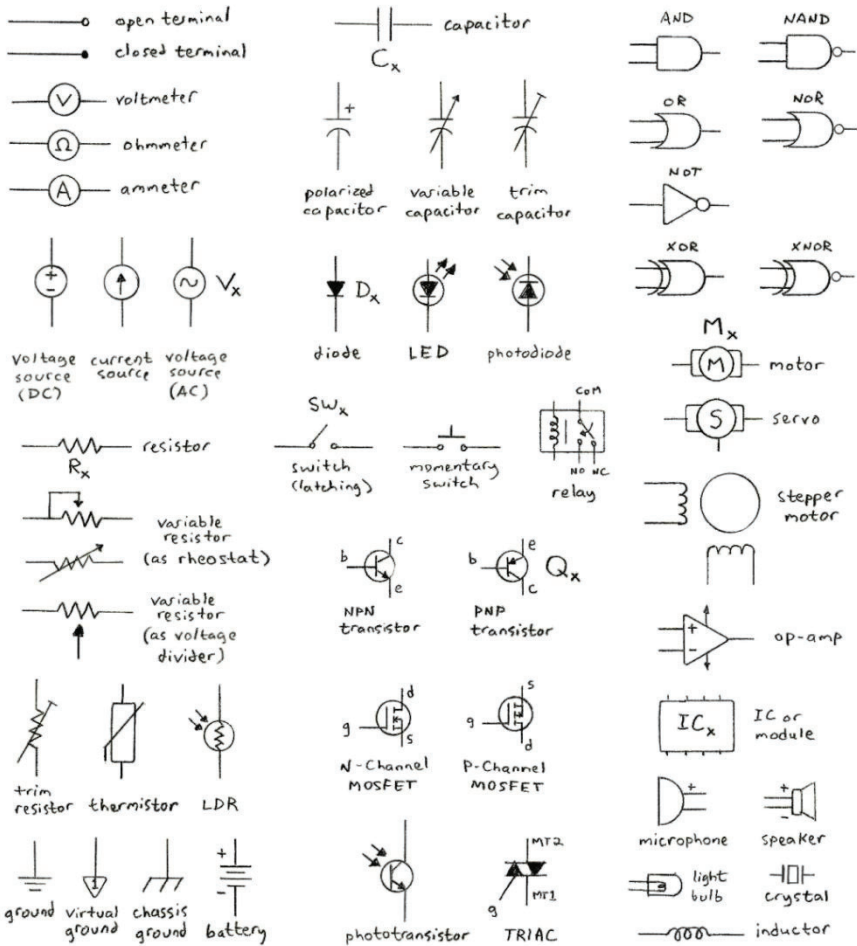


Figure A-8. List of circuit diagram symbols used in this text. Note that many variants of these symbols are common. LDRs, LEDs, photodiodes, transistors, and MOSFETs often appear in circuit diagrams without circles.





## List of Abbreviations

|           |                                                     |
|-----------|-----------------------------------------------------|
| %RSD      | percent relative standard deviation                 |
| AC        | alternating current                                 |
| ADC       | analog to digital converter                         |
| AREF      | analog reference pin                                |
| ASCII     | American standard code for information interchange  |
| AWG       | American wire gauge                                 |
| BJT       | bipolar junction transistor                         |
| bps       | bits per second                                     |
| CLK       | clock                                               |
| CR filter | high-pass capacitor-resistor filter                 |
| CSV       | comma separated values                              |
| CMOS      | complementary metal-oxide semiconductor             |
| dB        | decibel                                             |
| DC        | direct current                                      |
| DIP       | dual inline package                                 |
| divs      | divisions                                           |
| DPDT      | double pole double throw                            |
| EEPROM    | electrically erasable programmable read-only memory |
| e.g.      | exempli gratia                                      |
| FTDI      | Future Technology Devices International Ltd.        |
| GND       | ground                                              |
| HPF       | high-pass filter                                    |
| HV        | high voltage                                        |
| IC        | integrated circuit                                  |
| ICSP      | in-circuit serial programming                       |
| IDE       | integrated development environment                  |
| i.e.      | id est                                              |
| IR        | infrared                                            |
| ISR       | interrupt service routine                           |
| KCL       | Kirchhoff's current law                             |
| KVL       | Kirchhoff's voltage law                             |
| LED       | light emitting diode                                |
| LPF       | low-pass filter                                     |
| LSB       | least significant bit (to the far right)            |
| LV        | low voltage                                         |
| mAh       | milliamp hours                                      |
| MCU       | microcontroller unit                                |
| MISO      | master in slave out                                 |
| MOSFET    | metal oxide semiconductor field effect transistor   |

|                     |                                             |
|---------------------|---------------------------------------------|
| MOSI                | master out slave in                         |
| MSB                 | most significant bit (to the far right)     |
| NC                  | normally closed                             |
| NCHO                | normally closed held open                   |
| NiMH                | nickel metal hydride                        |
| NO                  | normally open                               |
| NOHC                | normally open held closed                   |
| NTC                 | negative temperature coefficient            |
| op-amp              | operational amplifier                       |
| PDIP                | plastic dual inline package                 |
| PID controller      | proportional integral derivative controller |
| PLCC                | plastic lead-chip carrier                   |
| PRV                 | peak reverse voltage                        |
| PTC                 | positive temperature coefficient            |
| PWM                 | pulse width modulation                      |
| Q-factor            | quality factor                              |
| RC filter           | low-pass resistor-capacitor filter          |
| RTC                 | real time clock                             |
| RTD                 | resistive temperature detector              |
| SCK                 | clock pin                                   |
| SNR                 | signal-to-noise ratio                       |
| SOIC                | small outline integrated circuit            |
| SPDT                | single pole double throw                    |
| SPST                | single pole single throw                    |
| SRAM                | static random-access memory                 |
| SW                  | switch                                      |
| TRIAC               | triode for attenuating current              |
| USB                 | universal serial bus                        |
| USP                 | United States Pharmacopeia                  |
| UTF-8               | unicode transformation format – 8 bit       |
| V <sub>GS(th)</sub> | threshold gate-source voltage (MOSFET)      |
| V <sub>cc</sub>     | voltage (common cathode)                    |

## BIBLIOGRAPHY

- Adriaensen, Jan. "Change Unipolar 28BYJ-48 to Bipolar Stepper Motor.", last modified October 27, 2013, accessed April 17, 2019, <http://www.jangeox.be/2013/10/change-unipolar-28byj-48-to-bipolar.html>.
- Alves, João. "Multiple Inputs - Parallel to Series.", last modified September 8, 2015, accessed March 1, 2019, <https://jpralves.net/post/2015/09/08/multiple-inputs-parallel-to-series.html#.XHm8IIhKiM8>.
- Analog Devices Inc. 2016. "AD623 Single and Dual-Supply, Rail-to-Rail, Low Cost Instrumentation Amplifier." *AD623 Datasheet Rev. F*. (January 11). <https://www.analog.com/media/en/technical-documentation/data-sheets/ad623.pdf>.
- Analog Devices Inc. "MT-033 Tutorial: Voltage Feedback Op Amp Gain and Bandwidth. Rev.0, 10/08, WK.", 2009, accessed February 21, 2019, <https://www.analog.com/media/en/training-seminars/tutorials/MT-033.pdf>.
- Andy Collinson. "Measuring Input and Output Impedance.", last modified March 20, 2018, accessed May 17, 2018, <http://www.zen22142.zen.co.uk/Theory/inzoz.htm>.
- Arduino.cc. "Arduino - Compare.", 2018, accessed May 26, 2018, <https://www.arduino.cc/en/Products/Compare>.
- Atmel Corporation. 2016. "8-Bit AVR Microcontrollers ATmega328/P Datasheet Complete." *Atmel-42735B-ATmega328/P\_Datasheet\_Complete-11/2016* (November). [https://cdn.sparkfun.com/assets/c/a/8/e/4/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P\\_Datasheet.pdf](https://cdn.sparkfun.com/assets/c/a/8/e/4/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf).
- Atmel Corporation. 2013. "Atmel 8-Bit AVR Microcontroller with 2/4/8K Bytes in-System Programmable Flash." Rev. 2586Q–AVR–08/2013 (ATTiny25/45/85 [DATASHEET]) (August). [https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-2586-AVR-8-bit-Microcontroller-ATTiny25-ATTiny45-ATTiny85\\_Datasheet.pdf](https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-2586-AVR-8-bit-Microcontroller-ATTiny25-ATTiny45-ATTiny85_Datasheet.pdf).
- Balachandran, Rama and Porter-Davis, Karen. "Using CDs and DVDs as Diffraction Gratings.", 2009, accessed January 6, 2019,

- [http://www.nnin.org/sites/default/files/files/Karen\\_Rama\\_USING\\_CD\\_s\\_AND\\_DVDs\\_AS\\_DIFFRACTION\\_GRATINGS\\_0.pdf](http://www.nnin.org/sites/default/files/files/Karen_Rama_USING_CD_s_AND_DVDs_AS_DIFFRACTION_GRATINGS_0.pdf).
- Bard, Allen J. and Larry R. Faulkner. 2001. *Electrochemical Methods*. 2. ed. ed. New York [u.a.]: Wiley.
- Bellman, Richard. 1964. "Control Theory." *Scientific American* 211 (3) (September): 186-200.
- Besenhard, Jürgen O. 1999. *Handbook of Battery Materials*. Weinheim ; New York: Wiley-VCH.  
<http://www.loc.gov/catdir/description/wiley032/99186348.html>;  
<http://www.loc.gov/catdir/toc/wiley022/99186348.html>.
- Boellmann, Werner e. a. "AVR Libc Home Page.", last modified February 9, 2016, accessed April 28, 2019, <http://www.nongnu.org/avr-libc/>.
- Burkett, Jaret. "Breadboard Arduino.", last modified February 6, 2016, accessed April 25, 2019,  
<https://github.com/oshlab/Breadboard-Arduino>.
- Camenzind, Hans R. 1997. "Redesigning the Old 555 [Timer Circuit]." *IEEE Spectrum* 34 (9): 80-85.
- Carolyn Mathas. "Light Sensors: An Overview.", last modified September 11, 2012, accessed December 18, 2018,  
<https://www.digikey.ca/en/articles/techzone/2012/sep/light-sensors-an-overview>.
- Carter, Bruce. 2006. "High-Speed Notch Filters." *Analog Applications Journal*: 19-26. <http://www.ti.com/lit/an/slyt235/slyt235.pdf>.
- Carter, Bruce and Thomas R. Brown. 2016. "Handbook of Operational Amplifier Applications." *Texas Instruments Inc. Application Report SBOA092B*. October 2001 [Revised September 2016] (Sep): 1-94.  
<http://www.ti.com/lit/an/sboa092b/sboa092b.pdf>.
- Chen, Chiachung. 2009. "Evaluation of Resistance-Temperature Calibration Equations for NTC Thermistors." *Measurement* 42 (7): 1103-1111. doi:10.1016/j.measurement.2009.04.004.  
[http://resolver.scholarsportal.info.myaccess.library.utoronto.ca/resolve/02632241/v42i0007/1103\\_eorcefnt](http://resolver.scholarsportal.info.myaccess.library.utoronto.ca/resolve/02632241/v42i0007/1103_eorcefnt).
- Coleman, T. "Shining the Light on Dimming.", last modified -01-22T11:50:00-05:00, 2015, accessed Mar 8, 2018,  
<http://www.ecmweb.com/lighting-control/shining-light-dimming>.
- Couto, Robson. "Arduino Reference - pulseIn().", last modified February 5, 2019, accessed February 22, 2019,  
<https://www.arduino.cc/reference/en/language/functions/advanced-io/pulsein/>.

- de Brabander, Frank. "Library for the LiquidCrystal LCD Display Connected to an Arduino Board.", last modified March 8, 2017, accessed March 5, 2018, <https://github.com/fdebrabander/Arduino-LiquidCrystal-I2C-library>.
- Dubins, David. "SRAMsimple: Library to Run 23LC1024 Chip for the Arduino Uno, using the Arduino IDE.", last modified November 14, 2018, accessed April 28, 2019, <https://github.com/dndubins/SRAMsimple>.
- Dummer, G. W. A. 2013. *Electronic Inventions and Discoveries: Electronics from its Earliest Beginnings to the Present Day*. Pergamon International Library. Kent: Elsevier Science & Technology.
- ELECFREAKS wiki. "Relay Module (Arduino Compatible).", last modified May 19, 2015, accessed Feb 22, 2018, [http://www.electfreaks.com/wiki/index.php?title=Relay\\_Module\\_\(Arduino\\_Compatible\)](http://www.electfreaks.com/wiki/index.php?title=Relay_Module_(Arduino_Compatible)).
- Everlight Electronics Co. Ltd. 2016. "5mm Phototransistor PT334-6C." *PT334-6C Datasheet DPT-0000185 Rev.4*. (December 19). <http://www.everlight.com/file/productfile/pt334-6c.pdf>.
- Fairchild Semiconductor Inc. 2014. "MOC3010M, MOC3011M, MOC3012M, MOC3020M, MOC3021M, MOC3022M, MOC3023M 6-PIN DIP Random-Phase Optoisolators Triac Driver Output (250/400 Volt Peak)." *MOC301XM, MOC302XM Rev. 1.0.3* (March). <http://www.farnell.com/datasheets/1806097.pdf>.
- Farzan, Azadeh. "Chapter 4: Set Theory (Course Notes, PMU199).", last modified December 24, 2018, <http://www.cs.toronto.edu/~azadeh/page11/page12/material/set-theory.pdf>.
- Fisher, Tim. "ATX 24 Pin 12V Power Supply Pinout.", last modified January 7, 2019, accessed January 10, 2019, <https://www.lifewire.com/atx-24-pin-12v-power-supply-pinout-2624578>.
- Gammon, Nick. "Gammon Forum: Interrupts.", last modified January 8, 2012, accessed April 14, 2019, <http://gammon.com.au/interrupts>.
- Gibilisco, Stan. 2013. *Beginner's Guide to Reading Schematics, 3E* McGraw-Hill/TAB Electronics. <http://lib.myilibrary.com?ID=525390>.
- gratefulfrog. "Arduino Playground - Keypad Tutorial.", last modified September 4, 2013, accessed February 28, 2019, <https://playground.arduino.cc/Main/KeypadTutorial>.
- Haffner, Sr D. "Using an Arduino R3 to Power the TCD1304AP CCD Chip.", last modified August 2, 2017, accessed December 18, 2018,

- <https://hackaday.io/project/18126-dav5-v301-raman-spectrometer/log/53099-using-an-arduino-r3-to-power-the-tcd1304ap-ccd-chip>.
- Hellma. 2008. *Calibration Standards for Spectrophotometers*. Müllheim, Germany: Hellma GmbH & Co. KG.
- Hobby CNC Australia. "Mounting Stepper Motors.", last modified February 15, 2015, accessed December 9, 2018, <http://www.hobbyncnaustralia.com.au/Instructions/iI8mountstepper.htm>.
- Hoffmann, Heiko. "A.3 Iterative Mean.", last modified March 22, 2005, accessed January 23, 2019, <http://www.heikohoffmann.de/htmlthesis/node134.html>.
- Intersil Corporation. 2013. "CL7660S, ICL7660A Super Voltage Converters." *ICL7660S, ICL7660A Datasheet FN3179 Rev 7.00*. (January 23). <https://www.renesas.com/us/en/www/doc/datasheet/icl7660.pdf>.
- Kanakaraja, Pamarthi. "RGB Color Detector using TCS3200 Sensor Module.", last modified April 11, 2017, accessed February 22, 2019, <https://electronicsforu.com/electronics-projects/rgb-color-detector-tcs3200-sensor-module>.
- Karki, Jim. 2002. "Active Low-Pass Filter Design." *Texas Instruments Inc. Application Report SLOA049B*. (Sep): 24. <http://www.ti.com/lit/an/sloa049b/sloa049b.pdf>.
- King, Terry. "Arduino Playground - ArduinoPinCurrentLimitations.", last modified July 15, 2017, accessed Feb 21, 2018, <https://playground.arduino.cc/Main/ArduinoPinCurrentLimitations>.
- Leger, George. "Unipolar Stepper Motor Vs Bipolar Stepper Motors.", last modified January 11, 2012, accessed December 18, 2018, <https://www.circuitspecialists.com/blog/unipolar-stepper-motor-vs-bipolar-stepper-motors/>.
- Loflin, Lewis. "Zero-Crossing Detectors Circuits and Applications.", last modified January, 2018, accessed April 14, 2019, [http://www.bristolwatch.com/ele2/zero\\_crossing.htm](http://www.bristolwatch.com/ele2/zero_crossing.htm).
- Macsimski, Simon. "Arduino Playground - ArduinoSleepCode.", last modified December 29, 2006, accessed March 16, 2019, <http://playground.arduino.cc/Learning/ArduinoSleepCode>.
- Maloberti, Franco and Anthony Davies. 2016. *A Short History of Circuits and Systems*. Aalborg: River Publishers. [https://ebookcentral.proquest.com/lib/\[SITE\\_ID\]/detail.action?docID=4530489](https://ebookcentral.proquest.com/lib/[SITE_ID]/detail.action?docID=4530489).
- Mancini, Ronald C. 2002. "Op Amps for Everyone: Design Reference." *Advanced Analog Products SLOD006B*. (Aug).

- [http://web.mit.edu/6.101/www/reference/op\\_amps\\_everyone.pdf](http://web.mit.edu/6.101/www/reference/op_amps_everyone.pdf).
- Mansfield, Anson. "TriacDimmer: A Library for Controlling a Triac Dimmer.", last modified February 24, 2017, accessed April 16, 2019, <https://www.arduinolibraries.info/libraries/triac-dimmer>.
- Margolis, Michael. "Time and TimeAlarms Libraries for Arduino.", last modified October 1, 2014, accessed April 25, 2019, [https://github.com/michaelmargolis/arduino\\_time](https://github.com/michaelmargolis/arduino_time).
- Maw, Carlyn and Igoe, Tom. "Serial to Parallel Shifting-Out with a 74HC595.", last modified November, 2006, accessed March 2, 2019, <https://www.arduino.cc/en/Tutorial/ShiftOut>.
- Maxim Integrated. "Application Note 1880. Charlieplexing - Reduced Pin-Count LED Display Multiplexing.", last modified February 10, 2003, accessed March 3, 2019, <https://www.maximintegrated.com/en/app-notes/index.mvp/id/1880>.
- Mellai, Stefania. "Arduino - EEPROM.", last modified May 17, 2018, accessed May 26, 2018, <https://www.arduino.cc/en/Reference/EEPROM>.
- . "Arduino - Introduction.", last modified Oct 20, 2017a, accessed March 5, 2018, <https://www.arduino.cc/en/Guide/Introduction>.
- . "Arduino Reference - analogWrite().", last modified November 15, 2017b, accessed May 26, 2018, <https://www.arduino.cc/reference/en/language/functions/analog-io/analogwrite/>.
- . "Arduino Reference: analogReference().", last modified Last Update: Nov. 24, 2017c, accessed Feb 20, 2018, <https://www.arduino.cc/reference/en/language/functions/analog-io/analogreference/>.
- Microchip Technology Inc. 2015. "23A1024/23LC1024: 1Mbit SPI Serial SRAM with SDI and SQI Interface." *23A1024/23LC1024 Datasheet Rev. C DS20005142C* (January). <http://ww1.microchip.com/downloads/en/DeviceDoc/20005142C.pdf>.
- . 2008. "25LC1024: 1 Mbit SPI Bus Serial EEPROM." *25LC1024 Datasheet Rev. B DS22064B* (May). <http://ww1.microchip.com/downloads/en/DeviceDoc/22064B.pdf>.
- Nagel, Sandra, Grant, Lyle, Mintzler, Janice, Mah, Dean, Bedeck, Bob, Hayduck, Penny and Gilbert, Trevor. "Tutorial 25: The Human Ear.", last modified October, 2016, accessed April 22, 2019, <https://psych.athabascau.ca/html/Psych402/Biotutorials/25/part1.html>.
- Navarro, Jamie. "Standalone Arduino / ATmega Chip on Breadboard.", last modified February 14, 2010,

- <https://www.instructables.com/id/Standalone-Arduino-ATMega-chip-on-breadboard/>.
- Nexperia. 2015a. "74HC04; 74HCT04 Hex Inverter." *74HC\_HCT04 V.5 Datasheet*. November 27, 2015 - Rev. 5. (November).  
[https://assets.nexperia.com/documents/data-sheet/74HC\\_HCT04.pdf](https://assets.nexperia.com/documents/data-sheet/74HC_HCT04.pdf).
- . 2015b. "74HC32; 74HCT32 Quad 2-Input OR Gate, Rev. 5." *74HC\_HCT32 V.6 Datasheet*. December 3, 2015 - Rev. 6..  
[https://assets.nexperia.com/documents/data-sheet/74HC\\_HCT32.pdf](https://assets.nexperia.com/documents/data-sheet/74HC_HCT32.pdf).
- NXP Semiconductors. 2013. "BT139-600E 4Q Triac." *BT139-600E Datasheet* September 27, 2013 (September 27,).  
[http://biakom.com/pdf/BT139-600E\\_NXP.pdf](http://biakom.com/pdf/BT139-600E_NXP.pdf).
- Omega Engineering. "pH Measurement Electrode Basics.", last modified August 28, 2018, accessed February 21, 2019,  
[https://www.omega.com/Green/pdf/pHbasics\\_REF.pdf](https://www.omega.com/Green/pdf/pHbasics_REF.pdf).
- ON Semiconductor Corp. 2018. "1N4001, 1N4002, 1N4003, 1N4004, 1N4005, 1N4006, 1N4007, Axial Lead Standard Recovery Rectifiers." *1N4001/D Datasheet*. June, 2018 – Rev. 14 (June).  
<http://www.onsemi.com/pub/Collateral/1N4001-D.PDF>.
- . 2011. "2N7000G Small Signal MOSFET 200 mAmps, 60 Volts N-Channel TO-92." *2N7000G Datasheet 2N7000/D* April 2011, Rev. 8. (April).  
<https://www.onsemi.com/pub/Collateral/2N7000-D.PDF>.
- . 2013. "P2N2222A Amplifier Transistors NPN Silicon." January, 2013 – Rev. 7. <https://www.onsemi.com/pub/Collateral/P2N2222A-D.PDF>.
- . 2014. "TIP120, TIP121, TIP122 (NPN); TIP125, TIP126, TIP127 (PNP) - Plastic Medium-Power Complementary Silicon Transistors." *TIP120/D Datasheet* November, 2014 – Rev. 9.  
<https://www.onsemi.com/pub/Collateral/TIP120-D.PDF>.
- Op de Coul, Misja. "Epoch Converter - Unix Timestamp Converter.", last modified January 20, 2019, accessed January 23, 2019,  
<https://www.epochconverter.com/>.
- Paul Stoffregen. "Teensy USB Development Board.", last modified February 8, 2019, accessed February 23, 2019,  
<https://www.pjrc.com/teensy/>.
- PeterEmbedded. "BH1750 / BH1750FVI Digital Light Sensor.", last modified November 2, 2018, accessed April 25, 2019,  
<https://github.com/PeterEmbedded/BH1750FVI>.
- PlaneTa MarTes. "ASCII Table, ASCII Codes: American Standard Code for Information Interchange. the Complete Table of ASCII Characters, Letters, Codes, Symbols and Signs.", last modified November 22, 2008, accessed April 28, 2019, <https://theasciicode.com.ar/>.



- Pololu Corporation. "Pololu - 9. Dealing with Motor Noise.", 2015, accessed Mar 8, 2018, <https://www.pololu.com/docs/0J15/9>.
- Poole, Ian. "Op Amp Input Impedance.", 2009, accessed December 15, 2018, [https://www.radioelectronics.com/info/circuits/opamp\\_basics/operational-amplifier-input-impedance.php](https://www.radioelectronics.com/info/circuits/opamp_basics/operational-amplifier-input-impedance.php).
- Portaluri, Bruno. "Fast Sampling from Analog Input - Yet another Arduino Blog.", last modified February 1, 2015, accessed March 12, 2019, <http://yaab-arduino.blogspot.com/2015/02/fast-sampling-from-analog-input.html>.
- Rabault, Jean. "Github - jerabaul29/ArduinoUseWatchdog: Examples of how to use the Arduino UNO Watchdog.", last modified January 9, 2016, accessed March 9, 2019, <https://github.com/jerabaul29/ArduinoUseWatchdog>.
- Roberts, S. "PID Control: A Brief Introduction and Guide, using Arduino.", last modified September 26, 2011, accessed March 23, 2018, [http://www.academia.edu/11086701/PID\\_Control\\_A\\_brief\\_introduction\\_and\\_guide\\_using\\_Arduino](http://www.academia.edu/11086701/PID_Control_A_brief_introduction_and_guide_using_Arduino).
- RobotFreak. "Arduino 101: Timers and Interrupts.", last modified August, 2011, accessed March 21, 2019, <https://www.robotshop.com/community/forum/t/arduino-101-timers-and-interrupts/13072>.
- Ross, Kevin. "The Basics - Bypass Capacitors.", last modified June, 1997, accessed May 17, 2018, <http://www.seattlerobotics.org/encoder/jun97/basics.html>.
- Save, Alok. "C++ - when to use Const Char \* and when to use Const Char[] - Stack Overflow.", last modified October 26, 2011, accessed January 22, 2019, <https://stackoverflow.com/questions/7903551/when-to-use-const-char-and-when-to-use-const-char>.
- Scherz, Paul and Simon Monk. 2016. *Practical Electronics for Inventors*. Fourth Edition ed. New York: McGraw-Hill Education. <http://www.loc.gov/catdir/enhancements/fy1605/2016932853-b.html>; <http://www.loc.gov/catdir/enhancements/fy1605/2016932853-d.html>.
- Schwager, Mike. "GitHub - GreyGnome/EnableInterrupt: New Arduino Interrupt Library, Designed for Arduino Uno/Mega 2560/Leonardo/Due.", last modified June 22, 2018, accessed March 9, 2019, <https://github.com/GreyGnome/EnableInterrupt>.
- Smith, J. "How to use the Conditional (Ternary) Operator.", last modified September 30, 2009, accessed January 25, 2019, <http://www.cplusplus.com/forum/articles/14631/>.

- Steve Hobley. "Light Theremin.", last modified December 18, 2012, accessed March 24, 2019, <https://makezine.com/projects/light-theremin/>.
- Stoffregen, Paul. "FreqMeasure Library.", last modified March 26, 2015, accessed April 25, 2019, <https://github.com/PaulStoffregen/FreqMeasure>.
- Stör, Marcel, Watson, Jim, Grokhotkov, Ivan, Diogosalazar and Christian, Alexander. "Libraries - ESP8266 Arduino Core 2.4.0 Documentation.", 2017, accessed May 26, 2018, <http://arduino-esp8266.readthedocs.io/en/latest/libraries.html>.
- Taranovich, Steve. "RTDs, PTCs, and NTCs: How to Effectively Decipher this Alphabet Soup of Temperature Sensors.", last modified September 14, 2011, accessed February 20, 2018, <https://www.digikey.ca/en/articles/techzone/2011/sep/rtds-ptcs-and-ntcs-how-to-effectively-decipher-this-alphabet-soup-of-temperature-sensors>.
- Texas Instruments Inc. 2013. "Application Report: AN-20 an Applications Guide for Op Amps." *Texas Instruments Inc.* (May): 1-26. <http://www.ti.com/lit/an/snoa621c/snoa621c.pdf>.
- . 2016a. "LM317 3-Terminal Adjustable Regulator." *LM317 Datasheet SLVS044X*. September 1997 [Revised September 2016] (September). <http://www.ti.com/lit/ds/symlink/lm317.pdf>.
- . 2016b. "LMC555 CMOS Timer." *LMC555 Datasheet SNAS558M*. February 2000 [Revised July 2016] (Jan). <http://www.ti.com/lit/ds/symlink/lmc555.pdf>.
- . 2015a. "LMx24-N, LM2902-N Low-Power, Quad-Operational Amplifiers." *LM124-N, LM224-N LM2902-N, LM324-N Datasheet SNOSC16D*. March 2000 [Revised January 2015]. <http://www.ti.com/lit/ds/symlink/lm124-n.pdf>.
- . 2014a. "LMx58-N Low-Power, Dual-Operational Amplifiers." *LM158-N, LM258-N, LM2904-N, LM358-N Datasheet SNOSBT3I*. January 2000 [Revised December 2014]. <http://www.ti.com/lit/ds/symlink/lm158-n.pdf>.
- . 2016c. "OPA541 High Power Monolithic Operational Amplifier." *OPA541 Datasheet SBOS153B*. September 2000 [Revised January 2016]. <http://www.ti.com/lit/ds/symlink/opa541.pdf>.
- . 2016d. "SNx4HC08 Quadruple 2-Input Positive-AND Gates." *SN54HC08, SN74HC08 Datasheet SCLS081G*. December 1982 [Revised June 2016]. <http://www.ti.com/lit/ds/symlink/sn74hc08.pdf>.

- . 2015b. "SNx4HC165 8-Bit Parallel-Load Shift Registers." December 1982 (Revised - December 2015) (SN54HC165, SN74HC165 datasheet SCLS116H).  
<http://www.ti.com/lit/ds/symlink/sn74hc165.pdf>.
- . 2017. "TL07xx Low-Noise JFET-Input Operational Amplifiers." *TL071, TL071A, TL071B TL072, TL072A, TL072B, TL074, TL074A, TL074B, TL072M, TL074M Datasheet SLOS080N*. September 1978 [Revised July 2017].  
<http://www.ti.com/lit/ds/symlink/tl072b.pdf>.
- . 2014b. "xx555 Precision Timers." *NA555, NE555, SA555, SE555 Datasheet SLFS022I*. September 1973 [Revised September 2014].  
<http://www.ti.com/lit/ds/symlink/ne555.pdf>.
- . 2015. "SNx4HC595 8-Bit Shift Registers with 3-State Output Registers." *SN54HC595, SN74HC595 Datasheet SCLS041I* December 1982 (Revised September 2015).  
<http://www.ti.com/lit/ds/symlink/sn74hc595.pdf>.
- Thal, Melissa and Michael Samide. 2001. "Applied Electronics: Construction of a Simple Spectrophotometer." *Journal of Chemical Education* 78 (11) (November): 1510-12.
- Tillaart, Rob. "Arduino Playground - TSL235R Light to Frequency Sensor.", last modified May 16, 2011, accessed February 22, 2019,  
<https://playground.arduino.cc/Main/TSL235R>.
- Tim Wescott. 2000. "PID without a PhD." *Embedded Systems Programming*, Oct 1, 86.
- Toshiba Corporation. 2001. "Toshiba CCD Linear Image Sensor TCD1304AP." *TCD1304AP Datasheet*. (October 15).  
<https://oceanoptics.com/wp-content/uploads/Toshiba-TCD1304AP-CCD-array.pdf>.
- Tower Pro Datasheet. 2017. "SG90 9g Micro Servo." *SG90 Datasheet*. (April 12). <http://akizukidenshi.com/download/ds/towerpro/SG90.pdf>.
- Tsai, Jane and Ltd Everlight Electronics Co. 2005. "PD638C Photodiode." *PD638C Datasheet*. (July 20). <http://www.everlight.com>.
- Vishay Siliconix. 2004. "TP0610L/T, VP0610L/T, BS250 P-Channel 60-V (D-S) MOSFET." Revision: July 18, 2008 (Document Number: 70209) (July 5). <https://www.vishay.com/docs/70209/70209.pdf>.
- Wenzel, Charles. "Battery Capacity.", 2017, accessed January 22, 2018,  
<http://www.techlib.com/reference/batteries.html>.
- Willistein, Jonathan. "Update to USBmicroISP: 6pin and 10pin Headers.", last modified September 13, 2015, accessed March 10, 2019,  
<http://jwillylinux.blogspot.com/2015/09/update-to-usbmicroisp-6pin-and-10pin.html>.

# INDEX

- #define*, 87, 145
- #ifdef*, 147
- #include*, 87
- 10% Rule*, 28, 51
- 3.3V logic*, 305
- AC coupling*, 283
- ADC*, 2, 238, 294, 326, 327
- alternating current*, 9, 11, 177
- ammeter*, 14
- analogRead()*, 382
- AND gate*, 70
- anode*, 7, 8, 35, 163, 164
- Arduino IDE*, 84, 85, 86, 87, 88, 90, 100, 120, 137, 147, 182, 184, 187, 190, 205, 224, 225, 226, 228, 230, 301, 304, 309, 323, 330, 334, 425
- AREF*, 134, 135, 151, 152, 254, 266, 269, 294, 408
- array*, 107, 108, 111, 114, 120, 142, 145, 308, 352, 395, 396, 397, 400, 401, 403, 415, 417
- ATtiny85*, 68, 419
- band reject filter*, 286
- band-pass filter*, 284
- bandwidth*, 284
- battery*, 7
- bias*, 246
- bias voltage*, 246
- binary*, 91
- Bipolar Junction Transistors*, 165
- bipolar stepper motor*, 197
- bit depth*, 92
- bit masking*, 354
- bit shifting*, 357
- bitwise AND*, 353
- bitwise NOT*, 355
- bitwise operations*, 353
- bitwise OR*, 354
- bitwise XOR*, 356
- Bode Magnitude Plot*, 277, 281
- boolean operators*, 102
- boolean variables*, 101
- boost converter*, 53
- breadboard*, 37, 67
  - ravine*, 37
  - terminal strips*, 37
- break*, 115, 143
- buck converter*, 53
- byte variable*, 103
- C++ shorthand*, 112
- call-by-reference*, 141
- call-by-value*, 141
- capacitance*, 44
- capacitor*, 44, 45, 46, 47, 48, 49, 50, 78, 82, 182, 275, 279, 293, 307, 406, 422
  - bypass capacitor*, 293
  - coupling capacitor*, 283
  - decoupling capacitor*, 161, 293
- casting*, 105, 106, 107, 111, 439
- cathode*, 7, 8, 34, 35, 163, 164
- centre frequency*, 284
- char*, 110
- char variable*, 103
- char[]*, 415
- charge*, 6
- charge coupling*, 283
- charge emitter*, 35
- charlieplexing*, 341
- chip programmer*, 349, 350
- closed loop control*, 206
- common return*, 10
- conditional operator*, 100
- constrain()*, 186
- control algorithm*, 207, 208
- conventional current*, 7
- current*, 7
  - load current*, 51
- current capacity*, 15, 22
- current divider equation*, 30, 31, 32
- current gated*, 173

- current regulator*, 55
- current source*, 55
- cutoff frequency*, 275, 276, 280, 281
  - decade*, 277
  - octave*, 277
- daisy chaining*, 173
- data logging*, 300
- data smoothing*, 295
- datasheet*, 53
- delay()*, 302
- delayMicroseconds()*, 303
- digital pin*, 124
  - INPUT mode*, 124
  - OUTPUT mode*, 124
- diode*, 3, 13, 33, 55, 162, 163, 164, 166, 180, 181, 189
  - brick wall*, 164
  - forward biased*, 163
  - Peak Reverse Voltage*, 164
- DIP*, 67
- direct current*, 9
- divs*, 131
- do...while*, 112, 114, 115
- drift velocity*, 8
- DRIVE*, 208
- dropout voltage*, 54
- duty cycle*, 126
- earth return*, 21
- EEPROM memory*, 320
- else if*, 116
- engineering control theory*, 206
- epoch time*, 300
- feed forward*, 206, 209, 223
- feedback*, 206, 211, 223
  - critically damped*, 217
  - integral threshold*, 218
  - over-damped response*, 215
  - undamped*, 216
  - undamped response*, 214
  - under-damped*, 216
  - winding up*, 218
- feed-forward gain constant*, 210
- flash memory*, 414
- Fleming's Left Hand Rule*, 192
- float*, 98
- float function*, 152
- flyback diode*, 181
- for loop*, 111, 113
- frequency range of human hearing*, 278, 283
- function*
  - input argument*, 107
  - loop() function*, 87, 96, 114, 121, 138, 144
  - setup() function*, 96, 114, 121, 125, 134, 137, 138, 144, 152
- gain*, 170, 237, 256, 267
  - gain in dB*, 239
- graded control*, 213
- ground*, 9, 10, 17, 18, 19, 20, 21, 39, 69, 71, 80, 122, 124, 129, 156, 158, 159, 160, 161, 165, 176, 177, 181, 184, 185, 189, 196, 231, 235, 241, 243, 244, 250, 251, 253, 260, 264, 276, 292, 293
  - chassis ground*, 19
  - chassis return*, 20
  - earth ground*, 18
  - floating return*, 20
- ground bus*, 21
- ground looping*, 21
- H-bridge*, 193
- headroom*, 54, 241
- heat sink*, 53, 56, 176, 178
- high side switching*, 160
- if...then...else*, 98, 116
- impedance*, 238
  - high impedance signal*, 238
  - low impedance signal*, 238
- impedance matching*, 288
- in parallel*, 22, 29
- in series*, 22
- inductive load*, 180
- input arguments*, 141
- input impedance*, 283
- integer*, 93
- Interrupt Service Routine*, 325, 331, 369
- interrups*, 369
  - cli()*, 373
  - external interrupts*, 336, 369

- pin change interrupts*, 370
- sei()*, 373
- inverter*, 73
- isnan()*, 299
- iterative mean*, 296
- LCD module*, 85
- lcd.print()*, 89
- LED matrix display*, 338
- LEDs*, 33
- LM35*, 132, 148, 230, 231, 234, 427
- load*, 8, 59
- load side*, 159
- logic level*, 69, 71
- logic shifter*, 305
- logic side*, 159
- logic table*, 70, 71, 73, 74, 75, 76, 77, 102
- long integer*, 95
- low side switching*, 160
- low-pass filter*, 275
- mains electricity*, 178, 180
- map()*, 134
- matrix keypad*, 338, 339, 340
- maximum power rating*, 33
- maximum power theorem*, 289
- mean filter*, 295
- median filter*, 296
- Mesh Current Method*, 61
- micros()*, 303
- millis()*, 302
- mode filter*, 297
- MOSFET*, 174
- motor driver*, 194
- multimeter*, 14, 39, 41, 42, 43, 58, 61, 148
- NaN*, 299
- negative voltage*, 242, 244, 250, 251, 256, 260, 261, 262, 264, 267, 276, 282
- negative voltage generator*, 261
- NOR gate*, 74
- Norton Equivalent Circuit*, 61
- NOT gate*, 73
- notch filter*, 286
- N-type silicon*, 163
- null character*, 108, 415
- Ohm's Law*, 12
- ohmic device*, 12
- ohmmeter*, 14
- on-off controller*, 212, 216, 217, 230, 236
- open-loop control*, 206
- operational amplifiers*, 235
  - buffer*, 237
  - comparator*, 236
  - compensating resistor*, 288
  - differential amplifier*, 250
  - gain-bandwidth product*, 243
  - headroom*, 239
  - ideal op-amp*, 236, 237, 238, 244, 287, 433
  - inverter*, 245
  - inverting AC amplifier*, 281
  - inverting amplifier*, 243
  - inverting input*, 236, 243
  - maximum peak output voltage*, 240
  - non-inverting input*, 236, 248
  - output short-circuit current*, 239
  - slew rate*, 242
  - transimpedance amplifier*, 326, 327
  - unity gain bandwidth*, 243
  - voltage follower*, 237
- OR gate*, 72
- P-Controller*, 217
- persistence of vision*, 126, 340
- photoresistor*, 325
- pinout diagram*, 81
- PLCC*, 69
- P-N junction*, 163, 164
- pointers*, 109, 396
- port manipulation*, 352, 360, 361, 367
- potentiometer*, 56, 80, 132, 133, 138, 172, 187, 191, 225, 226, 228, 229, 264, 278
- proportional feedback controller*, 213
- proportional gain constant*, 213
- protection diode*, 181
- P-type silicon*, 163

- pull-down resistor*, 129
- pull-up resistor*, 129
- pulseIn()*, 331
- PWM*, 122, 126, 127, 128, 132, 134, 136, 138, 153, 155, 165, 173, 177, 178, 186, 194, 195, 196, 205, 229, 274, 360, 373, 420, 427, 429
  - fast PWM mode*, 374
- Q-factor*, 284
- relational operator*, 100
- relay*, 157, 159
  - normally closed*, 159
  - normally open*, 159
  - poles*, 157
  - throw count*, 157
- resistance*, 5, 10, 11, 12, 13, 14, 15, 21, 24, 25, 26, 27, 32, 33, 34, 39, 41, 42, 48, 50, 51, 57, 58, 60, 66, 80, 148, 149, 150, 151, 152, 167, 168, 202, 210, 238, 245, 263, 264, 279, 286, 287, 289, 290, 424, 425
- resistor*, 5, 13, 14, 24, 25, 26, 27, 29, 31, 32, 33, 34, 39, 40, 41, 42, 43, 48, 49, 50, 51, 52, 56, 57, 58, 61, 63, 70, 82, 123, 125, 129, 130, 133, 150, 151, 153, 155, 166, 167, 168, 169, 171, 172, 173, 178, 187, 254, 256, 257, 263, 264, 271, 276, 278, 279, 288, 289, 290, 291, 366, 369, 406, 407, 422, 423, 424, 433
  - fixed-value resistor*, 13
  - potentiometer*, 13, 56
- resonant frequency*, 284
- return*, 19, 143
- reverse biased*, 164
- ripple*, 55
- RTC module*, 301
- second-order passive HPF*, 286
- sense resistor*, 150, 151
- sensitivity*, 242
- sensor*, 209
- serial monitor*, 122, 137, 138, 144, 148, 152, 153, 155, 156, 186, 187, 225, 226, 228, 229, 231, 266, 269, 270, 301, 302, 304, 305, 309, 354, 392, 397, 408, 411, 424, 429
  - serial plotter*, 138
  - Serial.parseFloat()*, 187
  - Serial.parseInt()*, 187
  - Serial.print()*, 137, 391, 416
  - Serial.println()*, 137, 391
  - Serial.read()*, 187
  - Serial.readString()*, 187
  - Serial.write()*, 396
  - setpoint*, 207
  - settling time*, 216
  - setup()*, 114
  - shift-in register*, 345
  - shiftIn()*, 347
  - shift-out register*, 343
  - shiftOut()*, 345
  - short circuit*, 19, 21, 29, 36, 160, 178, 181, 194, 239
  - signal*, 243
  - signal attenuation*, 263
  - signal-to-noise ratio*, 294
  - sizeof()*, 397
  - sketch*, 87
  - sleep mode*, 386, 387
  - SOIC*, 69
  - solid-state relays*, 160
  - SPDT*, 157
  - split supply*, 10, 53
  - static variable*, 139
  - stepper motor*, 191, 194, 197, 198, 200, 201, 202, 203, 204, 205, 206, 225, 226, 227, 231, 325, 333
  - stop-band filter*, 285
  - strcat()*, 398
  - strcpy()*, 109, 398
  - String*, 103, 110, 415
  - Strings*
    - Array of Strings*, 395
  - strlen()*, 397, 398
  - switch*, 35
    - high side switch*, 36
    - high-side switch*, 161, 167

- low-side switch*, 160, 161, 167, 175
- momentary switch*, 5, 36, 40, 43, 128, 129, 130, 266, 369
- switch...case*, 116, 117
- thermistor*, 148
- thermocouple*, 148, 235, 259, 270, 298
- Thévenin Equivalent Circuit*, 60
- Thévenin's Theorem*, 44, 57, 58, 59, 61, 62, 64, 82, 290
- Timers*
  - CTC mode*, 382
  - output compare registers*, 374
  - Timer 0*, 373, 384, 385
  - Timer 1*, 373, 377, 382
  - Timer 2*, 373, 379, 383
  - Timer Counter/Control Registers*, 374
  - watchdog timer*, 386, 387, 389, 390
- toggle switch*, 36
- tolerance*, 33, 34
- tone()*, 379
- transistor*
  - 10% Current Rule*, 168
  - active mode*, 166
  - active region*, 170
  - biased*, 170
  - cutoff mode*, 166, 170
  - Darlington pair*, 172
  - hFE*, 169
  - quiescent current*, 171
  - saturation mode*, 166, 170
- TRIAC*, 177
- truth table*, 70
- two-term exponential thermistor equation*, 149
- union*, 402
- unipolar stepper motor*, 197
- unsigned long*, 95, 96, 121, 301, 302, 303, 304, 415, 430
- virtual ground*, 10, 260
- void function*, 139, 140, 141
- volatile*, 330, 369, 370, 371, 372, 382, 383, 384
- voltage*, 7, 9
- voltage divider equation*, 5, 26, 27, 29, 41, 50, 67, 151, 152, 264, 433
- voltage gated*, 173
- voltage regulator*, 53
- voltage source*, 7
- voltmeter*, 14
- Wheatstone bridge*, 263
- while loop*, 114, 115
- working range*, 186
- XOR gate*, 75