Butter for your Breadboard:
An RLC Network Identification System

by

Joshua Michael Gordonson

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2015

Author .....
Department of Electrical Engineering and Computer Science
September 08, 2015

Certified by .....
Gerald Jay Sussman
Panasonic Professor of Electrical Engineering
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Theses

Butter for your Breadboard:

An RLC Network Identification System

by

Joshua Michael Gordonson

Submitted to the Department of Electrical Engineering and Computer Science
on September 08, 2015, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Science and Engineering

Abstract

A system for identifying networks of circuit elements was designed and constructed in the pursuit of better educational electronic prototyping tools. This document contains an analysis of a circuit network identification system, with a focus on determining RLC networks using impedance sensing techniques. A design of both hardware and software implementations of the circuit network identification system are presented, and the results of the implementations are analyzed. Also presented is a method for surface-mounting breadboard finger springs to printed circuit boards.

Thesis Supervisor: Gerald Jay Sussman
Title: Panasonic Professor of Electrical Engineering

Acknowledgments

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 History of Breadboards

The breadboard has been a staple substrate for electronic construction over the last century. At the dawn of a growing interest in amateur radio, resourceful tinkerers used planks of wood to secure and ruggedize their electrical handiwork. Conductive nodes, such as nails or copper rails, were driven into the non-conductive boards, providing anchors and contact points that were electrically isolated from the rest of the circuit. Components were soldered or wire-wrapped to the nodes, and sometimes secured by non-energized nails or screws. This construction technique provided a lot of artistic freedom in circuit construction, but was time consuming and required relatively heavy hand tools such as a hammer or drill. From a performance perspective, sensitive high-frequency circuits were infeasible due to a lack of a ground plane and long wires between components.

1.2 Modern Breadboards

The solderless breadboard is the canonical tool given to students taking introductory courses in the field. Rather than driving nodes into arbitrary locations, component leads are inserted into contact points arranged on a grid that allow rapid semi-rigid construction of circuits with no other tools. The layout of a solderless breadboard is designed to be compatible with a plethora of powerful integrated circuits - enabling complex electronic designs - but is still limited to low-frequency operation due to the parasitic capacitance between adjacent breadboard rails. Modern solderless breadboards have come a long way from their namesake

wooden ancestors, but there is still room for improvement.

The intent of breadboarding is to physically realize a circuit. Often, this involves designing or using a reference schematic to guide construction, but circuit improvisation is not uncommon. A meticulous breadboarder can successfully realize a circuit without error by correctly placing components and jumper wires - taking care not to introduce undesired 'parasitic' components. However, for the uninitiated it is difficult to justify the additional time and care required to plan and build. Inserting components and jumper wires into contact points is straightforward, but poor contacts, broken wires (inside insulation), and mis-inserting leads can plague designers for hours on end and potentially destroy components. Breadboarding is a skill that is learned over time, but small errors can lead to excessive frustration and turn students off to the field. I propose a solution to some of the issues with the solderless breadboard.

A confident linkage between the the electrical and mechanical domains is required to construct a circuit. On larger scales, the mechanical structures (eg. contacts, wires, components) that create electrical circuits are visible in plain sight. It is simple and reliable, then, to determine the circuit representation of a mechanical structure that has no hidden connections. Breadboards often obscure connections due to their construction. The regular nature of a breadboard, a grid of contact points arranged in rails, makes it difficult to keep track of which rail is connected to what circuit node. When combined with the opaque nature of most components and the poor reliability of breadboard contacts, visually verifying complex circuits on a breadboard becomes infeasible. Many of the inherent problems with breadboards stem from this open-loop nature of breadboarding, where visual cues are not enough to determine the electrical circuit from the mechanical structure. A symptom of open-loop construction techniques is a mismatch between the mental model and the physical realization of the system at hand. I seek to close this loop by designing and implementing a circuit-sensing breadboard.

1.3 Proposed Solution

A circuit-sensing breadboard is able to reverse-engineer circuits built on it by applying test voltages and currents and measuring the results. As a proof-of-concept, the proposed circuit-sensing breadboard completed for this thesis is able to reverse engineer circuits composed

of resistors built on a small section of the breadboard and display a schematic of the circuit in the browser.

The circuit-sensing breadboard is composed of a hardware system that interfaces various pieces of test equipment with eight rails on a breadboard. The system is composed of a pcb-mounted breadboard, a voltage source, current meter, and voltage meter multiplexing board, a microcontroller development board, firmware to control ADC sampling, AC voltage source testing, and streaming data to a computer for analysis and display. The hardware is controlled by a circuit sensing algorithm, which determines the equivalent circuit elements between any two nodes on the breadboard. A software simulation of the hardware system verifies operation of the network sensing algorithm and enables testing for networks larger than eight nodes. Once the circuit sensing algorithm reconstructs the entire circuit it generates a schematic diagram of the circuit, closing the loop on breadboard construction.

Chapter 2

Theory

In this chapter, a Network Sensing Algorithm that can reverse-engineer an arbitrary n-node RLC network given access to every node is described. First, it is shown how basic circuit theory is used to analyze and reverse engineer n-node resistive circuits. These methods are then generalized to reverse-engineer n-node impedance networks down to the individual impedance branch circuits, constructed from resistors, capacitors, and inductors. The impedance networks are then further decomposed into their constituent element by method of frequency-domain analysis. Finally, the original network is reconstructed.

2.1 RLC Elements

The design of a Network Sensing Algorithm requires an understanding of the networks to be analyzed. In this thesis, the networks of interest are composed of three circuit elements: resistors, capacitors, and inductors. The constitutive relations for these electrical circuit elements relate the voltage across an element to the current passing through it as a function of time. The constitutive relation for resistors, inductors, and capacitors are as follows:

Resistive element behavior is characterized by a linear relationship between current and voltage. That is, $v = iR$

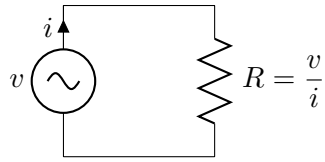


Figure 2-1: Constitutive Relations for Resistance

Inductive element behavior is characterized by a linear relationship between the time derivative of current and voltage. That is, $v = L\frac{di}{dt}$

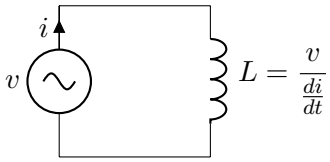


Figure 2-2: Constitutive Relations for Inductance

Capacitive element behavior is characterized by a linear relationship between the time derivative of voltage and current. That is, $i = C\frac{dv}{dt}$

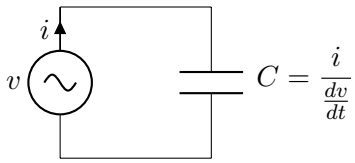


Figure 2-3: Constitutive Relations for Capacitance

When combined to form larger networks these constitutive relations hold, but two other laws of conservation are useful for further analysis.

2.2 Network Analysis

Kirchoff's Current Law

Kirchoff's current law comes from the idea of conservation of charge. That is, charge is neither created nor destroyed. The amount of charge moving along a conductor away from a circuit node is equal to the amount of charge moving along a conductor into that same

circuit node. More succinctly, The sum of the currents going into a node are equal to the sum of currents coming out of that node.

Kirchoff's Voltage Law

Kirchoff's voltage law can be derived from conservation of energy. KVL states that the total amount of energy put into a circuit is equal to the amount of energy lost in that circuit. Since voltage is a measure of the amount of energy per unit charge, the voltage across a circuit branch is indicative of the amount of energy dissipated as work or provided by an electromotive force. Following the last two statements, the sum of all of the voltages around a loop in a circuit is zero.

The following sections will use KVL. KCL, and the constitutive relations to reverse-engineer an n-node network.

2.3 Network Sensing Algorithm

Given access to the set of nodes in an electrical network, the objective of a network sensing algorithm is to determine the set of branches and the elements that compose them. In order to illustrate how a network sensing algorithm operates, it is necessary to begin with a simple example and then build in complexity.

2.3.1 Two Node Network

Take the example of a network with two nodes below:

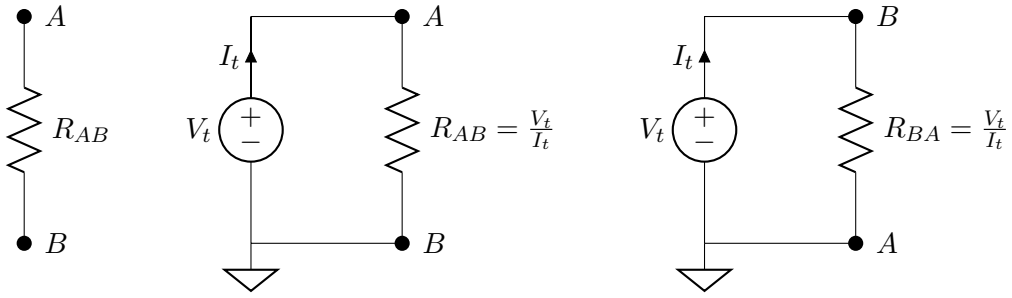


Figure 2-4: Finding R_{AB} in a two node network

In the case of a resistive network with two nodes, there is only one possible branch in the network and thus one possible element to characterize. From Section 2.1, the resistance between two nodes equal to the voltage across the nodes divided by the current through the nodes when power is applied. Here, the resistance R_{AB} is found by placing a test voltage V_t across the nodes and measuring the resulting current, I_t, then taking the ratio $\frac{V_t}{I_t}$. This measurement is called the driving point impedance [1]. If there are no circuit elements between the two nodes, the driving point impedance test will find zero current in the test voltage source, resulting in an infinite resistance between two nodes. An infinite resistance between two nodes in a circuit indicates that there are no elements in that branch of the network. Thus, the network can subsequently be simplified by removing that branch from the network.

[1] To measure a driving point impedance, a test voltage is applied between the node of interest and ground, and the resulting current in the voltage source is measured. The driving point impedance is then computed by dividing the test voltage by the resulting current.

2.3.2 Three Node Network

A resistive network with two nodes is simple, but provides an introduction to the methodology used in analyzing larger networks. In the case of a resistive network with three nodes, it is insufficient to utilize driving point impedance measurements alone, because each node has more than one path to any other node. The ability to ground arbitrary nodes of a network greatly simplifies the process of analyzing and reverse-engineering a network. The network-sensing algorithm relies on the ability to shrink the effective network by conglomerating nodes into the ground node.

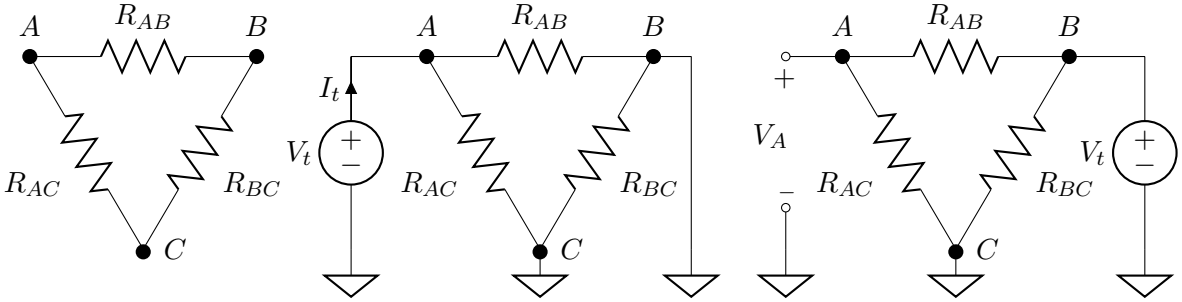


Figure 2-5: Finding R_{AB} in a three node network

Consider a resistive network with three nodes: A, B, and C. In order to determine the resistance in branch AB, the driving point impedance at node A is measured with nodes B and C grounded. This provides the resistance of the parallel combination of the branches with an endpoint at node A:[2]

$$R_{A_{||}} = R_{AB}||R_{AC}$$

Next, a test voltage source is applied to node B, node C is grounded, and the voltage at node A, V_A, is observed.

$$V_A \quad = \quad V_t \frac{R_{AC}}{R_{AB} + R_{AC}} \quad = \quad V_t \frac{R_{AB}||R_{AC}}{R_{AB}} \quad = \quad V_t \frac{R_{A_{||}}}{R_{AB}}$$

The branch resistance of interest, R_{AB} is calculated using the known quantities V_t, $V_{A||}$, and V_A.

$$V_t \frac{R_{A||}}{V_A} \quad = \quad V_t \frac{R_{AB}||R_{AC}}{V_t \frac{R_{AC}}{R_{AB}+R_{AC}}} \quad = \quad \frac{\frac{R_{AB}R_{AC}}{R_{AB}+R_{AC}}}{\frac{R_{AC}}{R_{AB}+R_{AC}}} \quad = \quad R_{AB}$$

[2] $R_1||R_2 = \frac{R_1 R_2}{R_1 + R_2} = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2}}$

This procedure is repeated for the remaining branches to determine the entire network.

2.3.3 N Node Network

A resistive network with any number of nodes can be reduced to a resistive network with three nodes by grounding the nodes that are not of interest. The resulting network does not modify the branch of interest, but connects the remaining branches attached to the nodes of interest in parallel. This collapses the network into a three node network or three branch equivalent circuit. Consider a resistive network with five nodes: A-E. To determine

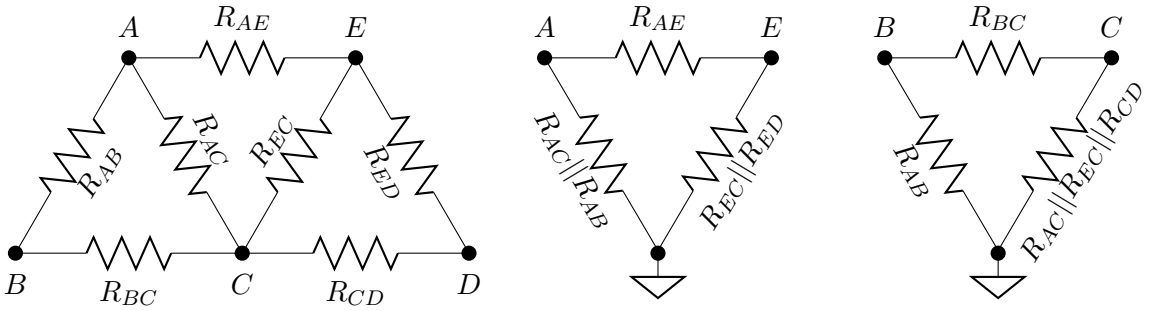


Figure 2-6: Collapsing five node network, collapsed to find R_{AE} and R_{BC}

the resistance between nodes A and E, the network is reduced to a three-node network by connecting all nodes except nodes A and E to ground. The three resistances of the branches that remain are R_{AE}, $R_{AB}||R_{AC}$, and $R_{EC}||R_{ED}$. The reduced network is then solved using the three node network method, and this procedure is repeated for all branches in the network.

The above thought experiment satisfies the problem at hand intuitively, but it can also be shown mathematically.

To find R_{AB} of an n-node complete network, first the driving point impedance at node A, $R_{A_{||}}$, is measured with all other nodes grounded.

$$\begin{aligned} R_{A_{||}} &= R_{AB}||R_{AC}||R_{AD}||... &&= \frac{1}{\frac{1}{R_{AB}}+\frac{1}{R_{AC}}+\frac{1}{R_{AD}}+...} \\ &&&= \frac{1}{\frac{1}{R_{AB}}+\frac{1}{(\frac{1}{\frac{1}{R_{AC}}+\frac{1}{R_{AD}}+...})}} \\ &&&= R_{AB}||(R_{AC}||R_{AD}||...) \end{aligned}$$

To find V_A from node B, the voltage at node A is measured when a test source is placed at node B with all other nodes grounded.

$$V_A \quad = \quad V_t \frac{(R_{AC}||R_{AD}||...)}{R_{AB} + (R_{AC}||R_{AD}||...)}$$

Dividing $R_{A_{||}}$ by V_A and multiplying by V_t:

$$V_t \frac{R_{A_{||}}}{V_A} \quad = \quad V_t \frac{\frac{R_{AB}(R_{AC}||R_{AD}||...)}{R_{AB}+(R_{AC}||R_{AD}||...)}}{V_t \frac{(R_{AC}||R_{AD}||...)}{R_{AB}+(R_{AC}||R_{AD}||...)}} \quad = \quad R_{AB}$$

Finding a resistive element in an n-node network requires two measurements. The number of branches in a complete graph with n nodes is equal to $\frac{n(n-1)}{2}$. The total number of measurements required to reverse-engineer an n-node circuit is $n(n-1)$.

2.3.4 Element Identification

Networks composed of elements with complex impedance can be analyzed with the same algorithm. By replacing the test DC voltage sources with AC voltage sources, the reactance of capacitors and inductors can be measured in addition to the resistance of resistors.

From Resistance to Impedance

Impedance is a complex valued ratio of the voltage across a circuit to the current flowing through that circuit. source the sum of resistance and reactance. Here, frequency-domain complex impedance is useful for identifying circuit elements based on the change in branch impedance over inputs of various frequencies.

MAKE NOTE ABOUT USING THE AMPLITUDE OF THE MEASURED SIGNALS

Parallel RLC Branches

To determine if there are multiple element types [in parallel] between two nodes, the impedance is measured over a range of frequencies, and the resulting changes in impedance are measured.

The impedance of a parallel RLC 'tank' circuit can be characterized and analyzed over all frequencies.

$$Z_R = R \qquad Z_L = j\omega L \qquad Z_C = \frac{1}{j\omega C}$$

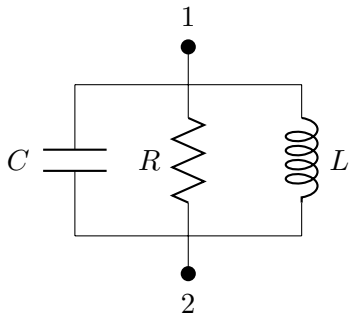
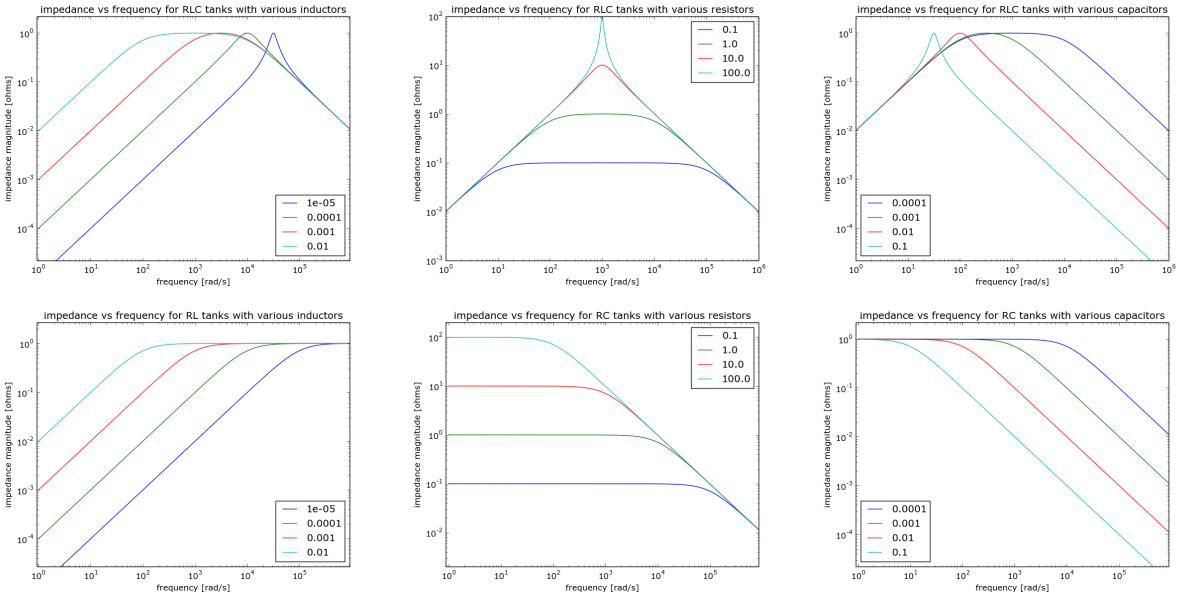


Figure 2-7: Example RLC Tank Circuit

$$Z_{RLC} = Z_C||Z_R||Z_L = \frac{1}{j\omega C + \frac{1}{R} + \frac{1}{j\omega L}} = \frac{j\omega RL}{-\omega^2 RLC + j\omega L + R}$$

Z_{RLC} has a zero at $\frac{1}{RL}$ and two poles at $\frac{-L \pm \sqrt{L^2+4R^2LC}}{-2RLC}$



The plots above illustrate various RLC tank circuits where one component value changes over several orders of magnitude. In the top-center and bottom-center plots, a decrease in resistance drops the maximum impedance over all frequencies. In the top-left and bottom-left plots, a decrease in inductance drops the asymptotic impedance at low frequencies In the top-right and bottom-right plots a decrease in capacitance drops the asymptotic impedance at high frequencies. As illustrated above, there are three regimes on the impedance vs frequency plot that divide the operation of an RLC tank into three elements:

When the slope of $|Z|$ *vs.* f is +1, the tank behaves like an inductor with inductance

$$L = |Z|/(j\omega)$$

When the slope of $|Z|$ *vs.* f is -1, the tank behaves like a capacitor with capacitance

$$C = j\omega|Z|$$

When the slope of $|Z|$ *vs.* f is 0, the tank behaves like a resistor with resistance

$$R = |Z|$$

Finite Difference Stencil

To reliably determine the regions of effective resistance, inductance, and capacitance, the slope of the $log(|Z_{nm}|)$ vs. $log(\omega)$ is computed via the midpoint method. The midpoint finite difference stencil takes the two points on either side of the point of interest and computes the slope between them.

$$Z'[n] = \frac{1}{2}(Z[n+1] - Z[n-1])$$

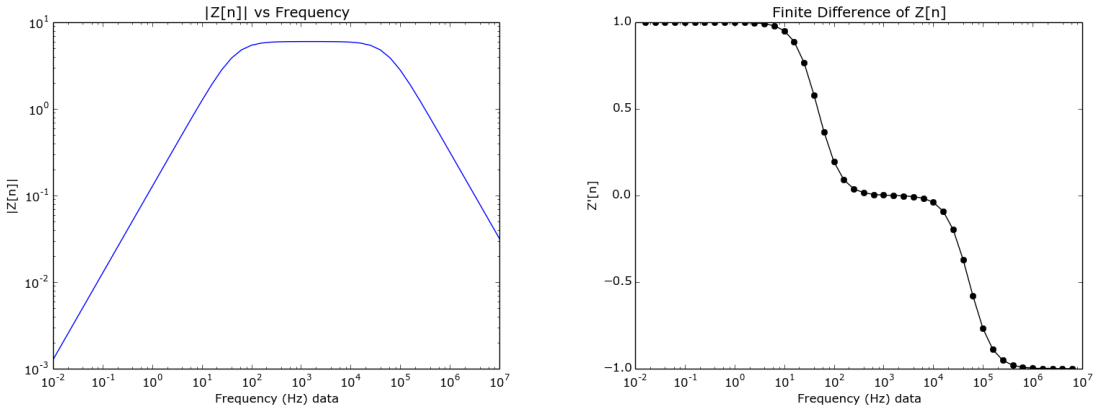


Figure 2-8: Finite difference of an RLC tank where R=6Ω, L=20mH, and C=500nF

In figure **??** the distinct regions of inductance, resistance, and capacitance are clearly visible and easily found by setting thresholds for the finite difference. The thresholds on

the region of inductance are between 1.1 and 0.9, the thresholds on the region of resistance are between 0.1 and -0.1, and the thresholds on the region of capacitance are between -0.9 and -1.1.

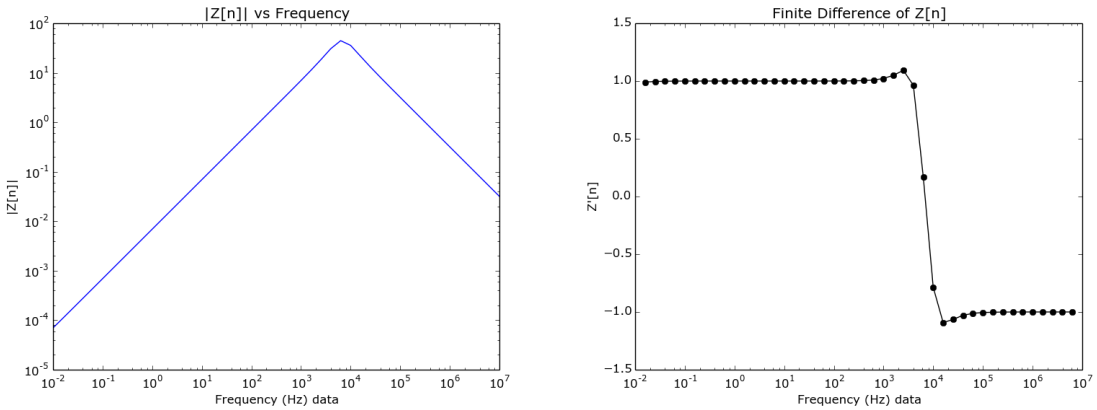


Figure 2-9: Finite difference of an RLC tank where R=44Ω, L=1.1mH, and C=500nF

In figure **??**, the resistance is not represented in the finite difference of `Z[n]`. The parallel resistance of an underdamped RLC tank is the maximum recorded impedance across all frequencies. In a simulation, using the maximum recorded impedance is a valid solution because measurement accuracy has a large (64-bit floating point) dynamic range, producing results that are accurate of ideal systems. In practice, spurious noise and lack of dynamic range over measurement data can cause poor results. Occasionally these poor results include large spikes of inaccurate impedance readings. In this case, a practical approach would be to use the finite difference stencil where possible and when an LC network is discovered, sample at the resonant frequency to find R. In the case of figure **??**, the sampling frequency would be 17KHz.

$$f_0 = \frac{1}{\sqrt{2\pi LC}} \approx 17000Hz$$

Component Value Calculation

Once the regions of the impedance plot are thresholded, the sampling frequency domains corresponding to those regions are assigned to an element type [$\omega_r[n]$ for R, $\omega_l[n]$ for L, $\omega_c[n]$ for C]. Then for each sampling frequency the component values are computed and

averaged to filter noisy data.

$$R[\omega_r] = Z[\omega_r] \qquad L[\omega_l] = \frac{Z[\omega_l]}{\omega_l} \qquad C[\omega_c] = \frac{1}{\omega_c Z[\omega_c]}$$

$$R = \frac{1}{rnum} \sum_{\omega_r} Z[\omega_r] \qquad L = \frac{1}{lnum} \sum_{\omega_l} \frac{Z[\omega_l]}{\omega_l} \qquad C = \frac{1}{cnum} \sum_{\omega_c} \frac{1}{\omega_c Z[\omega_c]}$$

Where *rnum*, *lnum*, and *cnum* are the number of sample frequencies in the R, L, and C regions, respectively.

2.3.5 Reconstructing the Network

With a record of all of the elements and their connections in the network, the network can be reconstructed and a schematic of the original circuit can be drawn.

Chapter 3

Simulation

In this chapter, a simulation of the circuit sensing breadboard is described. A simulated circuit sensing breadboard is useful for rapid iteration of hardware topologies and validation of network sensing algorithms. The simulator is also not limited by the hardware complexity, so simulating circuits with more than eight nodes and elements other than resistors, capacitors, and inductors is possible. The simulator generates random networks of resistors, inductors, and capacitors, and proceeds to analyze, reconstruct, and draw a schematic of the network using the network sensing algorithm.

3.1 NgSpice and Netlists

The open source software package NgSpice was used to simulate the networks and test circuits applied to the network. NGspice operates on text files called netlists, where each component in the network is specified on a single line. An example netlist is shown below.

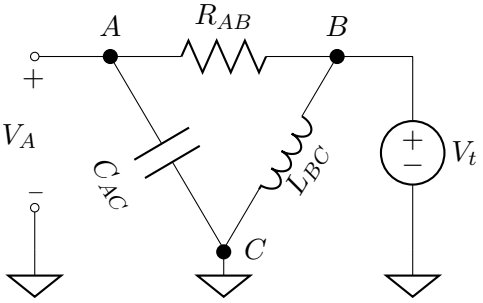


Figure 3-1: Five node network

```
fiveNodeNetlist
Vt        0 B 1
Rab A B 10k
Cac A C 1e-6
Lbc B C 1e-3
Vcg 0 C 0
.control
        op
        print(v(A))
        .endc
.end
```

The first letter of each element line designates the element to simulate:

`V start stop value`→ Voltage Source between `start` and `stop` nodes, `value` Volts [V]

`R start stop value`→ Resistor between `start` and `stop` nodes, `value` Ohms [Ω]

`L start stop value`→ Inductor between `start` and `stop` nodes, `value` Henries [H]

`C start stop value`→ Capacitor between `start` and `stop` nodes, `value` Farads [F]

Node 0 is always designated as ground, and all simulations require a ground node.

The control commands are as follows:

`op` → Operating Point Simulation `print()` → Print the relevant data passed as an argument

`v(N)` → The voltage at node N `i(E)` → The current through element E

3.2 Methods

The NSA simulator was written in python and uses the methods below.

3.2.1 Generate Random Netlist

`writeRandomNet(netlist,num,elements):`

Generates `num`-node random graph with no self-linking nodes (symmetric matrix with zeros on the diagonal) for each `[elements]` type (R,L,C). Subsequently assigns random values between two realistic limits for each element and writes the network to netlist `netlist`. 1-1k ohms, 10nF-10uF, 100uH-100mH.

When writing the capacitive and inductive elements, care must be taken to prevent a DC operating point simulation from failing. The infinite resistance across a capacitor and

zero resistance across an inductor are responsible for DC operating point simulation failure, and can be fixed by including a small resistor in series with inductors and a large resistor in parallel with capacitors.

`L0 1 2 1mH`	→	`L0 1 tl0 1mH` `Rl0 tl0 2 1e-5`	`C0 1 2 1e-6F`	→	`C0 1 2 1e-6F` `Rc0 1 2 1e8`

3.2.2 Inserting Voltage Sources, Grounds and Probes

`insertProbe2(target,nodes,groundNodes,probes,source='DC'):`

Inserts a 1V voltage source from ground to each node in `[nodes]`, grounds each node in `[groundNodes]`, and adds a voltage print statement for each node in `[probes]`. By default, the voltage sources are written as DC sources, but if 'AC' is passed into the last argument the sources are written as AC sources and the AC control statement is added.

`AC dec 5 10m 10meg`

Which runs a small-signal AC simulation and returns the amplitudes of the resulting voltage and current waveforms, five sample points per decade from 10mHz to 10MHz.

3.2.3 Run Simulation

`def runSim(target,results,source='DC'):`

Makes a system call to NgSpice in batch mode with netlist `target` and outputs the result to text file `results`. The last argument indicates how to parse the resulting data, as NgSpice returns DC data in the following format:

```
No. of Data Rows : 1
i(v) = -1.18295e-01
v(5) = 2.532846e-07}
```

and AC data is returned in this format:

```
No. of Data Rows : 6
                                    mynetlist
                                    AC Analysis  Sun Aug 30 18:35:07  2015
--------------------------------------------------------------------------------
Index   frequency       i(v)
--------------------------------------------------------------------------------
0        1.000000e+00        -1.72598e+00,        3.978810e+02
1        1.000000e+01        -1.58689e-01,        3.978827e+01
```

```
2        1.000000e+02        -1.43015e-01,        3.974287e+00
3        1.000000e+03        -1.42859e-01,        3.520201e-01
4        1.000000e+04        -1.42857e-01,        -4.18884e-01
5        1.000000e+05        -1.42857e-01,        -4.58275e+00

                                    mynetlist
                                    AC Analysis  Sun Aug 30 18:35:07  2015
--------------------------------------------------------------------------------
Index   frequency       v(3)
--------------------------------------------------------------------------------
0        1.000000e+00        1.000000e+00,        0.000000e+00
1        1.000000e+01        1.000000e+00,        0.000000e+00
2        1.000000e+02        1.000000e+00,        0.000000e+00
3        1.000000e+03        1.000000e+00,        0.000000e+00
4        1.000000e+04        1.000000e+00,        0.000000e+00
5        1.000000e+05        1.000000e+00,        0.000000e+00
```

DC data is returned as a dictionary with indices for the voltage source current and each node of interest:

`['i':i(V),'1':v(1),'2':v(2),..]`

AC data is returned as a dictionary of lists with indices for the voltage source current and each node of interest:

`['i':[i(`V_{f_1}`), i(`V_{f_2}`),..., i(`V_{f_n}`)], '1':[v(`1_{f_1}`), v(`1_{f_2}`),..., v(`1_{f_n}`)], ...]`

3.2.4 Print Matrix

`def printMatrix(m):` Prints matrix `m` in nice command-line output. It can handle both 2-dimensional and 3-dimensional matrices, for n-by-n matrices with lists of impedances over many frequencies.

3.2.5 Write JSON

`def writeJSON(name, network,group, elem):` Writes a JSON file named `name.json` of the n-by-n symmetric matrix `network`. Three separate JSON files are written, one for each element `[resistor.json, capacitor.json, and inductor.json]`. The `group` parameter is used when drawing multiple schematics on one page. The `elem` parameter is used to specify what type of element the network represents.

3.3 Executing NSA

3.3.1 Calculate $Z_{n||}(f)$

$Z_{n||}(f)$ is found by grounding all nodes except for node n and adding a voltage source to that node, then taking the ratio of the amplitudes of the resulting current into the node of interest and the voltage source.

3.3.2 Calculate $V_n(f)$

$V_n(f)$ is found by grounding all nodes except for nodes n and m, adding a voltage source to node m, and measuring the amplitude of the voltage at node n.

3.3.3 Calculate $Z_{nm}(f)$

$Z_{nm}(f)$ is calculated by the ratio of $Z_{n||}(f)$ to $V_n(f)$ scaled by V_t. In the case of this simulation, V_t is one.

3.3.4 Finite Difference

Z_{nm} is passed through the finite-differencer, returning a list $Z'_{nm}[f]$. $Z'_{nm}[f]$ is thresholded to find the frequencies for which $Z'_{nm}[f] \approx -1$, $Z'_{nm}[f] \approx 0$, and $Z'_{nm}[f] \approx 1$.

3.3.5 Element Identification

A list of inductances is calculated from $Z_{nm}[f]$ on all frequencies for which $Z'_{nm}[f] \approx -1$, $[f_L]$.

$L[f_L] = \frac{|Z_{nm}[f_l]|}{(f_l)}$

A list of resistances is calculated from $Z_{nm}[f]$ on all frequencies for which $Z'_{nm}[f] \approx 0$, $[f_R]$.

$R[f_R] = |Z_{nm}[f_R]|$

A list of capacitances is characterized from $Z_{nm}[f]$ on all frequencies for which $Z'_{nm}[f] \approx -1$, $[f_C]$.

$C[f_C] = \frac{1}{|Z_{nm}[f_C]|(f_C)}$

The element lists are averaged to to filter out inaccurate data.

3.3.6 Network Reconstruction

A new matrix is written for each element type, then the matrices are written to JSON and processed by javascript running D3 to display the final schematic.

Chapter 4

Hardware

In this chapter, hardware system design is covered. The hardware system is decomposed into its subsystems, and the design choices, specifications, and operation of those subsystems are explained. The hardware system is composed of a set of eight breadboard rails connected to a node voltage reading system, a test voltage source system, a grounding system, and a 32-bit microcontroller development board. The node voltage reading system is composed of a voltage multiplexer and an external A/D converter. The test voltage source system is composed of an internal D/A converter, an external buffer, a current sensing amplifier, an external A/D converter, and eight high-side analog switches. The grounding system is composed of eight N-channel MOSFETs, handled by the microcontroller. A simplified block diagram is below:

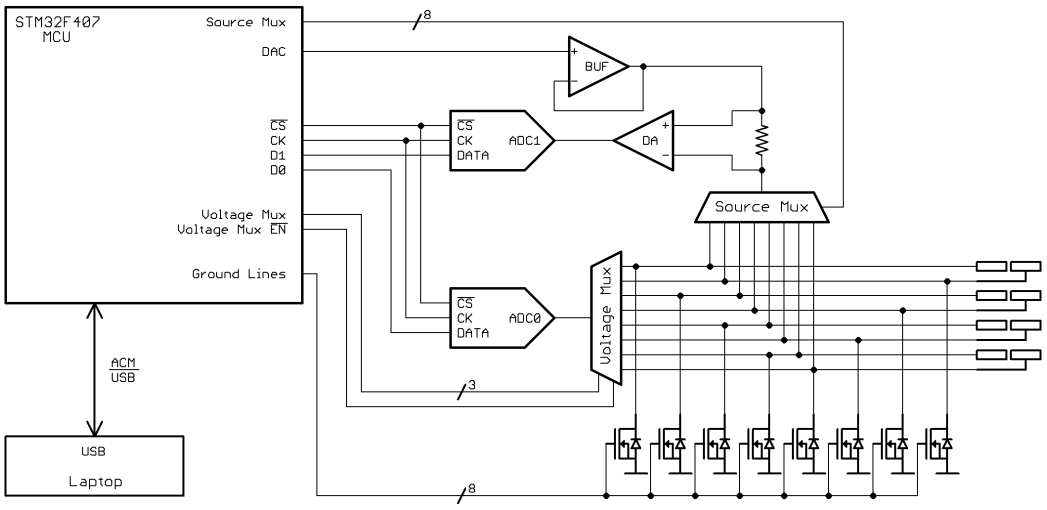


Figure 4-1: Simplified block diagram of hardware

4.1 Node Voltage Reading

An ADCS7476 12-bit A/D converter is used to measure the voltages at each node. The ADCS7476 can sample up to 1MSPS with ±1 LSB of total unadjusted error from -40^oC to 85^oC and $< \pm 0.2$ LSB of error at 25^oC. No bits are wasted in the ADCS7476 A/D converter. Additionally, the input circuitry to the ADC is a 100Ω resistor in series with a 26pF capacitor. This places a pole at 61MHz, causing a .03% error at 100KHz and .3% error at 1MHz. The additional performance is well worth the additional cost of $1.56375 in quantities of 1Ku.

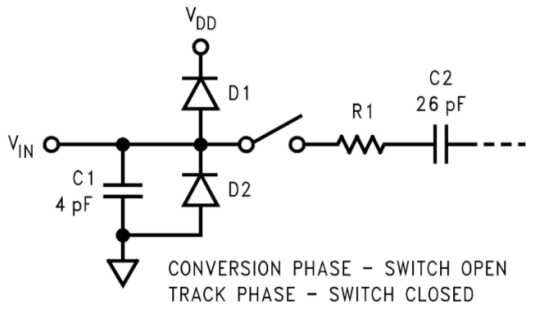


Figure 4-2: ADCS7476 Equivalent Input Circuit

The ADC is connected to the common pin of a CD4051 1:8 analog multiplexer. The multiplexer is connected to eight breadboard rails, which allows the ADC to measure the voltage on any of the eight rails, one rail at a time.

The CD4051 has about 200Ω of series resistance and 30pF of output capacitance when its supply voltage is 12V, which places an additional pole at 26MHz, again well above the Nyquist frequency.

So far, the voltage-reading signal chain has two poles - one at 61MHz and one at 26MHz.

4.1.1 Onboard A/D converters

Although the STM32F407 has three onboard 12-bit A/D converters, their specifications are lacking. Each is able to sample at 2MSPS, and it's possible to interleave them to attain a sampling rate of 6MSPS or higher if you're willing to throw away bits. The total unadjusted error (offset error, gain error, differential linearity error, and integral linearity error) is between ±2 LSB and ±5 LSB. With a minimum of ±2 LSB's of error, the lowest significant two bits in the 12-bit A/D are virtually useless. Another specification to consider

is the input circuitry to the ADC. The input circuitry to the ADC while it's in tracking mode looks like a 6KΩ resistor charging a 4pF capacitor. This puts a pole at 6.6MHz, causing .3% error in measurement at 100KHz, 3% error in measurement at 1MHz, and 7% at 3MHz. When sampling at the maximum sample rate of 6MSPS, the ADC's input network begins to introduce significant error. Granted, it's likely that there will be an alternative bandwidth bottleneck, this is still a metric of concern. [DM00037051.pdf, pages 133-134]

4.2 Signal Generator

The STM32F407 has an onboard D/A converter that is good enough to use as a signal generator. The onboard DAC is configured to output a cosine wavetable with a DC offset, as described in the next chapter.

4.3 Test Voltage Current Sensing

The DAC output is buffered by half of an MCP6L92 10MHz op-amp. The onboard DAC can be configured with an optional onboard buffer, but the buffer limits the DAC output range from 0.2V to Vdd-0.2V. Without the onboard buffer, the DAC output is 15kΩ. When configured as a voltage buffer, the MCP6L92 has an input impedance larger than 1GΩ, resulting in no signal attenuation.

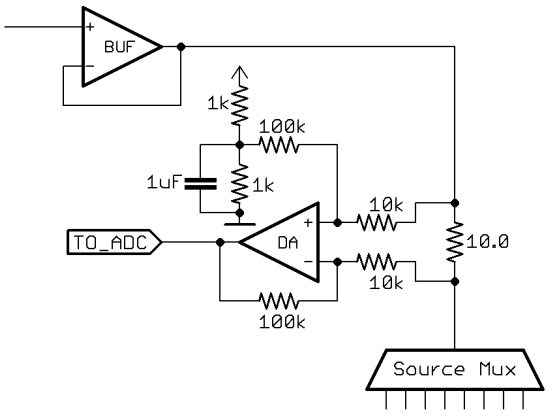


Figure 4-3: Difference Amplifier Schematic

The buffer sources current through a 10.0Ω, 1% sense resistor, which can be switched onto any of the breadboard rails through an array of eight high-side switches. The voltage

across the sense resistor is measured by the other half of the MCP6L92 op-amp configured as a simple difference amplifier. Using two 10kΩresistors and two 100kΩresistors, the difference amplifier is configured with a gain of 10.

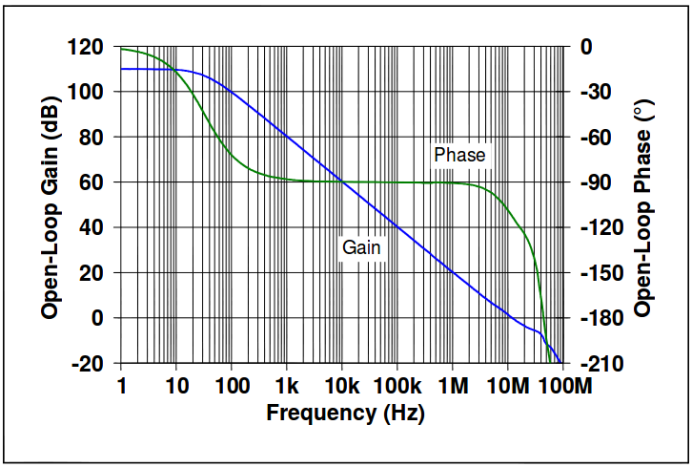


Figure 4-4: MCP6L92 Open Loop Bode Plot

According to the MCP6L92 datasheet, the difference amplifier should have a -3dB bandwidth of 1MHz and plenty of phase margin driving the 100Ω- 26pF input impedance to the current sense ADC.

4.4 High-side Switches

The high-side switches were selected for high-voltage operation, so that any rail of the breadboard could swing between 0 and 30V and there wouldn't be a problem. The selected switches were Vishay DG468 normally open analog switches. They have 9Ωresistance and 76pF of capacitance in the on-state, 1nA of leakage current and 30pF of capacitance in the off-state. The high and low side switches are the main limits of bandwidth due to their high amounts of input capacitance in both on and off states.

4.5 Low-side Switches

The low-side switches have a low logic-level threshold voltage, low drain-source on-resistance, can handle up to 30V, and are low-cost. The IRLML2803 N-FETs have an $R_{DS_{ON}}$ of about 1Ωwith a V_{GS} of 3.3V. With 10V V_{GS}, $R_{DS_{ON}}$ drops to 250mΩ. The downside of these

FETs is the high C_{DS} that comes from a wide transistor. C_{DC} is on the order of 60pF for each transistor. When combined with the additional capacitances mentioned above, each breadboard rail has a total of 140pF to ground. This has a significant impact on the maximum usable frequency for even moderate impedances on the breadboard. For example, consider a 10kΩ-10kΩresistor divider. At 100kHz, the 140 pF capacitance on each rail has 11kΩof impedance.

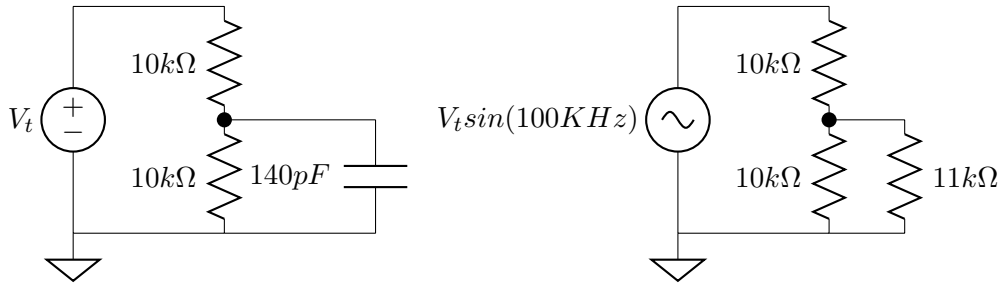


Figure 4-5: Resistor Divider Parasitic Capacitance

4.6 PCB Mounted Breadboard

Modern solderless breadboards are composed of three components: a plastic face, metal finger springs (rails), and a foam backing. The finger springs are designed to fit snugly in the plastic face, which provides structure for the entire board. The foam backing keeps the finger springs from being pushed out the back of the plastic face..

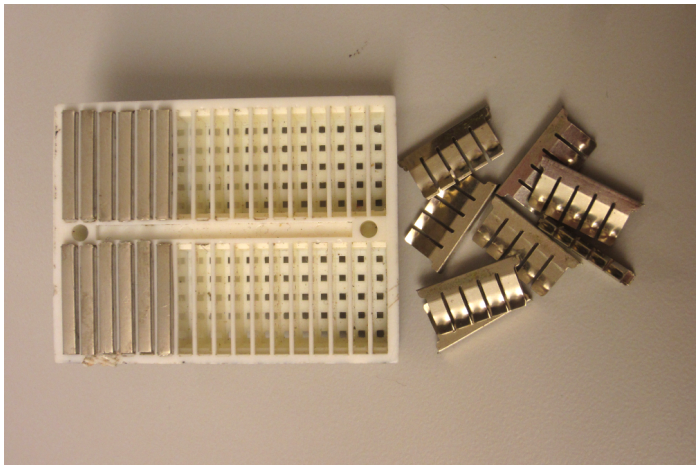


Figure 4-6: Solderless breadboard plastic face with finger spring rails removed

A breadboard can be mounted to a PCB by removing the foam backing and surface mount soldering the finger springs to a PCB, then pressing the plastic face onto the mounted rails Initial tests of this technique involved small numbers of finger springs hand-soldered one-by-one to a PCB, using nail polish as soldermask.

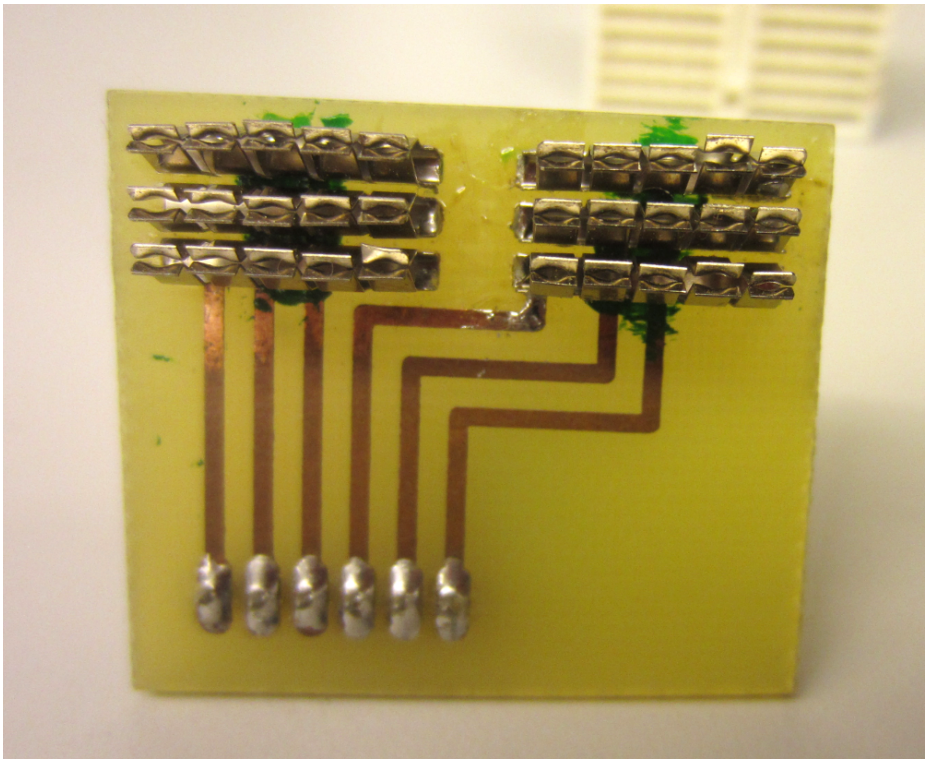


Figure 4-7: Breadboard rails hand-mounted on copper clad PCB

The results of these tests were promising, but alignment proved to be an issue. To mitigate this, a finger spring jig was 3d-printed to loosely hold each finger spring in precise alignment for PCB mounting.

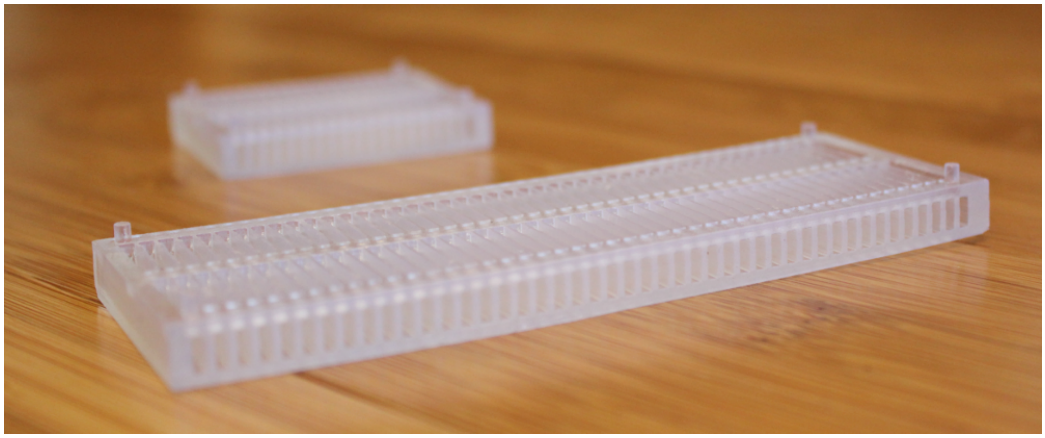


Figure 4-8: 3D printed finger spring jigs

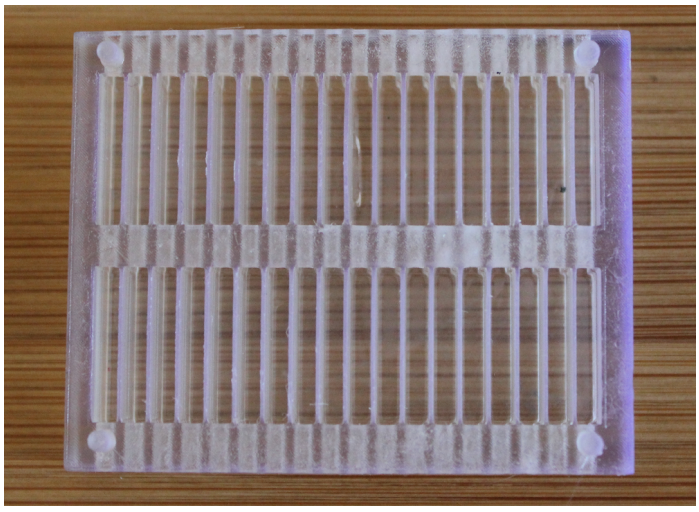


Figure 4-9: Top view of 36-position finger spring jig

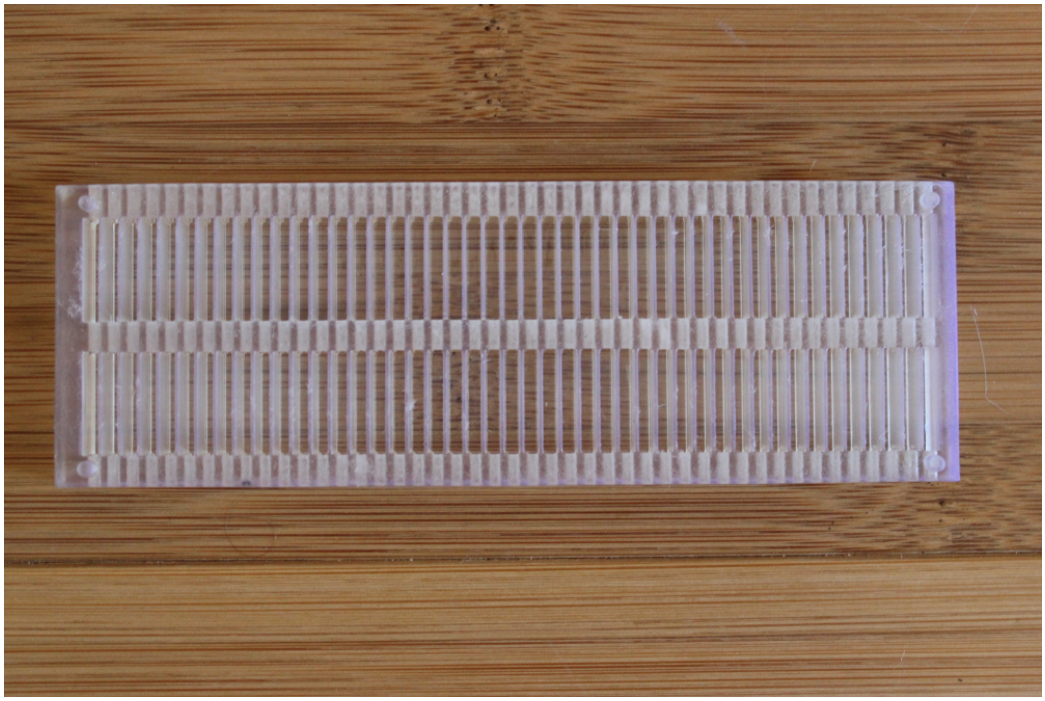


Figure 4-10: Top view of 90-position finger spring jig

The finger spring jig was designed in openSCAD and has four alignment pegs to mate with alignment holes on a PCB. To assemble a PCB mounted breadboard, the finger spring rails are removed from the breadboard and inserted into the finger spring jig.

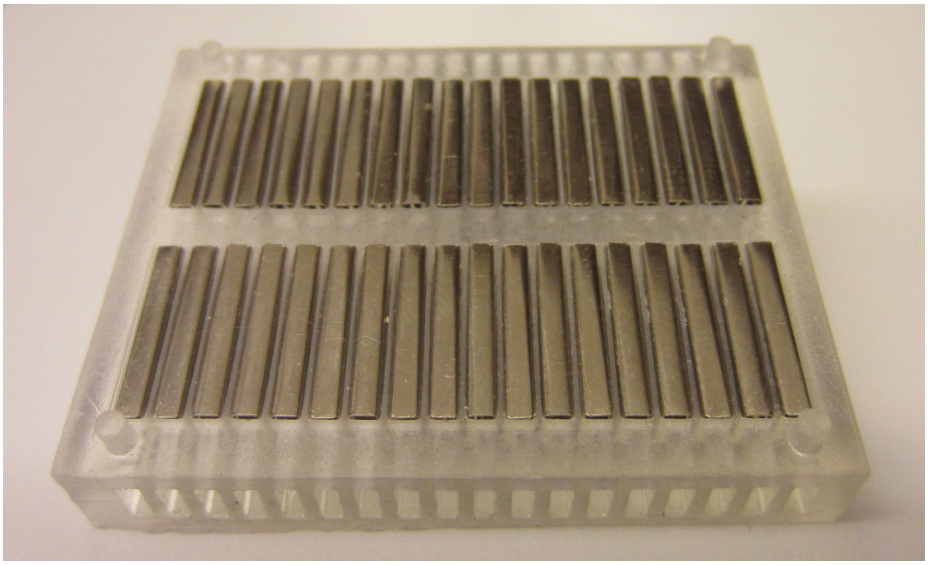


Figure 4-11: Fully loaded 36-position finger spring jig

Once all of the rails are placed, a dot of super glue is dispensed in the center of each finger spring, between two dots of solderpaste. The rails are then adhered to the PCB by pressing the PCB upside-down onto the finger spring jig while the alignment pegs are properly mated with the alignment holes in the PCB. This ensures that the array of finger spring rails are aligned with the pads on the PCB. After about ten seconds the rails adhere to the PCB and the PCB can be carefully separated from the finger spring jig. Finally the assembly is reflowed with hot air or in a reflow oven, and the plastic faceplate can be pressed on.

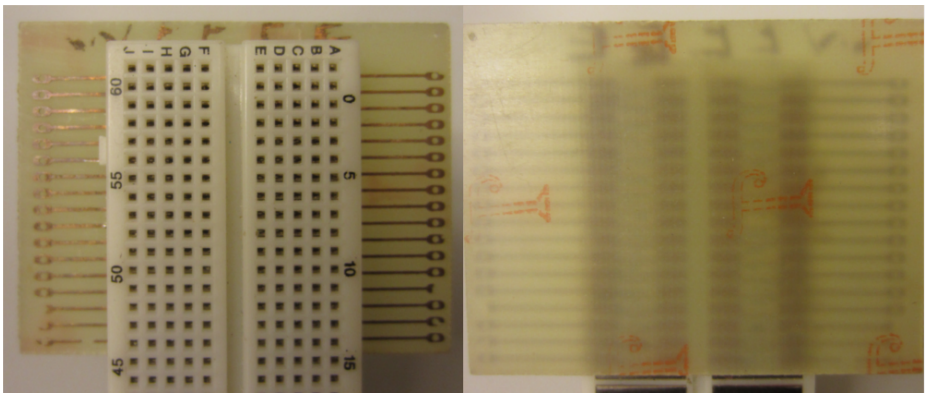


Figure 4-12: Front and back of PCB mounted breadboard

4.7 Hardware Prototypes

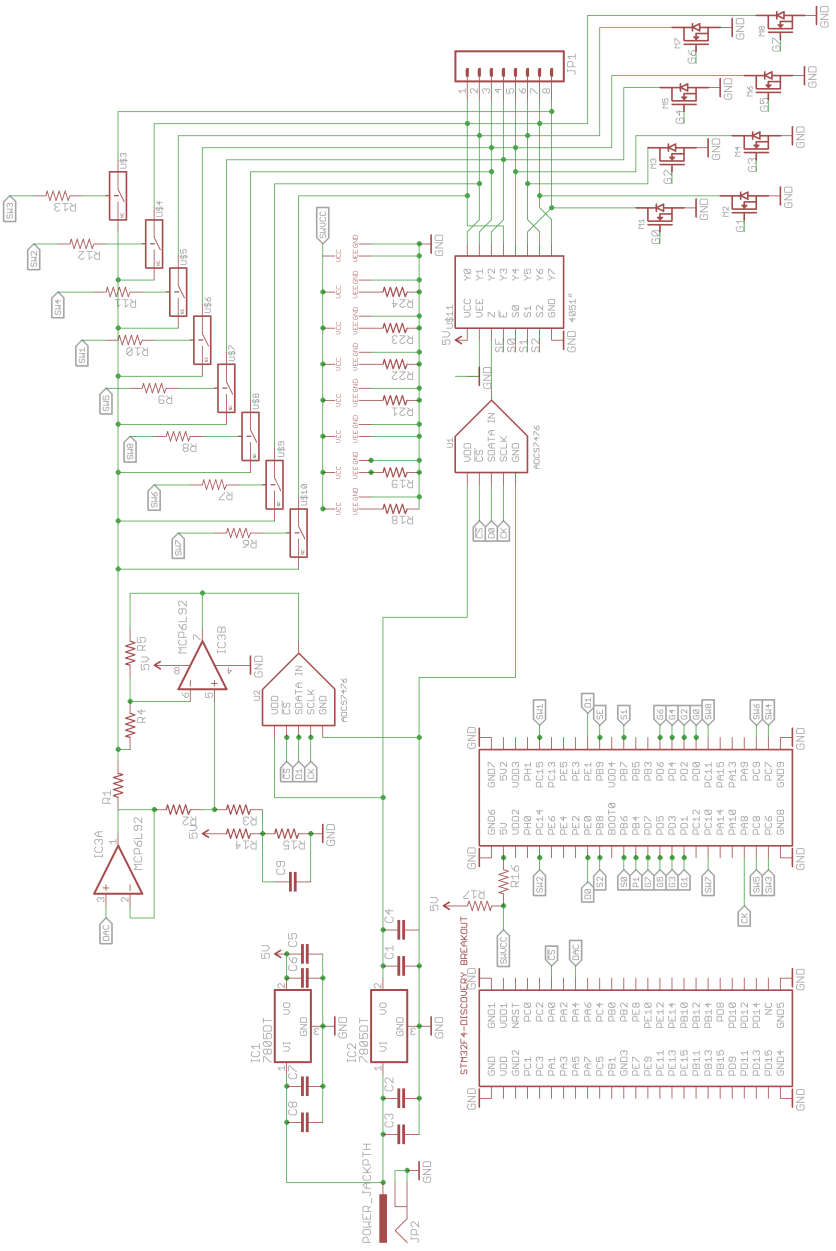


Figure 4-13: Schematic of full hardware system

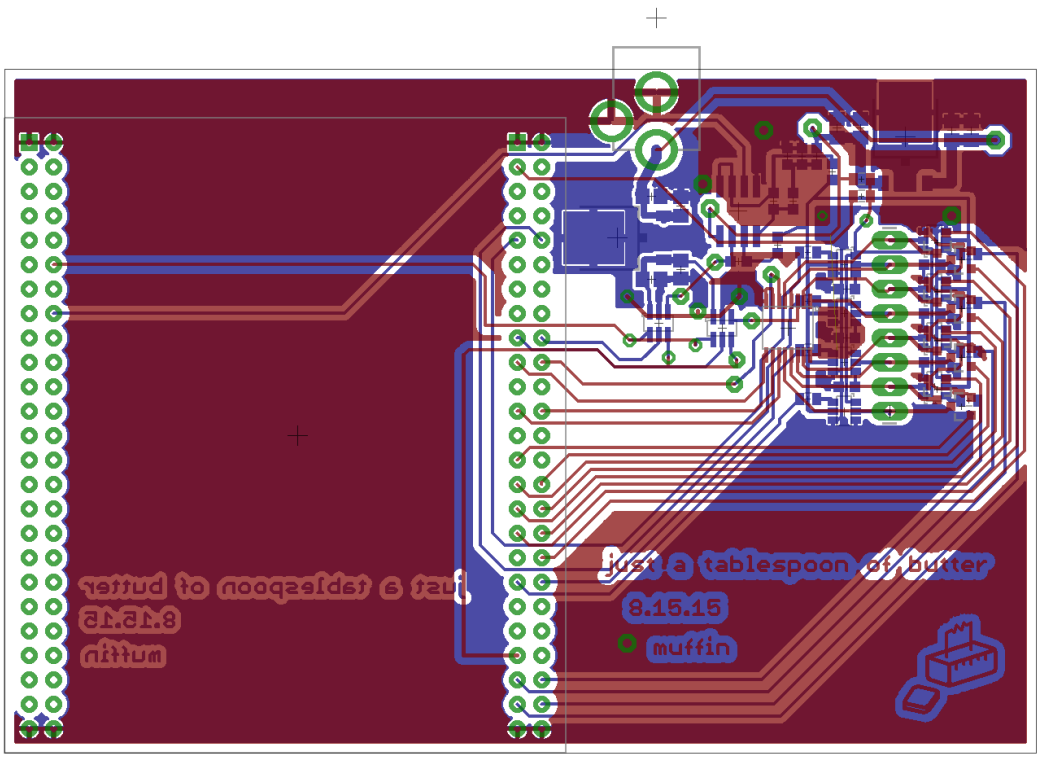


Figure 4-14: Layout of full hardware system

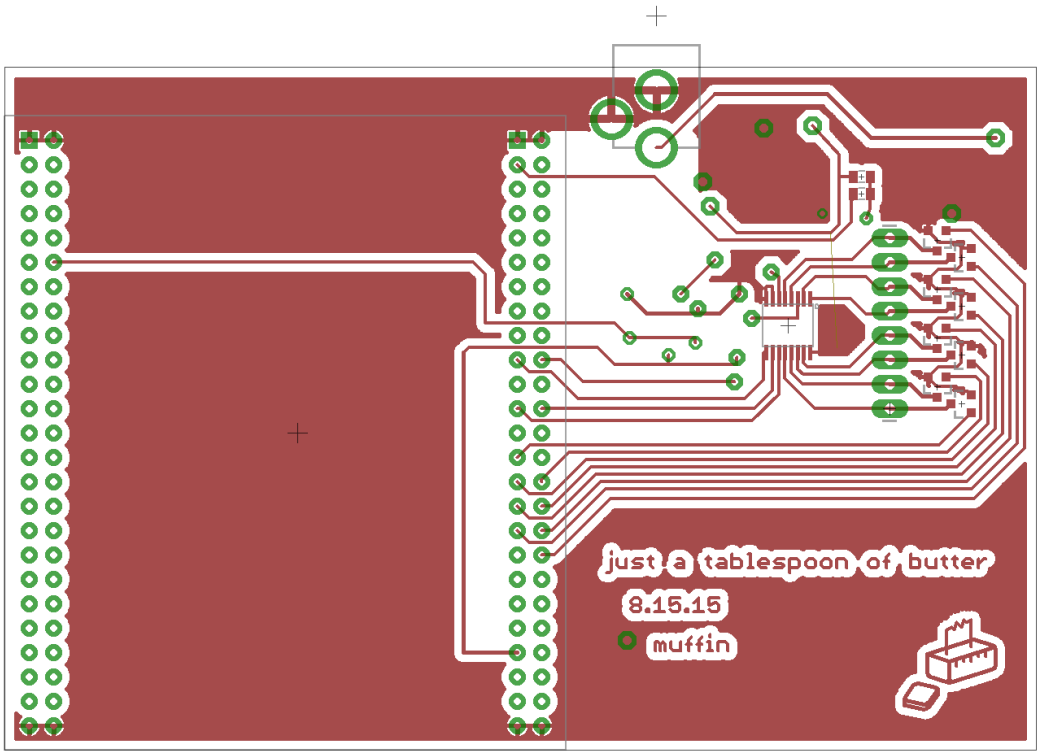


Figure 4-15: Top layer of full hardware system

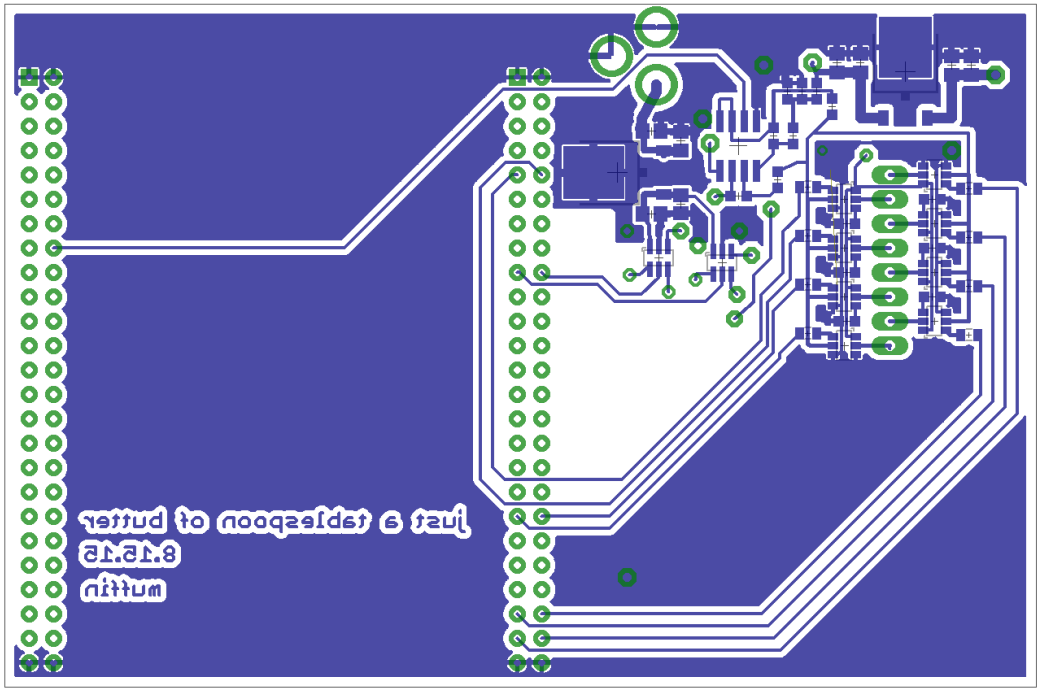


Figure 4-16: bottom layer of full hardware system

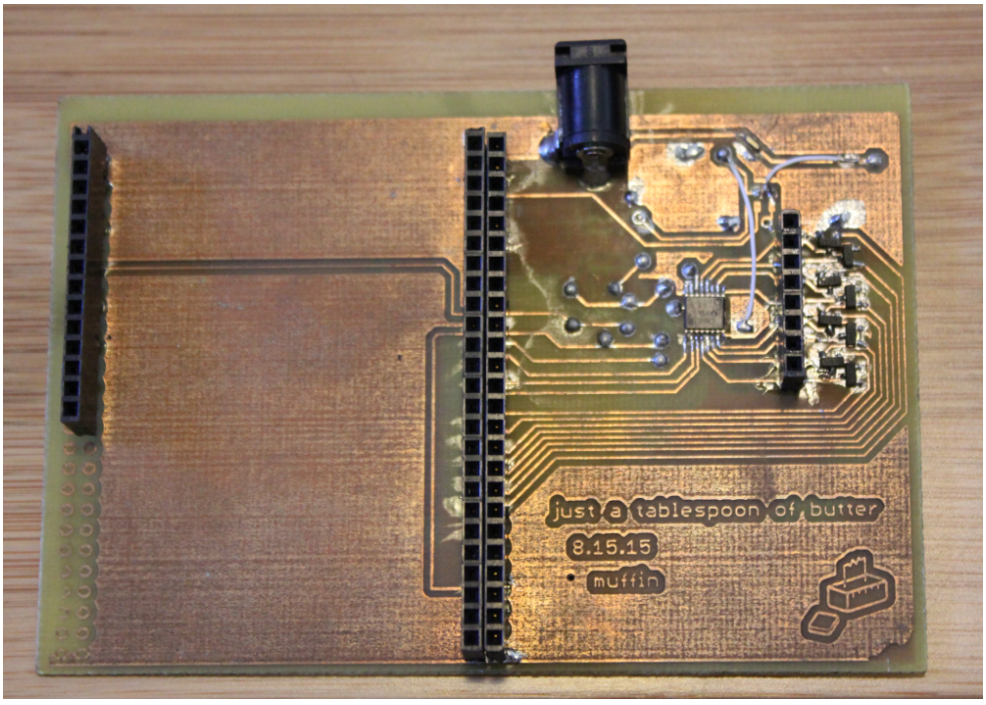


Figure 4-17: Top side of completed hardware system - power jack, voltage reading multiplexer, low-side switches, and 8-pin breadboard header are visible.

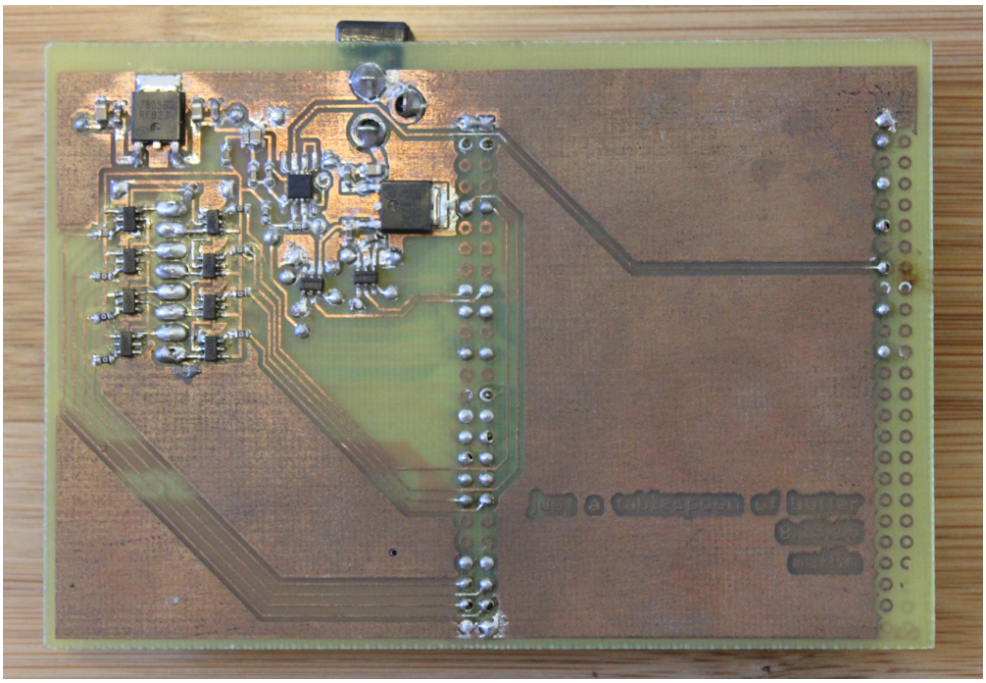


Figure 4-18: Bottom side of completed hardware system - voltage regulators, A/D converters, high-side switches, and current-sense amplifier are visible

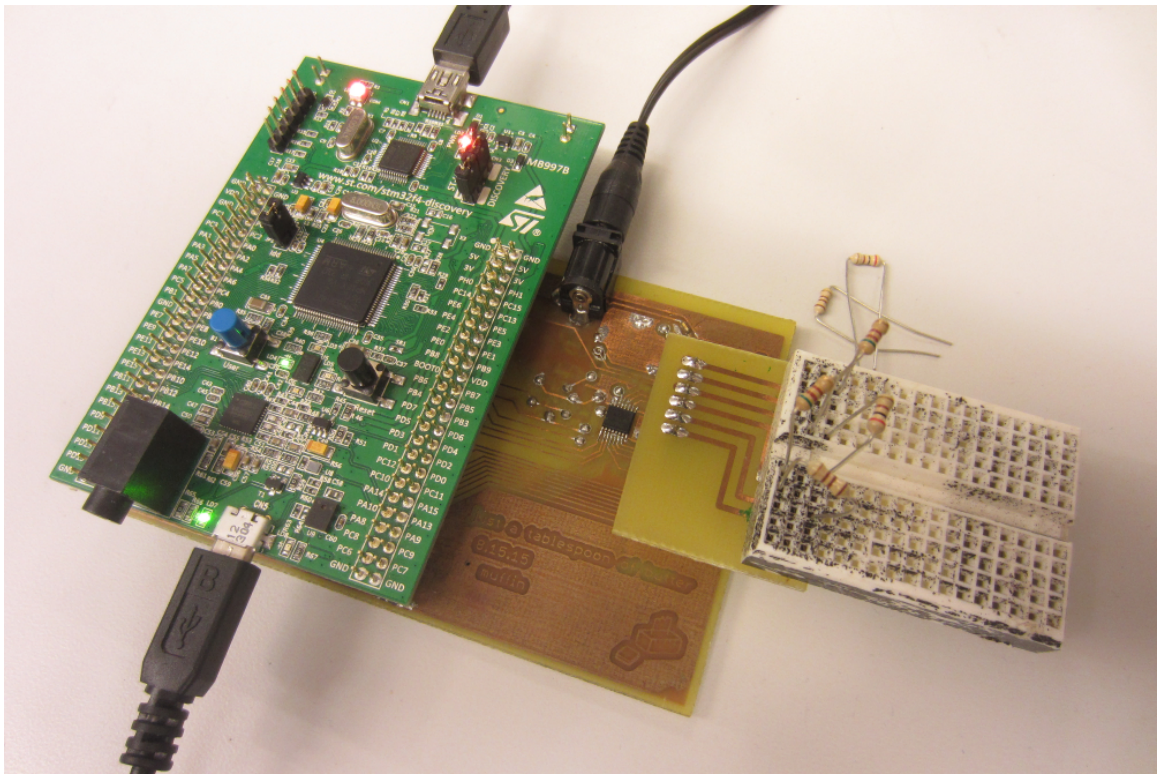


Figure 4-19: 8-node circuit sensing breadboard in operation

Chapter 5

Firmware

In this chapter, firmware configuration and operation is covered. The firmware is structured in three sections - USB data reception and transmission, DAC operation, and ADC operation. The USB subsystem receives commands from a host computer, specifying which nodes to ground, which nodes to connect to the voltage source, which nodes to probe, and what sample frequency to use. On a 'sample' command, the USB subsystem triggers both the DAC and ADC subsystems to operate synchronously, accumulating AC test samples in the memory buffer. When the buffer is full, the ADC subsystem sends the recorded data back to the host computer through the USB subsystem and waits for the next command.

5.1 USB

Libopencm3, licensed under the GNU LGPL v3, is used extensively in this firmware. Of particular utility, the USB subsystem relies entirely on example USB CDC ACM firmware included in the libopencm3 library. USB CDC (Communications Device Class) ACM (Abstract Control Module) is a USB protocol that is used to create virtual serial ports over USB. The firmware components that implement USB-CDC-ACM are a polling routine that is the only code running in the main loop and two callbacks - one for receiving data and one for transmitting data. The callback used to transmit data is called after the ADC subsystem has finished sampling and sends all of the collected data over USB. The callback used to receive data is called when the laptop sends commands over USB.

Command List

Commands are sent in sequences of characters over USBACM. The command list is as follows:

`rcvBuf=['g',nodes]` grounds all of the nodes represented by `nodes`. In this command, `nodes` is an 8-bit binary representation of which nodes to ground. For example, `['g',0x03]` grounds nodes 1 and 2.

`rcvBuf=['w',node]` connects the voltage source to `node` through a high-side switch. `node` is a decimal representation of which node to connect to the voltage source. For example, `['w',0x03]` connects the voltage source to node 3.

`rcvBuf=['m',node]` configures the voltage multiplexer to connect `node` to the A/D converter. `node` is a decimal representation of which node to connect. For example, `['m',0x03]` connects the A/D to node 3.

`rcvBuf=['s',f0,f1,f2]` configures the DAC to generate a cosine wave of frequency `(f2<<16) + (f1<<8) + f0` and configures the ADC to sample between 100 and 250 times a cycle when possible, with a maximum sample rate of 500Khz. For example, `['s',0xe8,0x03,0x00]` configures the DAC to generate a 1KHz cosine wave and configures the ADC to sample at 250KHz.

5.2 ADC

The ADCS7476 is a successive approximation ADC with an SPI interface. In operation, a master device initiates a conversion by pulling $\overline{\texttt{CS}}$ low, then clocks in data using the `CK` line and reading `SDATA`.

The ADC firmware configures two 32-bit timers and the DMA to record data from up to sixteen external ADCs. The two timers are responsible for driving the Chip Select and Clock lines on all of the ADCs. The timing of the Chip Select line determines the sampling frequency and the Clock line is driven at the ADCS7476's maximum clock frequency of 20MHz. On the rising edge of each clock cycle, the DMA is triggered to sample the data

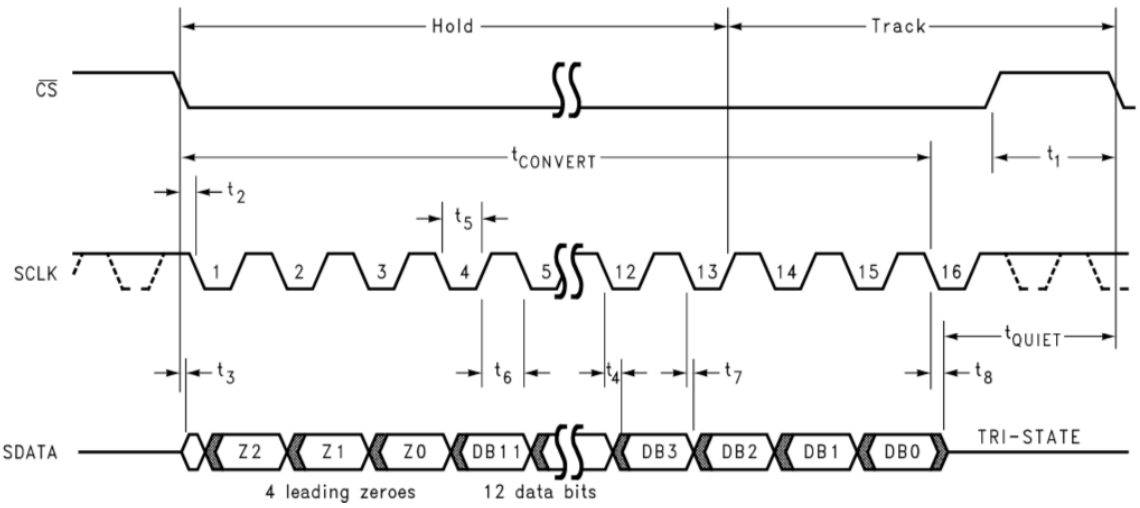


Figure 5-1: ADCS7476 Serial Interface Timing Diagram

lines of each ADC. A trigger causes the DMA to store each of the 16 pin states on `PORT E` as a 16-bit integer in the buffer `datas`.

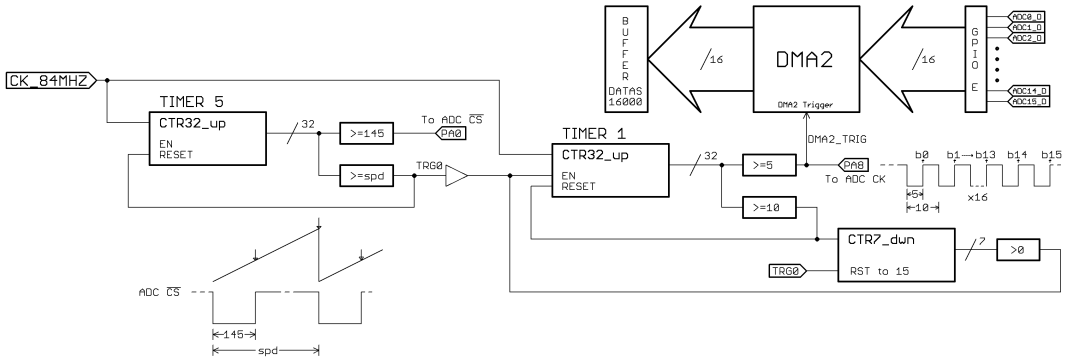


Figure 5-2: Full ADC Driver with DMA

5.2.1 ADC Timers

Timer 5 drives the Chip Select lines. It is configured as a 32-bit up-counting timer clocked at 84MHz, uses output compare unit 1 to control pin PA0 (connected to ADC $\overline{\texttt{CS}}$), and

is set to run in continuous mode. The ADC sampling frequency is controlled by the $\overline{\texttt{CS}}$ line, which is controlled by the period of Timer 5. When a host computer requests an ADC trigger, it sends a sampling frequency along with it. The appropriate $\overline{\texttt{CS}}$ period is calculated and assigned to the variable `spd` (sampling period). Timer 5's output compare mode 1 is set to PWM2 mode, where PA0 is low when the counter is less than the compare value and high when the counter is greater than the compare value. At update, Timer 5's count value is reset to 0, so PA0 falls to 0V and initiates a conversion. When Timer 5 counts to 145, $1.7\mu S$ later, the conversion is complete and PA0 rises to 3.3V, until the count reaches `spd` and the cycle starts again. Timer 5 is also configured in Master Mode and sends a trigger signal on TRG0 at each update event. In this case, the signal on TRG0 is used to enable Timer 1.

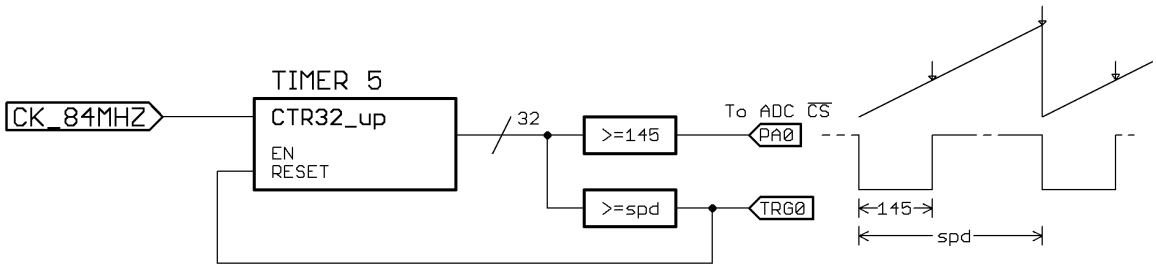


Figure 5-3: ADC Chip Select Timer

Timer 1 drives the ADC Clock lines and behaves as a $1\mu S$ on - $1\mu S$ off 16-shot timer. It's also configured as a 32-bit up-counting timer clocked at 84MHz, but it's only enabled by the TRG0 signal from Timer 5. Like Timer 5, it uses output compare unit 1 to control an output pin, in this case `PA8`. Unlike Timer 5, it's set to run in one-shot mode with a repetition counter that is preloaded with a value of 15. The output compare unit is set to run in PWM2 mode with fixed compare and update registers. When Timer 1 counts to 5, the compare unit fires and PA8 goes high. When the count reaches 10, the timer is updated, PA8 goes low, and the repetition counter decrements. The timer continues to run, toggling PA8 and decrementing the repetition counter, until the repetition counter reaches a value of 0. Once the repetition counter reaches 0, the next Timer 1 update disables the timer.

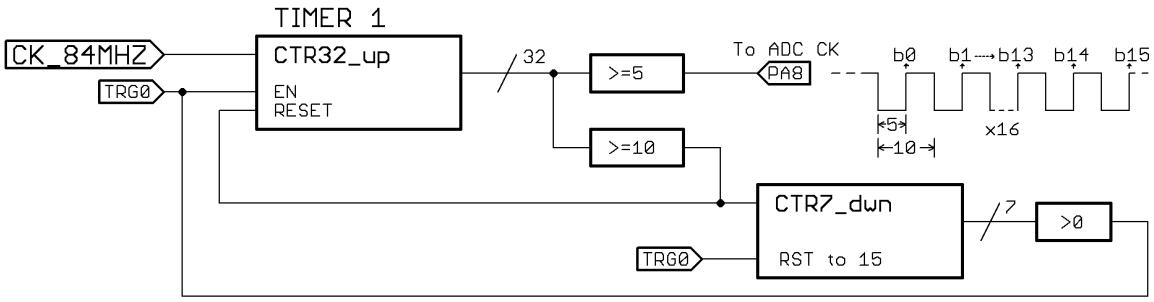


Figure 5-4: ADC Clock Timer

5.2.2 ADC DMA

A DMA controller is used to store serial ADC data from up to 16 ADCs in parallel. On each clock cycle, DMA2 takes the 16-bit integer value represented by the state of the 16 pins on `PORT E` and stores it in memory. As shown in Figure **??**, the ADC data pin is updated on a falling clock signal. To record the state of the data pin, DMA2 is triggered to start a data transfer on every compare event from Timer 1, which corresponds with the rising edge of the clock signal.

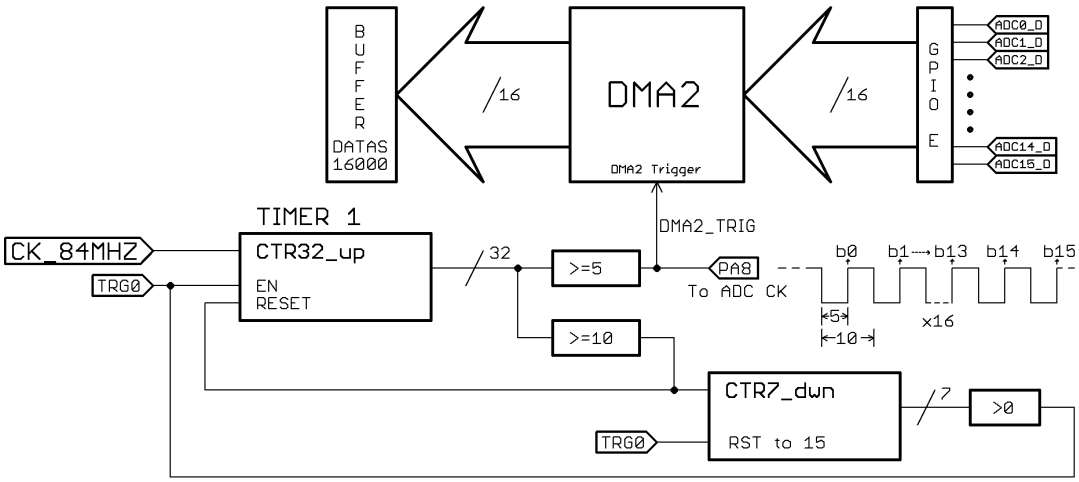


Figure 5-5: ADC Clock Timer Triggers DMA2

When Timer 5 is enabled, DMA2's 'number of data' [to transfer] register is set to 16000 and the ADCs continuously sample at the sample period defined by `spd`. In each cycle of

Timer 5, Timer 1 toggles the clock line 16 times, triggering DMA2 on each rising edge. With each DMA trigger, DMA2's number of data register decrements. When the register reaches 0, an interrupt fires that disables Timer 5 and begins the data reconstruction process.

5.2.3 Data Reconstruction

Since the serial data stream from each ADC is stored sequentially, the actual A/D readings need to be reconstructed after they are recorded. In Figure **??**, the binary data stored in `datas` is examined. Each 16-bit integer in `datas` contains one bit of data from each of the 16 pins on `PORT E`, which connect to the data pins on the external A/D converters. Here, each letter represents the data from a particular external ADC.

Bit Index	b_{15}	b_{14}	b_{13}	b_{12}	...	b_3	b_2	b_1	b_0
`datas[0]`	p_{15}	o_{15}	n_{15}	m_{15}	...	d_{15}	c_{15}	b_{15}	a_{15}
`datas[1]`	p_{14}	o_{14}	n_{14}	m_{14}	...	d_{14}	c_{14}	b_{14}	a_{14}
`datas[2]`	p_{13}	o_{13}	n_{13}	m_{13}	...	d_{13}	c_{13}	b_{13}	a_{13}
`datas[3]`	p_{12}	o_{12}	n_{12}	m_{12}	...	d_{12}	c_{12}	b_{12}	a_{12}
⋮	⋮	⋮	⋮	⋮	...	⋮	⋮	⋮	⋮
`datas[13]`	p_2	o_2	n_2	m_2	...	d_2	c_2	b_2	a_2
`datas[14]`	p_1	o_1	n_1	m_1	...	d_1	c_1	b_1	a_1
`datas[15]`	p_0	o_0	n_0	m_0	...	d_0	c_0	b_0	a_0
`datas[16]`	p_{15}	o_{15}	n_{15}	m_{15}	...	d_{15}	c_{15}	b_{15}	a_{15}

Table 5.1: `datas[0:16]`

To reconstruct the data collected from $\text{ADC}_a[0]$, a routine iterates through `datas[0:15]`, summing appropriately bit-shifted b_0's.

$\text{ADC}_a[0] = (a_{15} << 15) + (a_{14} << 14) + (a_{13} << 13) + ... + (a_2 << 2) + (a_1 << 1) + a_0$

To reconstruct all of the waveform recorded, the routine iterates through `datas[0:16000]`, reconstructing 1000 samples from each ADC.

After reconstruction, the data is sent to a host program over USB for further processing.

5.3 DAC

The STM32F407's onboard DAC is used as a signal source for impedance testing. The DAC is configured to update on a timer, and is fed new waveform values from a wavetable by DMA1.

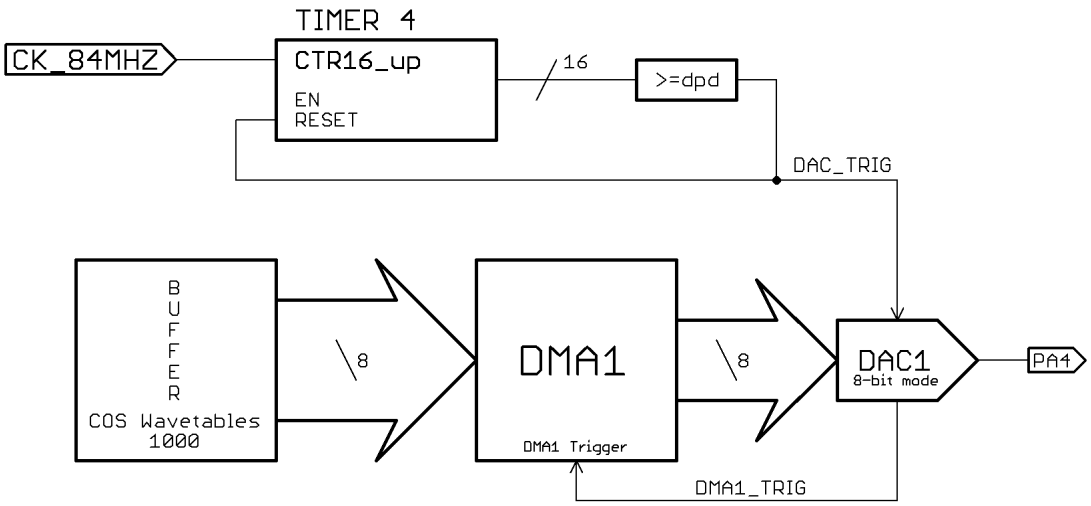


Figure 5-6: DAC Timer Triggers DMA1

Timer 4 is configured as a 16-bit up-counting timer with period `dpd`, set by command over USB. On update, Timer 4 is a master to DAC1, and triggers the DAC to output a new value. The DAC, in turn, triggers a data transfer from DMA1, which transfers the next value in the cosine wavetable to DAC1.

5.3.1 DAC Wavetables

There are three DAC wavetables, each 1000 bytes long. The first wavetable has four cycles of cosine, the second has 10, and the last has 40. The DAC synthesizes sinusoids by taking in a sequence of numbers from a wavetable and converting them, sequentially, to analog voltages. There are two factors that contribute to the output sinusoid's frequency: the rate at which the DAC takes in a new number and the number of samples in the wavetable it takes to complete one cycle. In the case of the first wavetable, where there are four cycles in 1000 samples, the DAC can synthesize a 1Hz signal by converting one number every 4mS, an 0.4% of the period time-step. Had the DAC been looking up the second wavetable, where there are ten cycles in 1000 samples, the DAC would have needed to wait 10mS between samples, a 1% of the period time-step. Using the third wavetable, the wait would be 40mS between samples, a 4% of the period time-step. This amount of delay in the DAC output creates an unacceptable amount of distortion in the output signal.

Wavetables with a higher number of samples per cycle can feed lower-frequency signals at higher sample rates. Additionally, wavetables with a high number of samples per cycle have smaller jumps between samples, resulting in a smoother waveform. According to these observations, a wavetable with an extremely large number of samples per cycle should be used to generate waveforms with the smallest amount of distortion.

However, the DAC's speed is limited by the internal system clock, and to generate sinusoids at 100KHz+, the DAC needs to be fed by a waveform with fewer samples per cycle.

Chapter 6

Software Control

In this chapter, software control of the hardware system is covered. The software that controls the hardware system is very similar to the software simulation in Chapter 3. A control routine parses first circuit 'experiment' requests and configures the hardware to follow out the experiment. The routine then analyzes the returned measurement data. Unlike in the simulation, the measurement data is returned in the time domain rather than the frequency domain. The software transforms the returned data to the frequency domain to calculate the magnitude of the returned waveforms. With magnitude data, the same network sensing algorithm from Chapter 2 is used to reconstruct the network.

6.1 Control Routine

"Just a teaspoon of sugar makes the medicine go down" - Mary Poppins

The control routine is quite self-explanatory.

```
def medicine(drive,chan,freq=1000,adc=0,uglist=[]):
    nog=0        #a number representing nodes not to ground
    for x in uglist:
        nog+=(1<<x)
    ser.write(['g',0xff&(~((1<<chan)|(1<<drive)|nog))])
                #ground all of the nodes whose bits are set to 1
    ser.write(['w',drive+1])
            #connect the voltage source to node 'drive'
    freq = int(freq)
    f=[freq&0xff,(freq&0xff00)>>8,(freq&0xff0000)>>16]
```

```
        #send three bytes representing the desired frequency
ser.write(['m',chan])
        #set the voltage multiplexer to the channel of interest, chan
ser.flushInput()
        #clear everything already in the input buffer
ser.write((['s']+f+[adc]))
        #tell the microcontroller to begin sampling
        # adc value of 0 is for voltage and 1 for current
a = ser.read(2000)
b = [5*((ord(a[2*x+1])<<8)+ord(a[2*x]))/1024. for x in range(len(a)/2)]
b = b[2:]
        # the ADC data is 10-bits per sample, but the serial channel only supports
        # sending 8-bits at a time, so it has to be reconstructed.
ft=fft(b)
aind=argmax(abs(ft[1:]))+1
        #calculate the magnitude of the most prominent sinusoid and return it
amp = abs(ft[aind])/500
return amp
```

6.1.1 FFT

The normalized FFT is used to extract the amplitude data out of the measured sinusoidal voltages and currents. Since the measured voltages and currents should be close to purely sinusoidal, the magnitude of the FFT of those voltage and currents should be close to a delta function at the frequency of interest. The magnitude peak value of the normalized FFT is the amplitude of the measured sinusoidal voltages and currents.

6.2 Network Sensing Algorithm

The implementation of the network sensing algorithm in the hardware control software is identical to that in the hardware simulator, mentioned in Chapter 3.