2010

Use of a Segmented Electrode To Control Current Distribution In Electrochemical Systems

Barry Van Tassell

CUNY City College

How does access to this work benefit you? Let us know!
Follow this and additional works at: http://academicworks.cuny.edu/cc_etds_theses

Part of the Chemical Engineering Commons

Recommended Citation
Van Tassell, Barry, "Use of a Segmented Electrode To Control Current Distribution In Electrochemical Systems" (2010). CUNY Academic Works.
http://academicworks.cuny.edu/cc_etds_theses/7

This Thesis is brought to you for free and open access by the City College of New York at CUNY Academic Works. It has been accepted for inclusion in Master's Theses by an authorized administrator of CUNY Academic Works. For more information, please contact AcademicWorks@cuny.edu.
Use of a Segmented Electrode To Control Current Distribution In Electrochemical Systems

Thesis
Submitted in partial fulfillment
for the requirement for the degree
Master of Engineering (Chemical)
at
The City College of New York
of the
City University of New York
by
Barry Van Tassell
January 2011
Approved:

[Signature]
Professor Daniel Steingart, Thesis Advisor

[Signature]
Professor Alexander Couzis, Chairman
Department of Chemical Engineering
# TABLE OF CONTENTS

Abstract................................................................................................................................2
List of Figures......................................................................................................................3
Acknowledgements..............................................................................................................4
Introduction..........................................................................................................................5
Motivation............................................................................................................................5
Wagner Number Theory.......................................................................................................7

Part I: The Ardustat..............................................................................................................8
  Hardware Design.....................................................................................................8
  Software Design.....................................................................................................13
  PID Theory.............................................................................................................17
  PID Experimental Procedure.................................................................................19
  Results....................................................................................................................20
  Discussion..............................................................................................................22

Part II: The Multistat..........................................................................................................23
  Theory....................................................................................................................23
  Experimental Setup................................................................................................24
  Results and Discussion..........................................................................................26

Conclusions and Future Work............................................................................................30

Appendix A: Ardustat Firmware........................................................................................32
Appendix B: Interface........................................................................................................47
Appendix C: Multistat Firmware.......................................................................................57
References Cited................................................................................................................70
Abstract

The goal of this project was to investigate the use of segmented electrodes to control the current distribution through an electrochemical cell. To achieve this goal, the project was separated into two parts, designing a scalable piece of hardware that could create a current distribution, and then using that hardware to create noticeable changes in a system by changing the current distribution within the system.

Prior to building the current distribution device, a basic 3-electrode galvanostat was built in order to test its fundamental design characteristics. The design stressed a simplistic hardware, relying on software to control the current. A series of trials were performed to test an implementation of proportional-integral-derivative (PID) control in the software, with the aim of improving response to system changes. In some cases, a noticeable improvement in response time was seen using PID control. However, more in depth testing will be required to implement PID reliably.

The device used with the segmented electrode was based on the design of this simpler galvanostat. Once built, it was tested in a system of two copper electrodes submerged in a ZnO/KOH solution. Results showed a verifiable ability to control the distribution of current along the segmented electrode, as well as an ability to use this control to affect dendrite growth on both the segmented electrode and its opposing electrode.
List of Figures

Figure 1: Ardustat Operational Diagram.................................................................8
Figure 2: ADC Deviation from Correlation............................................................10
Figure 3: Measured Resistance vs. Potentiometer State........................................11
Figure 4: Measured Resistance Deviation from Correlation.................................11
Figure 5: Operational Diagram of the Ardustat in a Voltage Divider Configuration....12
Figure 6: Demonstration of the Ardustat's Ability to Achieve Negative Currents.........13
Figure 7: Potential-time curves in DI water for select $K_p$ values.........................20
Figure 8: Current-time curves in ZnO/KOH for select $K_p$ values.........................21
Figure 9: Current-time curves in NaCl solution for select $K_p$ values....................22
Figure 10: Current Distribution Across Electrode....................................................27
Figure 11: Segmented Electrode with $I=1.8\text{mA}, L=3.6\text{mm}$..........................28
Figure 12: Segmented Electrode with $I=1.8\text{mA}, L=1.0\text{mm}$..........................29
Acknowledgements

The first person deserving recognition is my adviser and mentor, Professor Dan Steingart. His actions and words have shown me that I am able to succeed, and that I should enjoy myself while doing so. Additionally, his earlier work on the ardustat laid the foundation for most of the design methodologies used in this project, and none of the work on this project would be possible without his direct involvement. I would also like to extend my gratitude to my father, James Van Tassell, and Dr. Melody Festa. Their constant support, time investment, and insistence that I keep going strong despite all distractions and discouragements were paramount in my journey to this point. Lastly, I need to recognize all of the friends and family members who have helped me. Contributions as selfless as financial support, or as unexpected as offering an ear to ramble to, have been vital to my progress and sanity. Thank you, everyone.
Introduction

The distribution of current across an electrode can be controlled, and can be used to affect metal growth within an electrochemical cell. Specifically, an electrode can be segmented, with each segment having an independently controlled current. Direct control of the current distribution along an electrode can be used either to directly control growth at that electrode, or to indirectly influence growth on the opposite electrode. The latter can be described through the use of the Wagner number (Wa)\(^1\). In this project, two custom potentiostatic and galvanostatic devices were constructed to test the viability of this approach in real battery chemistries.

Motivation

The primary motivation for building a device to create a potential or current distribution comes from applications in existing battery chemistries, specifically lithium-based systems. Batteries based on these chemistries are desirable for their high energy densities\(^2\), but are presently limited to primary-cell use; meaning that they are discarded after one discharge. This is because during recharge, lithium grows on the cathode unevenly, forming dendritic structures; tree-like growths protruding from the electrode\(^2\). These dendrites become the primary location of electron flow and grow through the cell quickly, piercing any membranes within the battery and eventually reaching the other electrode. When this occurs, the cell will generate excess heat and gas, eventually rupturing the container and causing a sizable explosion. In some cases, the results of this reaction have been compared to an incendiary bomb\(^3\). The ability to charge these batteries safely relies on the prevention or limitation of dendrite growth.
A conventional assumption in electrochemistry is that current distribution along an electrode is constant\(^5\). However, uneven growth during reduction at an electrode is a clear example where this assumption is invalid. Since each molecule that binds to an electrode requires electrons to complete the reduction reaction (one, in lithium's case), a dendritic growth implies high current at the point of origination, and relatively low current elsewhere on the electrode, i.e. an uneven current distribution. A circuit that could force current to be kept at a certain level at points across the electrode could control dendrite growth.

Much of the work completed in this project could be facilitated, and done with a higher accuracy, through the use of commercial potentiostatic/galvanostatic devices that are readily available. Building a customizable galvanostat has several benefits. The first is scalability. Using high-accuracy commercial devices may work for low-resolution tests, but high resolution will eventually be required to control growth and current distribution at varying length scales. As an example, creating a distribution with only 2 or 4 points of control could easily be done with a commercial device, but creating a distribution with 32, 64, 128 or more points would require many large and expensive pieces of equipment, thus eliminating the possibility of this approach being employed anywhere except in excessively funded research labs. Avoiding the commercial approach at the beginning will allow scaling to a high resolution while keeping cost and physical size small.

A second reason to avoid commercial devices lies in their operational limitations. As is the case with most commercially built, single purpose equipment today, using them
in ways beyond the scope of the designer’s original intent can be difficult. Building this work on open, easily changeable hardware and custom software avoids possible limitations down the road in terms of improved automation or new functionality.

**Wagner Number Theory**

The Wagner number is a dimensionless quantity used to describe effects on growth along an electrode. Specifically, it denotes the “throwing power” of the system; the ability of a solution to plate evenly on a cathode\(^4\). It's typical form is

\[
Wa = \frac{\kappa}{l} \frac{d \eta}{dj},
\]

where \(\kappa\) denotes conductivity of the solution, \(d\eta/dj\) denotes the slope of the overpotential vs current density curve at the current density of the system, and \(l\) is a characteristic length of the system.

Since conductivity is the inverse of resistivity, which can be calculated as the product of cell resistance, \(R\), and distance between electrodes, \(L\), the equation for \(Wa\) can be defined as

\[
Wa = \kappa \frac{1}{l} \frac{d \eta}{dj} = \frac{1}{RL} \frac{1}{l} \frac{d \eta}{dj},
\]

which allows for a direct calculation of \(Wa\) from measured quantities in the cell, as long as the slope of the overpotential vs current density is determined for the system beforehand.

The literature indicates that for growth to be affected, \(Wa\) must be above approximately \(10^4\). This would imply a high current density or a small distance between the electrodes.
part i: the ardustat

hardware design

the design of a potentiostat/galvanostat is relatively straightforward. for this project, a 2-electrode device was built without a segmented electrode to test the underlying design and facilitate development of the various control algorithms. this particular device will be referred to as an ardustat.

the ardustat is able to control potential and current by manipulating the following main components: digital-to-analog converters (dac), analog-to-digital converters (adc), and a digital potentiometer. a survey of the literature turned up no published reports of potentiostats or galvanostats being built in this fashion. below is the basic operational diagram:

figure 1: ardustat operational diagram

the dac is the source of potential and current, taking a numerical value as input, and converting that number into a potential. the current from the dac passes through the potentiometer, which acts as a variable resistor. from there, current passes into one electrode of the test cell, while the other electrode is connected to ground.

the adc closest to the dac is used to verify the dac’s potential. the two adcs on either side of the potentiometer are used to measure the potential of the cell
during discharge and the current during both charge and discharge. This follows from a trivial adjustment of Ohm’s Law:

\[ V = IR \]
\[ (V_2 - V_1) = IR \]
\[ I = \frac{V_2 - V_1}{R} \]

From this, the current across the potentiometer, and thus the current going through the cell, can be determined.

The ardustat is built on an open hardware platform called the Arduino. Arduinos come equipped with 6 ADCs, each with 10-bit resolution (or 1024 possible states) capable of reading between 0 and 5 volts, digital I/O pins, and the ability to run custom firmware.

The DAC used in the ardustat has a voltage range and resolution identical to the arduino’s ADCs. The potentiometer is capable of resistances between 100 and 10,000 ohms with 8-bit resolution (256 states).

A common, but sometimes imprecise assumption is that the ADCs’ reported states correlate exactly with the advertised voltage range (0-5V). Namely,

\[ V = \frac{\text{state}}{1023} \cdot 5. \]

However, if the ADCs deviate from this even slightly, the reported voltage and current will be off by an amount that is non-negligible, especially when working at low currents.

To improve accuracy and correct for these deviations, a reference voltage chip
was added to the ardustat, which outputs a constant 2.500V. This was attached to an additional ADC, and the new correlation became:

$$V = \frac{\text{state}}{\text{state}_{\text{ref}}} \times 2.500 .$$

To determine the effectiveness of this method, direct measurements were taken. The figure below represents the difference between the ADC’s reported potential and the true potential.

![Figure 2: ADC Deviation from Correlation](image)

The data for these tests were done with the same hardware. The $R^2$ value for the reference correlation was 0.99993, and was 0.99942 for the original correlation. Though small, the difference is important for two reasons. First, when working in a very sensitive current range, a reported voltage that is off by 50 or 60 mV (as with the unreferenced correlation) could cause the difference between the true, desired, and reported currents to be significant. Second, each ADC is unique and may read potentials in different ways. Using a reference voltage will create a deviation from the correlation that is regular and more predictable.
The potentiometer did not fit a linear correlation as neatly as the ADCs. Figure 3 below shows the steps of the potentiometer matched to a linear correlation based on its minimum and maximum values.

![Figure 3: Measured Resistance vs. Potentiometer State.](image)

Alternatively, Figure 4 shows the same data as a deviation from the correlation.

![Figure 4: Measured Resistance Deviation from Correlation](image)

The deviations are irregular enough to warrant another approach, measuring the resistance of each potentiometer step directly.

This was accomplished in automation by connecting a resistor of known
resistance in place of a cell in the ardustat’s circuit, creating a voltage divider. Then, the DAC was set to a maximum, constant value and the potentiometer was stepped through all 256 states several times. The resulting ADC values from each of these steps was used in combination with the voltage divider equation (a form of Ohm’s Law) and the fixed resistor’s value to determine the resistance of each of the 256 states:

\[ V_2 = IR_2 \quad V_1 = I \left( R_1 + R_2 \right) \]

\[ V_2 = \frac{V_1}{R_1 + R_2} \cdot R_2 \]

\[ R_1 + R_2 = \frac{V_1}{V_2} \cdot R_2 \]

\[ R_1 = R_2 \left( \frac{V_1}{V_2} + 1 \right) \]

Additionally, because \( V_1 \) and \( V_2 \) only show up in this equation together as a proportion, the reported states can be used directly instead of converted voltage values.

One final piece of hardware design in the ardustat allows current to be reversed without having components that are capable of creating a negative current. This can be useful when precise control over a cell’s discharge is desired beyond the limits of the cell
itself. This design, and the resulting use technique, will be referred to as double-DAC. The design replaces the connection to ground with a connection to a second DAC. Since the DACs are able to sink a low current, this second DAC can be used to mimic ground at a higher potential.

![Figure 6: Demonstration of the Ardustat's Ability to Achieve Negative Currents](image)

Figures 6 demonstrates how a resistor was used in place of an electrochemical cell to display the ardustat’s ability to create both positive and negative currents, using double-DAC, without the need for more complicated hardware.

**Software Design**

The ardustat was controlled using two separate pieces of software: the program loaded onto the Arduino (the firmware), and the program running on the attached computer (the interface).

The firmware was written in a subset of the C programming language, and serves primarily as a feedback loop; reading ADCs and making adjustments to the potentiometer and DACs based on commands given by the interface. The full source code can be found in Appendix A.

When the interface sends a command to the ardustat, it is given as a 5-character
string, one letter followed by four numbers. The first character tells the ardustat the mode (potentiostatic, galvanostatic, etc...) and the remaining numbers contain a value that functions as the setting for that mode. Depending on which mode was set, the firmware will enter a loop consisting of reading the ADCs, determining what change, if any, must be made to the DACs and potentiometer to reach the desired setting, and adjusting those components accordingly.

For example, a command of “p0512” tells the ardustat to maintain a potential of 512. The number given refers not to the voltage but to the associated ADC state, which based on conversions discussed earlier is approximately 2.46V. If the ADC between the potentiometer and the cell shows a reading greater than 512, the firmware would adjust the DAC down the correct amount to reach that value.

In potentiostatic mode, if the firmware calculates that the DAC must be set above its maximum value to maintain the set potential, it decreases the potentiometer's resistance in order to decrease the voltage setting while maintaining current. This comes as a result of Ohm's Law:

\[
(V_1 - V_2) = IR
\]

If, for example, the desired cell voltage is 4V, R is set to 1000 ohms, and the current associated with that potential in the cell is 2mA, That would mean \( V_1 \) would need to be 6V, beyond the range of the DAC. If the resistance were decreased to 250 ohms, the DAC would only need to be set to 4.5V to achieve the same cell voltage.
Galvanostatic mode works on the same theory, but is set up differently. A potentiometer state is set beforehand, with a command such as “r0045,” where “r” does not signify a mode, but rather that the potentiometer should be set to a state of 45. Following this, another command is sent, such as “g0300,” which puts the device into galvanostatic mode. Here, the value is not a ADC state, but a difference of ADC states. Specifically it is $\Delta V$, $(V_1 - V_2)$. With this information, the firmware merely has to adjust the DAC setting to achieve this voltage difference across the potentiometer to maintain a constant current.

The firmware also returns information back to the user. Data like the current setting, ADC readings, potentiometer state, and DAC states are all reported back in the form of a single line of comma-delimited values. Each time the control loop cycles, this data is sent, assuring that the user is returned as much information about the ardustat and the cell as possible.

The firmware's commands and data are all transmitted across a serial USB connection between the Arduino and an attached computer. Software already exists (such as the Arduino's own IDE) that allows raw commands to be sent and received with the Arduino. Controlling an ardustat in this manner would yield a constant, dense stream of data about ADC and potentiometer states, as well as require hand calculations to convert real-world desired values into state information that the ardustat can understand. This is possible, but difficult and time consuming, especially when more complicated, dynamic functionality is desired. A separate interface was written to simplify communication and data manipulation.
The interface was written in the Python programming language, which allows for fast prototyping and easily understood code. The interface sends, receives, parses, and reports communications between the user and the ardustat, while also allowing a high degree of finely-tuned, controllable automation.

Most of the interface's functionality is straightforward. All data sent by the ardustat is automatically saved to file, both in its raw form and in a parsed form which includes converted voltage values, resistance information (from the calibration data, described earlier), and current. These values are also made available through Python directly, so user-written scripts can read them and make decisions based on these data. The interface also takes raw voltage and current values from the user, converts them into states, and sends the ardustat the appropriate command.

A more complex function of the interface is to determine the appropriate voltage and resistance values for a given current in galvanostatic mode. The ardustat must take more into account than the relationship between resistance, voltage, and current. Notably, the resistance chosen has an effect on the achievable accuracy. This concept is best demonstrated through example:

If desired current is 3mA, and at the time in question, the potential required to achieve that current is roughly 2V, the interface has a wide assortment of possible resistance values. Again, this process is dictated by Ohm's law:

\[ \Delta V = (V_1 - V_2) = IR \]
\[ V_1 = IR - V_2 \]
\[ I = \frac{(V_1 - V_2)}{R} \]

The \( \Delta V \) range for this situation is about [-2V,3V], correlating to a DAC range of
If a resistance of 100Ω is chosen, the current range is [-20mA,30mA]. Since the DAC has 1023 states, each state change will cause a jump of 48μA. Repeating the calculations, a resistance of 1000ohms would give a range of [-2mA,3mA] and 4.8μA/state. Ideally, the μA/state value will be low, so that any fluctuations in the voltage will cause the smallest possible fluctuations in the current.

Additionally, since the relationships defined are linear, we can say that any change in the required cell voltage will translate the current range up or down, while changing the resistance will expand or contract the range, as well as change the precision.

The purpose of the interface, then, is to choose a resistance with a range wide enough so that fluctuations in the system will not cause the target current to fall out of range, while choosing one high enough to ensure the highest precision possible. The method used for this project was a simple correlation between resistance and current. In the future, system-specific correlations might be required for maximum accuracy.

PID Theory

Initially, the algorithm used in both galvanostatic and potentiostatic modes to reach the required potential was simple. If the ADC reading deviated from the desired value, move the DAC up or down proportionally. This often works well, but in some cases this procedure can cause a long delay before the desired value is achieved, and sometimes can cause oscillations that continue indefinitely.

One way to alleviate such a problem is through PID control theory. PID is a control method with terms that account for a system's previous state, which can help dampen oscillations and bring the system to the desired state faster. If “error” is defined
as the difference between the desired state and the present state, then to achieve the desired state, PID theory states that the controlled variable must be moved by a value equal to:

\[
\frac{K_p \text{error} + K_i \int_0^t \text{error}(t) + K_d \frac{\text{d(error)}}{\text{dt}}}{1}
\]

where \(K_p\), \(K_i\), and \(K_d\) are constants determined for the particular system. In the ardustat's firmware, a form of PID was implemented in both potentiostatic and galvanostatic modes for DAC adjustments. It should be noted that the Integral term of the specific implementation employed here differs slightly from the above ideal equation in that the error history does not go back to time 0, it instead only takes into account the previous 10 data points. Additionally, the Integral term is estimated using the Trapezoidal Rule:

\[
\int_a^b f(x) \, dx \approx (b-a) \frac{f(a) + f(b)}{2}
\]

Optimizing the PID system at this point would be a very time consuming process; choosing values for \(K_p\), \(K_i\), and \(K_d\), testing, checking the results, and changing one of the 3 K-values again, hoping to see improvement. Some work has been done on facilitating this process. In a paper published in 1942, J. Ziegler and N. Nichols present what is considered the Ziegler-Nichols method for tuning a PID controller. The basic procedure is to set \(K_i\) and \(K_d\) to 0, and increase \(K_p\) until the system oscillates with a constant amplitude. The \(K_p\) value at this point, \(K_c\), is then used in conjunction with the oscillation's frequency, \(T_c\), to determine near-optimal values for \(K_p\), \(K_i\), and \(K_d\), using one of several correlations.

As a prelude to testing a multi-channel ardustat, the Ziegler-Nichols method was
applied to the PID implementation in the ardustat, in both potentiostatic and galvanostatic modes, in an attempt to improve the performance of the hardware, using the following correlations:

1. \( K_p = 0.6K_c \)  \( K_i = \frac{2K_p}{T_c} \)  \( K_d = K_p \frac{T_c}{8} \)
2. \( K_p = 0.2K_c \)  \( K_i = \frac{2K_p}{T_c} \)  \( K_d = K_p \frac{T_c}{3} \)

**PID Procedure**

The method used to obtain data was identical for all PID tests:

1. Prepare a small beaker containing the solution (DI water, ZnO/KOH) required for the test. Place two strips of copper sheet into the solution, making sure they are on opposite sides of the beaker.
2. Connect one of the computer-connected ardustat's leads to each of the copper strips.
3. In the firmware source, set \( K_p \) to the desired test value. Compile and upload this to the ardustat.
4. Use the Python interface to set the ardustat to the desired potential/current.
5. After a period of time allowing the system to come to equilibrium (usually 12 seconds) open the circuit and terminate the test.
6. Plot the resultant potential/current as a function of time. If it does not oscillate, increase \( K_p \) and repeat steps 3-6, taking care to replace the copper electrodes if they are tarnished or plated. If the potential/current does oscillate, note its oscillation frequency, \( T_c \).
7. Apply the Ziegler-Nichols method with \( T_c \) and the \( K_p \) at which oscillation
occurred to determine new values for $K_p$, $K_i$, and $K_d$.

**PID Results**

PID control was applied to the potentiostatic case first, using DI water as the system. Figure 7 below displays the potential-time curves for several notable values of $K_p$:

![Figure 7: Potential-time curves in DI water for select $K_p$ values](image)

Oscillations in this system, at 2V, occur precisely at $K_p=2.7$.

Figure 8 shows the results of galvanostatic control in a system of 0.6M ZnO and 2M KOH.

20
In this case, constant-amplitude oscillation occurs with $K_p=2.11$. Figure 9 shows the results from galvanostatic operation in a 0.5M NaCl solution.
Figure 9: Current-time curves in NaCl solution for select K_p values

**PID Discussion**

In the potentiostatic case, the data does not show any substantial improvement through PID operation compared to the standard proportional control. It is important to note from these experiments that the value of K_p, whether true PID control is used or not, is significant in determining how quickly the target potential is reached. When control is implemented in software as it is in this project, the relationship between K_p and the time required to reach the targeted value will always be a concern and should be adjusted ideally for each system and set of parameters used.

In the galvanostatic case, there are several points that should be noted. First, there is a noticeable difference in the oscillation value of K_p, referred to as K_c, between the different systems. This leads to the conclusion that a single one-size-fits-all set of K values cannot be used. Instead, each system being tested must be individually calibrated.
The behavior of the PID control used in the 0.5M NaCl system is also noteworthy. At first glance it converges about as fast as using a low $K_p$ value alone, but upon closer examination, there is a dip in current that occurs 3-5 seconds after convergence is reached. The dip is a result of the kinetics of the system, and indicates when a reaction begins and the potential changes, requiring the ardustat to adjust itself to account for it. With only the proportional term, the dip is noticeable and the ardustat takes a few seconds to recover. Using full PID, the dip is hardly noticeable. This result indicates that the usefulness of PID extends beyond a way to quickly arrive at a set value. PID control can automatically adjust to system changes to maintain pre-set system conditions.

**Part II: The Multistat**

**Theory**

The successor to the ardustat, the multistat, is a device built to test the feasibility of controlling current and potential distributions across an electrode. Like the ardustat, it is built using an Arduino, and has an external multi-channel DAC and potentiometer. However, both the DAC and potentiometer are different models than the ones used on the ardustat, primarily because the ardustat only required a single channel for current to flow. The multistat does not have a relay as the ardustat does, because the DAC used on the multistat has the ability to set individual DAC channels to a high-impedance state, accomplishing the same goal of inhibiting current flow.

The control algorithms used in the multistat's firmware are the same as in the ardustat's; in fact most of the multistat's firmware is taken directly from the ardustat's. The major differences are new methods for controlling the new DAC and potentiometer,
and organizing potentiostatic and galvanostatic functions to control multiple, individual channels instead of just one.

Overall, the multistat can be viewed as an ardustat that has one connection for a single electrode, and one connection allowing for an electrode segmented into up to 6 pieces. Normal operation is set up so that the segmented electrode will have a higher potential than the single electrode, with the single electrode attached to ground. However, Double-DAC is also implemented on the multistat, so these roles can be easily reversed.

Experimental Setup

Two primary tests were performed, using the segmented electrode as either the working electrode (WE) or the counter electrode (CE). Additionally, a test was done to determine the relationship between overpotential and current density in the system for use in the Wagner number calculations. This third test, requiring a 3-electrode system, was performed using an ardustat.

All tests were performed with an identical experimental design, and so their setups are comparable. In all cases, the electrolyte was 2M KOH and 0.6M ZnO, with the segmented and single electrodes made from copper wire and copper sheet, respectively.

In an effort to increase current density, the single electrode was masked beforehand so that only a small, measurable area would be in contact with the electrolyte. A 1cm wide, 0.8mm thick strip of copper sheet was sandwiched between two thin pieces of acrylic, with excess copper protruding from both ends. It was attached
using 2-part epoxy such that the epoxy filled all space between the acrylic pieces. Once set, excess epoxy around the edges of the acrylic was scraped away, and the strip of copper was cut up against the acrylic at one end, such that only a 2-dimensional surface was uncovered. For visibility, the acrylic was also sanded down to a wedge near the cut side of the electrode.

The segmented electrode was built from 6 pieces of 20-gauge insulated copper wire. Each piece was stripped of insulation at both ends, and were attached together side-by-side. At one end, the stripped ends are arranged so they were equidistant from their neighbors. They were then secured in place by a piece of plastic with similarly spaced holes.

The cell itself was also constructed from acrylic. Several designs were attempted but the one eventually settled upon was simply a block of acrylic with a deep, wide channel etched into it. The channel being wide enough to accommodate the single masked electrode, long enough to allow for 1cm to 2cm of additional space for the segmented electrode, and sloped to reduce the amount of electrolyte required to fill.

The preparation for each of the tests was done in essentially the same way:

1. Place the single masked electrode in the channel, pointing downwards, and secure it with electrical tape.

2. Place the segmented electrode opposite the single electrode, in parallel. The distance between the electrodes is a variable, but should be approximately 1-3mm. Secure the electrode with electrical tape.

3. Using a pipet, add electrolyte to the cell until both electrodes are sufficiently
covered. The volume required for coverage is less than 1mL.

4. Connect the electrodes.

1. For the overpotential test, attach the positive (red) wire from the ardustat to one of the center segments of the segmented electrode, the ground (black) wire to the single electrode, and the reference wire (labeled R) to a second thin piece of copper sheet, placing it in the electrolyte, keeping distance from both of the other electrodes.

2. For all other experiments, connect the multistat.

   1. For the segmented electrode, clip each wire labeled 1 through 6 to one copper wire each, being careful to maintain the wiring order.

   2. If the segmented electrode is to be used as the WE, clip the double-DAC (red) wire to the single electrode.

**Results and Discussion**

From this setup, the overpotential-current test was run by setting the ardustat to hold a current below the desired current for the other experiments (which in this case is 1.8mA), waiting for the potential required to hold that current to reach a constant state, and record the value. This was then done for a current above the target current. Because the ardustat's interface automatically reduces the measured voltage by the reference voltage, the difference in overpotential is being reported. These two data points are enough to find the change in overpotential per unit current, and since the area of the electrode surface is known, the current can be converted to current density:

\[
\frac{V_2 - V_1}{I_2 - I_1} \frac{1}{area} = \frac{\partial \eta}{\partial j}
\]
For a current of 1.8 mA, this method yielded a density of 145.8 $V/A-mm$.

The first true test of the multistat concept is to display its ability to directly control a current distribution; having different currents through different segments of the electrode. The most direct way to do this is to use the double-DAC functionality and use the segmented electrode as the WE, plating the different segments individually.

Figure 10: Current Distribution Across Electrode

Figure 10 is a distribution of current across the segmented electrode from one such test. The current through each segment is controlled individually, and the amount of metal plated on each segment is directly proportional to the current passing through it.

By far the most interesting result comes from the tests done using the segmented electrode as the CE, shown in Figure 11:
From the highlighted portion of Figure 11 it is possible to see a degree of uniformity across the electrode, as opposed to a directional change. For this test, current was passed
through a single segment denoted by the red dot in the lower right-hand corner. The growth on the right corner of the electrode is not a protrusion, rather it is growth along the acrylic masking used. This occurs on both sides of the electrode equally, and not caused by the location of the CE.

The distance between the electrodes is 3.6mm. The current passed is 1.8mA. The resistance through the cell was calculated as $1320\, \Omega$. Using this data and a characteristic length of 0.9mm (the width of a segment), we find the Wagner number of this test to be $Wa=0.034$.

*Figure 12: Segmented Electrode with $I=1.8mA$, $L=1.0mm$. Bottom image colored to highlight growth*
Figure 12 shows the result of a test done at the same current, with a distance between electrodes of 1mm. Across from each of the active electrodes, non-uniform zinc growth is clearly visible. Resistance through the cell was about $1130\Omega$, so $Wa=0.14$.

According to the value of the Wagner numbers, there should be no noticeable difference in growth between the two setups, yet clearly growth is affected in the second, but not the first. This implies that either the methodology used to calculate the Wagner numbers is not sufficiently accurate for this type of system, or that the technique used to measure the overpotential explicitly was not sound. In either case, growth was affected using a segmented electrode, which was the main goal this project was performed to complete. Further understanding of why the theory disagrees with the physical results is currently beyond the scope of this project.

**Conclusions and Future Work**

The primary goal of this project was to prove the plausibility of an adoptable and functional device for controlling current distribution, using inexpensive, easily modifiable hardware and software. The above results clearly demonstrate the feasibility of this approach. Additionally, despite the difficulties in describing the phenomenon mathematically, the results demonstrate that a device that can control current distribution can also influence the growth elsewhere in the cell.

Though much has been accomplished, there are still many places to improve, and questions to be answered. I would like to see work continue both on the multistat and ardustat designs. If the multistat were rebuilt with a DAC capable of providing higher currents, growth could be controlled while maintaining a larger distance between
electrodes. The multistat's design caters to scalability, and thus a multistat with the ability to control a higher number of segments simultaneously should be possible, and would be the next logical step once the present iteration is perfected.
Appendix A: Ardustat Firmware

#define DATAOUT 11 //MOSI
#define DATAIN 12 //MISO - not used, but part of builtin SPI
#define SPICLOCK 13 //sck
#define SLAVESELECTD 10 //ss
#define SLAVESELECTP 7 //ss

int adc; //out of pot
int dac; //out of main dac
int adcgnd; //adc at ground
int adcref; //ref electrode
int refvolt; //ref voltage 2.5V
int firstdac = 0;
int seconddac = 0;
int dacaddr = 0;
int dacmode=3;
boolean dactest = false;
boolean rtest = false;
int testcounter = 0;
int testlimit = 0;
int outvolt=1023;
byte pot = 0;
int temp;
byte resistance1=0;
int res=0;
int fixedres=0;
int cl=2;
int pdl = 4;
int counter = 0;
int sign = 1;
int waiter = 0;
int mode = 1;
int pMode = 0; //saved variable to remember if the last mode was pstat or not
int lastData[10]; //previous error values for use in pstat's PID algorithm
int dacrun;
int adcrun;
int resmove;

//Serial Comm Stuff
int incomingByte;
boolean setVoltage;
char serInString[100];
char sendString[99];
char holdString[5];
//int adcArray[100];
//int adcArrayCounter = 0;
int output;
boolean whocares = false;
boolean positive = false;
boolean gstat = false;
boolean pstat = false;
boolean ocv = true;
boolean cv = false;
int setting = 0;
int speed = 100;
int countto = 0;
byte clr;

void setup()
{

    // Startup Serial
    Serial.begin(57600);
    Serial.println("Hi Dan!");

    // SPI
    byte i;
    // byte clr;
    pinMode(DATAOUT, OUTPUT);
    pinMode(3, OUTPUT);
    pinMode(DATAIN, INPUT);
    pinMode(SPICLOCK, OUTPUT);
    pinMode(SLAVESELECTD, OUTPUT);
    pinMode(5, OUTPUT);
    pinMode(6, OUTPUT);
    pinMode(SLAVESELECTP, OUTPUT);
    pinMode(cl, OUTPUT);
    digitalWrite(SLAVESELECTD, HIGH); // disable device
    digitalWrite(SLAVESELECTP, HIGH); // disable device
    digitalWrite(cl, LOW);
    delay(1000);
    digitalWrite(cl, HIGH);
    // SPCR is 01010000. write_pot turns off the SPI interface,
    // which means SPCR becomes 00010000 temporarily
    // SPCR = (1<<SPE)|(1<<MSTR);
    SPCR = B00010000;
    clr = SPSR;
    clr = SPDR;
    delay(10);
    // power on reset
    pot = 144;
    resistance1 = 0;
    res = 0;
    write_pot(pot, resistance1, res);
    // wake up
    pot = B00010000;
    write_pot(pot, resistance1, res);
    // set resistance to High
    pot = B01000000;
    resistance1 = B00000000;
}
res=255;
write_pot(pot,resistance1,res);
for (int i=1;i<11;i++) lastData[i]=0;
}

void loop()
{

//read the serial port and create a string out of what you read
readSerialString(serInString);
if( isStringEmpty(serInString) == false) { //this check is optional

//delay(500);
holdString[0] = serInString[0];
holdString[1] = serInString[1];
holdString[2] = serInString[2];
holdString[3] = serInString[3];
holdString[4] = serInString[4];

//try to print out collected information. it will do it only if there actually is some info.
if (serInString[0] == 43 || serInString[0] == 45 || serInString[0] == 114 || serInString[0] == 103 ||
serInString[0] == 112|| serInString[0] == 80 ||  serInString[0] == 82 || serInString[0] == 99)
{
    if (serInString[0] == 43) positive = true;
    else if (serInString[0] == 45) positive = false;
pstat = false;
if (serInString[0] != 114) gstat = false;
dactest = false;
rtest = false;
ocv = false;
sign = 1;
for (int i = 0; i < 98; i++)
{
    sendString [i] = serInString[i+1];
}
int out =  stringToNumber(sendString,4);
//Serial.print(out,DEC);
if (serInString[0] != 112)
{
pMode = 0;
}
if (serInString[0] == 43)
{
    outvolt = out;
    write_dac(0,outvolt);
digitalWrite(3,HIGH);
speed = 5;
countto = 10;
}
if (serInString[0] == 45)
{
}
ocv = true;
if(out >= 1) write_dac(1,out);
digitalWrite(3,LOW);
speed = 5;
countto = 10;
}

if (serInString[0] == 80)
{
dactest = true;
testcounter = 0;
testlimit = 1023;
//speed = 5;
//countto = 10;
}

if (serInString[0] == 82)
{
rtest = true;
testcounter = 0;
testlimit = 255;
//speed = 10;
//countto = 10;
}

if (serInString[0] == 114)
{
res = out;
write_pot(pot,resistance1,res);
}

if (serInString[0] == 103)
{
dacon();
gstat = true;

outvolt = analogRead(0);
write_dac(0,outvolt);
//speed = 5;
//countto = 20;

if (out > 2000)
{
out = out - 2000;
sign = -1;
}
else if (out < 2000)
{
out = out;
    sign = 1;
}

setting = out;
outvolt = analogRead(0)+(sign*out);
write_dac(0,outvolt);
digitalWrite(3, HIGH);
}
if (serInString[0] == 112)
{
    if (pMode == 0)
    {
        dacon();
    }
pstat = true;
pMode = 1;
    //speed = 5;
    //countto = 20;
    setting = out;
    //outvolt = setting; //initial guess
digitalWrite(3, HIGH);
    write_dac(0,setting);
}
if (serInString[0] == 99)
{
    pstat = true;
    //speed = 5;
    //countto = 20;
    setting = out;
    //outvolt = setting; //initial guess
dacon();
digitalWrite(3, HIGH);
}
}

else if (serInString[0] == 32)
{
    digitalWrite(5, HIGH);
digitalWrite(6, LOW);
delay(100);
digitalWrite(5, LOW);
digitalWrite(6, LOW);
}
flushSerialString(serInString);
}
//Work Section
if (pstat) potentiostat();
if (gstat) galvanostat();
//if (cv)
if (dactest) testdac();
if (rtest) testr();
delay(speed);
counter++;
adcrun = adc + adcrun;
dacrun = dac + dacrun;
if (counter > countto) {

dac = dacrun/counter;

adc = adcrun/counter;

sendout();
counter = 0;
adcrun =0;
dacrun = 0;
}
}

char spi_transfer(volatile char data)
{
SPDR = data; // Start the transmission
while (!(SPSR & (1<<SPIF))) // Wait the end of the transmission
{
};
return SPDR; // return the received byte
}

void write_dac(int address, int value)
{
//This function replaces the old write_dac, which
//is named send_dac now. It calls
//send_dac normally, but first makes sure
//that the 2 DACs aren't fighting each other.
if (address==1)
{
    send_dac(0,0);
    send_dac(1,value);
}
else
{
    send_dac(1,0);
    send_dac(address,value);
}
byte send_dac(int address, int value)
{
    SPCR = B01010000;
    firstdac = (address << 6) + (3 << 4) + (value >> 6);
    seconddac = (value << 2) & 255;
    digitalWrite(SLAVESELECTD, LOW);
    // 2 byte opcode
    spi_transfer(firstdac);
    spi_transfer(seconddac);
    digitalWrite(SLAVESELECTD, HIGH); // release chip, signal end transfer
    // delay(3000); */
    SPCR = B00010000;
}

byte dacoff()
{
    SPCR = B01010000;
    firstdac = (3 << 6);
    seconddac = 0;
    digitalWrite(SLAVESELECTD, LOW);
    // 2 byte opcode
    spi_transfer(firstdac);
    spi_transfer(seconddac);
    digitalWrite(SLAVESELECTD, HIGH); // release chip, signal end transfer
    // delay(3000); */
    SPCR = B00010000;
}

byte dacon()
{
    SPCR = B01010000;
    firstdac = (1 << 6);
    seconddac = 0;
    digitalWrite(SLAVESELECTD, LOW);
    // 2 byte opcode
    spi_transfer(firstdac);
    spi_transfer(seconddac);
    digitalWrite(SLAVESELECTD, HIGH); // release chip, signal end transfer
    // delay(3000); */
    SPCR = B00010000;
}

byte write_pot(int address, int value1, int value2)
{
    /* digitalWrite(SLAVESELECTP, LOW);
    // 3 byte opcode
    spi_transfer(address);
    spi_transfer(value1);
    spi_transfer(value2);
    digitalWrite(SLAVESELECTP, HIGH); */ // release chip, signal end transfer
    sendValue(0, 255 - value2);
// Below Here is Serial Comm Shizzle (for rizzle)

// utility function to know whether an array is empty or not
boolean isStringEmpty(char *strArray) {
    if (strArray[0] == 0) {
        return true;
    } else {
        return false;
    }
}

// Flush String
void flushSerialString(char *strArray) {
    int i = 0;
    if (strArray[i] != 0) {
        while (strArray[i] != 0) {
            strArray[i] = 0; // optional: flush the content
            i++;
        }
    }
}

// Read String In
void readSerialString(char *strArray) {
    int i = 0;
    if (Serial.available()) {
        Serial.println(" "); // optional: for confirmation
        while (Serial.available()) {
            strArray[i] = Serial.read();
            i++;
        }
    }
}

int stringToNumber(char thisString[], int length) {
    int thisChar = 0;
    int value = 0;
    for (thisChar = length - 1; thisChar >= 0; thisChar--) {
        char thisByte = thisString[thisChar] - 48;
        value = value + powerOfTen(thisByte, (length - 1) - thisChar);
    }
    return value;
}

/*
This method takes a number between 0 and 9,
and multiplies it by ten raised to a second number.
*/

long powerOfTen(char digit, int power) {
    long val = 1;
    if (power == 0) {
        return digit;
    }
    else {
        for (int i = power; i >= 1; i--) {
            val = 10 * val;
        }
        return digit * val;
    }
}

void potentiostat() {
    // read in values
    adc = analogRead(0);
    dac = analogRead(1);
    int refelectrode = analogRead(3);
    int diff_adc_ref = adc - refelectrode;
    float err = setting - diff_adc_ref;
    int move = 0;
    for (int i=2;i<=11;i++) lastData[i-1]=lastData[i];
    lastData[10] = err;
    // PID
    float p = 3/1.7/1.7*err; //4/1.7
    float i = 10*getIntegral(); //8/2
    float d = 3.5*(lastData[10]-lastData[9]); //8/8
    move = int(p+i+d);
    outvolt = outvolt + move;
    if (outvolt>1023) {
        res = res - (outvolt-1023)/1024.*255./2;
        outvolt = 1023;
        //res = res-res/6;
        if (res<0) res=0;
    } else if (outvolt<0){
        res = res+(outvolt+(lastData[10]-lastData[9]))/1024.*255.;
        outvolt = 20;
        //res = res - res/6;
        if (res<0) res=0;
    }
    if (abs(outvolt-diff_adc_ref)<100){
        res = res + abs(outvolt-diff_adc_ref)/10.;
        if (res>255) res = 255;
    }
    write_pot(0,resistance1,res);
    write_dac(0,outvolt);
/* //if potential is too high
  if ((diff_adc_ref > setting) && (outvolt > 0))
  {
    move = gainer(diff_adc_ref,setting);
    outvolt = outvolt - move;
    write_dac(0,outvolt);
  }

  //if potential is too low
  else if ((diff_adc_ref < setting) && (outvolt < 1023))
  {
    move = gainer(diff_adc_ref,setting);
    outvolt = outvolt + move;
    write_dac(0,outvolt);
  }

  // if range is limited decrease R
  if ((outvolt > 1022) && (res > 0))
  {
    outvolt = 1000;
    write_dac(0,outvolt);
    resmove = resgainer(adc,setting);
    res = res - resmove;
    write_pot(pot,resistance1,res);
  }

  else if ((outvolt < 1) && (res > 0))
  {
    outvolt = 23;
    write_dac(0,outvolt);
    resmove = resgainer(adc,setting);
    res = res - resmove;
    write_pot(pot,resistance1,res);
    delay(waiter);
  }

  //if range is truncated increase R
  int dude = abs(dac-adc);
  if ((dude < 100) && (res < 255))
  {
    res = res + 1;
    write_pot(pot,resistance1,res);
    delay(waiter);
  }*/

void galvanostat()
{
  //get values
  adc = analogRead(0);
  dac = analogRead(1);
int move = 1;
int diff = 0;

//if charging current
if (sign > 0)
{
    diff = dac - adc;
    //if over current step dac down
    if( ((diff) > (setting)) && (outvolt > 0))
    {
        move = gainer(diff,setting);
        outvolt = outvolt-move;
        write_dac(0,outvolt);
    }

    //if under current step dac up
    if ((diff) < (setting) && (outvolt < 1023))
    {
        move = gainer(diff,setting);
        outvolt = outvolt+move;
        write_dac(0,outvolt);
    }
}

//if discharge current
if (sign < 0)
{
    diff = adc - dac;
    //if over current step dac up
    if ( (diff) > (setting) && (outvolt < 1023))
    {
        move = gainer(diff,setting);
        outvolt = outvolt+move;
        write_dac(0,outvolt);
    }

    //if under current step dac down
    if ((diff) < (setting) && (outvolt > 0))
    {
        move = gainer(diff,setting);
        outvolt = outvolt-move;
        write_dac(0,outvolt);
    }
}

void sendout()
adc = analogRead(0);
dac = analogRead(1);
adcgnd = analogRead(2);
adcref = analogRead(3);
refvolt = analogRead(5);
if (pstat) mode = 2;
else if (gstat) mode = 3;
else if (ocv) mode = 1;
else if (dactest) mode = 4;
else mode = 0;
int setout = sign*setting;
Serial.print("GO,"壹");
Serial.print(outvolt,DEC);
Serial.print(",");
Serial.print(adc);
Serial.print(",");
Serial.print(dac);
Serial.print(",");
Serial.print(res);
Serial.print(",");
Serial.print(setout);
Serial.print(",");
Serial.print(mode);
Serial.print(",");
Serial.print(holdString[0]);
Serial.print(holdString[1]);
Serial.print(holdString[2]);
Serial.print(holdString[3]);
Serial.print(holdString[4]);
Serial.print(",");
Serial.print(adcgnd);
Serial.print(",");
Serial.print(adcref);
Serial.print(",");
Serial.print(refvolt);
Serial.println(",ST");
//res=res+1;
//if (res > 255) res = 0;
//sendValue(0,1);
//sendValue(0,255);
}

void testdac ()
{
digitalWrite(3,LOW);
write_dac(0,testcounter);
outvolt = testcounter;
testcounter = testcounter + 1;
if (testcounter > testlimit) testcounter = 0;
}
void testr ()
{
  digitalWrite(3,HIGH);
  write_dac(0,1023);
  outvolt = 1023;
  res = testcounter;
  write_pot(pot, resistance1, res);
  testcounter = testcounter + 1;
  if (testcounter > testlimit) testcounter = 0;
}

float getIntegral()
{ //Trapez. rule applied to data since pstat was last called
  float integral = lastData[1];
  for (int i=2;i<=9;i++)
  {
    integral = integral + 2*lastData[i];
  }
  integral = integral+lastData[10];
  return 0.5*integral;
}

int gainer(int whatitis, int whatitshouldbe)
{
  int move = abs(whatitis-whatitshouldbe);
  move = constrain(move,1,100);
  return move;
}

/*int regainer(int whatitis, int whatitshouldbe)
{
  int move = 0;
  int diff = abs(whatitis-whatitshouldbe);
  if (diff > 20) move = 30;
  else move = 10;
  //move = constrain(move,1,100);
  return move;
} */

//Barry's hacky functions

byte value;

byte sendBit(boolean state)
{
  digitalWrite(SPICLOCK,LOW);
  delayMicroseconds(10);
  digitalWrite(DATAOUT, state);
  digitalWrite(SPICLOCK, HIGH);
  delayMicroseconds(10);
}
byte sendValue(int wiper, int val)
//tested cycle time for this function is ~565 microseconds.
{
    value = byte(val);
    //digitalWrite(SPICLOCK,LOW);
    //digitalWrite(DATAOUT,LOW);
    digitalWrite(SLAVESELECTP,LOW);
    delayMicroseconds(10);

    //Select wiper
    for(int i=0;i<3;i++)
    {
        sendBit(false);
    }
    sendBit(wiper);

    //write command
    for(int i=0;i<4;i++)
    {
        sendBit(false);
    }
    //data
    sendBit(HIGH && (value & B10000000));
    sendBit(HIGH && (value & B01000000));
    sendBit(HIGH && (value & B00100000));
    sendBit(HIGH && (value & B00010000));
    sendBit(HIGH && (value & B00001000));
    sendBit(HIGH && (value & B00000100));
    sendBit(HIGH && (value & B00000010));
    sendBit(HIGH && (value & B00000001));
    //sendBit(true); //fudge
    digitalWrite(SLAVESELECTP,HIGH);
    //Serial.println(in);
    delayMicroseconds(10);
}

byte readWiper()
{
    //send read command
    digitalWrite(SLAVESELECTP,LOW);
    delayMicroseconds(10);
    sendBit(false);
    sendBit(false);
    sendBit(false);
    sendBit(false);
    sendBit(true);
    sendBit(true);

    //get data
    int data[9];
    Serial.print(" ");
    for(int i=0;i<9;i++)
    {
        digitalWrite(SPICLOCK,LOW);
        digitalWrite(SPICLOCK,HIGH);
        digitalWrite(SPIDATAOUT,LOW);
        digitalWrite(SPIDATAOUT,high);
        digitalWrite(SPIDATAOUT,low);
        digitalWrite(SPIDATAOUT,high);
        digitalWrite(SPIDATAOUT,low);
        digitalWrite(SPIDATAOUT,high);
        digitalWrite(SPIDATAOUT,low);
      }
delayMicroseconds(10);
digitalWrite(DATAOUT, LOW);
delayMicroseconds(10);
data[i] = digitalRead(DATAIN);
digitalWrite(SPI_CLOCK, HIGH);
delayMicroseconds(10);
Serial.print(data[i]);
}
digitalWrite(SLAVE_SELECT_P, HIGH);
Appendix B: Interface

#!/usr/bin/python

import serial
import time
import threading
import os
import re
from math import exp

import resistances

def autoConnect(id=1):
    port = guessUSB()
    if port==False:
        print("Unable to find an Ardustat.")
    else:
        a = Ardustat(port,id=id)
        a.start()
        return a

def listArdustats():
    """Scans ports, lists ardustats. Uses a lockfile to show which are in use."""

    ardList = []

    if os.name=='posix':
        possibles = [x for x in os.listdir("/dev") if \
            (x.find('usb')!=-1 or x.find('USB')!=-1) and x.find('tty')!=-1]
        if len(possibles)>=1:
            for i in range(len(possibles)):
                lockstuff = isLocked("/dev/"+possibles[i])
                if lockstuff != False:
                    ardList.append("/dev/"+possibles[i]+"ID #"+lockstuff+" *IN
USE\n")
                else:
                    aid = isArdustat("/dev/"+possibles[i])
                    if aid=="" or aid>0:
                        ardList.append("/dev/"+possibles[i]+" unused\n")
        else:
            pass

    elif os.name=="nt":
        for i in range(1,100):
            try:
                lockstuff = isLocked("\COM"+str(i))
                if lockstuff != False:
                    ardList.append("\COM"+str(i)+"ID #"+lockstuff+" *IN
USE\n")
            except:
                pass

    else:
        aid = isArdustat("\COM"+str(i))
        if aid=="" or aid>0:
            ardList.append("\COM"+str(i)+" unused\n")

except:
if ardList==[]:
    print("No ardustats detected")
    return
print("Ardustats:")
for i in ardList:
    print(i)

def guessUSB():
    """Attempts to guess the correct serial port."""
    if os.name=='posix':
        possibles = [x for x in os.listdir('/dev') if \n            (x.find('usb')!=-1 or x.find('USB')!=-1) and x.find('tty')!=-1]
        if len(possibles)>=1:
            for i in range(len(possibles)):
                aid = isArdustat("/dev/"+possibles[i])
                print aid
                if aid=="" or aid>0:
                    print("Ardustat found on /dev/"+possibles[i]+".")
                    return "/dev/"+possibles[i]
    elif os.name=""nt":
        for i in range(1,100):
            try:
                aid = isArdustat("COM"+str(i))
                if aid=="" or aid>0:
                    print("Ardustat found on "+"COM"+str(i)+".")
                    return "COM"+str(i)
            except:
                pass
        return False

def isArdustat(port):
    b=serial.Serial(port,57600)
    b.setDTR(False)
    time.sleep(0.022)
    b.setDTR(True)
    for j in range(10):
        line = b.readline()
        if line.find("Hi")!=-1:
            b.close()
            toreturn = ""
            for i in line:
                try:
                    int(i)
                    toreturn += i
                except:
                    continue
            return False
    return toreturn

def isLocked(port):
    try:
        f = open("LOCKFILE","r")
        while True:
            lockport = f.readline().split(" ")
        return False
if len(lockport)==1:
    f.close()
    break
if lockport[0]==port:
    f.close()
    return lockport[1]

f.close()
except:
    pass
return False

class Ardustat(serial.Serial):
    def __init__(self, port, rate=57600, id=None):
        serial.Serial.__init__(self, port, rate)
        self.commands = ['R', 'r', ',', 'p', ',', 'g']  # possibilities
        self.id = id
        self.port = port
        self.ruleBuffer = "1v100"
        self.log = ""

        ### setup and end-point functions
        def start(self):
            f = open("LOCKFILE",'a')
            f.write(self.port + " " + str(id))
            f.close()
            self.reset()
            self.log = Log(self, self.id)
            self.log.start()
            self.logRule(self.ruleBuffer)
            self.openCircuit()

        def reset(self):
            self.setDTR(False)
            time.sleep(0.022)
            self.setDTR(True)

        def end(self):
            self.log.stop()
            f = open("LOCKFILE",'r')
            list = f.readlines()
            list = [x for x in list if x.split(" ")[0]!=self.port]
            f.close()
            f = open("LOCKFILE",'w')
            for i in list:
                f.write(i)
            f.close()
            self.close()

        def calibrate(self, resistance):
            try:
                resistance/1.
            except:
                print("Circuit resistance must be a number!")
def logRule(self, rule):
    # record-every-x-points if-[voltage,current,resistance]-changes-x-%-reset-counter
    if not re.match("^[0-9]{1,4}[vcr][0-9]{1,3}$", rule) or \
        re.match("^[0-9]{1,4}$", rule):
        print("Incorrect expression format.")
        return
    self.ruleBuffer = rule
    try:
        self.log.setRule(self.ruleBuffer)
    except:
        pass

### control functions

def sendRaw(self, command):
    """Takes a command of the form Cxxxx.  C is in self.commands and xxxx is a 4-digit number."
    if self.commands.count(command[0]) == 1:
        try:
            int(command[1:])
        except:
            return "Invalid command."
        self.write(command)
        return "Command '"+command+"' sent."
    else:
        return "Invalid command."

def potentiostat(self, voltage):
    """Potentiostatic mode.  0-5V with respect to reference""
    unit = str(int(voltage/2.5*int(self.log.lastline.split(',')[10])))
    if unit<0: unit = -(unit-2000) #negative voltage
    if len(unit)==1:
        unit = "000"+unit
    elif len(unit)==2:
        unit = "00"+unit
    elif len(unit)==3:
        unit = "0"+unit
    self.sendRaw("p"+unit)

def galvanostat(self, current, initialV=1, charging=1):
    """Puts the Ardustat into galvanostatic mode, current is given in mA.  initialV is depreciated.""
    resAllowance = 100
    if initialV<0 or initialV>5:
        print("Ardustat can't reach that current!")
    return
    if current<0.08:  #ballpark a voltage
        initialV = 0.5

50
elif current<0.5:
    initialV = 1.
elif current<=1:
    initialV = 3.
elif current<=3:
    initialV = 3.
#elif current<6:
    #initialV = 4.
else:
    initialV = 3.5

guessR = initialV/float(current/1000.)    #get a resistance for that voltage
resistance = -1
for i in self.log.res:
    if i+resAllowance>guessR and i-resAllowance<guessR:
        resistance = i
        break
if resistance == -1:
    resistance = self.log.res[-1]
res = str(self.log.res.index(resistance))

if charging==1:
    #this messily prevents current spikes when setting
galvanostatic mode.
    self.sendRaw("g0000")
else:
    self.sendRaw("g2000")
time.sleep(0.2)
if len(res)==1:
    res = "000"+res
elif len(res)==2:
    res = "00"+res
elif len(res)==3:
    res = "0"+res
print(self.sendRaw("r"+res))
time.sleep(0.2)
realV = resistance*current/1000    #get real voltage based on resistance
unit = str(int(realV/2.5*int(self.log.lastline.split(","))[-1]))
if len(unit)==1:
    unit = "000"+unit
elif len(unit)==2:
    unit = "00"+unit
elif charging==1:
    print(self.sendRaw("g0"+unit))
else:
    print(self.sendRaw("g2"+unit))
return resistance

def rampVoltage(self,start,end,duration):
    #volts and seconds
    try:
        start = int(self.log.lastline.split(","))[-1]*float(start)/2.5
        end = int(self.log.lastline.split(","))[-1]*float(end)/2.5
        duration = float(duration)
    except:
        print("All arguments must be numbers!")
return
print("Beginning rampup..")
steps = end-start
timeperstep = duration/abs(steps)
if steps==abs(steps):
    steplist = range(int(steps))
else:
    steplist = range(int(steps),1)
steplist.reverse()
for i in steplist:
    unit = start+i
    if unit<0: unit = -(unit-2000) #negative voltage
    string = str(int(unit))
    if len(string)==1:
        string = "p000"+string
    elif len(string)==2:
        string = "p00"+string
    elif len(string)==3:
        string = "p0"+string
    elif len(string)==4:
        string = "p"+string
    print(self.sendRaw(string))
    time.sleep(timeperstep)
print("Voltage reached.")

def raiseGround(self,voltage):
    
    """Artificially raises ground to 'voltage'. Jumper pin on the Ardustat must be in the inward position."""
    if voltage>5 or voltage<0:
        print("Voltage out of range!")
    unit = str(int(voltage/2.5*int(self.log.lastline.split(",")[10])))
    if len(unit)==1:
        unit = "000"+unit
    elif len(unit)==2:
        unit = "00"+unit
    elif len(unit)==3:
        unit = "0"+unit
    self.sendRaw("d"+unit)

def CV(self,low,high,cycles=1):
    """Right now, does nothing. Placeholder."""
    pass

def openCircuit(self):
    """Go OCV, all DACs set to 0."""
    self.sendRaw('-0000')

def ask(self):
    """Prints a summary of the parsed data, as of now."""
    parsed = self.log.parse(self.log.lastline).split(" ")
    print("unix time: "+str(parsed[0]))
    print("mode: "+str(parsed[1]))
    print("DAC setting: "+str(parsed[2]))
print("cell voltage: "+str(parsed[3]))
print("set resistance: "+str(parsed[4]))
print("current: "+str(parsed[5]))
print("cell resistance: "+str(parsed[6]))
print("ground: "+str(parsed[7]))
print("reference electrode: "+str(parsed[8]))
print("raw: "+str(self.log.lastline))

### utility functions (don't call these explicitly)
def update(self,parsed):
    parsed = parsed.split(" ")
    if len(parsed)<8: return
    self.mode = parsed[1]
    self.setVoltage = parsed[2]
    self.cellVoltage = parsed[3]
    self.setResistance = parsed[4]
    self.current = parsed[5]
    self.cellResistance = parsed[6]
    self.ground = parsed[7]
    self.reference = parsed[8]

### don't call stuff from this class explicitly, either
class Log(threading.Thread):
    def __init__(self, serial, id="None"):
        threading.Thread.__init__(self)
        self.daemon = True
        self.ser = serial
        self.id = id
        self.rule = "1v100"  #rules for logging, defaults to "1", which is log everything.
        self.lastLogCount = 0  #also for timing purposes
        self._stop = threading.Event()
        self._calibrate = threading.Event()
        self.reslist = [[] for x in range(256)]
        self.res = resistances.reslist.get(str(self.id),[])
        if self.res == []:
            print("No calibration data was found, or you did not provide \n an id number. You should run calibrate so your data is more accurate.")
            for i in range(256):
                self.res.append(i/255.*10000)

    def stop(self):
        self._stop.set()
        print("bleck")  # threading.Thread.join(self)

    def setRule(self,rule):
        # from Ardustat.logRule
        self.rule = rule

    def runCal(self, set):
        while True:
            ard = raw_input("Which Ardustat is this? ")
if resistances.reslist.get(ard,None)==None:
    break
else:
    print("That one exists already, try again.")
self.id = ard
print("Beginning calibration. This will take a few minutes. Do not send any commands.")
self._calibrate.set()
self.set = set
self.rescount = 0

def saveRes(self,res,name):
    f = open("resistances.py","r")
    lines = f.readlines()
    f.close()
    f = open("resistances.py","w")
    lines[-1] = lines[-1][:-2]+
    lines.append("","+str(name)+","+str(res)+")")
    f.writelines(lines)
    f.close()

def run(self):
    tehm = str(int(time.time()))
    rawfile = "data.raw","ardustat"+tehm+".raw"
    parsedfile = "data.txt","ardustat"+tehm+".txt"
    self.rfile = open(rawfile,'w')
    self.pfile = open(parsedfile,'w')
    print("Data being recorded in "+parsedfile)
    print("Raw data can be found in "+rawfile)
    time.sleep(0.1)
    while not self._stop.isSet():
        try:
            line = self.ser.readline()
        except:
            continue
        try:
            if len(line.split(","))>=10:
                self.lastline = line
            else:
                continue
        except:
            continue
        if not self._calibrate.isSet():
            if self.shouldLog(line):  # will determine if this line should get
                self.rfile.write(line)
            if not self._calibrate.isSet():
                parsed = self.parse(line)
                self.pfile.write(parsed)
                self.ser.update(parsed)
                self.lastLogCount += 1
            # calibration algorithm starts here if called (skipped otherwise)
            if self._calibrate.isSet():
                #calibration algorithm starts here if called (skipped otherwise)
line = line.split("\,"\n)
try:
    self.reslist[int(line[4])].append(self.set*(1023./int(line[2])-1))
except ZeroDivisionError:
    print line[2]
    self.reslist[int(line[4])].append(0)
self.rescount +=1
#print self.rescount
if self.rescount>256*5:
    self._calibrate.clear()
    #calculate average
    self.res = [sum(x)/5. for x in self.reslist]
    #calculate standard deviation (it is right, I swear)
    self.resdev = [(sum(map(lambda y: (y-
    self.res[x])**2,self.reslist[x]))/5.)**0.5 for x in range(256)]
    if not self._calibrate.isSet():
        self.saveRes(self.res,self.id)
        print("Calibration completed and saved for Ardustat")
        "+str(self.id))

def shouldLog(self,line):
    if self.lastLogCount>=int(re.findall("^[0-9]{1,4},\1\2\3\4\5\6\7\$,self.rule[0]):
        self.lastLogCount = 0
        return True
    line = line.split("\,"\n)
    lastline = self.lastline.split("\,"\n)
    if len(line)<11 or len(lastline)<11: return False
    if re.findall("\1\2\3\4\5\6\7\$",self.rule)!=[0]:
        condition = re.findall("\1\2\3\4\5\6\7\$",self.rule[0]
        percent = int(re.findall("\1\2\3\4\5\6\7\$",condition)[0])
        R = lambda x: self.res[int(x)]
        I = lambda v,r: float(v)/float(r)
        if line[2]=='0': line[2]='0.0001'
        if lastline[2]=='0': lastline[2]='0.0001'
        if condition[0]=='v':
            if abs(1-float(lastline[2])/float(line[2]))*100>=percent:
                self.lastLogCount = 0
                print "2nd"
                return True
        elif condition[0]=='c':
            if abs(1-I(lastline[2],R(lastline[4]))/float(I(line[2],R(line[4]))))*100>=percent:
                self.lastLogCount = 0
                return True
        elif condition[0]=='r':
            if abs(1-R(lastline[4])/R(line[4]))*100>=percent:
                self.lastLogCount = 0
                return True

    return False


def parse(self,line):
    #time mode dac cell res current ref gnd

line = line.split(",")
if len(line)<12: return ""
line[0]="0"
line[11]="0"
if line[7][0]=="-":  #this ghetto's the current to 0 during OCV (when the
    ocvCurrent=0
else:
    ocvCurrent=1
line[7]="del"
line = [float(x) for x in line if x!="del"]
V = lambda x:x/1024*5  #without reference
Vr = lambda x: x/line[9]*2.5  #with reference
R = lambda x: self.res[int(x)]
I = lambda v,r: v/r
potres = R(line[4])
if potres==0: print(str(line[4])+"?? resistance was 0")
gndvolt = Vr(line[7])
refvolt = Vr(line[8])
adc = Vr(line[2])
setvolt = Vr(line[1])
current = (setvolt-adc)/potres *ocvCurrent #goes to 0 if OCV
cellres = 0
else:
cellres = (line[2]/float(line[3]))*potres/(1-line[2]/float(line[3]))
return str(time.time())+" "+str(int(line[6]))+" "+str(setvolt)+" "+str(adc)+" "+str(potres)+" "+str(current)+" "+str(cellres)+" "+str(gndvolt)+" "+str(refvolt)+"\n"

if __name__ == "__main__":
    port = guessUSB()
    a = Ardustat(port,id=2)
a.start()
    a.galvanostat(0.5)
time.sleep(600)
a.openCircuit()
a.end()
    time.sleep(5)
a.galvanostat(2)
time.sleep(30)
a.openCircuit()
a.raiseGround(2)
    #print a.sendRaw("d0520")
a.galvanostat(1,charging=0)
time.sleep(30)
a.openCircuit()
a.end()
Appendix C: Multistat Firmware

#define DATAOUT 12
#define DATAIN 13
#define SPICLOCK 11
#define SLAVESELECTD 10//DAC
#define SLAVESELECTP 7//Pot
#define CNVST 2
#define EOC 3

//DAC Commands
byte dacInit = B00010000;
byte dacUpdateMain1 = B11101111;
byte dacUpdateMain2 = B11110000;

//DAC locations
byte dac0 = B00100000;
byte dac1 = B00110000;
byte dac2 = B01000000;
byte dac3 = B01010000;
byte dac4 = B01100000;
byte dac5 = B01110000;
byte dac6 = B10000000;
byte dacByte; //for the function to use

byte potInit = B101;
byte potData = B10000000;
byte pot1 = B000;
byte pot2 = B001;
byte pot3 = B010;
byte pot4 = B011;
byte pot5 = B100;
byte pot6 = B101;

//Resistance (step)
int res[7]; //res[0] isn't anything
int adc[7]; //out of pot (0 is WE)
int dac[7]; //out of dac
int dacset[7];
int alladc[16];
//int outvolt=4096;
boolean charging = true;
int sign=1;
int mode = 0; //0 ocv, 1 pot, 2 galv, 3 manual
boolean pMode = 0; //saved variable to remember if the last mode was pstat or not
int lastData[10]; //previous error values for use in pstat's PID algorithm
int resmove;
boolean rtest;

//Serial Comm Stuff
int incomingByte;
boolean setVoltage;
char serInString[100];
char sendString[99];
char holdString[6];
int output;
//boolean whocares = false;
//boolean positive = false;
boolean gstat = false;
boolean pstat = false;
//boolean cv = false;
int setting = 0;
int throttle = 100;
int countto = 0;

void setup()
{
  millis();
  for(int i=0;i<7;i++)
  {
    dacset[i]=0;
    res[i]=0;
  }
  //initialize serial comm and pins
  Serial.begin(57600);
  Serial.println("Hi Barry!");
  //initialize pins, disable everything
  pinMode(DATAOUT,OUTPUT);
  pinMode(SPICLOCK,OUTPUT);
  pinMode(SLA VESELECTD,OUTPUT);
  pinMode(SLA VESELECTP,OUTPUT);
  pinMode(CNVST,OUTPUT);
  pinMode(DATAIN,INPUT);
  pinMode(EOC,INPUT);
  digitalWrite(SLA VESELECTD,HIGH);
  digitalWrite(SLA VESELECTP,HIGH);
  digitalWrite(CNVST,HIGH);
  //MAX-chip setup register
  digitalWrite(SLA VESELECTD,LOW);
  sendByte(B01000000,HIGH);
  digitalWrite(SLA VESELECTD,HIGH);
  delay(100);
  //MAX-chip setup register
  digitalWrite(SLA VESELECTD,LOW);
  sendByte(B01000000,HIGH);
  digitalWrite(SLA VESELECTD,HIGH);
  delay(100);
  //turn on the DACs
  counterOn();
  delay(100);
  workGround();
  delay(100);
  //ADC Conversion Register
  digitalWrite(SLA VESELECTD,LOW);
  sendByte(B01000000,HIGH);
  digitalWrite(SLA VESELECTD,HIGH);
  delay(100);
}
digitalWrite(SLAVESELECTD,HIGH);
delay(100);

//ADC Averaging Register
digitalWrite(SLAVESELECTD,LOW);
sendByte(B00110000,HIGH);
digitalWrite(SLAVESELECTD,HIGH);
delay(100);

digitalWrite(SLAVESELECTD,LOW);
sendByte(dacInit,HIGH); //set dacs to 0 to start
sendByte(B10100000,HIGH);
sendByte(B00000000,HIGH);
digitalWrite(SLAVESELECTD,HIGH);
}

void loop()
{
    //read the serial port and create a string out of what you read
    readSerialString(serInString);
    if( isStringEmpty(serInString) == false) { //this check is optional

        //try to print out collected information. it will do it only if there actually is some info.
        if (serInString[0] == 100 || serInString[0] == 45 || serInString[0] == 112 || serInString[0] == 107 ||
            serInString[0] == 48 || serInString[0] == 49 || serInString[0] == 50 || serInString[0] == 51 ||
            serInString[0] == 103 ||
            serInString[0] == 52 || serInString[0] == 53 || serInString[0] == 54 || serInString[0] == 114 ||
            serInString[0] == 82)
        {
            //if (serInString[0] == 43) positive = true;
            if (true)
                pstat = true;
            gstat = false;
            //if (serInString[0] != 114) gstat = false;
            dactest = false;
            rtest = false;
            charging = true;
            for (int i = 0; i < 98; i++)
            {
                sendString[i] = serInString[i+1];
            }
            int out = stringToNumber(sendString,4);
        }
        if(serInString[0]!=100) pMode=false;
switch (serInString[0])
{
    case 48: mode=0;   // 0
        write_dac(0,out);
        //if (out==0) workGround();
        /*else*/ workOn();
        break;
    case 49: mode=3;   // 1
        write_dac(1,out);
        break;
    case 50: mode=3;   // 2
        write_dac(2,out);
        break;
    case 51: mode=3;   // 3
        write_dac(3,out);
        break;
    case 52: mode=3;   // 4
        write_dac(4,out);
        break;
    case 53: mode=3;   // 5
        write_dac(5,out);
        break;
    case 54: mode=3;   // 6
        write_dac(6,out);
        break;
    case 100: mode=3;  // d
        write_dacs(out);
        break;
    case 45:  // incomplete
        counterFloat();
        write_dacs(0);
        dac_update(); // for good luck
        out=0;
        break;
    case 112: mode=1;  // p
        pstat=true;
        counterOn();
        setting = out;
        write_pots(255);
        write_dacs(out);
        break;
    case 103: mode=2;  // g
        gstat=true;
        counterOn();
        if(out>5000)
        {
            sign=-1;
            setting = out-5000;
            write_dacs(out-5000);
        }else{
            sign=1;
            setting = out;
write_dacs(out);
}
break;
case 107: mode = 0;  //k
    counterGround();
    write_dacs(0);
    break;
case 114: {int potnum = floor(out/1000);  //r
    if(potnum==0) write_pots(out);
    else write_pot(potnum,out-(potnum*1000));}
    break;
case 82:  rtest=true;  //R
    counterOn();
    write_dacs(4095);
    dac_update();
    break;
default: break;
    //still need to add r, g (above, too)
}
dac_update();
flushSerialString(serInString);
}

//Work Section
getADC(alladcs);
adcs[0]=alladcs[0];
adcs[1]=alladcs[1];
adcs[2]=alladcs[2];
adcs[3]=alladcs[3];
adcs[4]=alladcs[4];
adcs[5]=alladcs[5];
adcs[6]=alladcs[6];
dacs[1]=alladcs[7];
dacs[2]=alladcs[8];
dacs[3]=alladcs[9];
dacs[4]=alladcs[10];
dacs[6]=alladcs[12];
sendout();
if (pstat) potentiostat();
if (gstat) galvanostat();
    //if (cv)
    //if (dactest) testdac();
    // if (rtest) testr();
    delay(throttle);
}

void sendout()
{
    //Serial.print("ST");
    Serial.print(millis());
    //DAC Setting 0-6
}
Serial.print("|\nfor(int i=0;i<7;i++)
{
    Serial.print(dacset[i]);
    Serial.print(",");
}
//DAC Reading 0-6
Serial.print("|
for(int i=0;i<7;i++)
{
    Serial.print(dac[i]);
    Serial.print(",\nfor(int i=1;i<7;i++)
{
    Serial.print(res[i]);
    Serial.print(",");
}
//Pot Setting 1-6
Serial.print("|
for(int i=1;i<7;i++)
{
    Serial.print(res[i]);
    Serial.print(",");
}
//mode
//Main Setting
//Working DAC Setting
//Working Reading
Serial.println("\n
}

void potentiostat()
{
    //counterOn(); //for good luck
    //int diff_adc_ref = adc - refelectrode;
    for(int j=1;j<7;j++)
    {
        float diff_adc_ref = adc[j];
        float err = setting - diff_adc_ref;
        int move = 0;
        //for (int i=2;i<=11;i++) lastData[i-1]=lastData[i];
        //lastData[10] = err;
        //PID
        float p = 1*err; //4/1.7
        //float i = 10*getIntegral(); //8/2
        //float d = 3.5*(lastData[10]-lastData[9]); //8/8
        //move = int(p+i+d);
        move = int(p);
        dacset[j] = dacset[j] + move;
    }
}
//Serial.println(move);
if (dacset[j]>4090)
{
    res[j] = res[j] - 20;/(dacset[j]-4095)/4096.*255;
    dacset[j] = 4000;
    //res = res-res/6;
    if (res[j]<1) res[j]=1;
} else if (dacset[j]<0){
    res[j] = res[j]+20;/(dacset[j]+(lastData[10]-lastData[9]))/1024.*255.;
    dacset[j] = 1;
    //res = res - res/6;
    if (res[j]>255) res[j]=255;
}
if (abs(dacset[j]-diff_adc_ref)<100){
    res[j] = res[j] + abs(dacset[j]-diff_adc_ref)/10.;
    if (res[j]>255) res[j] = 255;
}
write_pot(j,res[j]);
write_dac(j,dacset[j]);
}
dac_update();


/*void galvanostat()
{
    float move;
    float diff;
    float err;
    float p;
    float i;
    float d;
    float Kp = 1;
    float Ki = 1;
    float Kd = 1;
    for(int j=1;j<7;j++)
    {
        if(charging)
        {
            diff=dac[j]-adc[j];
            if(diff>0) err = setting-diff;
            else err = setting;
            p = Kp*err;
            move = (p);
        } else {
            diff=adc[j]-dac[j];
            if(diff>0) err = setting-diff;
            else err = setting;
            p = Kp*err;
            move = (p);
        }
        if(dacset[j]+move>4090) dacset[j]=4090;
        else dacset[j]=dacset[j]+move;
        write_dac(j,dacset[j]);
    }*/
void dac_update();
}*/

void galvanostat()
{
  int move;
  int diff;
  for(int j=1;j<7;j++)
  {
    if (sign>0) diff = dac[j]-adc[j];
    else diff = adc[j]-dac[j];

    move = gainer(diff,setting,dacset[j]);
    if (sign>0) dacset[j] = dacset[j]-move;
    else dacset[j] = dacset[j]+move;
    dacset[j] = constrain(dacset[j],0,4000);
    write_dac(j,dacset[j]);
  }
}

int gainer(int whatitis, int whatitshouldbe, int huh)
{
  int move;
  int err = whatitis-whatitshouldbe;
  for (int i=2;i<=11;i++) lastData[i-1]=lastData[i];
  lastData[10] = err;
  int p = 0.5*err;
  int i = 0*err;
  int d = 0*err;
  move = (p+i+d);
  //if (sign>0) move = constrain(huh-move,0,4000)+huh;
  //else*/ move = constrain(huh+move,0,4000)-huh;
  return move;
}

void testr()
{
  for(int i=1;i<7;i++)
  {
    res[i] = res[i]+1;
    if(res[i]>255) res[i]=0;
    write_pot(i,res[i]);
  }
}

void write_dac(int dac, int value)
{
  switch(dac)
  {
    case 0:
      dacByte = dac0;
      break;
    /*case 1:
      dacByte = dac1;
      break;
    */
    /*case 2:
      dacByte = dac2;
      break;
    */
    case 3:
      dacByte = dac3;
      break;
    case 4:
      dacByte = dac4;
      break;
    case 5:
      dacByte = dac5;
      break;
    case 6:
      dacByte = dac6;
      break;
    case 7:
      dacByte = dac7;
      break;
    default:
      dacByte = 0;
      break;
  }
case 1:
    dacByte = dac1;
    break;
case 2:
    dacByte = dac2;
    break;
case 3:
    dacByte = dac3;
    break;
case 4:
    dacByte = dac4;
    break;
case 5:
    dacByte = dac5;
    break;
case 6:
    dacByte = dac6;
    break;
default:
    return;
}
dacset[dac]=value;
digitalWrite(SLAVESELECTD,LOW);
sendByte(dacInit,HIGH);
sendByte(dacByte+(value>>8),HIGH);
sendByte(value-((value>>8)*256),HIGH);
digitalWrite(SLAVESELECTD,HIGH);
delayMicroseconds(10);
}

void write_dacs(int value)
{
    for(int i=1;i<7;i++)
    {
        write_dac(i,value);
    }
}

void dac_update()
{
    digitalWrite(SLAVESELECTD,LOW);
    sendByte(dacInit,HIGH);
    sendByte(dacUpdateMain1,HIGH);
    sendByte(dacUpdateMain2,HIGH);
    digitalWrite(SLAVESELECTD,HIGH);
}

void write_pots(int value)
{
    Serial.println("sup");
    for (int i=1;i<7;i++)
    {
        write_pot(i,value);
    }
void write_pot(int pot, int value)
{
    res[pot] = value;
    value = 255 - value;
    byte potnum;
    switch(pot)
    {
        case 1: potnum = pot1;
            break;
        case 2: potnum = pot2;
            break;
        case 3: potnum = pot3;
            break;
        case 4: potnum = pot4;
            break;
        case 5: potnum = pot5;
            break;
        case 6: potnum = pot6;
            break;
    }
    digitalWrite(SLAVESELECTP, LOW);
    sendBit(HIGH && (potnum & B100), LOW);
    sendBit(HIGH && (potnum & B010), LOW);
    sendBit(HIGH && (potnum & B001), LOW);
    sendByte(value, LOW);
    digitalWrite(SLAVESELECTP, HIGH);
    delayMicroseconds(10);
}

byte sendBit(boolean state, boolean clk)
{
    // clk indicates on what edge data is sent
    digitalWrite(SPILOCK, clk);
    //delayMicroseconds(10);
    digitalWrite(DATOUT, state);
    digitalWrite(SPILOCK, !clk);
    delayMicroseconds(1);
}

byte sendByte(int val, boolean clk)
{
    delayMicroseconds(10);
    //data
    byte value = byte(val);
    sendBit(HIGH && (value & B10000000), clk);
    sendBit(HIGH && (value & B01000000), clk);
    sendBit(HIGH && (value & B00100000), clk);
    sendBit(HIGH && (value & B00010000), clk);
    sendBit(HIGH && (value & B00001000), clk);
    sendBit(HIGH && (value & B00000100), clk);
sendBit(HIGH && (value & B00000010), clk);
sendBit(HIGH && (value & B00000001), clk);
}

int getADC(int results[])
{
    int result;

digitalWrite(SLAVESELECTD, LOW);
sendByte(B11111000, HIGH);
digitalWrite(SLAVESELECTD, HIGH);

digitalWrite(SPICLOCK, HIGH); //needs to start high
digitalWrite(CNVST, LOW); //start the conversion
//delayMicroseconds(1);
digitalWrite(CNVST, HIGH);

countto = 0;
while (true)
{
    //Serial.println("haisdf");
    delay(1);
    if(!digitalRead(EOC)) break;
    if (countto>20)
    {
        digitalWrite(SLAVESELECTD, LOW);
        sendByte(B11111000, HIGH);
        digitalWrite(SLAVESELECTD, HIGH);

digitalWrite(SPICLOCK, HIGH); //needs to start high
digitalWrite(CNVST, LOW); //start the conversion
delayMicroseconds(1);
digitalWrite(CNVST, HIGH);
countto=0;
    }
    else countto++;
}

for(int i=0;i<16;i++)
{
    result = 0;

digitalWrite(SLAVESELECTD, LOW);
for (int i=0;i<8;i++)
{
    digitalWrite(SPICLOCK, LOW);
    if(digitalRead(DATAIN)) result = result*2 + 1;
    else result = result*2;
    digitalWrite(SPICLOCK, HIGH);
}
digitalWrite(SLAVESELECTD, HIGH);
digitalWrite(SLAVESELECTD, LOW);
for (int i=0;i<8;i++)
{  
digitalWrite(SPICLOCK,LOW);
  if(digitalRead(DATAIN)) result = result*2 + 1;
  else result = result*2;
  digitalWrite(SPICLOCK,HIGH);
  //Serial.println(result);
}
digitalWrite(SLAVESELECTD,HIGH);
results[i]=result;
}

void workGround()
{
  digitalWrite(SLAVESELECTD,LOW);
  sendByte(dacInit,HIGH);
  sendByte(B11110000,HIGH);
  sendByte(B00011000,HIGH);
  digitalWrite(SLAVESELECTD,HIGH);
}

void workOn()
{
  digitalWrite(SLAVESELECTD,LOW);
  sendByte(dacInit,HIGH);
  sendByte(B11110000,HIGH);
  sendByte(B00010010,HIGH);
  digitalWrite(SLAVESELECTD,HIGH);
  Serial.println("sup");
}

void counterGround()
{
  digitalWrite(SLAVESELECTD,LOW);
  sendByte(dacInit,HIGH);
  sendByte(B11111111,HIGH);
  sendByte(B11101000,HIGH);
  digitalWrite(SLAVESELECTD,HIGH);
}

void counterOn()
{
  digitalWrite(SLAVESELECTD,LOW);
  sendByte(dacInit,HIGH);
  sendByte(B11110111,HIGH);
  sendByte(B11100010,HIGH);
  digitalWrite(SLAVESELECTD,HIGH);
  delayMicroseconds(10);
}

void counterFloat()
{
  digitalWrite(SLAVESELECTD,LOW);
}
sendByte(dacInit,HIGH);
sendByte(B11110111,HIGH);
sendByte(B11100100,HIGH);
digitalWrite(SLAVESELECTD,HIGH);
}

boolean isStringEmpty(char *strArray) {
    if (strArray[0] == 0) {
        return true;
    }
    else {
        return false;
    }
}

//Flush String
void flushSerialString(char *strArray) {
    int i=0;
    if (strArray[i] != 0) {
        while(strArray[i] != 0) {
            strArray[i] = 0;                  // optional: flush the content
            i++;
        }
    }
}

//Read String In
void readSerialString (char *strArray) {
    int i = 0;
    if(Serial.available()) {
        Serial.println(" ");  //optional: for confirmation
        while (Serial.available()){
            strArray[i] = Serial.read();
            i++;
        }
    }
}

int stringToNumber(char thisString[], int length) {
    int thisChar = 0;
    int value = 0;

    for (thisChar = length-1; thisChar >=0; thisChar--) {
        char thisByte = thisString[thisChar] - 48;
        value = value + powerOfTen(thisByte, (length-1)-thisChar);
    }
    return value;
}

/*
This method takes a number between 0 and 9,
and multiplies it by ten raised to a second number.
*/
long powerOfTen(char digit, int power) {
    long val = 1;
    if (power == 0) {
        return digit;
    }
    else {
        for (int i = power; i >=1 ; i--) {
            val = 10 * val;
        }
        return digit * val;
    }
}

References Cited


