BIOMIMETIC APPLICATION OF ION-CONDUCTING-BASED MEMRISTIVE DEVICES IN SPIKE-TIMING-DEPENDENT-PLASTICITY

by

Kolton T. Drake

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Electrical Engineering

Boise State University

August 2015

© 2015

Kolton T. Drake

ALL RIGHTS RESERVED

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Kolton T. Drake

Thesis Title: Biomimetic Application of Ion-Conducting-Based Memristive Devices in Spike-Timing-Dependent-Plasticity

Date of Final Oral Examination: 12 June 2015

The following individuals read and discussed the thesis submitted by student Kolton T. Drake, and they evaluated his presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Kristy Campbell, Ph.D.	Chair, Supervisory Committee
Elisa Barney Smith, Ph.D.	Member, Supervisory Committee
Kurtis Cantley, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Kristy Campbell, Ph.D., Chair of the Supervisory Committee. The thesis was approved for the Graduate College by John R. Pelton, Ph.D., Dean of the Graduate College.

ACKNOWLEDGEMENTS

I would like to thank Dr. Campbell for her continual support, encouragement, and inspiration throughout my time as a research student. The skills I have gained both technically and academically can, in a large part, be credited to her tutelage and oversight since my Sophomore year as an undergraduate. I cannot express how truly thankful for the opportunities I’ve been afforded while working with her.

I would also like to thank Dr. Barney Smith and Dr. Cantley for the knowledge I’ve gained from their courses and their gracious participation on my graduate committee. Special thanks go to Kyler Palmer and Cody Breckenridge for their first-steps in developing the console application used to control the B1500. Without this interface, the tests that were performed and analyzed in this thesis would not yet be finished.

I would also like to thank Meredith for her patience and support during my time experimenting, researching, and writing. I would not have been able to do this without her.

ABSTRACT

The design and synthesis of artificial learning systems has been aided by the study of biological learning systems. Classic biological learning is driven by the strengthening and weakening of the synapses that connect neurons within the brain through a phenomenon known as Spike-Timing-Dependent-Plasticity. That is, synaptic connectivity between neurons is modulated by the relative timing of their spiking outputs. Similarly, neuromorphic computing architectures can implement a mesh of artificial neurons interconnected by a network of artificial synapses to mimic the learning behaviors found in nature.

Memristors, two-terminal devices whose resistance can be programmed as a function of voltage and current, offer a promising biomimetic solution for a hardware-based artificial synapse. This work focuses on characterizing the switching behavior of an ion-conducting, chalcogenide-based resistive memory in a test environment emulating the behavior of a two-neuron, single-synapse neuromorphic circuit to demonstrate learning at speeds significantly faster than those found in biological synapses.

The results from this study show that the ion-conducting memristors used in this work exhibit effective learning at time scales ranging over several orders of magnitude: from the biologically-relevant millisecond region to the faster-than-nature nanosecond region.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

STDP	Spike Timing Dependent Plasticity
ANN	Artificial Neural Network
MIM	Metal-Insulator-Metal
ESD	Electro-Static Discharge
DUT	Device Under Test
SMU	Semiconductor Measurement Unit
WGFMU	Waveform Generator/Fast Measurement Unit
RSU	Remote-Sense and Switch Unit
AC	Alternating Current
DC	Direct Current
FWHM	Full-Width-Half-Max

CHAPTER ONE: INTRODUCTION

This work focuses on characterizing the switching behavior of an ion-conducting, chalcogenide-based resistive memory in a test environment that emulates the biologically equivalent neuron-synapse connection. This includes the demonstration of memristive state adjustment, or “learning,” at speeds significantly faster than those found in biological synapses, which establishes an exciting precedent in the synthesis and operation of a hardware-based synapse for use in neuromorphic computing. An introduction to a few key concepts is necessary before diving into the details of how this is performed, and is provided in the first chapters of this work.

This chapter gives a brief introduction to memristor theory and neuromorphic computing, specifically Spike-Timing-Dependent-Plasticity (STDP). The motivation for and an outline of the body of this work is also included.

1.1 Memristor Overview

The term “memristor” is a portmanteau created from “memory” and “resistor,” as the term describes a device whose resistance can be changed, by application of potential or current to the device, to a value that is “remembered.” Additionally, the resistance of a memristor is a function of its history [1][2].

In 1971, Leon Chua proposed that there was a fourth circuit element called the memristor in addition to the three fundamental circuit elements: resistors, capacitors, and inductors [2]. A summary of the four basic circuit variables and their associated circuit elements is shown in Figure 1. The relationships between the four fundamental circuit variables, voltage (*v*), current (*i*), charge (*q*), and flux linkage (*φ*), define the values of each of these basic circuit elements, but not all of these variables were appropriately related by an elementary circuit component. Thus, Chua proposed the necessary existence of the memristor, to be defined as a two terminal circuit element relating flux linkage and charge and characterized by the *φ*-*q* curve [2].

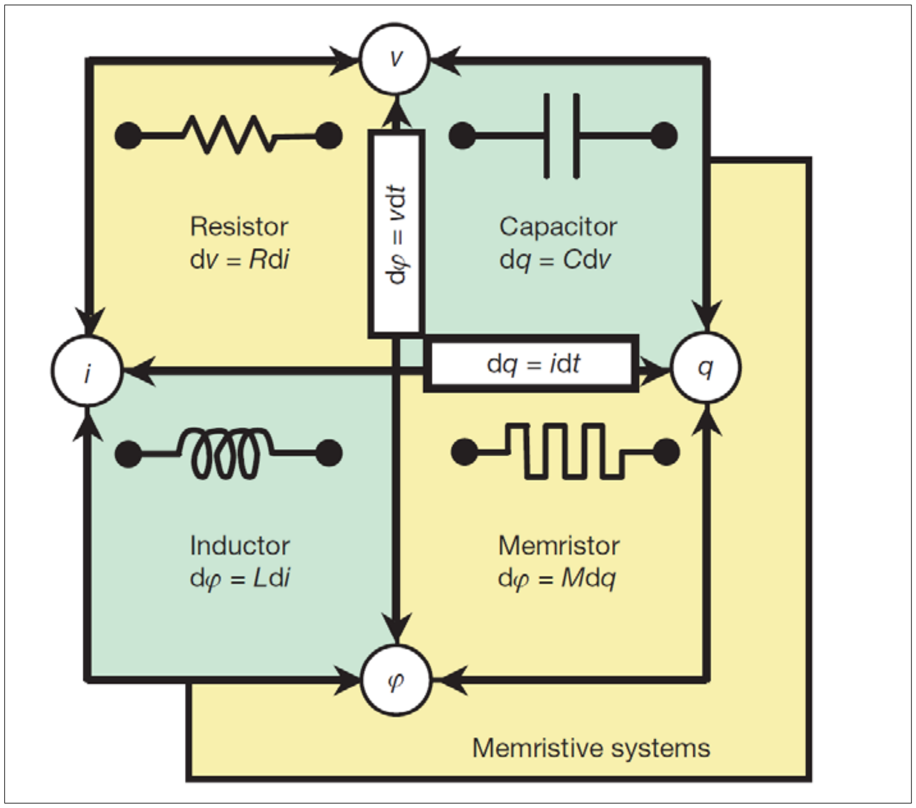


Figure 1. The four basic circuit elements and their relation to the four fundamental circuit variables [3].

In a recent publication [4], Chua claims that if a device exhibits a pinched hysteresis loop on the voltage-current plane, as shown in Figure 2, that device can be classified as a memristor. For this reason, the devices characterized by this work will be alternately referred to as "memristors" or "resistive memory." This means that devices ranging from those described by Hirose and Hirose in 1976 [5] to Kozicki and West in

1998 [6] should also be classified as memristors. While Strukov and Snider et al. in their *Nature* publication, “The Missing Memristor Found” [3], claimed to have found the first memristor, this is clearly not the case. Their TiO_2 based resistive memory and the models they developed to describe the resistive switching phenomena show a pinched hysteresis loop, but they are not unique in this respect.

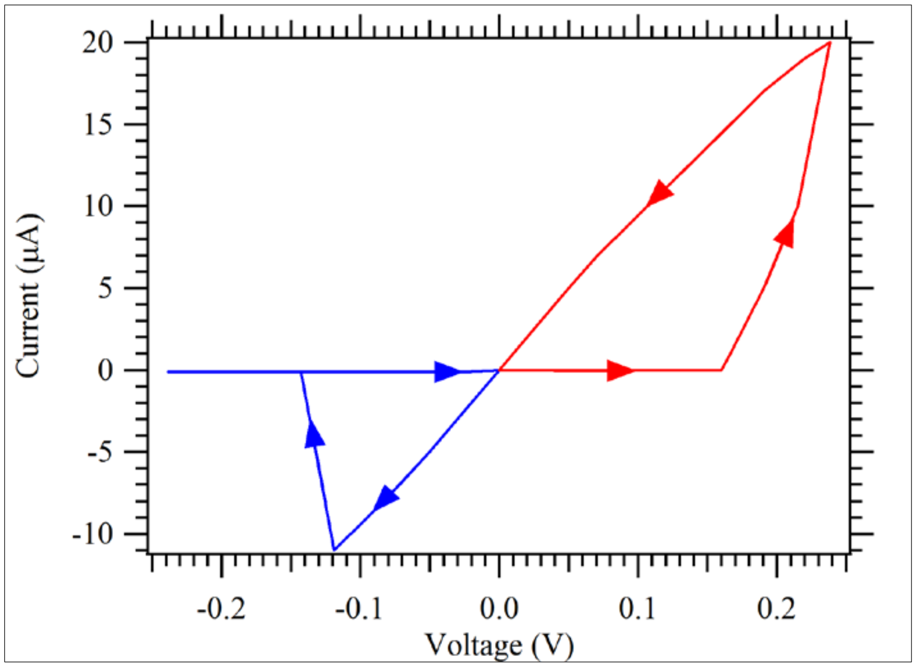


Figure 2. An example of a pinched I-V hysteresis loop for a memristor.

1.2 Neuromorphic Computing

Neuromorphic computing is a field that seeks to increase the speed and efficiency of solving complex computing problems by developing hardware and software solutions that emulate or simulate biological learning systems. For certain problems, such as natural scene recognition, humans regularly outperform computer vision models in terms of accuracy [7]. Simulations on the scale of a mammalian brain are incredibly computationally expensive. For example, a simulation of a cat brain performed by IBM in 2009 ran 83 times slower than its biological counterpart despite using the IBM Blue

Gene/P supercomputer, which was equipped with 147,456 CPUs and 144 TB of main memory [8].

Biological systems are also extremely efficient in terms of power, space, and time requirements for processing sensory input [9]. These benefits are primarily attributed to the use of elementary physical phenomena, such as the interaction of charged neurotransmitter ions with biologically generated electrical signals, as the computational operators and analog memory storage elements, which allow for massively parallel processing of multiple sensory inputs [10]. In mimicking the synaptic connections between neurons within brains, learning within artificial neural networks (ANN) is accomplished by varying the strength of the connections between individual “neurons” within the network. This is accomplished by introducing the network to stimuli that should produce a known output, and incrementally adjusting the strength of the internal connections of the network until the stimuli-response reproduces the expected output [9]. These incremental adjustments typically require computationally expensive software-based algorithms or low-density, complex circuitry [10][11]. Memristors, which have the ability to incrementally adjust their conductance and act as a form of analog memory, offer a promising solution for a hardware-based synapse.

1.3 STDP: “Neurons that fire together, wire together.”

To understand how Spike-Timing-Dependent-Plasticity (STDP) can affect communication between neurons, it’s necessary to understand some of how neurons facilitate communication. Figure 3 shows a diagram of a neuron, its dendrites, and its axon connecting it to another neuron [12].

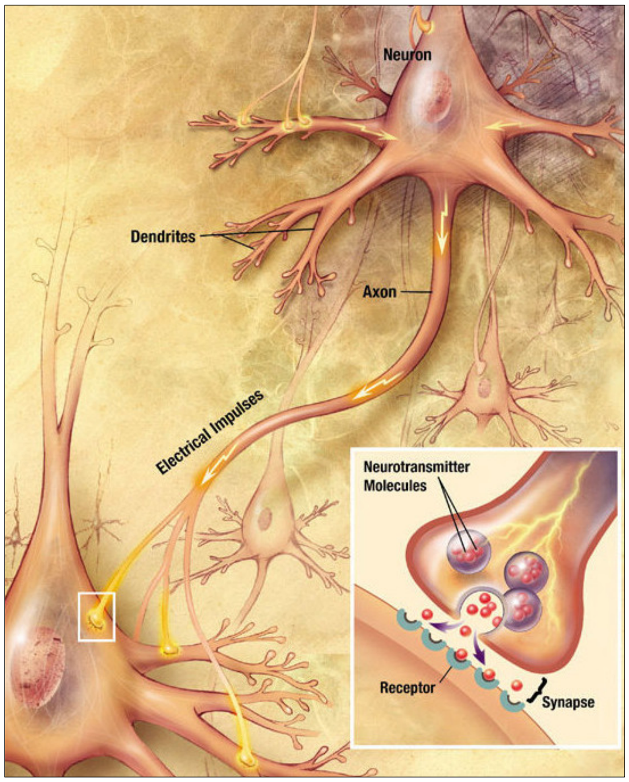


Figure 3. A diagram of a synaptic connection between neurons [12].

A neuron receives signals on its dendrites in the form of neurotransmitters sent from other neurons' axons. The neuron sends signals by "firing" electrical pulses down its own axon membrane, which can release neurotransmitters onto another neuron's dendrite. These signals are formally known as action potentials, but are commonly referred to as "spikes" due to their abrupt voltage-time signature [13].

The interfacial gap between an axon and a dendrite is called the synapse, where the axonal neuron (the sender) is denoted as the "presynaptic" neuron and the receiving dendrites are connected to the "postsynaptic" neuron [13]. When the presynaptic action potential reaches the synapse, channels open in the presynaptic axon that allow neurotransmitters to flow out of the axon and into the receptors on the dendrites of the postsynaptic neuron [14]. The receptors on the dendrite respond to the presence of these neurotransmitters and create a postsynaptic action potential. If this neurotransmitterinduced action potential is large enough, it can cause the postsynaptic neuron to fire its own action potential [14].

Simply stated, the strength of the synaptic connection between two neurons determines how well they communicate. In 1949, Donald Hebb postulated that the strength of a synapse between two neurons is increased when the presynaptic and postsynaptic neurons are simultaneously active, allowing increased flow of neurotransmitters across the synaptic gap [13]. Hebb's work was overly simplified in that the synaptic strength was modulated by the average firing rate of pre- and postsynaptic neurons, which did not take into consideration the impact of individual synaptic events. Subsequent studies by Gerstner et al. expanded Hebb's work in 1993 to shift the emphasis from ensembles of spikes (average firing rate) to the impact of individual spiking events [15]. This provided the foundation for what is now known as Spike-Timing-Dependent-Plasticity [16].

Spike-Timing-Dependent-Plasticity describes a learning mechanism by which the strength of the synaptic connection between neurons can be modulated by the relative timing of individual spikes from the neurons' outputs; if there is an action potential present on the dendrite of the postsynaptic neuron that is well timed with the firing of the presynaptic neuron, the strength of the synaptic connection will either be potentiated (strengthened) or depressed (weakened). Originally proposed in the context of machine learning, experimental work in 1998 by Bi and Poo demonstrated that STDP is the driving force for synaptic strength modulation in real neural tissue by electrically stimulating cultures of rat hippocampal neurons [17].

Bi and Poo’s work showed that if the postsynaptic neuron fires at the same time that the receptors on the dendrite have their own action potential, the action potential on the postsynaptic dendrite created by the reception of neurotransmitters is effectively amplified, resulting in strong communication between the neurons and potentiation of their synaptic connection. Conversely, if the action potential sourced by the postsynaptic neuron is not coincident with the passage of neurotransmittors, the reception of the neurotransmitters is effectively rebuffed, resulting in poor communication and depression of the synaptic connection.

As the proposed electrical analogue to a biological synapse, the memristor is desired to be able to have its resistance modulated by STDP learning pulses [18]. Chapter 2 includes a more in-depth discussion about STDP theory and how it works with memristors.

1.4 Motivation and Outline

This work is motivated by the possibility that the ion-conducting memristor used for these experiments is a good biomimetic analog to the chemical synapse, and the desire to quantify the response of the ion-conducting memristor used in this work under STDP test conditions. While these devices demonstrate learning functionality similar to natural synapses, we seek to investigate the range of speeds at which these devices can be programmed, including speeds significantly faster than those found in nature. Optimization of test conditions to fit this type of memristor’s specific programming characteristics is necessary to improve the programming response of the device. This includes investigating device behavior as a function of the shape, amplitude, and timing of the programming pulse applied to the device.

Chapter 2 includes an overview of the structure and fabrication of the ion-conducting memristors used in this work and a review of STDP implemented with ideal models of memristors.

Chapter 3 provides an overview of the experiments performed and the measurement tools used to gather the data. It also includes a brief overview of the device design, structure, and fabrication, as well as the typical programming characteristics of the ion-conducting memristors used in this work. This section aims to elucidate the selection of experimental parameters.

The results and discussions of the experiments are presented in Chapter 4. Chapter 5 summarizes this thesis.

CHAPTER TWO: BACKGROUND

This chapter includes background information about the structure of the ion-conducting chalcogenide memristor used in this work and a brief overview of STDP using ideal memristive models.

2.1 Ion-Conducting Resistive Memory

Ion-conducting devices that exhibit nonvolatile resistive switching have been identified as potential candidates for scalable, fast switching, and low current memory elements [19]. Typically, resistive memories are built with a Metal-Insulator-Metal (MIM) structure that starts at a very high resistance as shown in Figure 4 (a). Chalcogenide-based ion-conducting resistive memories modulate their resistances through a mechanism that involves the generation of mobile metal ions when a potential is applied across the device. An easily oxidized metal is typically used for the active metal layer, such as Ag or Cu. For devices based on chalcogenide glasses, such as Ge_xSe_{1-x}, the mobile metal ions move into and through the amorphous chalcogenide (as shown in Figure 4 (b)) when a potential above a certain threshold, commonly denoted as v_{th}, is applied [20].

As the mobile ions are reduced at the more negative electrode, a conductive channel, also referred to as a conductive filament, begins to form through the amorphous material, which reduces the device's resistance as shown in Figure 4 (c) and (d). By reversing the applied potential, the reverse reaction occurs; the metal in the channel is oxidized and forced to migrate towards the more negative electrode, which is now the opposite electrode. This severs the conductive filament and increases the device's resistance as shown in Figure 4 (e) [21]. Not all of the metal in the channel is removed as shown in Figure 4 (f), so subsequent programming operations often have a reduced threshold requirement [20].

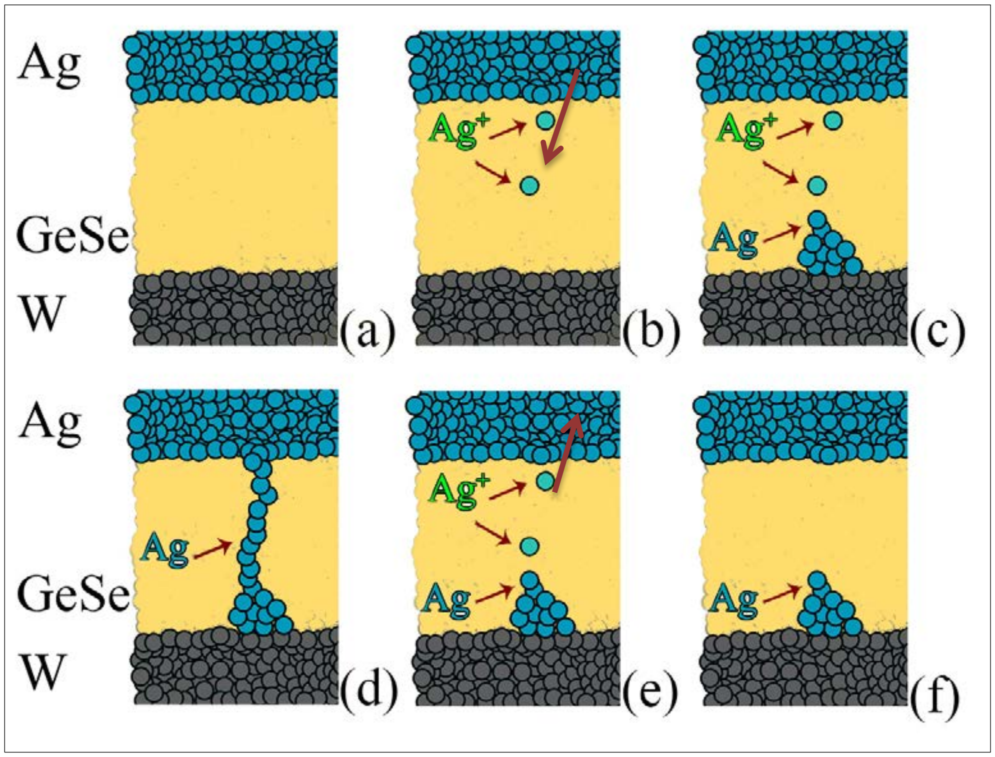


Figure 4. An example of a MIM ion-conducting memristor structure with an active Ag metal layer (top), a GeSe amorphous glass (center) and a W metal layer (bottom). (a) Immediately after fabrication, the device is very high resistance and there is no movement of the Ag into the GeSe layer. (b) After applying a slight positive voltage bias to the top electrode Ag^+ ions move into the glass. (c) As the bias increases, more Ag^+ ions move into the glass and some reduce to begin forming an Ag base. (d) When enough Ag^+ ions reduce to form an Ag filament, the conductance of the device is greatly decreased. (e) By applying a negative potential to the top electrode, most of the Ag metal in the channel again ionizes into Ag^+ and reduces at the top electrode, thus severing the conductive filament and increasing device resistance. (f) Some of the reduced Ag is left behind and makes subsequent filament formation easier.

2.2 STDP in Ideal Memristors

The characteristic ability of memristors to incrementally change their conductance as a function of the potentials applied to the device is crucial for their use as a synapse in STDP. Ideally, sub-threshold potentials will not affect the conductance of the memristor [22], which allows non-destructive verification of device state. Similarly, the strength of a synapse's connection, also known as its weight, does not change with sub-threshold action potentials [17]. Figure 5 shows the relationship between a biological synapse and a memristor, where the neuron circuits are labeled as somas [23].

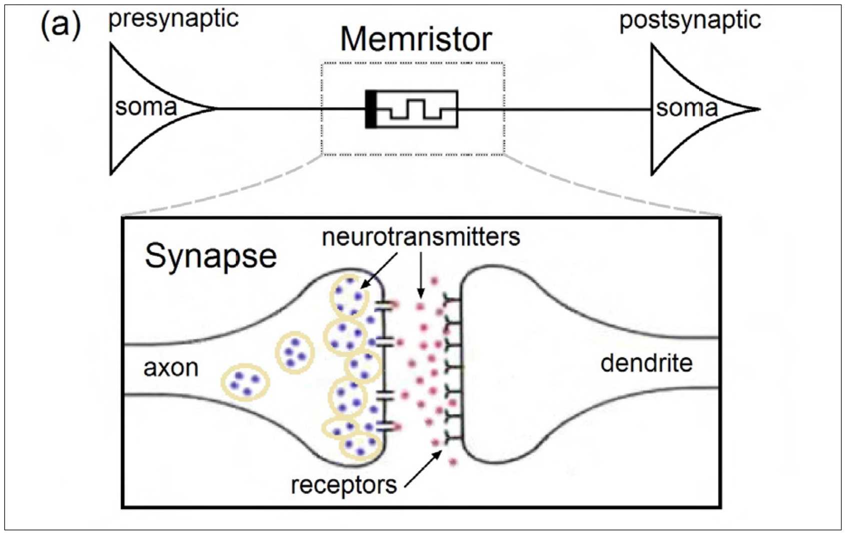


Figure 5. Schematic diagram showing the analogous relationship between the memristor and the synapse [23].

In the biological system, the relative timing of sub-threshold action potentials present on the axon and dendrite membranes can create a pro-threshold action potential capable of affecting the synapse's strength [17]. These potentials, also known as spikes, are sourced by the presynaptic (axon) and postsynaptic (dendrite) neurons, and the total synaptic action potential is the difference between the membrane action potentials [17]. Similarly, in the memristive system, if each neuron circuit fires a sub-threshold voltage pulse (spike) at slightly different times, the interaction of these potentials across the memristor can create a resultant voltage that is above the threshold required to affect the device's conductance [18]. When a device increases its conductance, this is referred to as a positive weight change, which is analogous to synaptic potentiation. A decrease in device conductance is referred to as a negative weight change.

In this work, a pro-threshold resultant voltage is created when the presynaptic and postsynaptic neurons send identical voltage pulses at slightly different times, denoted by ΔT. The notation for ΔT is such that a positive ΔT indicates that the postsynaptic neuron fired before the presynaptic neuron, and vice versa [18]. If ΔT is greater than the time that the pulse is active, the presynaptic (Vpre) and postsynaptic (Vpost) voltage pulses do not interact, and the resultant voltage (V_{MR}) across the synapse (analogously, the memristor) will not have a magnitude greater than the individual pulses. Figure 6 shows the interaction of presynaptic and postsynaptic voltage pulses with positive and negative ΔT values, and their subsequent creation of a resultant voltage larger than the individual pulses.

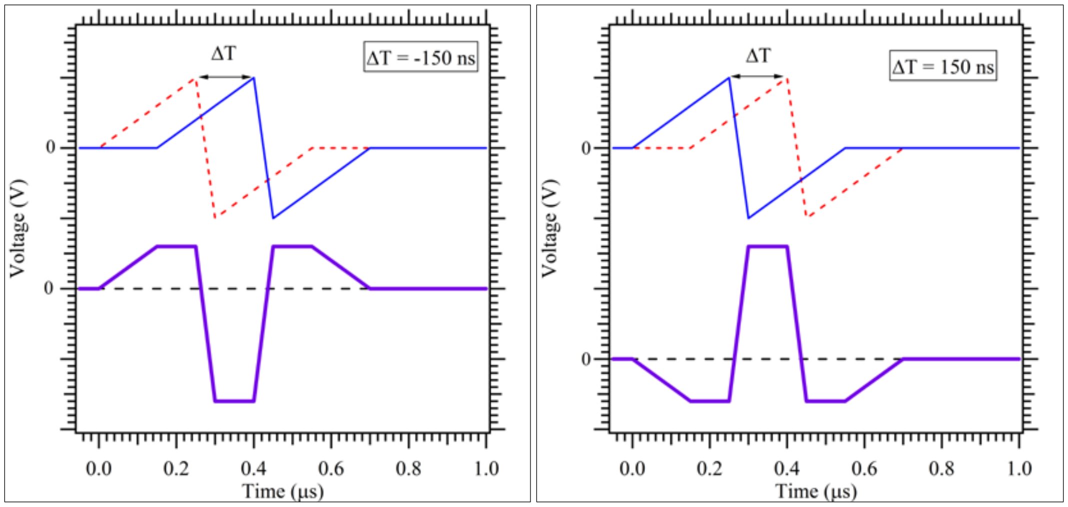


Figure 6. Differences in resultant voltage due to pulse timing. Vpre is shown in dashed red, Vpost in solid blue, and the resultant V_{MR} is the thick purple trace.

The function governing the change in conductance of the memristor is generically denoted as *f*(V_{MR}), which is a function of the current (I_0) and the polarity of the voltage applied to the memristor [18], where

$$f(V_{MR}) = \begin{cases} I_0 \times sign(V_{MR}) & if \quad |V_{MR}| > v_{th} \\ 0 & otherwise \end{cases}. \tag{1}$$

The efficacy of these pulses is known as the “learning function,” which is given by ξ. It is commonly represented as a function of the time differential between pulses, and is equivalent to:

$$\xi(\Delta T) = \int f(V_{MR})\, dt \ . \tag{2}$$

Equations 1 and 2 state that for sub-threshold pulses, the change in conductance of the memristor should be 0, therefore the learning function should be 0. This occurs when the presynaptic and postsynaptic pulses don’t interact to form a pro-threshold resultant voltage. For pro-threshold voltage pulses, the sign of the region in which the resultant waveform is greater than the device threshold determines how the weight is updated.

Figure 7 shows the resultant voltage and pro-threshold region caused by well-timed, sub-threshold pulses. The area highlighted in red indicates the time during which the weight of the device is adjusted by the resultant waveform; device weight is decreased for a value less than the negative threshold and the weight is increased for a value above the positive threshold.

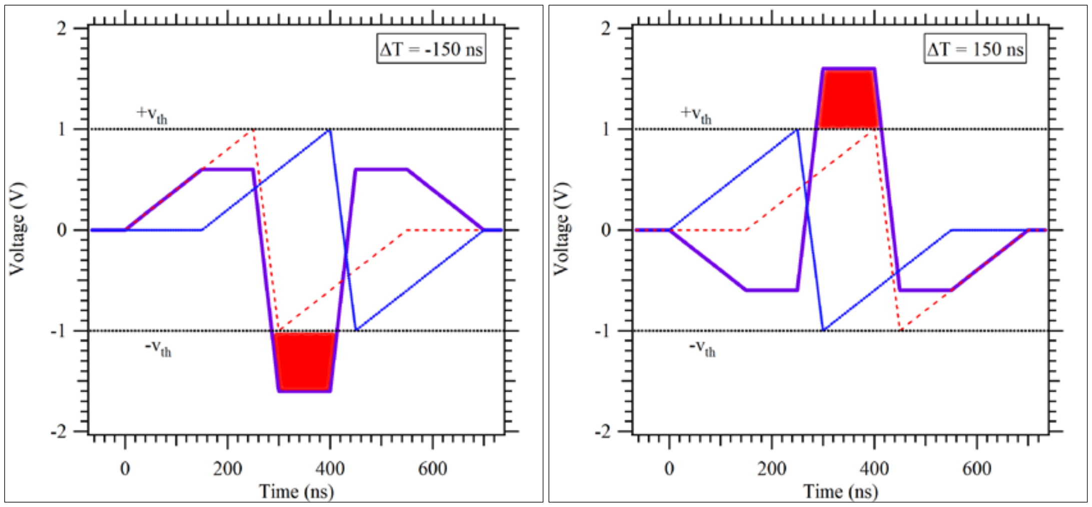


Figure 7. Positive and negative weight adjustments due to pulse timing. Threshold Voltage, Vth, is ± 1 V. Vpre (top electrode) is shown in dashed red, Vpost (bottom electrode) in solid blue, and V_{MR} is the thick purple trace. The resultant waveform is highlighted where it is above device threshold.

The shape of the resultant waveform is entirely dependent on the shape of the individual pulses, which in turn affects the shape of the learning function. Serrano-Gotarredona et al. explored simulations of various pulse shapes (V_{mem} in their notation) and their learning functions, two of which are shown in Figure 8 [22]. They propose that the ability to tune the STDP pulse shape and the corresponding learning function is essential for getting STDP to work with different material systems and circuit topologies [22]. This is significant in that it shows that various pulse shapes can still fall under the umbrella of STDP, including those that have been optimized for programming the ion-conducting memristors in this work. The results and analyses of the STDP experiments performed in Chapter 4 show the correlation between our pulse shape and the calculated learning functions.

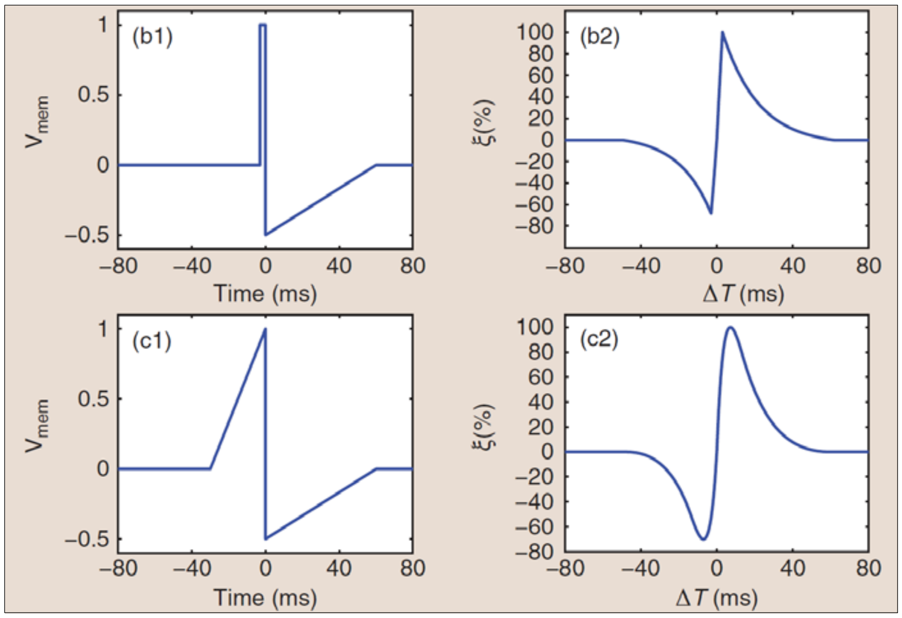


Figure 8. Two STDP pulse shapes and their corresponding learning functions. Inset (b1) shows a sharp positive square pulse followed by a longer negative ramp, which results in the sharp transitions shown in the learning function in inset (b2). Inset (c2) shows rounded transitions as a result of the pulse shape in (c1), which extends the positive pulse with a ramp [22].

CHAPTER THREE: EXPERIMENTAL OVERVIEW

The goal of this work is the characterization of a type of chalcogenide-based ion-conducting memristive device in a test environment that mimics the circuit outlined in Fig. 5. The experiments have been performed over a range of pulse timing parameters with a ΔT ranging from 50 nanoseconds to 950 milliseconds.

This chapter gives an overview of the structure and fabrication of the ion-conducting memristors used in this work, the terminology and naming conventions used in the experimental analyses, as well as the measurement tools used to perform the experiments.

Four primary experiments were performed for this thesis. They are:

1. The AC Pulsing experiment is a set of tests designed to show that the device can be programmed with the minimum timing window present for the Sub-Microsecond STDP experiment.
2. The Sub-Microsecond STDP experiment was the first STDP experiment performed for this work, and seeks to investigate the functionality of these memristors as circuit analogs to biological synapses.
3. The Trailing Edge Cancellation experiment is a refinement of the one-sided STDP experiment, which uses a modified STDP resultant waveform to improve device programming characteristics.
4. The Extended ΔT experiment is an extension of the sub-microsecond STDP experiment, which features slower pulses that extend the ΔT range into the hundreds of milliseconds. We hope to show that these memristors are able to

demonstrate learning over a broad range of timing windows including those found in biological synapses.

3.1 Device Structure and Fabrication

The resistive memory used in this work was manufactured in the Idaho Microfabrication Laboratory at Boise State University using technology available from the U.S. patent and trademark office, referred to as a resistance variable memory device, or programmable conductor [24] [25]. All processing steps were performed at Boise State University in the Idaho Microfabrication Laboratory. The photomasks used in this work were fabricated by HTA Photomask [1605 Remuda Lane, San Jose, CA 95112]. The device structure is shown in Figure 9 and Figure 10 shows a top-down view of an actual device. This structure contains an easily oxidized Ag metal layer, an amorphous chalcogenide layer of Ge_2Se_3, and W top and bottom electrodes.

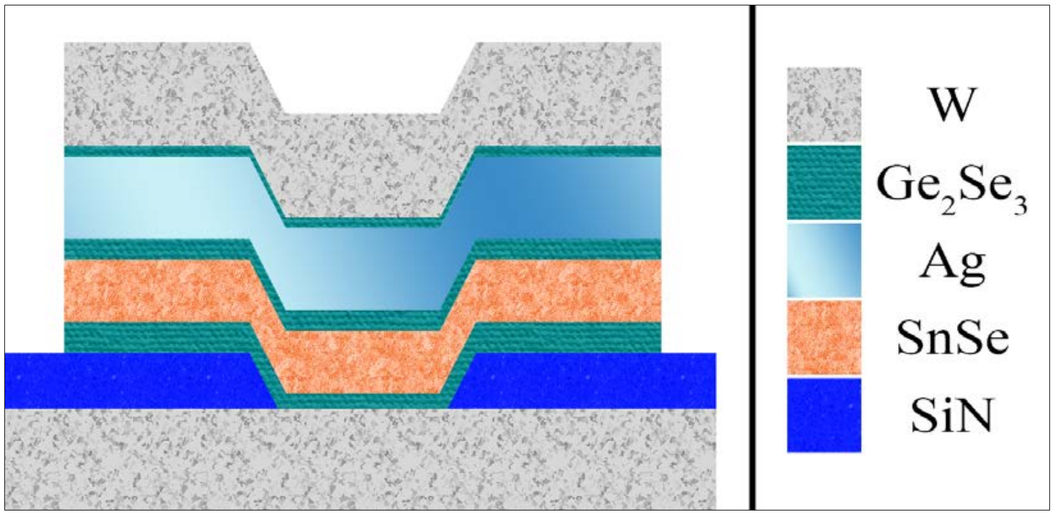


Figure 9. The ion-conducting resistive memory device stack featuring tungsten top and bottom electrodes, an active metal layer of Ag, and an active Ge_2Se_3 chalcogenide glass layer between the conductor layer and the bottom electrode [24] [25].

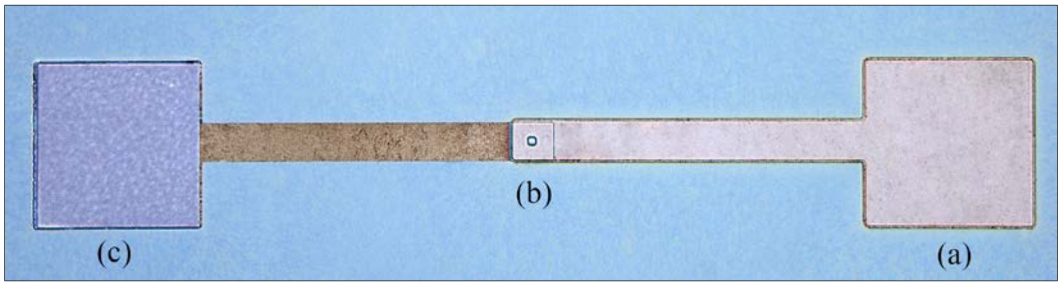


Figure 10. Top-down view of a fabricated 4 µm device with a top electrode (a), device via (b), and a bottom electrode (c).

Ion-conducting devices were fabricated with a via structure and top and bottom electrodes as shown in Figure 9, each of which extends to a metal pad for wirebonding or electrical probing access as shown in Figure 10. The structure consists of, in thin film layer order from bottom to top electrode, 600 Å W/300 Å Ge_2Se_3/800 Å SnSe/150 Å Ge_2Se_3/500 Å Ag/100 Å Ge_2Se_3/380 Å W. The active switching layer is the 300 Å Ge_2Se_3 layer adjacent to the bottom electrode.

The devices were fabricated on 100 mm p-type Si wafers. Isolated W bottom electrodes were patterned on the wafers and a nitride layer was used for device isolation. Vias were etched through the nitride layer to provide contact to the bottom electrode and to define the device active region. This via defines the device size and ranges from 1 µm to 30 µm. No difference in the electrical response was observed between the differing device sizes, therefore the 4 and 5 µm devices were used throughout the work presented here. The wafers were sputtered with Ar^+ to clean the W electrode followed by in-situ deposition of all devices stack layers using an AJA International ATC Orion 5 UHV Magnetron. The Ge_2Se_3 and SnSe targets were from Process Materials [5625 Brisa Street, Livermore, CA 94550]. Etching was performed with a Veeco ME1001 ion-mill by etching through the W and the memristor device materials and stopping on nitride. The top and bottom electrode bond pad contacts were 80 µm x 80 µm.

3.2 Device Programming Characteristics

The devices in this work typically have an initial resistance of more than 1 GΩ immediately after fabrication and can be programmed to less than 100 Ω. The ion-conducting memristors in this work decrease resistance when programmed by applying a positive voltage to the top electrode above their threshold voltage. Conversely, these devices dramatically increase their resistance when a negative potential exceeding the erase threshold voltage magnitude is applied to the top electrode. Figure 11 shows that the positive threshold is approximately 250 mV for DC "Write" operations and approximately -175 mV for DC "Erase" operations. Under AC pulsing conditions, however, the voltage required to affect the device increases as the width of the applied pulse decreases [4], but the polarity remains the same.

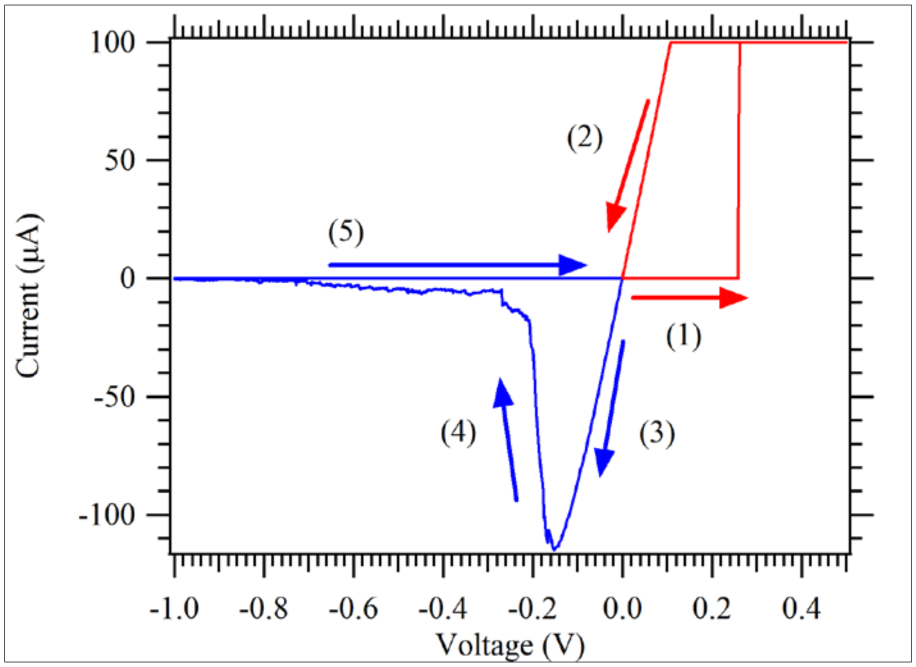


Figure 11. An example of a pinched I-V hysteresis loop for an ion-conducting chalcogenide memristor used in this work. Starting with a sweep to 0.5 V (1) showing a very high initial resistance and a large increase in conductivity at approximately 250 mV followed by (2), a reverse sweep showing the lower resistance state. The low resistance state is maintained during (3), a sweep to -1 V until (4)

where the device decreases in conductivity. The high resistance state is retained during the reverse sweep shown by (5).

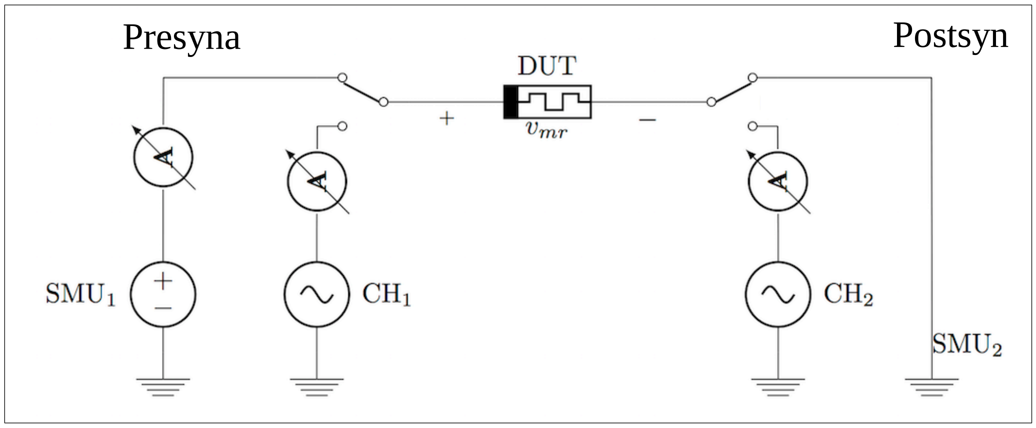
3.3 Electrical Characterization

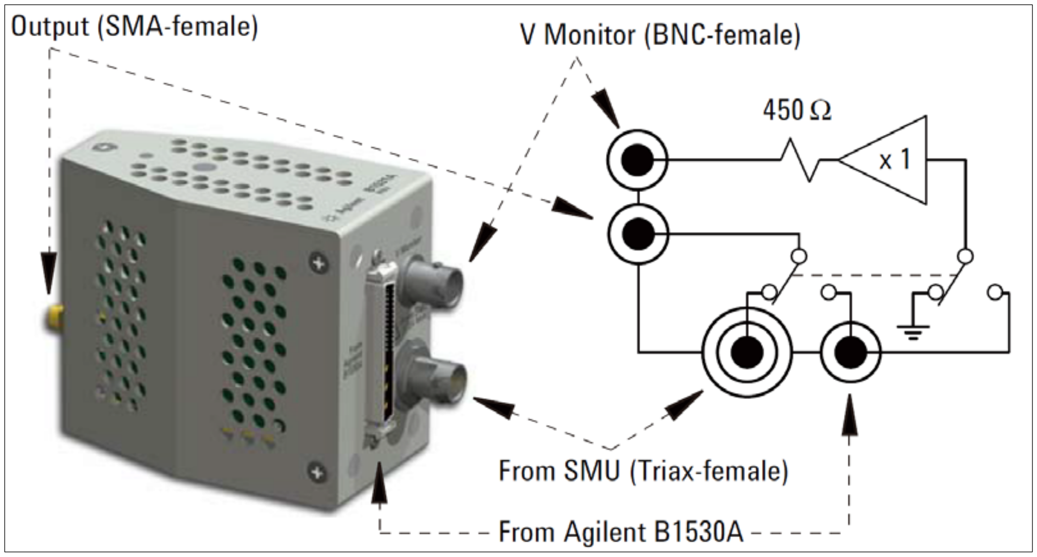
All probing was performed with a MicroManipulator 6200 microprobe station resting on a Technical Manufacturing Corporation MICRO-g air table for vibration reduction. Electrical measurements were performed with an Agilent B1500 Semiconductor Parameter Analyzer equipped with two B1511A medium-power Semiconductor Measurement Units (SMU) for DC measurements and a two channel B1530A Waveform Generator/Fast Measurement Unit (WGFMU) with two B1531A Remote-sense and Switch Units (RSU) for AC (alternating current) pulsing measurements.

The two-channel WGFMU is a self-contained module with each channel able to generate arbitrary linear waveforms with a 10 ns minimum time step. Each channel can also simultaneously measure current or voltage with a variety of options for measurement range and speed, and the channels share a common ground. The ability to simultaneously apply a test voltage while measuring voltage and current makes it a good tool for rapidly observing changes in device under test (DUT) resistance. Each channel of the WGFMU is connected to an RSU located near the probes to improve timing and sourcing. Additionally, each RSU features a switch that allows a direct connection from the SMUs to the DUT to facilitate high precision DC measurements without the need to lift the probes, which could disturb the device state. Figure 12 shows a block diagram of the electrical connections for this test setup [26].

The measurement equipment was controlled over GPIB with a Visual C++ console program that was developed for these experiments to provide a platform for rapid device testing and data acquisition. This code is provided in Appendix A.

(a)





(b)

Figure 12. (a) Block diagram showing the electrical connections for circuit test set up. Each RSU (CH$_1$, CH$_2$) features a buffer that is connected to an oscilloscope for monitoring the voltage at the top and bottom electrodes. Each channel of the WGFMU can monitor the current through or the voltage applied to the device. (b) Block diagram and picture showing the RSU’s switch configuration between WGFMU (B1530A) and SMU [26].

3.3.1 Calculating Weight Change

To remain consistent with neuromorphic literature, device state is reported most often as ΔG or ξ, where Δ represents "change in," G represents conductance (Ω^{-1}), and ξ is the synaptic weight update function in %. A device is said to increase its weight if its conductance increased, and decrease its weight if its conductance is decreased. The learning function is empirically calculated as:

$$\xi(\Delta T) = \frac{(G_{post} - G_{pre})}{G_{max}} \times 100\ \%\ . \tag{3}$$

G_{pre} is the conductance before an STDP pulse is sent, G_{post} is the conductance after the STDP pulse is sent, and G_{max} is the maximum conductance value of the device during each ΔT test. Dividing by the maximum conductance normalizes the range of ξ to be within ±100 %.

3.3.2 Minimum Timing Considerations

When creating the pulse shapes for the AC pulsing and Sub-Microsecond ΔT experiments, consideration had to be made for the recommended minimum pulse width of a waveform sourced by the WGFMU, which is 145 ns [26]. The shape that was created for the fast tests (Sub-Microsecond ΔT) has subsequently been stretched for the Extended ΔT STDP experiment; its rise and fall times have been adjusted so that the overall pulse shape remains the same for each ΔT sequence.

3.3.3 Resistance Measurements

While the WGFMU is a very good tool for programming and rapidly observing changes in device state, its ability to accurately measure resistance is limited by the resolution of the ammeter. Additionally, when the DUT rapidly changes its resistance, the impedance matching provided by the WGFMU and RSU is invalid until the tool can “catch up.” This can cause ringing when the device significantly changes its resistance (see Figure 13(d)), which further reduces measurement accuracy [26].

The two SMUs are much more accurate for measuring resistance than the WGFMU, but they are also much slower. Thus, every AC programming pulse is preceded and succeeded by a DC “Read” sweep from 0 to 20 mV sub threshold sweep sourced by the SMUs for accurate resistance measurements.

3.3.4 Two-sided vs. One-sided Measurements

A two-neuron, single-synapse neuromorphic circuit contains the memristor between the two neurons. The programming voltages sourced by the neurons must be bipolar, and each neuron is responsible for referencing its output voltage from a common ground and the output of the other neuron. This implies a requirement for each neuron to be a fully bipolar programming circuit, which significantly increases the footprint of the overall neuromorphic circuit.

With this in mind, the AC characterization has been performed in two ways. The two-sided measurement directly mimics the original neuromorphic circuit by using both WGFMU channels as pulse sources. The one-sided measurement, however, involves a pre-programming calculation of the resultant waveform, which is then sourced from Channel 1 (top electrode) exclusively, while the bottom electrode of the device is connected through Channel 2, which is set to force 0 V. The resultant waveform is the simple subtraction of the separate waveforms, as shown by V_{MR} in Chapter 2. This one-sided pulsing is intended to show that these devices do not require two fully differential neurons for neuromorphic applications, which should dramatically simplify the required circuitry.

3.3.5 Device Conditioning

Given that the formation and destruction characteristics of conductive filaments are dependent on the device's initial conditioning [27], every device probed for this work has been programmed with a set of AC conditioning pulses or gentle DC sweeps to initialize its state. Under DC bias conditions, as shown in Figure 11, 300 mV can be large enough to affect the state of the device. Under AC pulsing conditions, however, the voltage required to affect the device increases as the width of the applied pulse decreases [4].

For some of the experiments in this work, conditioning was performed using the AC Conditioning waveform sourced by the WGFMU and repeated 10 times immediately after breaking through the photo resist on each device. These tests ensure that if the results of this work are carried forward into actual circuit design, extra circuitry, such as a current-limiting SMU, will not be required to initialize the devices. One caveat with the AC Conditioning is that it is quite harsh; it can send as much as 10 mA of current through the device, which places the device into a very low resistance state. While this device state works for the STDP experiments performed in this thesis, it is not the normal operating region for these devices. For the Extended ΔT STDP experiment, however, the device was programmed with a gentler DC conditioning sweep from 0 to 1 V with a 10 µA compliance current. No additional AC conditioning was used.

The AC conditioning waveform shown in Figure 13 features a low-amplitude, 200 mV 600 ns full-width-half-max (FWHM) read pulse before and after the two programming pulses, which are a -3.5 V 300 ns FWHM Erase pulse and a +2 V, 150 ns

FWHM Write pulse. Voltage data is measured on Channel 1 of the WGFMU, and current data is measured on Channel 2 (hence the negative current for positive bias).

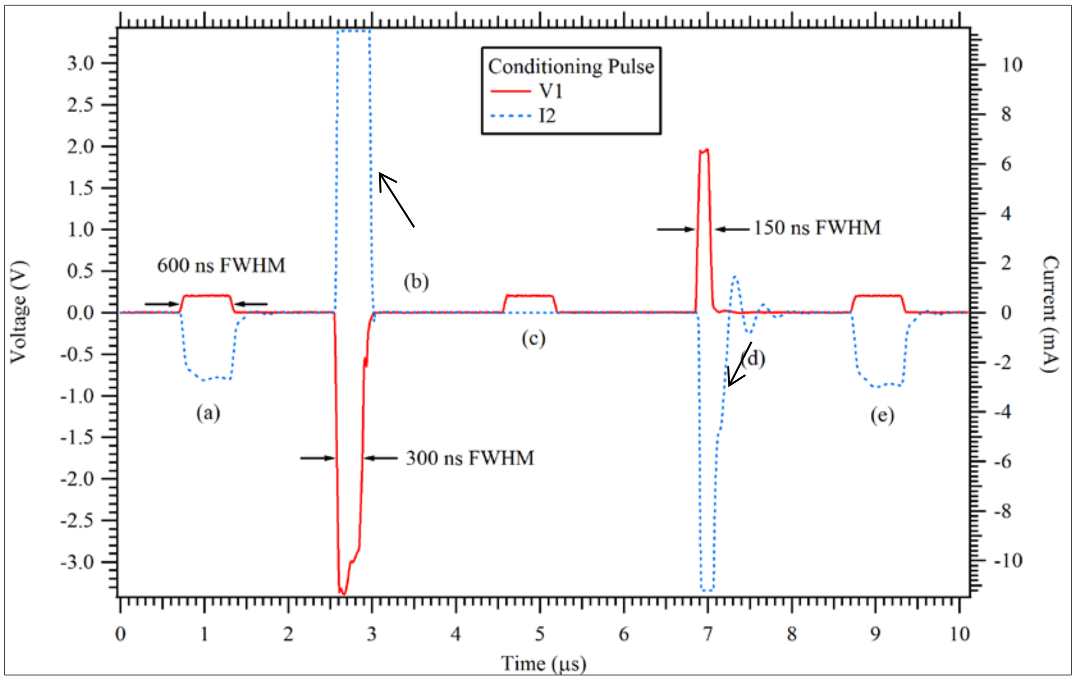


Figure 13. The AC conditioning pulse for the devices in this work and the typical device response. Voltage is measured from Channel 1 (V1, solid red line) and current is measured from Channel 2 (I2, dashed blue line). Shown at (a) is the first 200 mV 600 ns read, (b) is the -3.5 V 300 ns FWHM erase pulse, (c) is the erase-verification read, (d) is the 2 V 150 ns FWHM write pulse, and (e) is the write-verification read.

The large current in Fig. 13(a) shows that the device is already programmed to a relatively low resistance from a previous conditioning pulse because the current response is in the milliamp range for an applied voltage of 0.2 V. Fig. 13(b) shows the clipped current response of the much larger amplitude erase pulse. This pulse greatly increases the resistance of the device, which results in a very low current response to the second read at Fig. 13(c). Fig. 13(d) shows the ringing response of the ammeter when the device switches from the high resistance shown in Fig. 13(c) to the low resistance shown by Fig. 13(e). The typical resistance of a device after this type of conditioning is less than 500 Ω.

Again, it should be reiterated that this harsh conditioning is not how the devices typically operate, but is a way that we chose to use them for this study.

3.4 Motivation

The over-arching goal of this work is the characterization of these memristive devices in a test environment that mimics the theoretical neuromorphic circuit outlined in Chapter 2. Particular emphasis is placed on sub-microsecond pulse widths to demonstrate rapid adjustments of device conductance. Optimization of these pulsing parameters allows us to demonstrate that these devices are capable of state modification at speeds much faster than their biological counterparts.

CHAPTER FOUR: EXPERIMENTAL RESULTS AND DISCUSSION

This chapter includes the details of how and why each of the four experiments described in Chapter 3 were performed as well as a discussion of their results.

4.1 AC Pulsing

The AC pulsing experiment is a set of tests designed to verify that the pulsing parameters and timing requirements for the Sub-Microsecond STDP experiment are compatible with these memristors for both one- and two-sided tests.

4.1.1 Experimental Setup

A series of one- and two-sided pulses were applied using the same timing parameters (50 ns rise/fall time) as the waveform shown in Figure 13, but with write and erase pulse amplitudes of ± 1.8 V as shown in Figures 14 and 15. In the two-sided pulses, negative resultant voltages were created by applying a positive bias to the bottom electrode of the device and positive resultant voltages were created by applying a positive bias to the top electrode of the device. One-sided pulses used both positive and negative biases applied to the top electrode with 0 V applied to the bottom electrode. The state verification pulse (Fig. 14(a) and (c)) is used as an indicator of device state change, but is not as accurate as the measurement provided by the SMU, thus the device has its conductance measured using DC sweeps from 0 to 20 mV before and after each pulse, and the change in conductance is used to calculate the weight update function ξ.

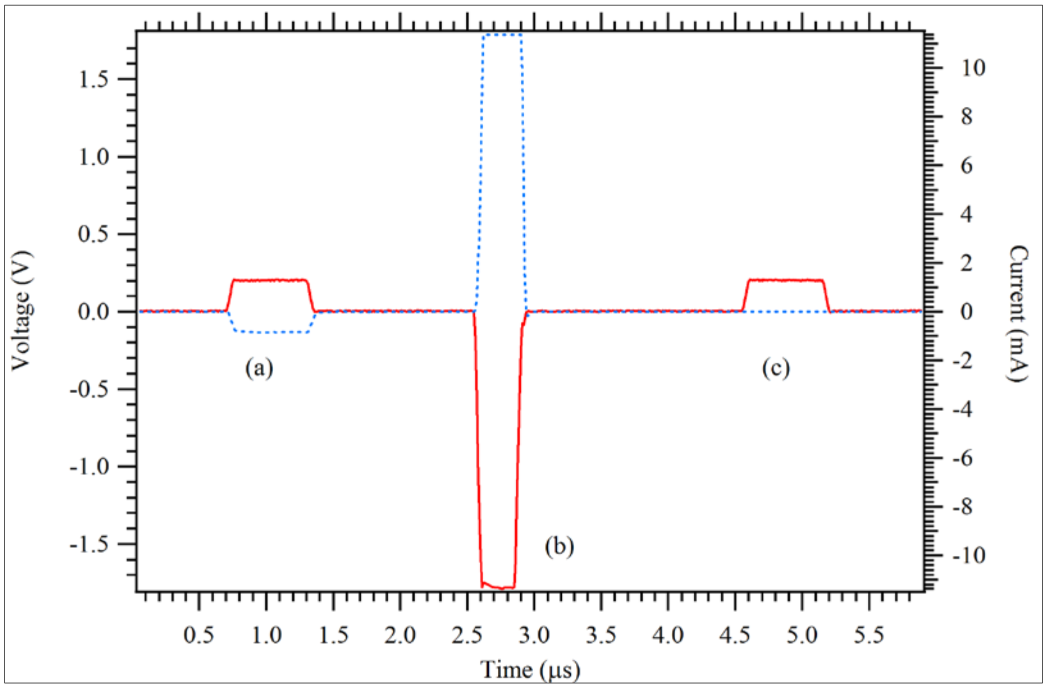


Figure 14. **AC Erase pulse and typical device response with Voltage as the solid red line and Current as the dotted blue line. (a) An initial low-resistance indicated by a large current. (b) The -1.8 V 300 ns FWHM erase operation. (c) The device has been programmed to a higher resistance as indicated by the decreased current.**

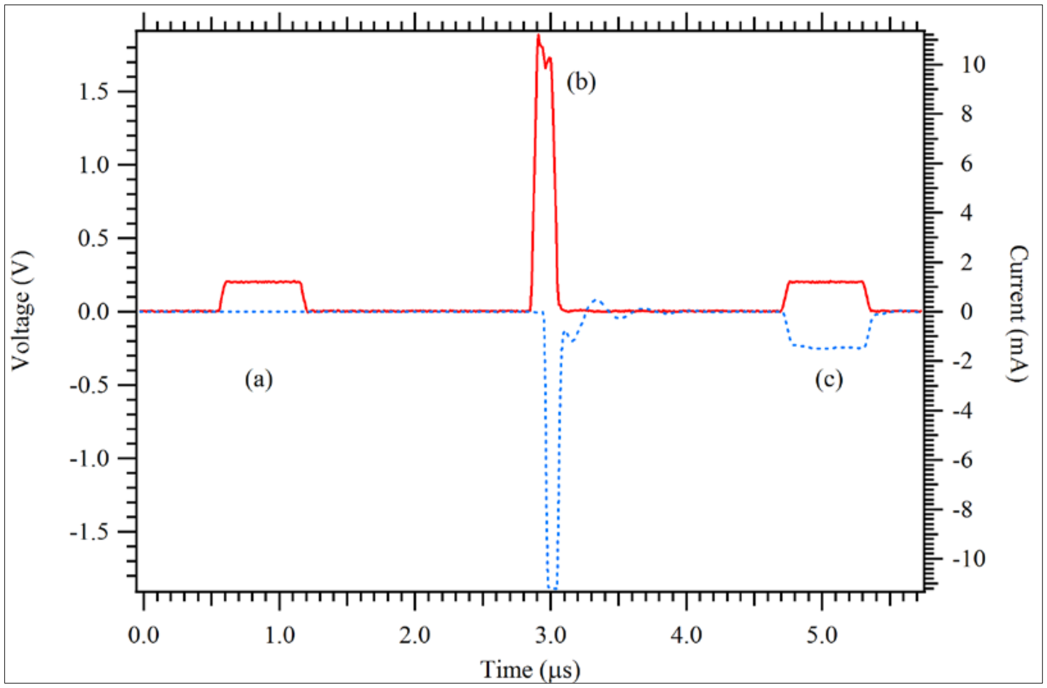


Figure 15. **AC Write pulse and typical device response with Voltage as the solid red line and Current as the dotted blue line. (a) An initial high-resistance indicated by a very low current. (b) The 1.8 V 150 ns FWHM write operation. (c) The device has been programmed to a lower resistance as indicated by the increased current.**

The full test sequence performed on each device for this experiment is as follows:

1. AC Conditioning Pulse Repeated 10 times
2. One-Sided AC Erase Pulse
3. One-Sided AC Write Pulse
4. AC Conditioning Pulse Repeated 10 times
5. Two-Sided AC Erase Pulse
6. Two-Sided AC Write Pulse

4.1.2 Results

The AC Pulsing Experiment was performed on a set of three 5 µm devices immediately after conditioning. The results in Figure 16 show that the one- and two-sided AC pulsing program in a similar fashion. The one-sided pulses appear to have a smaller standard deviation, but the average weight update function for each operation is within a single standard deviation for both one- and two-sided pulses.

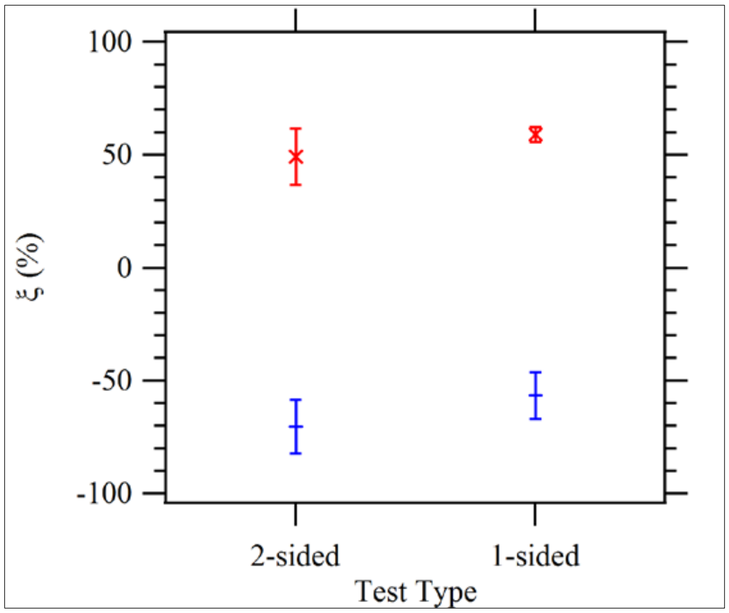


Figure 16. Results from the AC Pulsing test. Marker indicates the average, error bars are ± 0.5 standard deviations for three 5 µm devices. Results from the AC erase

pulses (– 1.8 V) are indicated by a blue dash and the results from the AC write pulses (+ 1.8 V) are indicated by a red “X”.

4.1.3 Discussion

These results show that a ±1.8 V, 150 ns FWHM pulse is able to significantly affect the conductance of these memristive devices in both directions (increasing and decreasing). It also confirms that both one- and two-sided measurements affect the device state in a similar manner.

4.2 Sub-Microsecond STDP

As discussed in Chapter 2, the potentiation provided by STDP is effective when the amplitude of the resultant is greater than the programming threshold of the device. This occurs when the normally sub-threshold pulses “fire together,” effectively amplifying their individual magnitudes by combining to form a larger magnitude waveform. As the first experiment performed demonstrating STDP programming functionality, this experiment seeks to demonstrate rapid adjustments to device conductance under STDP pulse conditions.

4.2.1 Experimental Setup

The STDP pulse shape was chosen to be a symmetric synaptic pulse with a ± 1 V amplitude as shown in Figure 17. Table 1 shows the timing convention used for ΔT and includes the expected response of a memristive device and is repeated below. ΔT is the difference in time from when the postsynaptic neuron fires to when the presynaptic neuron fires. For ΔT = 0, the resultant waveform is 0 V at all points and thus the device should remain unaffected.

Table 1. Polarity of ΔT for applied STDP pulse and expected device response.

ΔT	**First Neuron to Fire**	**Weight Update (ξ)**	**Conductance (G)**	**V_{MR}**	**Equivalent AC Operation**
$\Delta T > 0$	Postsynaptic	Positive	Increase	Mostly Positive	Write
$\Delta T < 0$	Presynaptic	Negative	Decrease	Mostly Negative	Erase

The maximum amplitude of the resultant is ± 2 V with a ΔT of ± 50 ns, respectively. The STDP pulse shape is non-zero for 550 ns, so the V_{MR} when ΔT is greater than ± 550 ns does not reflect interaction between V_{pre} and V_{post}. For this reason, the maximum ΔT tested is ± 600 ns. Figure 18 shows the resultant V_{MR} for a few ΔT values. Note that the V_{MR} is equal in magnitude but opposite in polarity for each pair of positive and negative ΔT values.

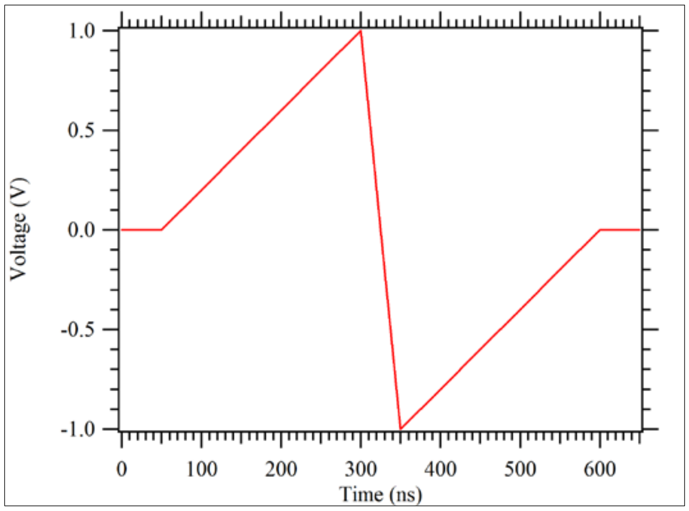


Figure 17. The STDP Pulse shape used for this experiment. The rise time from 0 to +1 V is 250 ns, the transition from +1 to -1 V is 50 ns (the minimum ΔT), and the rise time from -1 to 0 V is 250 ns.

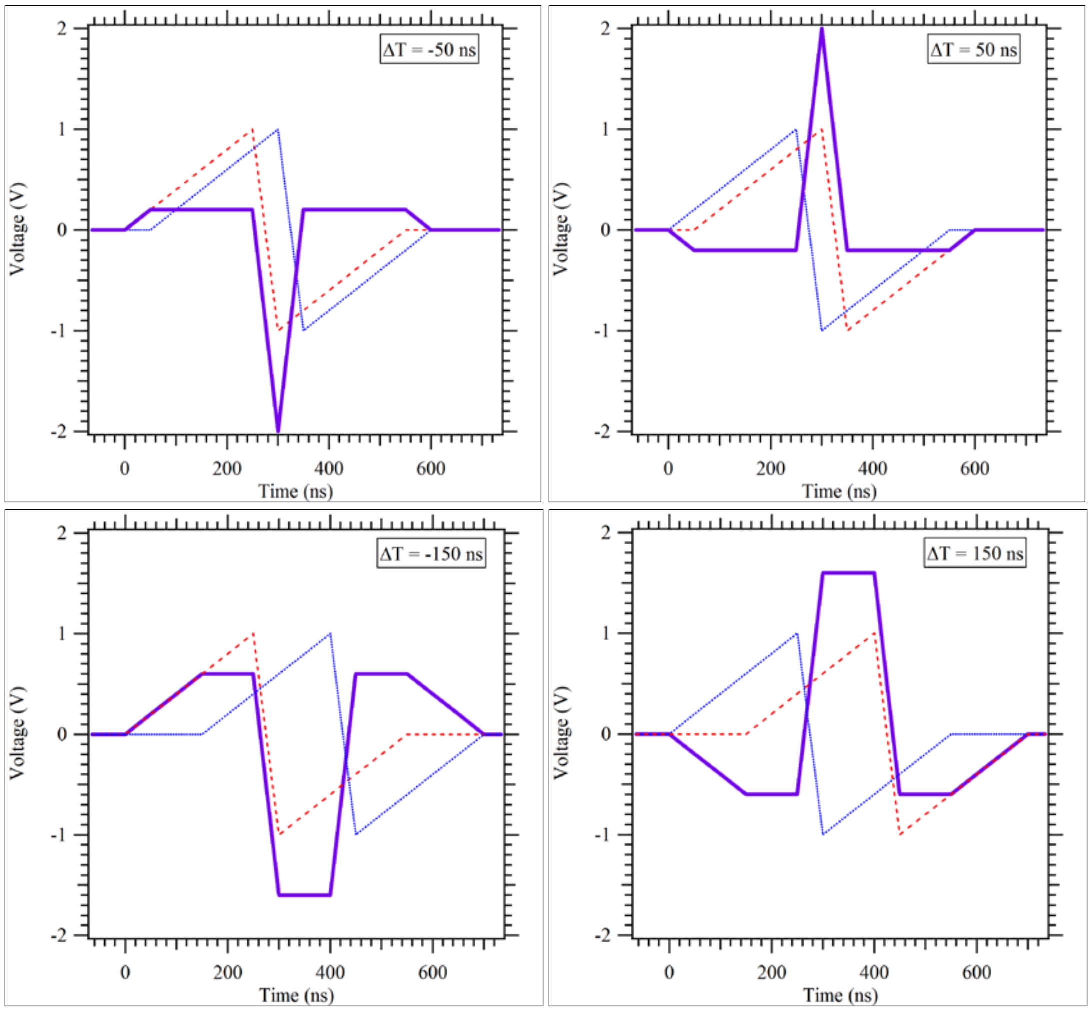


Figure 18. The resultant V_{MR} for various ΔT values. V_{pre} (top electrode) is shown in dashed red, V_{post} (bottom electrode) in solid blue, and V_{MR} is the thick purple trace.

The STDP test sequence begins with a ΔT = 0 ns (where V_{MR} is zero), and increases the ΔT by increments of ± 50 ns, i.e. ΔT = 0, -50 ns, 50 ns, -100 ns, 100 ns… up to ± 600 ns. This is referred to as the "ΔT sequence" or the "STDP test sequence" interchangeably.

One-Sided STDP is performed with the same sequence of ΔTs, but instead forces the calculated resultant V_{MR} for each ΔT at the top electrode of the device. The device has its conductance measured using DC sweeps before and after each pulse, and the change in conductance is used to calculate the weight update function ξ.

The full test sequence performed for this experiment is as follows:

1. AC Conditioning Pulse with an erase voltage of -4 V and a write voltage of 2 V, repeated 10 times.
2. Two-sided or one-sided ΔT sequence (three devices per sequence).

4.2.2 Results

Performed on two sets of three 5 µm devices after AC conditioning, which placed the device in a low resistance state (as verified by a pre-STDP pulse DC Read sweep), the results in Figures 19 and 20 confirm the expected result: STDP can be performed in both one- and two-sided circuit topologies. The results are fairly similar for both experiments, with the two-sided STDP experiment showing slightly tighter standard deviations than the one-sided experiment.

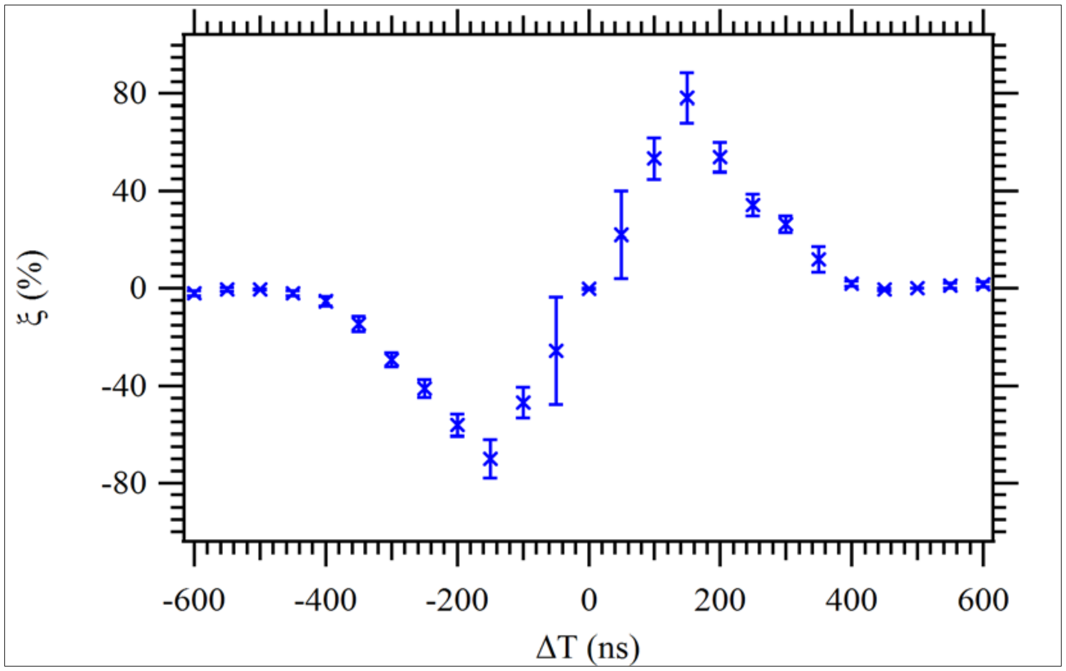


Figure 19. Results from the two-sided STDP experiment. Marker indicates the average, error bars are ± 0.5 standard deviations over 3 devices.

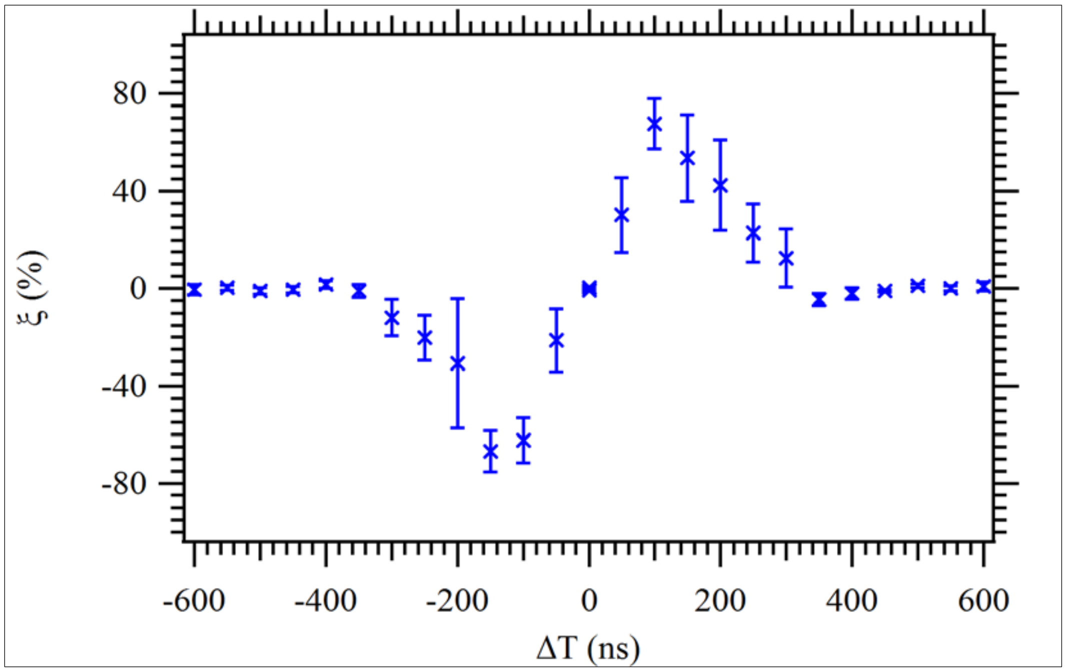


Figure 20. Results from the one-sided STDP experiment. Marker indicates the average, error bars are ± 0.5 standard deviations over three devices.

4.2.3 Discussion

For ΔT = 0, the resultant V_{MR} is 0 at all times because the pre- and post-synaptic pulses perfectly overlap one another and cancel each other out. As expected, there is no observed change in device conductance when the voltage across the device remains unchanged.

The learning function peaks near ΔT = ± 150 ns for both tests, with an average magnitude of ± 80 %. As shown in Figure 18, when ΔT = 150 ns, the resultant V_{MR} contains a 1.6 V, 160 ns FWHM pulse. As the opposing polarities in the post- and pre-synaptic pulses V_{post} and V_{pre} align, the resultant V_{MR} is effectively magnified.

For |ΔT| > ± 150 ns, the learning function falls off until it finally terminates at around ± 400 ns. This is due to the lack of the “magnification” of the resultant STDP pulse when the pulses are spread far enough apart so that they do not interact with one another; for ΔTs ≥ ± 300 ns, the resultant $|V_{MR}|$ is smaller than 1 V. As the separation between pulses grows, the resultant V_{MR} begins to look like two individual pulses because V_{pre} and V_{post} no longer overlap.

Given the results showing very little change in weight when the resultant V_{MR} is less than 1 V, it would seem that the estimated threshold of ± 1 V shown in Figure 7 is useful in estimating the effectiveness of an STDP pulse on these devices. Interestingly, the device conductance is affected when the resultant V_{MR} remains at ± 1 V for a greater amount of time than a single pulse would allow.

This verifies the key STDP principle that says learning should occur when pre- and post-synaptic pulses strongly interact, and device state should remain unchanged when they do not interact.

4.3 Trailing Edge Cancellation

The Trailing Edge Cancellation experiment seeks to show an increase in weight adjustment by modifying the shape of the resultant V_{MR}. As is shown in Figure 18, V_{MR} is comprised of a large-magnitude spike in between two small-magnitude spikes of the opposite polarity. This large spike is the workhorse of the STDP pulse; the device conductance should decrease for a V_{MR} whose large spike is negative and vice versa. Based upon the data from the STDP experiment, when the large and small magnitude spikes are of a similar magnitude, the device's conductance is not greatly affected.

4.3.1 Experimental Setup

By decreasing the amplitude of the final small spike (also called the trailing edge), we aim to show that the adjustment of the device's conductance will be greater as the trailing edge's amplitude approaches 0 V. Figure 21 shows the resultant V_{MR} for a ΔT = 250 ns and the four modified shapes. As before, a ΔT = -250 ns has the same shape but opposite polarity. $\Delta T = \pm 250$ ns was chosen because it had a small effect on device state as shown in Figure 20, and we wish to show an increase in pulse efficacy with modification to the resultant V_{MR}.

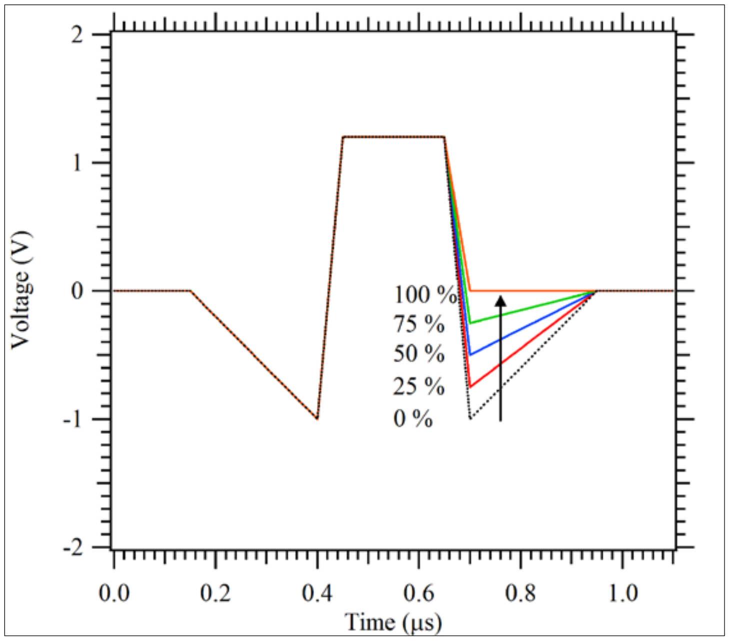


Figure 21. The Trailing Edge Cancellation starting with the standard V_{MR} for ΔT = 250 ns (black, dotted). The magnitude of the trailing edge cancellation is 25 (red), 50 (blue), 75 (green), and 100 % (orange).

After conditioning, the device is exposed to the one-sided resultant for ΔT = -250 ns and ΔT = 250 ns with a full-sized trailing edge (no cancellation). This is followed by the same $\Delta T = \pm$ 250 ns resultants with their trailing edge cancellation factors of 25, 50, 75, and 100 %. This can be visualized as a gradual decrease of trailing edge magnitude as shown in Figure 21. The device has its conductance measured using DC sweeps before and after each pulse, and the change in conductance is used to calculate the weight update function ξ.

4.3.2 Results

Performed on a set of two 5 µm devices for all tests immediately after AC conditioning, the results of the Trailing Edge Cancellation experiment shown in Figure 22 indicate a very strong correlation between the magnitude of the trailing edge cancellation and the impact of the resultant V_{MR} on the weight function. This leads us to the conclusion that for the case where the trailing edge is similar in magnitude to the main pulse, which occurs when the pre- and post-synaptic pulses are not heavily interacting, the last voltage applied to the device has the largest impact on its conductance.

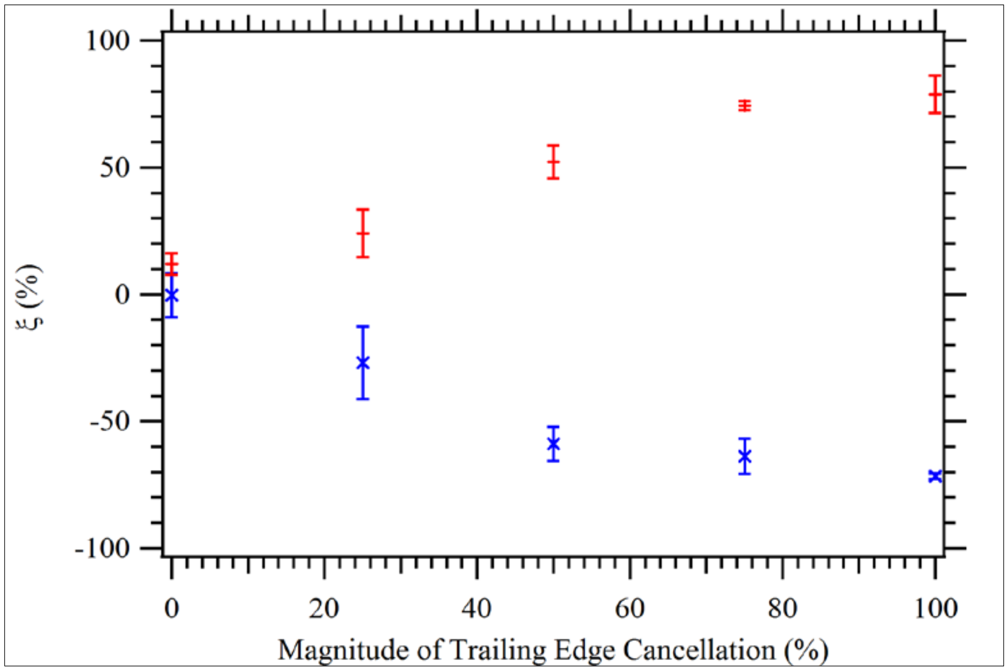


Figure 22. Results from the Trailing Edge Cancellation experiment. Red dash markers represent the ΔT = + 250 ns, blue "X" markers represent ΔT = - 250 ns. Markers indicate the average, error bars are ± 0.5 standard deviations over two devices.

4.3.3 Discussion

The 0 % cancellation has almost no effect on the device's state, similar to the ΔT = ± 250 ns tests in Figure 20, while 100 % cancellation brought the learning function up to 80 % for ΔT = + 250 ns and -70 % for ΔT = - 250 ns. This indicates that the magnitude of the trailing edge, which is opposite in polarity to the "intended" operation, heavily impacts the effectiveness of STDP pulse when the main pulse and the trailing edge are of similar magnitude.

4.4 Extended ΔT STDP

After the Sub-Microsecond STDP experiment was performed, a second STDP experiment was performed that sought to show that these memristors are able to demonstrate learning over a wide range of timing windows, including those found in biological synapses. The results from this Extended ΔT STDP experiment are discussed below.

4.4.1 Experimental Setup

In the Extended ΔT experiment, the STDP pulse features the same symmetric shape shown in Figure 17, but its pulse width and amplitude parameters have been adjusted for programming at four different timing windows covering six orders of magnitude. Table 2 contains the pulse parameters for each ΔT_{min} tested.

Table 2. Extended ΔT Pulse Amplitudes and Timing Parameters.

ΔT_{min}	$\lvert V_{peak} \rvert$	Max $\lvert V_{MR} \rvert$
50 ms	0.2 V	0.4 V
500 µs	0.35 V	0.7 V
5 µs	0.7 V	1.4 V
50 ns	0.9 V	1.8 V

The ΔT sequence starts at ΔT = 0 and increases in multiples of the minimum ΔT up to ± 19*(ΔT). For example, with a minimum ΔT = 50 ms, the range of ΔTs tested is from -950 to +950 ms in increments of ± 50 ms. This provides discretization of test points at each minimum ΔT for easy comparison of learning functions at 20 different points. The device has its conductance measured using DC sweeps before and after each pulse, and the change in conductance is used to calculate the weight update function ξ.

The full test sequence performed on each device for this experiment is as follows:

1. DC Conditioning Sweep 1 from 0 – 1 V, 10 μA compliance
2. ΔT sequence 1: Minimum ΔT = 50 ms, Repeated 10 times
3. DC Conditioning Sweep 2 from 0 – 1 V, 10 μA compliance
4. ΔT sequence 2: Minimum ΔT = 500 μs, Repeated 10 times
5. DC Conditioning Sweep 3 from 0 – 1 V, 10 μA compliance
6. ΔT sequence 3: Minimum ΔT = 5 μs, Repeated 10 times
7. DC Conditioning Sweep 4 from 0 – 1 V, 10 μA compliance
8. ΔT sequence 4: Minimum ΔT = 50 ns, Repeated 10 times

Each repetition of the ΔT sequence uses a maximum conductance from within that sequence's repetition for normalization of weight changes.

4.4.2 Results

Performed on a set of four 4 μm devices with the DC conditioning sweeps in between each ΔT sequence, the results in Figure 23 show very similar learning functions between each ΔT range tested. This test was performed as a double-sided STDP experiment only.

As expected, each learning function tends to peak at $\Delta T = \pm 2$ to $3 * \Delta T_{min}$, and settle to a baseline by $\Delta T = \pm 7 * \Delta T_{min}$. This fall off is due to the lack of interaction between the V_{Pre} and V_{post} pulses, which decreases the magnitude of the resultant STDP pulse when the pulses are spread far enough apart.

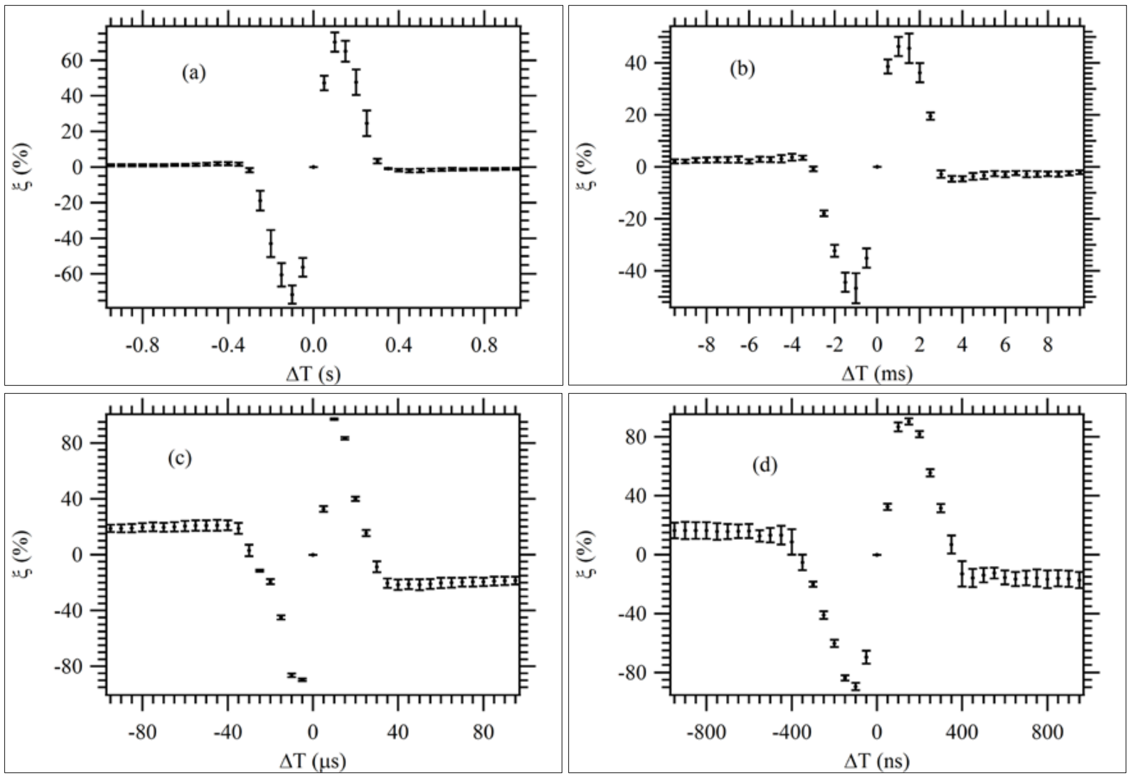


Figure 23. Results from the Extended ΔT STDP experiment for a ΔT_{min} of (a) 50 ms, (b) 500 μs, (c) 5 μs, and (d) 50 ns. Markers indicate the average of 40 pulses (10 per device), error bars are ± 0.5 standard deviations over four devices.

4.4.3 Discussion

In Fig. 23 (a), the ΔT_{min} is 50 ms. The devices show a near-symmetrical STDP learning function that is only active when the STDP pulses are within ± 300 ms of one another; or $\pm 6 * \Delta T_{min}$.

In Fig. 23 (b), however, we can see that the pulses applied slightly affect the device beyond the first six ΔT steps. This effect is magnified in Fig. 23 (c) and (d). The data shows a phenomena best described by an inversion of the learning function for ΔT values greater than $\pm$ 6 * ΔT_{min}. This inversion seems to plateau rather than increasing as a function of ΔT. Given that the trailing edge of a resultant waveform remains unchanged when the pre- and post-synaptic pulses no longer interact, and given the results from the Trailing Edge Cancellation experiment, which indicate a strong relationship between the magnitude of the trailing edge and the effectiveness of an STDP pulse, this leads us to the conclusion that the offset is related to the trailing edge of the resultant STDP pulse.

The trailing edge is opposite in polarity to the expected operation, and the fact that individual pulse amplitude is increased as the ΔT_{min} decreases indicates that the trailing edges created by individual pulses applied to the device are more likely to affect its state. This shows up in the increased deviation of weight updates for Fig. 23 (c) and (d) when $|\Delta T| >= 5 * \Delta T_{min}$ because of the increased effectiveness of this trailing edge.

4.5 Summary of Results

The results presented in this chapter show that STDP can be implemented in the ion-conducting chalcogenide memristors fabricated by Dr. Campbell’s research group. Distinctively, these experiments showed efficacy in both one- and two-sided pulsing topologies at speeds much faster than those found in biology. The addition of one-sided STDP presents a promising avenue for shrinking the circuit complexity and power requirements of a neuromorphic circuit by halving the required programming circuitry.

The extreme rapidity of synaptic weight change presented in this work showed that pulse conditions can be orders of magnitude faster than any publication to date. Biological synaptic updates are typically of the 1 – 100 millisecond time scale, but this work proves that incremental memristive synaptic weight updates can occur from the 50 nanosecond to the 50 millisecond time scales in the ion-conducting chalcogenide memristors fabricated by Dr. Campbell’s research group at Boise State.

CHAPTER FIVE: CONCLUSION

This thesis is the culmination of research and experimentation to understand and implement STDP learning algorithms in physical memristors. This conclusion summarizes the accomplishments of this thesis and ends with some recommended next-steps for further characterization.

5.1 Conclusion and Next Steps

The main goal of this thesis was to explore the switching behavior of Boise State's chalcogenide-based resistive memory fabricated by Dr. Campbell's research group in a test environment that mimics neuromorphic circuitry. The experiments performed for this thesis show that memristive STDP is possible using real devices in a lab environment.

The best next-step would be to deposit and package these devices directly into an integrated circuit with CMOS neurons. This would pull these devices out of the lab and allow exciting combinations of multiple neurons and synapses for use in neuromorphic computing architectures.

More characterization work is required to fill in the gaps of our understanding, but this work establishes an exciting precedent by demonstrating STDP in physical memristors from nanosecond to millisecond time scales.

REFERENCES

[1] L.O. Chua and S. M. Kang, “Memristive Devices and Systems,” Proceedings of the IEEE, 64(2):209 – 223, Feb. 1976.

[2] L. O. Chua, “Memristor – The Missing Circuit Element,” IEEE Trans. Circuit Theory 18, 507, 1971.

[3] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, “The missing memristor found,” Nature (London) vol. 453, p. 80-83, (2008). © Nature Publishing Group 2009, license no. 3580771103241.

[4] L. Chua, “If it’s pinched it’s a memristor,” Institute of Physics Semiconductor Science and Technology vol. 29, September 18, 2014.

[5] Y. Hirose, H. Hirose. “Polarity-dependent memory switching and behavior of Ag dendrite in Ag-photodoped amorphous As_2S_3 films,” J. Applied Physics, 47, 2767-2772, 1976.

[6] M. N. Kozicki, W. C. West, “Programmable metallization cell structure and method of making same,” U.S. Patent 5 761 115, Jun. 2, 1998.

[7] A. Borji, L. Itti, “Human vs. Computer in Scene and Object Recognition,” Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on, 113 – 120, June 2014.

[8] R. Ananthanarayanan, S. K. Esser, H. D. Simon, D. S. Modha, “The Cat is Out of the Bag: Cortical Simulations with 10^9 Neurons, 10^{13} Synapses,” High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference On, 1 – 12, Nov. 2009.

[9] C. Mead, A, “Neuromorphic Electronic Systems,” Proceedings of the IEEE. Vol. 78, No. 10, October 1990.

[10] S. H. Jo, T. Chang, I. Ebong, B. B. Bhadviya, P. Mazunder, and Wei Lu, “Nanoscale Memristor Device as Synapse in Neuromorphic Systems,” Nano Letters, 10(4):1297 – 1301, 2010.

[11] S.P. Adhikari, C. Yang, H. Kim, and L.O. Chua, “Memristor Bridge Synapse-Based Neural Network and Its Learning,” Neural Networks and Learning Systems, IEEE Transactions on, 23(9):1426 – 1435, Sept. 2012.

[12] National Institutes of Health. (2007, September 1). *Findings from memory research continue to fascinate* [Online]. Available: http://www.nia.nih.gov/alzheimers/features/findings-memory-research-continue-fascinate.

[13] D. Hebb, The Organization of Behavior: A Neuropsychological Theory, New York: Wiley & Sons, 1949.

[14] L. J. Elias, D. M. Saucier, *Neuropsychology: Clinical and Experimental Foundations*, Boston: Pearson, 2005.

[15] W. Gerstner, R. Ritz, J. Leo van Hemmen, “Why Spikes? Hebbian learning and retrieval of time-resolved excitation patterns,” Biol. Cybern. 69, 503-515, 1993.

[16] G. Howard, E. Gale, L. Bull, B. Costello, A. Adamatzky, “Evolution of Plastic Learning in Spiking Networks via Memristive Connections,” IEEE Transactions on Evolutionary Computation, vol. 16. no. 5. October 2012.

[17] G. Bi and M. Poo, “Synaptic Modifications in Cultured Hippocampal Neurons: Dependence on Spike Timing, Synaptic Strength, and Postsynaptic Cell Type,” The Journal of Neuroscience, 18(24):10464–10472, December 15, 1998.

[18] T. Serrano-Gotarredona, T. Prodromakis, T. Masquelier, G. Indiveri, and B. Linares-Barranco, “STDP and STDP variations with memristors for spiking neuromorphic learning systems,” Frontiers in Neuroscience, vol. 7, no. 2, 2013.

[19] R. Waser, R. Dittmann, G. Staikov, and K. Szot, “Redox-Based Resistive Switching Memories – Nanoionic Mechanisms, Prospects, and Challenges,” Adv. Mater. 21, p. 2632-2663, 2009.

[20] M. Mitkova and M. N. Kozicki, “Silver incorporation in Ge-Se glasses used in programmable metallization cell devices,” J. Non-Cryst. Solids, Vol. 299-302, pt. B, 1023-1027, 2002.

[21] S. Rahaman, S. Maikap, H. Chiu, C. Lin, T. Wu, Y. Chen, P. Tzeng, M. Kao, and M. Tsai, “Bipolar Resistive Switching Memory Using Cu Metallic Filament in Ge(0.4)Se(0.6) Solid Electrolyte,” Electrochem. and Solid-State Lett. vol. 13, issue 5, p. H159-H162, 2010.

[22] T. Serrano-Gotarredona, T. Prodromakis, and B. Linares-Barranco, “A Proposal for Hybrid Memristor-CMOS Spiking Neuromorphic Learning Systems,” IEEE Circuits and Systems Magazine, Q2, 74 - 88, 2013. © IEEE 2012.

[23] W. Cai and R. Tetzlaff, "Advanced memristive model of synapses with adaptive thresholds," Cellular Nanoscale Networks and Their Applications (CNNA), 2012 13th International Workshop on, pp. 1,6, 29-31, August 2012. © IEEE 2012.

[24] K. A. Campbell, J. Li, A. McTeer, and J. T. Moore, “Layered resistance variable memory device and method of fabrication,” U.S. Patent 7 723 713, May 25, 2010.

[25] K. A. Campbell, “SnSe-based limited reprogrammable cell,” U.S. Patent 8 101 936, Jan. 24, 2012.

[26] Agilent B1530A Waveform Generator/Fast Measurement Unit User’s Guide, 5th ed., Agilent Technologies, Santa Clara, CA, August 2012.

[27] F. Pan, S. Gao, C. Chen, C. Song, F. Zeng, “Recent progress in resistive random access memories: Materials, switching mechanisms, and performance,” Materials Science and Engineering R 83, 1-59, 2014.

APPENDIX

STDP Testing Program

```
// KDPulser.cpp
// Developed in Visual Studio 2013
// Used to perform all testing included in Kolton Drake's Master Thesis
// @Author Kolton Drake
// WGFMU Libraries from Agilent
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include "wgfmu.h"
#include <visa.h>
#include <cmath>
#include <sstream>
#include <string>
#include <iostream>
#include <iomanip> // std::setprecision
#include <fstream>
#include <vector>
#include <Windows.h>
#include <algorithm>
#include <functional>
#include <random>
#include <chrono>
#include <iostream>
#include <fstream>
#include <string>
#include <vector>

using namespace std;

struct Waveform
{
   vector<double> waveData;
   vector<double> rawData;
   double freq;
   double amp;
   double offset;
   int length;
};


//   Functions in this module:
void  csvline_populate(vector<string>  &record,  const  string&  line,  char
      delimiter);
int csvparse(string operation, double amplitude, int dt);
int csvparse1(string cmd, double wAmp, double eAmp);
int csvparse2(string cmd, double wAmp, double eAmp);
int csvparse3(string cmd, double wAmp, double eAmp);
int csvparseRead(string cmd, double wAmp, double eAmp);
double  dcSweep(double  amplitude,  double  compliance,  int  pts,  string
      testName);
double  dcSweep2(double  amplitude,  double  compliance,  int  pts,  string
      testName);
double  dcSweep3(double  amplitude,  double  compliance,  int  pts,  string
      testName);
double  dcSweep4(double  amplitude,  double  compliance,  int  pts,  string
      testName);
void wgfmu_arb();
void pulse(string fname, string ampl, string endtime, string timeStepStr,
      string currentRange, string repeatCount);
void pulse1(string fname, string wAmp, string eAmp, string endtime, string
      timeStepStr, string currentRange, string repeatCount);
void pulse2(string fname, string wAmp, string eAmp, string endtime, string
      timeStepStr, string currentRange, string repeatCount);
void pulse3(string fname, string wAmp, string eAmp, string endtime, string
      timeStepStr, string currentRange, string repeatCount);
void pulse4(string fname, string wAmp, string eAmp, string endtime, string
      timeStepStr, string currentRange, string repeatCount);
void pulseNoSave(string fname, string wAmp, string eAmp, string endtime,
      string timeStepStr, string currentRange, string repeatCount);
void resultant(string fname, string wAmp, string eAmp);
void  writeResults2ChannelP(int  channelId1,  int  channelId2,  const  char*
      fileName, string pulseParams);
void dSee(string ampl, string comp, string points);
void res(string timeStepStr);

// Global variables and constants

const int CH1 = 101;
const int CH2 = 102;
string  rootF  =  "C:/Users/koltondrake/Documents/STDP/STDP_data/Jan2015/";
      //root file location

string testID;
string dieNum;
string devNum="Dev16";
string temperature = "777";
string cwAmp;
string ceAmp;
ViSession defaultRM, vi;
Waveform testWave,testWave2,testWave3,testWave4;
vector<double> curWaveDT;
vector<double> curWaveV;
int k, m;
int fnum = 0, dcnum = 0;
double curAmp=0;


// Checks the error being returned from the WGFMU

void checkError(int ret) // 7
{
   if (ret < WGFMU_NO_ERROR) {
     throw ret;
   }
}

// Checks the error being returned from the WGFMU

int checkError2(int ret) //14
{
   if (ret < WGFMU_NO_ERROR) {
     int size;
     WGFMU_getErrorSize(&size);
     char* msg = new char[size + 1];
     WGFMU_getError(msg, &size);
     fprintf(stderr, "%s", msg);
     delete[] msg;
   } return ret;
}

static const int VISA_ERROR_OFFSET = WGFMU_ERROR_CODE_MIN - 1;


void checkError3(int ret) //29
{
   if  (ret  <  WGFMU_NO_ERROR  &&  ret  >=  WGFMU_ERROR_CODE_MIN  ||  ret  <
      VISA_ERROR_OFFSET)
   {
     throw ret;
   }
}

// Saves the file from the WGFMU

void writeResults(int channelId, const char* fileName) //36
{
   FILE* fp = fopen(fileName, "w");
   if (fp != 0)
   {
     int   measuredSize,   totalSize;   WGFMU_getMeasureValueSize(channelId,
      &measuredSize, &totalSize);
     for (int i = 0; i < measuredSize; i++)
     {
       double time, value;
       WGFMU_getMeasureValue(channelId, i, &time, &value);
       fprintf(fp, "%.9lf, %.9lf\n", time, value);
     } fclose(fp);
   }
}

// Saves the file from the WGFMU with a row offset
void writeResults2(int channelId, int offset, int size, const char* fileName)
      //51
{
   FILE* fp = fopen(fileName, "w");
   if (fp != 0) {
     int measuredSize, totalSize;
     WGFMU_getMeasureValueSize(channelId, &measuredSize, &totalSize);
     for (int i = offset; i < offset + size; i++) {
        double time, value;
        WGFMU_getMeasureValue(channelId, i, &time, &value);
        fprintf(fp, "%.9lf, %.9lf\n", time, value);
      }
      fclose(fp);
    }
}

// Saves the file from the WGFMU with a row offset for each channel

void writeResults3(int channelId1, int channelId2, int offset, int size,
      const
   char* fileName) //66
{
   FILE* fp = fopen(fileName, "w");
   if (fp != 0) {
     int measuredSize, totalSize;
     WGFMU_getMeasureValueSize(channelId2, &measuredSize, &totalSize);
     for (int i = offset; i < offset + size; i++) {
       double time, value, voltage;
       WGFMU_getMeasureValue(channelId2, i, &time, &value);
       WGFMU_getInterpolatedForceValue(channelId1, time, &voltage);
       fprintf(fp, "%.9lf, %.9lf\n", voltage, value);
     }
     fclose(fp);
   }
}

// Saves the results from the WGFMU with a row offset for each channel // and
      the measurement information is placed in the header.

void writeResults2Channel(int channelId1, int channelId2, const char*
      fileName)
{
   FILE* fp = fopen(fileName, "w");
   if (fp != 0)
   {
     fprintf(fp, "Chan 1 Mode:,FastIV,Chan 2 Mode:,FastIV\n");
     fprintf(fp, "Chan 1 Meas Mode:,IMeas,Chan 2 Meas Mode:,IMeas\n");
     fprintf(fp, "Chan 1 IMeas Range:,1 mA,Chan 2 IMeas Range:,1 mA\n");
     fprintf(fp, "time, V1, I2\n");
     int measuredSize, totalSize;
     WGFMU_getMeasureValueSize(channelId1, &measuredSize, &totalSize);
     for (int i = 0; i < measuredSize; i++)
     {
       double time1, value1, time2, value2;
       WGFMU_getMeasureValue(channelId1, i, &time1, &value1);
       WGFMU_getMeasureValue(channelId2, i, &time2, &value2);
       fprintf(fp, "%.9lf, %.9lf, %.9lf\n", time1, value1, value2);

     } fclose(fp);
   }
}
```

```
// Saves the results from the WGFMU with a row offset for each channel // and
      the measurement information is placed in the header, as well as // the
      actual pulse conditions used (how to call the STDP test)

void  writeResults2ChannelP(int  channelId1,  int  channelId2,  const  char*
      fileName, string pulseParams)
{
   FILE* fp = fopen(fileName, "w");
   if (fp != 0)
   {
     fprintf(fp, "Chan 1 Mode:,FastIV,Chan 2 Mode:,FastIV\n");
     fprintf(fp, "Chan 1 Meas Mode:,VMeas,Chan 2 Meas Mode:,IMeas\n");
     fprintf(fp, pulseParams.c_str());
     fprintf(fp, "\n");
     fprintf(fp, "time, V1, I2\n");
     int measuredSize, totalSize;
     WGFMU_getMeasureValueSize(channelId1, &measuredSize, &totalSize);
     for (int i = 0; i < measuredSize; i++)
     {
       double time1, value1, time2, value2;
       WGFMU_getMeasureValue(channelId1, i, &time1, &value1);
       WGFMU_getMeasureValue(channelId2, i, &time2, &value2);
       fprintf(fp, "%.9lf, %.9lf, %.9lf\n", time1, value1, value2);

     } fclose(fp);
   }
}

// This is the main menu for the console application
// Saves a log file to the root file location.


void wgfmu_arb()
{

   //set up log file
   ostringstream os;
   //os << rootF << dieNum << "_" << devNum << "_" <<  "log.csv"; //for the
   //os  <<  rootF  <<  dieNum  <<  "/"  <<  devNum  <<  "/"<<  devNum<<  "_"  <<
      "log.csv"; //Original STDP Form.
   os << rootF << dieNum << "/" << devNum << "_" << "log.csv"; //New Form
   string f_name = os.str();

   FILE *fp = fopen(f_name.c_str(), "w");
   fprintf(fp, "LOG\n");
   fprintf(fp, devNum.c_str());
   fprintf(fp, "\n");

   ostringstream rName;
   ostringstream rNum;

   string cmd="y";
   cout << "w to send positive pulse\n";
   cout << "e to send negative pulse\n";
   cout << "r to DC Read device\n";
   cout << "c to DC condition device\n";
   cout << "n to exit\n";
   cout << "p to pulse the device\n";
   cout << "s to pulse the device frome one side\n";
   cout << "d to pulse the device frome both sides\n";
   cout << "h to show this information again\n";
   //int i = 0;

   while (cmd != "n")
   {
     cout << "KDPulser:>";
     cin >> cmd;
     fprintf(fp, cmd.c_str());
     fprintf(fp, "\n");
     string  fname,  amplitude,  endTime,  timeStepStr,  compStr,  repeatCount,
      wAmp, eAmp,points;
     if (cmd == "n")break; //exit
     if (cmd == "h")
     {
       cout << "w to send positive pulse\n";
       cout << "e to send negative pulse\n";
       cout << "r to DC Read device\n";
       cout << "c to DC condition device\n";
       cout << "n to exit\n";
       cout << "p to pulse the device\n";
       cout << "h to show this information again\n";
     }
     else if (cmd == "r")
     {
       cout << "Number of Points: ";
       cin >> points;
       fprintf(fp, points.c_str());
       fprintf(fp, "\n");

       res(points);
     }
     else if (cmd == "rPost")
     {
       rNum << setfill('0') << setw(3)<< fnum-1;
       rName << "rPost" << rNum.str();
       //dcSweep4(0.02, 10E-3, 51, rName.str()); //dc sweep to read
       fnum -= 1;
       dcSweep4(0.02, 10E-3, 51, "rPost");

       rName.clear();
       rName.str("");
       rNum.clear();
       rNum.str("");
     }
     else if (cmd == "rPre")
     {
       rNum << setfill('0') << setw(3) << fnum;
       rName << "rPre_" << rNum.str();

       //dcSweep4(0.02, 10E-3, 51, rName.str()); //dc sweep to read
       dcSweep4(0.02, 10E-3, 51, "rPre");
       fnum -= 1;
       rName.clear();
       rName.str("");
       rNum.clear();
       rNum.str("");
     }
     else if (cmd == "c") //condition
     {
       condition();
     }
     else if (cmd == "p") //pulse
     {
       cout << "Input File: ";
       cin >> fname;

       //fprintf(fp, "Input File: ");
       fprintf(fp, fname.c_str());
       fprintf(fp, "\n");

       cout << "Amplitude: ";
       cin >> amplitude;

       //fprintf(fp, "Amplitude: ");
       fprintf(fp, amplitude.c_str());
       fprintf(fp, "\n");

       cout << "End time: ";
       cin >> endTime;

       //fprintf(fp, "End time: ");
       fprintf(fp, endTime.c_str());
       fprintf(fp, "\n");

       cout << "Time step: ";
       cin >> timeStepStr;


       //fprintf(fp, "Time step: ");
       fprintf(fp, timeStepStr.c_str());
       fprintf(fp, "\n");

       cout << "Compliance: ";
       cin >> compStr;

       //fprintf(fp, "Compliance: ");
       fprintf(fp, compStr.c_str());
       fprintf(fp, "\n");

       cout << "Repeats: ";
       cin >> repeatCount;

       //fprintf(fp, "Repeats: ");
       fprintf(fp, repeatCount.c_str());
       fprintf(fp, "\n");

       pulse(fname, amplitude, endTime, timeStepStr,compStr,repeatCount);
     }
     else if (cmd == "d") //double side
     {
       cout << "Input File: ";
       cin >> fname;

       //fprintf(fp, "Input File: ");
       fprintf(fp, fname.c_str());
       fprintf(fp, "\n");

       cout << "Write Amplitude: ";
       cin >> wAmp;

       //fprintf(fp, "Amplitude: ");
       fprintf(fp, wAmp.c_str());
       fprintf(fp, "\n");

       cout << "Erase Amplitude: ";
       cin >> eAmp;

       //fprintf(fp, "Amplitude: ");
       fprintf(fp, eAmp.c_str());
       fprintf(fp, "\n");

       cout << "End time: ";
       cin >> endTime;

       //fprintf(fp, "End time: ");
       fprintf(fp, endTime.c_str());
       fprintf(fp, "\n");

       cout << "Time step: ";
       cin >> timeStepStr;


       //fprintf(fp, "Time step: ");
       fprintf(fp, timeStepStr.c_str());
       fprintf(fp, "\n");

       cout << "Compliance: ";
       cin >> compStr;

       //fprintf(fp, "Compliance: ");
       fprintf(fp, compStr.c_str());
       fprintf(fp, "\n");

       cout << "Repeats: ";
       cin >> repeatCount;

       //fprintf(fp, "Repeats: ");
       fprintf(fp, repeatCount.c_str());
       fprintf(fp, "\n");

       pulse2(fname, wAmp, eAmp, endTime, timeStepStr, compStr, repeatCount);
     }
     else if (cmd == "s") //single side
     {
       cout << "Input File: ";
       cin >> fname;

       //fprintf(fp, "Input File: ");
```

```
    fprintf(fp, fname.c_str());
    fprintf(fp, "\n");

    cout << "Write Amplitude: ";
    cin >> wAmp;

    //fprintf(fp, "Amplitude: ");
    fprintf(fp, wAmp.c_str());
    fprintf(fp, "\n");

    cout << "Erase Amplitude: ";
    cin >> eAmp;

    //fprintf(fp, "Amplitude: ");
    fprintf(fp, eAmp.c_str());
    fprintf(fp, "\n");

    cout << "End time: ";
    cin >> endTime;

    //fprintf(fp, "End time: ");
    fprintf(fp, endTime.c_str());
    fprintf(fp, "\n");

    cout << "Time step: ";
    cin >> timeStepStr;


    //fprintf(fp, "Time step: ");
    fprintf(fp, timeStepStr.c_str());
    fprintf(fp, "\n");

    cout << "Compliance: ";
    cin >> compStr;

    //fprintf(fp, "Compliance: ");
    fprintf(fp, compStr.c_str());
    fprintf(fp, "\n");

    cout << "Repeats: ";
    cin >> repeatCount;

    //fprintf(fp, "Repeats: ");
    fprintf(fp, repeatCount.c_str());
    fprintf(fp, "\n");

    pulse1(fname, wAmp, eAmp, endTime, timeStepStr, compStr, repeatCount);
  }
  else if (cmd == "stdp") //double side STDP
  {
    cout << "Input File: ";
    cin >> fname;

    //fprintf(fp, "Input File: ");
    fprintf(fp, fname.c_str());
    fprintf(fp, "\n");
    cout << "Write Amplitude: ";
    cin >> wAmp;

    //fprintf(fp, "Amplitude: ");
    fprintf(fp, wAmp.c_str());
    fprintf(fp, "\n");

    cout << "Erase Amplitude: ";
    cin >> eAmp;

    //fprintf(fp, "Amplitude: ");
    fprintf(fp, eAmp.c_str());
    fprintf(fp, "\n");

    cout << "End time: ";
    cin >> endTime;

    //fprintf(fp, "End time: ");
    fprintf(fp, endTime.c_str());
    fprintf(fp, "\n");

    cout << "Time step: ";
    cin >> timeStepStr;


    //fprintf(fp, "Time step: ");
    fprintf(fp, timeStepStr.c_str());
    fprintf(fp, "\n");

    cout << "Compliance: ";
    cin >> compStr;

    //fprintf(fp, "Compliance: ");
    fprintf(fp, compStr.c_str());
    fprintf(fp, "\n");

    cout << "Repeats: ";
    cin >> repeatCount;

    //fprintf(fp, "Repeats: ");
    fprintf(fp, repeatCount.c_str());
    fprintf(fp, "\n");

    //pulse3(fname,   wAmp,   eAmp,   endTime,   timeStepStr,   compStr,
   repeatCount);
    pulseNoSave(fname,   wAmp,   eAmp,   endTime,   timeStepStr,   compStr,
   repeatCount);
  }
  else if (cmd == "stdpl") //double side STDP with longer pos-neg
  {
    cout << "Input File: ";
    cin >> fname;

    //fprintf(fp, "Input File: ");
    fprintf(fp, fname.c_str());
    fprintf(fp, "\n");
    cout << "Write Amplitude: ";
    cin >> wAmp;

    //fprintf(fp, "Amplitude: ");
    fprintf(fp, wAmp.c_str());
    fprintf(fp, "\n");

    cout << "Erase Amplitude: ";
    cin >> eAmp;

    //fprintf(fp, "Amplitude: ");
    fprintf(fp, eAmp.c_str());
    fprintf(fp, "\n");

    cout << "End time: ";
    cin >> endTime;

    //fprintf(fp, "End time: ");
    fprintf(fp, endTime.c_str());
    fprintf(fp, "\n");

    cout << "Time step: ";
    cin >> timeStepStr;


    //fprintf(fp, "Time step: ");
    fprintf(fp, timeStepStr.c_str());
    fprintf(fp, "\n");

    cout << "Compliance: ";
    cin >> compStr;

    //fprintf(fp, "Compliance: ");
    fprintf(fp, compStr.c_str());
    fprintf(fp, "\n");

    cout << "Repeats: ";
    cin >> repeatCount;

    //fprintf(fp, "Repeats: ");
    fprintf(fp, repeatCount.c_str());
    fprintf(fp, "\n");

    pulse4(fname, wAmp, eAmp, endTime, timeStepStr, compStr, repeatCount);
  }

  else if (cmd == "resl") //double side resultant with longer pos-neg
  {
    cout << "Input File: ";
    cin >> fname;

    //fprintf(fp, "Input File: ");
    fprintf(fp, fname.c_str());
    fprintf(fp, "\n");

    cout << "Write Amplitude: ";
    cin >> wAmp;
```

```
        //fprintf(fp, "Amplitude: ");
        fprintf(fp, wAmp.c_str());
        fprintf(fp, "\n");

        cout << "Erase Amplitude: ";
        cin >> eAmp;

        resultant(fname, wAmp, eAmp);
      }

      else if (cmd == "dsee")
      {
        fprintf(fp, fname.c_str());
        fprintf(fp, "\n");

        cout << "Amplitude: ";
        cin >> wAmp;

        fprintf(fp, wAmp.c_str());
        fprintf(fp, "\n");

        cout << "Compliance: ";
        cin >> compStr;

        fprintf(fp, compStr.c_str());
        fprintf(fp, "\n");

        cout << "Number of Points: ";
        cin >> points;
        fprintf(fp, points.c_str());
        fprintf(fp, "\n");


        dSee(wAmp, compStr, points);
      }
      else if (cmd == "time")
      {
        cout << chrono::system_clock::now().time_since_epoch().count();
        Sleep(1000);
        cout << chrono::system_clock::now().time_since_epoch().count();
      }
    }

    fclose(fp);
    WGFMU_closeSession();
    viClose(vi);
    viClose(defaultRM);
    exit(0);
}
// The very first write/erase STDP pulse creator. Deprecated function
// left for archiving purposes.

void datPulse(string cmd, string ampl,string dt)
{
    string amplitude=ampl;
    csvparse(cmd, stod(amplitude), stoi(dt));
```

```
WGFMU_clear();
int dtint = stoi(dt);
testWave2.waveData.push_back(0);
testWave.waveData.push_back(0);
double* v1 = &testWave.waveData[0];
double* v2 = &testWave2.waveData[0];
vector<double> testDT, test2DT;
//vector<double> zeroV = { 0, 0 };
char* leader = "ch1";
char* lagger = "ch2";
int leaderch = 101;
int laggerch = 102;

double minTime = 10E-9;
double datStep = .00100;
for (int i = 0; i < dtint; i++)
{
//   if (cmd == "w")test2DT.push_back(datStep);
//   if (cmd == "e")testDT.push_back(datStep);
}
for (int i = 0; i < testWave.length; i++)
{
  //cout << i*datStep << "testing doub convert\n";
  testDT.push_back(datStep);
  test2DT.push_back(datStep);
}
for (int i = 0; i < dtint; i++)
{
//   if (cmd == "e")test2DT.push_back(datStep);
//   if (cmd == "w")testDT.push_back(datStep);
}
double* dt1 = &testDT[0];
double* dt2 = &test2DT[0];
double endTime = datStep*(testWave.length);
//vector<double> zeroDT = { datStep, endTime };
//double* zDT = &zeroDT[0];
//double* zV = &zeroV[0];
WGFMU_createPattern(leader, 0);
WGFMU_createPattern(lagger, 0);

//WGFMU_addVectors(leader, leaderdts, v1, 6);
WGFMU_addVectors(leader, dt1, v1, testWave.length);
WGFMU_addVectors(lagger, dt2, v2, testWave2.length);

int numPoints = 20000; //was 100000
//double timeStep = window / (numPoints -1);
double timeStep = 1E-4;


// Set the measurement events for both channels
//WGFMU_setMeasureEvent(leader,   "evt",   0,   numPoints,   timeStep,   0,
   WGFMU_MEASURE_EVENT_DATA_AVERAGED);
//WGFMU_setMeasureEvent(lagger,   "evt2",   0,   numPoints,   timeStep,   0,
   WGFMU_MEASURE_EVENT_DATA_AVERAGED); //no averaging
WGFMU_setMeasureEvent(leader,  "evt",  0,  numPoints,  timeStep,  timeStep-
   minTime, WGFMU_MEASURE_EVENT_DATA_AVERAGED);
```

```
    WGFMU_setMeasureEvent(lagger,  "evt2",  0,  numPoints,  timeStep,  timeStep-
       minTime, WGFMU_MEASURE_EVENT_DATA_AVERAGED);


    WGFMU_addSequence(101, leader, 1); // Add the waveform to WGFMU channel 1
    WGFMU_addSequence(102, lagger, 1);

    //ONLINE
    WGFMU_initialize();

    // Set the operation mode for each channel
    WGFMU_setOperationMode(101, WGFMU_OPERATION_MODE_FASTIV);
    WGFMU_setOperationMode(102, WGFMU_OPERATION_MODE_FASTIV);

    // B1500 Defaults to measuring voltage. Change it to current. Also set the
       resolution
    // for the ADC in the WGFMU
    WGFMU_setMeasureMode(101, WGFMU_MEASURE_MODE_CURRENT);
    WGFMU_setMeasureMode(102, WGFMU_MEASURE_MODE_CURRENT);
    WGFMU_setMeasureCurrentRange(101, WGFMU_MEASURE_CURRENT_RANGE_1MA);
    WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_1MA);



    WGFMU_setTriggerOutMode(leaderch,    WGFMU_TRIGGER_OUT_MODE_START_SEQUENCE,
       WGFMU_TRIGGER_OUT_POLARITY_POSITIVE);
    WGFMU_setTriggerOutEvent(leader, "trig", 0, 0);

    WGFMU_connect(101);
    WGFMU_connect(102);
    WGFMU_execute();
    WGFMU_waitUntilCompleted();

    // Write the data to appropriate file
    ostringstream thefilename;

    //thefilename << "L:/MEC 107 Data/STDP/" << testID << "/1006301/Wafer10/"
       <<  dieNum  <<  "/"  <<  devNum  <<  "/"  <<  posAmplitude  <<  "_"  <<
       abs(negAmplitude) << "_" << dT << units << ".csv";
    thefilename << rootF << dieNum << "/" << devNum << "/Pulse_" << curAmp <<
       "_" << dt << "_" << fnum << ".csv";
    //thefilename   <<   "C:/Users/koltondrake/Documents/STDP/STDP_data/"   <<
       "resultant/" << devNum << "/" << curAmp << "_" << fnum << ".csv";
    string f_name = thefilename.str();
    writeResults2Channel(101, 102, f_name.c_str());
    //int ret = WGFMU_exportAscii(f_name.c_str());

    thefilename.str("");
    thefilename.clear();
    fnum++;
}


void  pulse(string  fname,  string  ampl,  string  endtime,  string  timeStepStr,
       string currentRange, string repeatCount)
{
    string amplitude = ampl;
```

```
csvparse(fname, stod(amplitude), 0);
WGFMU_clear();
double endCount=0;
int error = 1;
int curPeat=0;
int wasZero = 1;
//int dtint = stoi(dt);
//testWave2.waveData.push_back(0);
testWave.waveData.push_back(0);
double* v1 = &testWave.waveData[0];
//double* v2 = &testWave2.waveData[0];
int repeats = 1;
while (error == 1)
{
  error = 0;
  try
  {
    repeats = stoi(repeatCount);
  }
  catch (const std::invalid_argument& ia)
  {
    error = 1;
    cout << "Invalid Repeat Count \n";
    cout << "Repeat Count: ";
    cin >> repeatCount;
    cout << "\n";
  }
}
error = 0;



vector<double> testDT;
vector<double> zeroV = { 0, 0 };
char* leader = "ch1";
char* lagger = "ch2";
int leaderch = 101;
int laggerch = 102;
string pulseParams = fname+","+ampl + "," + endtime + "," + timeStepStr +
   "," + currentRange + "," + repeatCount;

size_t size = 0;

if             (endtime.find_first_of("m",             size)            !=
   string::npos)endtime.replace(endtime.find_first_of("m",  size),  1,  "e-
   3");
else        if        (endtime.find_first_of("u",         size)         !=
   string::npos)endtime.replace(endtime.find_first_of("u",  size),  1,  "e-
   6");
else        if        (endtime.find_first_of("n",         size)         !=
   string::npos)endtime.replace(endtime.find_first_of("n",  size),  1,  "e-
   9");

if            (timeStepStr.find_first_of("m",            size)          !=
   string::npos)timeStepStr.replace(timeStepStr.find_first_of("m",   size),
   1, "e-3");
```

```
else        if        (timeStepStr.find_first_of("u",        size)        !=
   string::npos)timeStepStr.replace(timeStepStr.find_first_of("u",   size),
   1, "e-6");
else        if        (timeStepStr.find_first_of("n",        size)        !=
   string::npos)timeStepStr.replace(timeStepStr.find_first_of("n",   size),
   1, "e-9");


double minTime = 10E-9;
//double datStep = 10E-9;
double datStep = stod(endtime) / testWave.length;
if (datStep <= 10e-9) datStep = 10e-9;
double bigStep = 3 * datStep;

/*if (datStep <= 14.49E-9) datStep = 10E-9;
else if (datStep <= 24.49E-9) datStep = 20E-9;
else if (datStep <= 34.49E-9) datStep = 30E-9;
else if (datStep <= 44.49E-9) datStep = 40E-9;
else if (datStep <= 54.49E-9) datStep = 50E-9;
else if (datStep <= 64.49E-9) datStep = 60E-9;
else if (datStep <= 74.49E-9) datStep = 70E-9;*/



for (int i = 0; i < testWave.length*repeats; i++)
{
  //cout << i*datStep << "testing doub convert\n";
  //if(datStep<=100e-9) testDT.push_back(datStep+minTime);
  if ((i>1 && i<testWave.length && v1[i] == 0 && v1[i-1]==0))
  {
    testDT.push_back(datStep * 3);
    endCount += bigStep;
  }
  else if (i > testWave.length)
  {
    if (v1[i - testWave.length*curPeat] == 0 && v1[i-1]==0)
    {
      testDT.push_back(datStep * 3);
      endCount += bigStep;
    }
    else
    {
      wasZero = 0;
      testDT.push_back(datStep);
      endCount += datStep;
    }
  }
  else
  {
    wasZero = 0;
    testDT.push_back(datStep);
    endCount += datStep;
  }
  if (i%testWave.length == 0)curPeat++;

  //test2DT.push_back(datStep);
}
```

```
//cout << endCount << "\n";
double* dt1 = &testDT[0];
//double* dt2 = &test2DT[0];


//double endTime = datStep*testWave.length*repeats;



//double endTime = stod(endtime)*repeats;
//double endTime = testDT.size()*datStep;
double endTime = endCount;

vector<double> zeroDT = {datStep, endTime};
double* zDT = &zeroDT[0];
double* zV = &zeroV[0];
WGFMU_createPattern(leader, 0);
WGFMU_createPattern(lagger, 0);

//WGFMU_addVectors(leader, leaderdts, v1, 6);
for(int    i=0;i<repeats;i++)    WGFMU_addVectors(leader,    dt1,    v1,
   testWave.length); //Account for repeats.
WGFMU_addVectors(lagger, zDT, zV, 2);


//double timeStep = window / (numPoints -1);
//double timeStep = 10E-9;
double timeStep = stod(timeStepStr);
if (timeStep <= 10e-9) timeStep = 10e-9;
//int numPoints = (endTime / timeStep); //was 100000
//int numPoints = (datStep*testDT.size())/timeStep;
int numPoints = endTime / timeStep;

// Set the measurement events for both channels
//WGFMU_setMeasureEvent(leader,   "evt",   0,   numPoints,   timeStep,   0,
   WGFMU_MEASURE_EVENT_DATA_AVERAGED);
//WGFMU_setMeasureEvent(lagger,   "evt2",   0,   numPoints,   timeStep,   0,
   WGFMU_MEASURE_EVENT_DATA_AVERAGED); //no averaging
WGFMU_setMeasureEvent(leader,  "evt",  0,  numPoints,  timeStep,  timeStep  -
   minTime, WGFMU_MEASURE_EVENT_DATA_AVERAGED);
WGFMU_setMeasureEvent(lagger,  "evt2",  0,  numPoints,  timeStep,  timeStep  -
   minTime, WGFMU_MEASURE_EVENT_DATA_AVERAGED);


WGFMU_addSequence(101, leader, 1); // Add the waveform to WGFMU channel 1
WGFMU_addSequence(102, lagger, 1);

//ONLINE
WGFMU_initialize();

// Set the operation mode for each channel
WGFMU_setOperationMode(101, WGFMU_OPERATION_MODE_FASTIV);
WGFMU_setOperationMode(102, WGFMU_OPERATION_MODE_FASTIV);

// B1500 Defaults to measuring voltage. Change it to current. Also set the
   resolution
// for the ADC in the WGFMU
```

```
WGFMU_setMeasureMode(101, WGFMU_MEASURE_MODE_VOLTAGE);
WGFMU_setMeasureMode(102, WGFMU_MEASURE_MODE_CURRENT);
WGFMU_setMeasureVoltageRange(101, WGFMU_MEASURE_VOLTAGE_RANGE_10V);
if (currentRange == "1u" || currentRange == "1E-6" || currentRange == "1e-
   6" || currentRange == "1uA" || currentRange == "1ua" || currentRange ==
   "0.000001")
  WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_1UA);
else if (currentRange == "10u" || currentRange == "10E-6" || currentRange
   == "10e-6" || currentRange == "10uA" || currentRange == "10ua" ||
   currentRange == "0.00001")
  WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_10UA);
else  if  (currentRange  ==  "100u"  ||  currentRange  ==  "100E-6"  ||
   currentRange == "100e-6" || currentRange == "100uA" || currentRange ==
   "100ua" || currentRange == "0.0001")
  WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_100UA);
else if (currentRange == "1m" || currentRange == "1E-3" || currentRange ==
   "1e-3"  ||  currentRange  ==  "1mA"  ||  currentRange  ==  "1ma"  ||
   currentRange == "0.001")
  WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_1MA);
else if (currentRange == "10m" || currentRange == "10E-3" || currentRange
   == "10e-3" || currentRange == "10mA" || currentRange == "10ma" ||
   currentRange == "0.01")
  WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_10MA);
else
{
  cout << "wat";
  WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_10MA);
}

WGFMU_setTriggerOutMode(leaderch,    WGFMU_TRIGGER_OUT_MODE_START_SEQUENCE,
   WGFMU_TRIGGER_OUT_POLARITY_POSITIVE);
WGFMU_setTriggerOutEvent(leader, "trig", 0, 0);

WGFMU_connect(101);
WGFMU_connect(102);
WGFMU_execute();
WGFMU_waitUntilCompleted();

// Write the data to appropriate file
ostringstream thefilename;
ostringstream fileNumber;
fileNumber << setfill('0') << setw(3) << fnum;

//thefilename << "L:/MEC 107 Data/STDP/" << testID << "/1006301/Wafer10/"
   <<  dieNum  <<  "/"  <<  devNum  <<  "/"  <<  posAmplitude  <<  "_"  <<
   abs(negAmplitude) << "_" << dT << units << ".csv";
thefilename << rootF << dieNum << "/" << devNum << "/" << devNum << "_" <<
   fname <<"_" << curAmp << "_" << fileNumber.str() << ".csv";
//thefilename   <<   "C:/Users/koltondrake/Documents/STDP/STDP_data/"   <<
   "resultant/" << devNum << "/" << curAmp << "_" << fnum << ".csv";
string f_name = thefilename.str();
writeResults2ChannelP(101, 102, f_name.c_str(),pulseParams);
//int ret = WGFMU_exportAscii(f_name.c_str());

thefilename.str("");
thefilename.clear();
fileNumber.str("");
```

```
   fileNumber.clear();
   fnum++;
}

/*
This pulse is for taking a csv file and placing all amplitudes on Channel 1.
Write amplitude and Erase amplitude should be entered as positive values.
*/
void pulse1(string fname, string wAmp, string eAmp, string endtime, string
      timeStepStr, string currentRange, string repeatCount)
{
   //string amplitude = ampl;
   csvparse1(fname, stod(wAmp), stod(eAmp));
   WGFMU_clear();
   double endCount = 0;
   int error = 1;
   int curPeat = 0;
   int wasZero = 1;
   //int dtint = stoi(dt);
   testWave2.waveData.push_back(0);
   testWave.waveData.push_back(0);
   double* v1 = &testWave.waveData[0];
   double* v2 = &testWave2.waveData[0];
   int repeats = 1;
   while (error == 1)
   {
     error = 0;
     try
     {
       repeats = stoi(repeatCount);
     }
     catch (const std::invalid_argument& ia)
     {
       error = 1;
       cout << "Invalid Repeat Count \n";
       cout << "Repeat Count: ";
       cin >> repeatCount;
       cout << "\n";
     }
   }
   error = 0;



   vector<double> testDT;
   vector<double> zeroV = { 0, 0 };
   char* leader = "ch1";
   char* lagger = "ch2";
   int leaderch = 101;
   int laggerch = 102;
   string pulseParams = fname + "," + wAmp + "," + eAmp + "," + endtime + ","
      + timeStepStr + "," + currentRange + "," + repeatCount;

   size_t size = 0;
```

```
if              (endtime.find_first_of("m",             size)          !=
   string::npos)endtime.replace(endtime.find_first_of("m",  size),  1,  "e-
   3");
else         if          (endtime.find_first_of("u",         size)         !=
   string::npos)endtime.replace(endtime.find_first_of("u",  size),  1,  "e-
   6");
else         if          (endtime.find_first_of("n",         size)         !=
   string::npos)endtime.replace(endtime.find_first_of("n",  size),  1,  "e-
   9");

if             (timeStepStr.find_first_of("m",            size)          !=
   string::npos)timeStepStr.replace(timeStepStr.find_first_of("m",   size),
   1, "e-3");
else        if        (timeStepStr.find_first_of("u",        size)        !=
   string::npos)timeStepStr.replace(timeStepStr.find_first_of("u",   size),
   1, "e-6");
else        if        (timeStepStr.find_first_of("n",        size)        !=
   string::npos)timeStepStr.replace(timeStepStr.find_first_of("n",   size),
   1, "e-9");


double minTime = 10E-9;
//double datStep = 10E-9;
double datStep = stod(endtime) / testWave.length;
if (datStep <= 10e-9) datStep = 10e-9;
double bigStep = 3 * datStep;

/*if (datStep <= 14.49E-9) datStep = 10E-9;
else if (datStep <= 24.49E-9) datStep = 20E-9;
else if (datStep <= 34.49E-9) datStep = 30E-9;
else if (datStep <= 44.49E-9) datStep = 40E-9;
else if (datStep <= 54.49E-9) datStep = 50E-9;
else if (datStep <= 64.49E-9) datStep = 60E-9;
else if (datStep <= 74.49E-9) datStep = 70E-9;*/



for (int i = 0; i < testWave.length*repeats; i++)
{
  //cout << i*datStep << "testing doub convert\n";
  //if(datStep<=100e-9) testDT.push_back(datStep+minTime);
  if ((i>1 && i<testWave.length && v1[i] == 0 && v1[i - 1] == 0 && v2[i]
   == 0 && v2[i - 1] == 0))
  {
    testDT.push_back(datStep * 3);
    endCount += bigStep;
  }
  else if (i > testWave.length)
  {
    if    (v1[i    -    testWave.length*curPeat]    ==    0    &&    v2[i    -
   testWave.length*curPeat] == 0 && v1[i - 1] == 0 && v2[i - 1] == 0)
    {
      testDT.push_back(datStep * 3);
      endCount += bigStep;
    }
    else
    {
      wasZero = 0;
      testDT.push_back(datStep);
      endCount += datStep;
    }
  }
  else
  {
    wasZero = 0;
    testDT.push_back(datStep);
    endCount += datStep;
  }
  if (i%testWave.length == 0)curPeat++;

  //test2DT.push_back(datStep);
}
//cout << endCount << "\n";
double* dt1 = &testDT[0];
double* dt2 = &testDT[0];


//double endTime = datStep*testWave.length*repeats;



//double endTime = stod(endtime)*repeats;
//double endTime = testDT.size()*datStep;
double endTime = endCount;

//vector<double> zeroDT = { datStep, endTime };
//double* zDT = &zeroDT[0];
//double* zV = &zeroV[0];
WGFMU_createPattern(leader, 0);
WGFMU_createPattern(lagger, 0);

//WGFMU_addVectors(leader, leaderdts, v1, 6);
for (int i = 0; i < repeats; i++)
{
  WGFMU_addVectors(leader,  dt1,  v1,  testWave.length);  //Account  for
   repeats.
  WGFMU_addVectors(lagger, dt2, v2, testWave.length);
}
//WGFMU_addVectors(lagger, zDT, zV, 2);


//double timeStep = window / (numPoints -1);
//double timeStep = 10E-9;
double timeStep = stod(timeStepStr);
if (timeStep <= 10e-9) timeStep = 10e-9;
//int numPoints = (endTime / timeStep); //was 100000
//int numPoints = (datStep*testDT.size())/timeStep;
int numPoints = endTime / timeStep;

// Set the measurement events for both channels
//WGFMU_setMeasureEvent(leader,  "evt",  0,  numPoints,  timeStep,  0,
   WGFMU_MEASURE_EVENT_DATA_AVERAGED);
//WGFMU_setMeasureEvent(lagger,  "evt2",  0,  numPoints,  timeStep,  0,
   WGFMU_MEASURE_EVENT_DATA_AVERAGED); //no averaging
```

```
WGFMU_setMeasureEvent(leader, "evt", 0, numPoints, timeStep, timeStep -
   minTime, WGFMU_MEASURE_EVENT_DATA_AVERAGED);
WGFMU_setMeasureEvent(lagger, "evt2", 0, numPoints, timeStep, timeStep -
   minTime, WGFMU_MEASURE_EVENT_DATA_AVERAGED);


WGFMU_addSequence(101, leader, 1); // Add the waveform to WGFMU channel 1
WGFMU_addSequence(102, lagger, 1);

//ONLINE
WGFMU_initialize();

// Set the operation mode for each channel
WGFMU_setOperationMode(101, WGFMU_OPERATION_MODE_FASTIV);
WGFMU_setOperationMode(102, WGFMU_OPERATION_MODE_FASTIV);

// B1500 Defaults to measuring voltage. Change it to current. Also set the
   resolution
// for the ADC in the WGFMU
WGFMU_setMeasureMode(101, WGFMU_MEASURE_MODE_VOLTAGE);
WGFMU_setMeasureMode(102, WGFMU_MEASURE_MODE_CURRENT);
WGFMU_setMeasureVoltageRange(101, WGFMU_MEASURE_VOLTAGE_RANGE_10V);
if (currentRange == "1u" || currentRange == "1E-6" || currentRange == "1e-
   6" || currentRange == "1uA" || currentRange == "1ua" || currentRange ==
   "0.000001")
  WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_1UA);
else if (currentRange == "10u" || currentRange == "10E-6" || currentRange
   == "10e-6" || currentRange == "10uA" || currentRange == "10ua" ||
   currentRange == "0.00001")
  WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_10UA);
else if (currentRange == "100u" || currentRange == "100E-6" ||
   currentRange == "100e-6" || currentRange == "100uA" || currentRange ==
   "100ua" || currentRange == "0.0001")
  WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_100UA);
else if (currentRange == "1m" || currentRange == "1E-3" || currentRange ==
   "1e-3" || currentRange == "1mA" || currentRange == "1ma" ||
   currentRange == "0.001")
  WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_1MA);
else if (currentRange == "10m" || currentRange == "10E-3" || currentRange
   == "10e-3" || currentRange == "10mA" || currentRange == "10ma" ||
   currentRange == "0.01")
  WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_10MA);
else
{
  cout << "wat";
  WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_10MA);
}

WGFMU_setTriggerOutMode(leaderch, WGFMU_TRIGGER_OUT_MODE_START_SEQUENCE,
   WGFMU_TRIGGER_OUT_POLARITY_POSITIVE);
WGFMU_setTriggerOutEvent(leader, "trig", 0, 0);

WGFMU_connect(101);
WGFMU_connect(102);
WGFMU_execute();
WGFMU_waitUntilCompleted();
```

```
   // Write the data to appropriate file
   ostringstream thefilename;
   ostringstream fileNumber;
   fileNumber << setfill('0') << setw(3) << fnum;

   //thefilename << "L:/MEC 107 Data/STDP/" << testID << "/1006301/Wafer10/"
      << dieNum << "/" << devNum << "/" << posAmplitude << "_" <<
      abs(negAmplitude) << "_" << dT << units << ".csv";
   thefilename << rootF << dieNum << "/" << devNum << "/" << devNum << "_" <<
      fname << "_" << wAmp << "_" << eAmp << "_" << fileNumber.str() <<
      ".csv";
   //thefilename << "C:/Users/koltondrake/Documents/STDP/STDP_data/" <<
      "resultant/" << devNum << "/" << curAmp << "_" << fnum << ".csv";
   string f_name = thefilename.str();

   writeResults2ChannelP(101, 102, f_name.c_str(), pulseParams);
   //int ret = WGFMU_exportAscii(f_name.c_str());

   thefilename.str("");
   thefilename.clear();
   fileNumber.str("");
   fileNumber.clear();
   fnum++;
}

/*
This pulse is for taking a csv file and placing negative amplitudes on
      Channel 2 and positive amplitudes on Channel 1.
Write amplitude and Erase amplitude should be entered as positive values.s
*/
void pulse2(string fname, string wAmp, string eAmp, string endtime, string
      timeStepStr, string currentRange, string repeatCount)
{
   //string amplitude = ampl;
   csvparse2(fname, stod(wAmp), stod(eAmp));
   WGFMU_clear();
   double endCount = 0;
   int error = 1;
   int curPeat = 0;
   int wasZero = 1;
   //int dtint = stoi(dt);
   testWave2.waveData.push_back(0);
   testWave.waveData.push_back(0);
   double* v1 = &testWave.waveData[0];
   double* v2 = &testWave2.waveData[0];
   int repeats = 1;
   while (error == 1)
   {
     error = 0;
     try
     {
       repeats = stoi(repeatCount);
     }
     catch (const std::invalid_argument& ia)
     {
       error = 1;
       cout << "Invalid Repeat Count \n";
```

```
    cout << "Repeat Count: ";
    cin >> repeatCount;
    cout << "\n";
  }
}
error = 0;



vector<double> testDT;
vector<double> zeroV = { 0, 0 };
char* leader = "ch1";
char* lagger = "ch2";
int leaderch = 101;
int laggerch = 102;
string pulseParams = fname + "," + wAmp + "," + eAmp + "," + endtime + ","
   + timeStepStr + "," + currentRange + "," + repeatCount;

size_t size = 0;

if             (endtime.find_first_of("m",             size)            !=
   string::npos)endtime.replace(endtime.find_first_of("m",  size),  1,  "e-
   3");
else         if         (endtime.find_first_of("u",          size)         !=
   string::npos)endtime.replace(endtime.find_first_of("u",  size),  1,  "e-
   6");
else         if         (endtime.find_first_of("n",          size)         !=
   string::npos)endtime.replace(endtime.find_first_of("n",  size),  1,  "e-
   9");

if            (timeStepStr.find_first_of("m",            size)           !=
   string::npos)timeStepStr.replace(timeStepStr.find_first_of("m",   size),
   1, "e-3");
else        if        (timeStepStr.find_first_of("u",        size)         !=
   string::npos)timeStepStr.replace(timeStepStr.find_first_of("u",   size),
   1, "e-6");
else        if        (timeStepStr.find_first_of("n",        size)         !=
   string::npos)timeStepStr.replace(timeStepStr.find_first_of("n",   size),
   1, "e-9");


double minTime = 10E-9;
//double datStep = 10E-9;
double datStep = stod(endtime) / testWave.length;
if (datStep <= 10e-9) datStep = 10e-9;
double bigStep = 3 * datStep;

/*if (datStep <= 14.49E-9) datStep = 10E-9;
else if (datStep <= 24.49E-9) datStep = 20E-9;
else if (datStep <= 34.49E-9) datStep = 30E-9;
else if (datStep <= 44.49E-9) datStep = 40E-9;
else if (datStep <= 54.49E-9) datStep = 50E-9;
else if (datStep <= 64.49E-9) datStep = 60E-9;
else if (datStep <= 74.49E-9) datStep = 70E-9;*/
```

```
for (int i = 0; i < testWave.length*repeats; i++)
{
  //cout << i*datStep << "testing doub convert\n";
  //if(datStep<=100e-9) testDT.push_back(datStep+minTime);
  if ((i>1 && i<testWave.length && v1[i] == 0 && v1[i - 1] == 0 && v2[i]
   == 0 && v2[i - 1] == 0))
  {
    testDT.push_back(datStep * 3);
    endCount += bigStep;
  }
  else if (i > testWave.length)
  {
    if    (v1[i    -    testWave.length*curPeat]    ==    0    &&    v2[i    -
   testWave.length*curPeat] == 0 && v1[i - 1] == 0 && v2[i - 1] == 0)
    {
      testDT.push_back(datStep * 3);
      endCount += bigStep;
    }
    else
    {
      wasZero = 0;
      testDT.push_back(datStep);
      endCount += datStep;
    }
  }
  else
  {
    wasZero = 0;
    testDT.push_back(datStep);
    endCount += datStep;
  }
  if (i%testWave.length == 0)curPeat++;

  //test2DT.push_back(datStep);
}
//cout << endCount << "\n";
double* dt1 = &testDT[0];
double* dt2 = &testDT[0];


//double endTime = datStep*testWave.length*repeats;



//double endTime = stod(endtime)*repeats;
//double endTime = testDT.size()*datStep;
double endTime = endCount;

//vector<double> zeroDT = { datStep, endTime };
//double* zDT = &zeroDT[0];
//double* zV = &zeroV[0];
WGFMU_createPattern(leader, 0);
WGFMU_createPattern(lagger, 0);

//WGFMU_addVectors(leader, leaderdts, v1, 6);
for (int i = 0; i < repeats; i++)
{
  WGFMU_addVectors(leader, dt1, v1, testWave.length); //Account for
    repeats.
  WGFMU_addVectors(lagger, dt2, v2, testWave.length);
}
//WGFMU_addVectors(lagger, zDT, zV, 2);


//double timeStep = window / (numPoints -1);
//double timeStep = 10E-9;
double timeStep = stod(timeStepStr);
if (timeStep <= 10e-9) timeStep = 10e-9;
//int numPoints = (endTime / timeStep); //was 100000
//int numPoints = (datStep*testDT.size())/timeStep;
int numPoints = endTime / timeStep;

// Set the measurement events for both channels
//WGFMU_setMeasureEvent(leader, "evt", 0, numPoints, timeStep, 0,
   WGFMU_MEASURE_EVENT_DATA_AVERAGED);
//WGFMU_setMeasureEvent(lagger, "evt2", 0, numPoints, timeStep, 0,
   WGFMU_MEASURE_EVENT_DATA_AVERAGED); //no averaging
WGFMU_setMeasureEvent(leader, "evt", 0, numPoints, timeStep, timeStep -
   minTime, WGFMU_MEASURE_EVENT_DATA_AVERAGED);
WGFMU_setMeasureEvent(lagger, "evt2", 0, numPoints, timeStep, timeStep -
   minTime, WGFMU_MEASURE_EVENT_DATA_AVERAGED);


WGFMU_addSequence(101, leader, 1); // Add the waveform to WGFMU channel 1
WGFMU_addSequence(102, lagger, 1);

//ONLINE
WGFMU_initialize();

// Set the operation mode for each channel
WGFMU_setOperationMode(101, WGFMU_OPERATION_MODE_FASTIV);
WGFMU_setOperationMode(102, WGFMU_OPERATION_MODE_FASTIV);

// B1500 Defaults to measuring voltage. Change it to current. Also set the
   resolution
// for the ADC in the WGFMU
WGFMU_setMeasureMode(101, WGFMU_MEASURE_MODE_VOLTAGE);
WGFMU_setMeasureMode(102, WGFMU_MEASURE_MODE_CURRENT);
WGFMU_setMeasureVoltageRange(101, WGFMU_MEASURE_VOLTAGE_RANGE_10V);
if (currentRange == "1u" || currentRange == "1E-6" || currentRange == "1e-
   6" || currentRange == "1uA" || currentRange == "1ua" || currentRange ==
   "0.000001")
  WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_1UA);
else if (currentRange == "10u" || currentRange == "10E-6" || currentRange
   == "10e-6" || currentRange == "10uA" || currentRange == "10ua" ||
   currentRange == "0.00001")
  WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_10UA);
else if (currentRange == "100u" || currentRange == "100E-6" ||
   currentRange == "100e-6" || currentRange == "100uA" || currentRange ==
   "100ua" || currentRange == "0.0001")
  WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_100UA);
else if (currentRange == "1m" || currentRange == "1E-3" || currentRange ==
   "1e-3" || currentRange == "1mA" || currentRange == "1ma" ||
   currentRange == "0.001")
```

```
     WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_1MA);
   else if (currentRange == "10m" || currentRange == "10E-3" || currentRange
      == "10e-3" || currentRange == "10mA" || currentRange == "10ma" ||
      currentRange == "0.01")
     WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_10MA);
   else
   {
     cout << "wat";
     WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_10MA);
   }

   WGFMU_setTriggerOutMode(leaderch,    WGFMU_TRIGGER_OUT_MODE_START_SEQUENCE,
      WGFMU_TRIGGER_OUT_POLARITY_POSITIVE);
   WGFMU_setTriggerOutEvent(leader, "trig", 0, 0);

   WGFMU_connect(101);
   WGFMU_connect(102);
   WGFMU_execute();
   WGFMU_waitUntilCompleted();

   // Write the data to appropriate file
   ostringstream thefilename;
   ostringstream fileNumber;
   fileNumber << setfill('0') << setw(3) << fnum;

   //thefilename << "L:/MEC 107 Data/STDP/" << testID << "/1006301/Wafer10/"
      << dieNum << "/" << devNum << "/" << posAmplitude << "_" <<
      abs(negAmplitude) << "_" << dT << units << ".csv";
   thefilename << rootF << dieNum << "/" << devNum << "/" << devNum << "_" <<
      fname << "_" << wAmp << "_" << eAmp  << "_" << fileNumber.str() <<
      ".csv";
   //thefilename   <<   "C:/Users/koltondrake/Documents/STDP/STDP_data/"   <<
      "resultant/" << devNum << "/" << curAmp << "_" << fnum << ".csv";
   string f_name = thefilename.str();

   writeResults2ChannelP(101, 102, f_name.c_str(), pulseParams);
   //int ret = WGFMU_exportAscii(f_name.c_str());

   thefilename.str("");
   thefilename.clear();
   fileNumber.str("");
   fileNumber.clear();
   fnum++;
}


/*
This is for STDP. Requires two files _ch1 and _ch2
Includes bumpy read.
*/
void pulse3(string fname, string wAmp, string eAmp, string endtime, string
      timeStepStr, string currentRange, string repeatCount)
{
   //string amplitude = ampl;
   csvparse3(fname, stod(wAmp), stod(eAmp));

   //Generates the testWave3 data.
```

```
csvparseRead("bumpy", 1, 1);

WGFMU_clear();
double endCount = 0;
double endReadCount = 0;
double readIdx[100];
double readTimeStep[100];
int error = 1;
int curPeat = 0;
int wasZero = 1;
int pointCount=0;
//Number of decades to read.
int decades = 2;
//Number of points per read waveform data; if bumpy_ch1.csv has 70 points,
   readRes takes 20 measurement points for each point.
//Number of read points calculation = numBumpyPts * readRes * decades.
int readRes = 5;
//int dtint = stoi(dt);

testWave.waveData.push_back(0); //Top Electrode
testWave2.waveData.push_back(0); //Bottom Electrode
testWave3.waveData.push_back(0); //Read Pulse (goes to top electrode).
testWave4.waveData.push_back(0); //Read pulse bot.


int repeats = 1;
while (error == 1)
{
  error = 0;
  try
  {
    repeats = stoi(repeatCount);
  }
  catch (const std::invalid_argument& ia)
  {
    error = 1;
    cout << "Invalid Repeat Count \n";
    cout << "Repeat Count: ";
    cin >> repeatCount;
    cout << "\n";
  }
}
error = 0;



vector<double> testDT;
vector<double> test2DT;
vector<double> test3DT;
vector<double> test4DT;
char* leader = "ch1";
char* lagger = "ch2";
int leaderch = 101;
int laggerch = 102;
string pulseParams = fname + "," + wAmp + "," + eAmp + "," + endtime + ","
   + timeStepStr + "," + currentRange + "," + repeatCount;
```

```
size_t size = 0;

if              (endtime.find_first_of("m",             size)           !=
   string::npos)endtime.replace(endtime.find_first_of("m",  size),  1,  "e-
   3");
else        if         (endtime.find_first_of("u",          size)          !=
   string::npos)endtime.replace(endtime.find_first_of("u",  size),  1,  "e-
   6");
else        if         (endtime.find_first_of("n",          size)          !=
   string::npos)endtime.replace(endtime.find_first_of("n",  size),  1,  "e-
   9");

if             (timeStepStr.find_first_of("m",             size)           !=
   string::npos)timeStepStr.replace(timeStepStr.find_first_of("m",   size),
   1, "e-3");
else       if        (timeStepStr.find_first_of("u",        size)          !=
   string::npos)timeStepStr.replace(timeStepStr.find_first_of("u",   size),
   1, "e-6");
else       if        (timeStepStr.find_first_of("n",        size)          !=
   string::npos)timeStepStr.replace(timeStepStr.find_first_of("n",   size),
   1, "e-9");

double endTimes = stod(endtime);
double minTime = 10E-9;
//double datStep = 10E-9;
double datStep = endTimes / testWave.length;
if (datStep <= 10e-9) datStep = 10e-9;

if (endTimes >= 1)
{
  decades = 2;
  readRes = 40;
}
else if (endTimes > 1e-3)
{
  decades = 3;
  readRes = 20;
}
else if (endTimes > 10e-6)
{
  decades = 5;
  readRes = 20;
}
else
{
  decades = 7;
  readRes = 20;
}

/*if (datStep <= 14.49E-9) datStep = 10E-9;
else if (datStep <= 24.49E-9) datStep = 20E-9;
else if (datStep <= 34.49E-9) datStep = 30E-9;
else if (datStep <= 44.49E-9) datStep = 40E-9;
else if (datStep <= 54.49E-9) datStep = 50E-9;
else if (datStep <= 64.49E-9) datStep = 60E-9;
else if (datStep <= 74.49E-9) datStep = 70E-9;*/
```

```
for (int i = 0; i < testWave.length; i++)
{
  //cout << "idx:\t" << i << "\t dt:\t" << datStep << "\n";
  //cout << "idx:\t" << testDT[i] << "\t dt:\t" << datStep;
  testDT.push_back(datStep);
  test2DT.push_back(datStep);
  //test2DT.push_back(datStep * 10);
  endCount += datStep;
}


endReadCount = endCount;
for (int j = 1; j < decades; j++)
{
  readIdx[j] = endReadCount;
  for (int i = 0; i < testWave3.length; i++)
  {
    if (j == 1)
    {
      pointCount += readRes;
      test3DT.push_back(datStep);
      test4DT.push_back(datStep);
      endReadCount += datStep;
    }

    else if (j > 1)
    {
      test3DT.push_back(datStep*pow(10.0, j));
      test4DT.push_back(datStep*pow(10.0, j));
      endReadCount += datStep*pow(10.0, j);

      testWave3.waveData.push_back(testWave3.waveData[i]);
      testWave4.waveData.push_back(testWave4.waveData[i]);
    }
  }
  readTimeStep[j] = (endReadCount-readIdx[j])/pointCount;
  if (readTimeStep[j] <= 10E-9)
  {
    readTimeStep[j] = 10e-9;
    pointCount = (endReadCount - readIdx[j]) / 10e-9;
  }
}
double* dt1 = &testDT[0];
double* dt2 = &test2DT[0];
double* dt3 = &test3DT[0];
double* dt4 = &test4DT[0];
double* v1 = &testWave.waveData[0];
double* v2 = &testWave2.waveData[0];
double* v3 = &testWave3.waveData[0];
double* v4 = &testWave4.waveData[0];

//double* dt23 = &test2DT[0];
//double* dt24 = &test2DT[0];
```

```
//double endTime = datStep*testWave.length*repeats;



//double endTime = stod(endtime)*repeats;
//double endTime = testDT.size()*datStep;
double endTime = endCount;
//endTime *= repeats;

double endReadTime = endReadCount;



WGFMU_createPattern(leader, 0);
WGFMU_createPattern(lagger, 0);

for (int i = 0; i < repeats; i++)
{
  WGFMU_addVectors(leader,   dt1,   v1,   testWave.length);   //Account   for
   repeats.
  WGFMU_addVectors(lagger, dt2, v2, testWave2.length);

  //readSection
  WGFMU_addVectors(leader, dt3, v3, testWave3.waveData.size()-1);
  WGFMU_addVectors(lagger, dt4, v4, testWave4.waveData.size()-1);
}

double timeStep = stod(timeStepStr);
if (timeStep <= 10e-9) timeStep = 10e-9;
int numPoints = endTime / timeStep;

// Set the measurement events for both channels

WGFMU_setMeasureEvent(leader,  "evt",  0,  numPoints,  timeStep,  timeStep  -
   minTime, WGFMU_MEASURE_EVENT_DATA_AVERAGED);
WGFMU_setMeasureEvent(lagger,  "evt2",  0,  numPoints,  timeStep,  timeStep  -
   minTime, WGFMU_MEASURE_EVENT_DATA_AVERAGED);

//read events

ostringstream eventName;
double averagingTime = 0.02;
for (int i = 1; i < decades; i++)
{
  while (readTimeStep[i]>1.3)
  {
    readTimeStep[i] /= 10;
    pointCount *= 10;
  }
  averagingTime = readTimeStep[i] - minTime;
  if (averagingTime > 0.02) averagingTime = 0.02;
  eventName.clear();
  eventName << "evtR" << i;
  WGFMU_setMeasureEvent(leader,    eventName.str().c_str(),    readIdx[i],
   pointCount,                readTimeStep[i],                averagingTime,
   WGFMU_MEASURE_EVENT_DATA_AVERAGED);
  eventName.clear();
  eventName << "evtRR" << i;
  WGFMU_setMeasureEvent(lagger,    eventName.str().c_str(),    readIdx[i],
   pointCount,              readTimeStep[i],               averagingTime,
   WGFMU_MEASURE_EVENT_DATA_AVERAGED);
}

WGFMU_addSequence(101, leader, 1); // Add the waveform to WGFMU channel 1
WGFMU_addSequence(102, lagger, 1);

//ONLINE
WGFMU_initialize();

// Set the operation mode for each channel
WGFMU_setOperationMode(101, WGFMU_OPERATION_MODE_FASTIV);
WGFMU_setOperationMode(102, WGFMU_OPERATION_MODE_FASTIV);

// B1500 Defaults to measuring voltage. Change it to current. Also set the
   resolution
// for the ADC in the WGFMU
WGFMU_setMeasureMode(101, WGFMU_MEASURE_MODE_VOLTAGE);
WGFMU_setMeasureMode(102, WGFMU_MEASURE_MODE_CURRENT);
WGFMU_setMeasureVoltageRange(101, WGFMU_MEASURE_VOLTAGE_RANGE_10V);
if (currentRange == "1u" || currentRange == "1E-6" || currentRange == "1e-
   6" || currentRange == "1uA" || currentRange == "1ua" || currentRange ==
   "0.000001")
  WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_1UA);
else if (currentRange == "10u" || currentRange == "10E-6" || currentRange
   == "10e-6" || currentRange == "10uA" || currentRange == "10ua" ||
   currentRange == "0.00001")
  WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_10UA);
else if (currentRange == "100u" || currentRange == "100E-6" ||
   currentRange == "100e-6" || currentRange == "100uA" || currentRange ==
   "100ua" || currentRange == "0.0001")
  WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_100UA);
else if (currentRange == "1m" || currentRange == "1E-3" || currentRange ==
   "1e-3" || currentRange == "1mA" || currentRange == "1ma" ||
   currentRange == "0.001")
  WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_1MA);
else if (currentRange == "10m" || currentRange == "10E-3" || currentRange
   == "10e-3" || currentRange == "10mA" || currentRange == "10ma" ||
   currentRange == "0.01")
  WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_10MA);
else
{
  cout << "wat";
  WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_10MA);
}

WGFMU_setTriggerOutMode(leaderch,    WGFMU_TRIGGER_OUT_MODE_START_SEQUENCE,
   WGFMU_TRIGGER_OUT_POLARITY_POSITIVE);
WGFMU_setTriggerOutEvent(leader, "trig", 0, 0);

WGFMU_connect(101);
WGFMU_connect(102);
WGFMU_execute();
WGFMU_waitUntilCompleted();
// Write the data to appropriate file
```

```
   ostringstream thefilename;
   ostringstream fileNumber;
   fileNumber << setfill('0') << setw(3) << fnum;

   //thefilename << "L:/MEC 107 Data/STDP/" << testID << "/1006301/Wafer10/"
      << dieNum << "/" << devNum << "/" << posAmplitude << "_" <<
      abs(negAmplitude) << "_" << dT << units << ".csv";
   thefilename << rootF << dieNum << "/" << devNum << "/" << devNum << "_" <<
      fname << "_" << wAmp << "_" << eAmp << "_" << fileNumber.str() <<
      ".csv";
   //thefilename << "C:/Users/koltondrake/Documents/STDP/STDP_data/" <<
      "resultant/" << devNum << "/" << curAmp << "_" << fnum << ".csv";
   string f_name = thefilename.str();
   /*
   This is the section where it returns a value
   */

   writeResults2ChannelP(101, 102, f_name.c_str(), pulseParams);


   thefilename.str("");
   thefilename.clear();
   fileNumber.str("");
   fileNumber.clear();
   fnum++;
}

/*
This is for STDP, does not perform a bumpy read, does not save data.
Requires ch1 and ch2
*/
void pulseNoSave(string fname, string wAmp, string eAmp, string endtime,
      string timeStepStr, string currentRange, string repeatCount)
{
   //string amplitude = ampl;
   csvparse3(fname, stod(wAmp), stod(eAmp));

   WGFMU_clear();
   double endCount = 0;
   double endReadCount = 0;
   double readIdx[100];
   double readTimeStep[100];
   int error = 1;
   int curPeat = 0;
   int wasZero = 1;
   int pointCount = 0;


   testWave.waveData.push_back(0); //Top Electrode
   testWave2.waveData.push_back(0); //Bottom Electrode

   int repeats = 1;
   while (error == 1)
   {
     error = 0;
     try
     {
```

```
    repeats = stoi(repeatCount);
  }
  catch (const std::invalid_argument& ia)
  {
    error = 1;
    cout << "Invalid Repeat Count \n";
    cout << "Repeat Count: ";
    cin >> repeatCount;
    cout << "\n";
  }
}
error = 0;



vector<double> testDT;
vector<double> test2DT;
char* leader = "ch1";
char* lagger = "ch2";
int leaderch = 101;
int laggerch = 102;
string pulseParams = fname + "," + wAmp + "," + eAmp + "," + endtime + ","
   + timeStepStr + "," + currentRange + "," + repeatCount;

size_t size = 0;

if              (endtime.find_first_of("m",             size)           !=
   string::npos)endtime.replace(endtime.find_first_of("m",  size),  1,  "e-
   3");
else        if         (endtime.find_first_of("u",         size)         !=
   string::npos)endtime.replace(endtime.find_first_of("u",  size),  1,  "e-
   6");
else        if         (endtime.find_first_of("n",         size)         !=
   string::npos)endtime.replace(endtime.find_first_of("n",  size),  1,  "e-
   9");

if             (timeStepStr.find_first_of("m",            size)          !=
   string::npos)timeStepStr.replace(timeStepStr.find_first_of("m",   size),
   1, "e-3");
else       if        (timeStepStr.find_first_of("u",        size)        !=
   string::npos)timeStepStr.replace(timeStepStr.find_first_of("u",   size),
   1, "e-6");
else       if        (timeStepStr.find_first_of("n",        size)        !=
   string::npos)timeStepStr.replace(timeStepStr.find_first_of("n",   size),
   1, "e-9");

double endTimes = stod(endtime);
double minTime = 10E-9;

double datStep = endTimes / testWave.length;
if (datStep <= 10e-9) datStep = 10e-9;


for (int i = 0; i < testWave.length; i++)
{

  testDT.push_back(datStep);
  test2DT.push_back(datStep);
  endCount += datStep;
}


double* dt1 = &testDT[0];
double* dt2 = &test2DT[0];

double* v1 = &testWave.waveData[0];
double* v2 = &testWave2.waveData[0];



double endTime = endCount;
endTime *= repeats;



WGFMU_createPattern(leader, 0);
WGFMU_createPattern(lagger, 0);

for (int i = 0; i < repeats; i++)
{
  WGFMU_addVectors(leader,  dt1,  v1,  testWave.length);  //Account  for
   repeats.
  WGFMU_addVectors(lagger, dt2, v2, testWave2.length);
}

double timeStep = stod(timeStepStr);
if (timeStep <= 10e-9) timeStep = 10e-9;
int numPoints = endTime / timeStep;

// Set the measurement events for both channels

WGFMU_setMeasureEvent(leader,  "evt",  0,  numPoints,  timeStep,  timeStep  -
   minTime, WGFMU_MEASURE_EVENT_DATA_AVERAGED);
WGFMU_setMeasureEvent(lagger,  "evt2",  0,  numPoints,  timeStep,  timeStep  -
   minTime, WGFMU_MEASURE_EVENT_DATA_AVERAGED);


WGFMU_addSequence(101, leader, 1); // Add the waveform to WGFMU channel 1
WGFMU_addSequence(102, lagger, 1);

//ONLINE
WGFMU_initialize();

// Set the operation mode for each channel
WGFMU_setOperationMode(101, WGFMU_OPERATION_MODE_FASTIV);
WGFMU_setOperationMode(102, WGFMU_OPERATION_MODE_FASTIV);

// B1500 Defaults to measuring voltage. Change it to current. Also set the
   resolution
// for the ADC in the WGFMU
WGFMU_setMeasureMode(101, WGFMU_MEASURE_MODE_VOLTAGE);
WGFMU_setMeasureMode(102, WGFMU_MEASURE_MODE_CURRENT);
WGFMU_setMeasureVoltageRange(101, WGFMU_MEASURE_VOLTAGE_RANGE_10V);
```

```
    if (currentRange == "1u" || currentRange == "1E-6" || currentRange == "1e-
       6" || currentRange == "1uA" || currentRange == "1ua" || currentRange ==
       "0.000001")
      WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_1UA);
    else if (currentRange == "10u" || currentRange == "10E-6" || currentRange
       == "10e-6" || currentRange == "10uA" || currentRange == "10ua" ||
       currentRange == "0.00001")
      WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_10UA);
    else if (currentRange == "100u" || currentRange == "100E-6" ||
       currentRange == "100e-6" || currentRange == "100uA" || currentRange ==
       "100ua" || currentRange == "0.0001")
      WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_100UA);
    else if (currentRange == "1m" || currentRange == "1E-3" || currentRange ==
       "1e-3" || currentRange == "1mA" || currentRange == "1ma" ||
       currentRange == "0.001")
      WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_1MA);
    else if (currentRange == "10m" || currentRange == "10E-3" || currentRange
       == "10e-3" || currentRange == "10mA" || currentRange == "10ma" ||
       currentRange == "0.01")
      WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_10MA);
    else
    {
      cout << "wat";
      WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_10MA);
    }

    WGFMU_setTriggerOutMode(leaderch,    WGFMU_TRIGGER_OUT_MODE_START_SEQUENCE,
       WGFMU_TRIGGER_OUT_POLARITY_POSITIVE);
    WGFMU_setTriggerOutEvent(leader, "trig", 0, 0);

    WGFMU_connect(101);
    WGFMU_connect(102);
    WGFMU_execute();
    WGFMU_waitUntilCompleted();


    fnum++;
}

/*
This is for STDP. Requires two files _ch1 and _ch2
*/
void pulse4(string fname, string wAmp, string eAmp, string endtime, string
       timeStepStr, string currentRange, string repeatCount)
{
    //string amplitude = ampl;
    csvparse3(fname, stod(wAmp), stod(eAmp));
    WGFMU_clear();
    double endCount = 0;
    int error = 1;
    int curPeat = 0;
    int wasZero = 1;
    //int dtint = stoi(dt);
    testWave2.waveData.push_back(0);
    testWave.waveData.push_back(0);
    double* v1 = &testWave.waveData[0];
    double* v2 = &testWave2.waveData[0];
```

```
int repeats = 1;
while (error == 1)
{
  error = 0;
  try
  {
    repeats = stoi(repeatCount);
  }
  catch (const std::invalid_argument& ia)
  {
    error = 1;
    cout << "Invalid Repeat Count \n";
    cout << "Repeat Count: ";
    cin >> repeatCount;
    cout << "\n";
  }
}
error = 0;



vector<double> testDT;
vector<double> test2DT;
vector<double> zeroV = { 0, 0 };
char* leader = "ch1";
char* lagger = "ch2";
int leaderch = 101;
int laggerch = 102;
string pulseParams = fname + "," + wAmp + "," + eAmp + "," + endtime + ","
   + timeStepStr + "," + currentRange + "," + repeatCount;

size_t size = 0;

if              (endtime.find_first_of("m",              size)             !=
   string::npos)endtime.replace(endtime.find_first_of("m",  size),  1,  "e-
   3");
else         if         (endtime.find_first_of("u",          size)          !=
   string::npos)endtime.replace(endtime.find_first_of("u",  size),  1,  "e-
   6");
else         if         (endtime.find_first_of("n",          size)          !=
   string::npos)endtime.replace(endtime.find_first_of("n",  size),  1,  "e-
   9");

if             (timeStepStr.find_first_of("m",             size)            !=
   string::npos)timeStepStr.replace(timeStepStr.find_first_of("m",   size),
   1, "e-3");
else        if        (timeStepStr.find_first_of("u",        size)          !=
   string::npos)timeStepStr.replace(timeStepStr.find_first_of("u",   size),
   1, "e-6");
else        if        (timeStepStr.find_first_of("n",        size)          !=
   string::npos)timeStepStr.replace(timeStepStr.find_first_of("n",   size),
   1, "e-9");


double minTime = 10E-9;
//double datStep = 10E-9;
double datStep = stod(endtime) / testWave.length;
```

```
if (datStep <= 10e-9) datStep = 10e-9;
double bigStep = 3 * datStep;

/*if (datStep <= 14.49E-9) datStep = 10E-9;
else if (datStep <= 24.49E-9) datStep = 20E-9;
else if (datStep <= 34.49E-9) datStep = 30E-9;
else if (datStep <= 44.49E-9) datStep = 40E-9;
else if (datStep <= 54.49E-9) datStep = 50E-9;
else if (datStep <= 64.49E-9) datStep = 60E-9;
else if (datStep <= 74.49E-9) datStep = 70E-9;*/



for (int i = 0; i < testWave.length; i++)
{
  if (testWave.rawData[i]==-1)
  {
    testDT.push_back(datStep*10);
    endCount += datStep*10;
  }
  else
  {
    testDT.push_back(datStep);
    endCount += datStep;
  }

  if (testWave2.rawData[i] == -1)
  {
    test2DT.push_back(datStep * 10);
  }
  else
  {
    test2DT.push_back(datStep);
  }
}

double* dt1 = &testDT[0];
double* dt2 = &test2DT[0];


//double endTime = datStep*testWave.length*repeats;



//double endTime = stod(endtime)*repeats;
//double endTime = testDT.size()*datStep;
double endTime = endCount;

//vector<double> zeroDT = { datStep, endTime };
//double* zDT = &zeroDT[0];
//double* zV = &zeroV[0];
WGFMU_createPattern(leader, 0);
WGFMU_createPattern(lagger, 0);

//WGFMU_addVectors(leader, leaderdts, v1, 6);
for (int i = 0; i < repeats; i++)
{
  WGFMU_addVectors(leader,  dt1,  v1,  testWave.length);  //Account  for
   repeats.
  WGFMU_addVectors(lagger, dt2, v2, testWave.length);
}
//WGFMU_addVectors(lagger, zDT, zV, 2);


//double timeStep = window / (numPoints -1);
//double timeStep = 10E-9;
double timeStep = stod(timeStepStr);
if (timeStep <= 10e-9) timeStep = 10e-9;
//int numPoints = (endTime / timeStep); //was 100000
//int numPoints = (datStep*testDT.size())/timeStep;
int numPoints = endTime / timeStep;

// Set the measurement events for both channels
//WGFMU_setMeasureEvent(leader,   "evt",   0,   numPoints,   timeStep,   0,
   WGFMU_MEASURE_EVENT_DATA_AVERAGED);
//WGFMU_setMeasureEvent(lagger,   "evt2",   0,   numPoints,   timeStep,   0,
   WGFMU_MEASURE_EVENT_DATA_AVERAGED); //no averaging
WGFMU_setMeasureEvent(leader, "evt", 0, numPoints, timeStep, timeStep -
   minTime, WGFMU_MEASURE_EVENT_DATA_AVERAGED);
WGFMU_setMeasureEvent(lagger, "evt2", 0, numPoints, timeStep, timeStep -
   minTime, WGFMU_MEASURE_EVENT_DATA_AVERAGED);


WGFMU_addSequence(101, leader, 1); // Add the waveform to WGFMU channel 1
WGFMU_addSequence(102, lagger, 1);

//ONLINE
WGFMU_initialize();

// Set the operation mode for each channel
WGFMU_setOperationMode(101, WGFMU_OPERATION_MODE_FASTIV);
WGFMU_setOperationMode(102, WGFMU_OPERATION_MODE_FASTIV);

// B1500 Defaults to measuring voltage. Change it to current. Also set the
   resolution
// for the ADC in the WGFMU
WGFMU_setMeasureMode(101, WGFMU_MEASURE_MODE_VOLTAGE);
WGFMU_setMeasureMode(102, WGFMU_MEASURE_MODE_CURRENT);
WGFMU_setMeasureVoltageRange(101, WGFMU_MEASURE_VOLTAGE_RANGE_10V);
if (currentRange == "1u" || currentRange == "1E-6" || currentRange == "1e-
   6" || currentRange == "1uA" || currentRange == "1ua" || currentRange ==
   "0.000001")
  WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_1UA);
else if (currentRange == "10u" || currentRange == "10E-6" || currentRange
   ==  "10e-6"  ||  currentRange  ==  "10uA"  ||  currentRange  ==  "10ua"  ||
   currentRange == "0.00001")
  WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_10UA);
else  if  (currentRange  ==  "100u"  ||  currentRange  ==  "100E-6"  ||
   currentRange == "100e-6" || currentRange == "100uA" || currentRange ==
   "100ua" || currentRange == "0.0001")
  WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_100UA);
else if (currentRange == "1m" || currentRange == "1E-3" || currentRange ==
   "1e-3"  ||  currentRange  ==  "1mA"  ||  currentRange  ==  "1ma"  ||
   currentRange == "0.001")
```

```
      WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_1MA);
    else if (currentRange == "10m" || currentRange == "10E-3" || currentRange
       == "10e-3" || currentRange == "10mA" || currentRange == "10ma" ||
       currentRange == "0.01")
      WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_10MA);
    else
    {
      cout << "wat";
      WGFMU_setMeasureCurrentRange(102, WGFMU_MEASURE_CURRENT_RANGE_10MA);
    }

    WGFMU_setTriggerOutMode(leaderch,    WGFMU_TRIGGER_OUT_MODE_START_SEQUENCE,
       WGFMU_TRIGGER_OUT_POLARITY_POSITIVE);
    WGFMU_setTriggerOutEvent(leader, "trig", 0, 0);

    WGFMU_connect(101);
    WGFMU_connect(102);
    WGFMU_execute();
    WGFMU_waitUntilCompleted();
    // Write the data to appropriate file
    ostringstream thefilename;
    ostringstream fileNumber;
    fileNumber << setfill('0') << setw(3) << fnum;

    //thefilename << "L:/MEC 107 Data/STDP/" << testID << "/1006301/Wafer10/"
       << dieNum << "/" << devNum << "/" << posAmplitude << "_" <<
       abs(negAmplitude) << "_" << dT << units << ".csv";
    thefilename << rootF << dieNum << "/" << devNum << "/" << devNum << "_" <<
       fname << "L_" << wAmp << "_" << eAmp << "_" << fileNumber.str() <<
       ".csv";
    //thefilename   <<   "C:/Users/koltondrake/Documents/STDP/STDP_data/"   <<
       "resultant/" << devNum << "/" << curAmp << "_" << fnum << ".csv";
    string f_name = thefilename.str();

    writeResults2ChannelP(101, 102, f_name.c_str(), pulseParams);
    //int ret = WGFMU_exportAscii(f_name.c_str());

    thefilename.str("");
    thefilename.clear();
    fileNumber.str("");
    fileNumber.clear();
    fnum++;
}

/*
This is for printing resultant waves. Requires two files _ch1 and _ch2
*/
void resultant(string fname, string wAmp, string eAmp)
{
    //string amplitude = ampl;
    csvparse3(fname, stod(wAmp), stod(eAmp));
    int l = fname.length() - 1;
    //WGFMU_clear();
    string outputF = fname.replace(2,1,"RL");
    cout << "\n" << outputF << "\n";

    double endCount = 0;
```

```
testWave2.waveData.push_back(0);
testWave.waveData.push_back(0);
double* v1 = &testWave.waveData[0];
double* v2 = &testWave2.waveData[0];


vector<double> actualT;
vector<double> actualV1;
vector<double> actualV2;
//actualT.push_back(0);
//actualV1.push_back(0);
//actualV2.push_back(0);


vector<double> testDT;
vector<double> test2DT;
vector<double> zeroV = { 0, 0 };

size_t size = 0;

double minTime = 10E-9;
//double datStep = 10E-9;
double datStep = 50e-9;
//double datStep = 10e-9;

for (int i = 0; i < testWave.length; i++)
{
  if (testWave.rawData[i] == -1)
  {
    for (int j = 0; j < 10; j++)
    {
      double tempV = (((double)j + 1) / 10)*-2 + 1;
      actualV1.push_back(tempV);

      testDT.push_back(datStep);
      actualT.push_back(endCount);
      endCount += datStep;
    }
  }
  else
  {
    actualV1.push_back(testWave.rawData[i]);
    testDT.push_back(datStep);
    actualT.push_back(endCount);
    endCount += datStep;
  }

  if (testWave2.rawData[i] == -1)
  {
    for (int j = 0; j < 10; j++)
    {
      double tempV = (((double)j + 1) / 10)*-2 + 1;
      actualV2.push_back(tempV);

      test2DT.push_back(datStep);
    }
  }
  else
  {
    actualV2.push_back(testWave2.rawData[i]);
    test2DT.push_back(datStep);
  }
}
double* actT = &actualT[0];
double* actV1 = &actualV1[0];
double* actV2 = &actualV2[0];
double* dt1 = &testDT[0];
double* dt2 = &test2DT[0];
double endTime = endCount;
double timeStep = 50e-9;
int numPoints = endTime / timeStep;

cout << "V\t" << "T\n";
for (int i = 0; i < numPoints; i++)
{
  cout << actV1[i]-actV2[i] << "\t" << actT[i] << "\n";
}

ostringstream thefilename;

thefilename  <<  rootF  <<  dieNum  <<  "/"  <<  devNum  <<  "/"  <<  outputF
   <<"_ch2.csv";

string f_name = thefilename.str();

FILE* fp = fopen(f_name.c_str(), "w");
if (fp != 0)
{
  fprintf(fp, "V1, T\n");
  for (int i = 0; i < numPoints; i++)
  {
    if (i == 0)
    {
      fprintf(fp, "%.9lf,", 0);
      fprintf(fp, "VOLATILE,1,0.1,0,%i\n",numPoints);
    }
    else fprintf(fp, "%.9lf\n", 0);
  } fclose(fp);
}

//int ret = WGFMU_exportAscii(f_name.c_str());

thefilename.str("");
thefilename.clear();
actualT.clear();
actualV1.clear();
actualV2.clear();
//actualT.push_back(0);
//actualV1.push_back(0);
//actualV2.push_back(0);


testDT.clear();
```

```
    test2DT.clear();
    fnum++;
}



double  dcSweep2(double  amplitude,  double  compliance,  int  pts,  string
      testName){

    ViSession defaultRM, vi;
    viOpenDefaultRM(&defaultRM);
    viOpen(defaultRM, "GPIB0::17::INSTR", VI_NULL, VI_NULL, &vi);

    viPrintf(vi, "*RST\n");

    viPrintf(vi, "CN 2,3\n");

    // Set ch2 to sweep measure mode (mode=2)
    // Params: mode(sweep),chnum
    viPrintf(vi, "MM 2,2\n");

    // Set ch2 sweep parameters
    char dcString[100];
    sprintf(dcString, "WV 2,1,200,0,%f,%i,%e\n", amplitude, pts, compliance);
    //cout << dcString;
    ostringstream os;
    //os << "WV 2,1,0,0," << amplitude << ",51," << compliance << "\n";
    //string sweepParam = os.str();
    string sweepParam = dcString;
    // Params: chnum,mode(linear),range(auto),start,stop,step,icomp
    viPrintf(vi,  (char*)sweepParam.c_str());  //  A  little  magic  necessary  to
      turn string into Char*

    os.str("");
    os.clear();

    // Force 0V at ch3 with auto-ranging and 100mA current limit.
    viPrintf(vi, "DV 3,0,0,0.1\n");

    // Set format to return 12 digits with a header, and return the sourcing
    // data
    // Params: format, mode
    viPrintf(vi, "FMT 2,1\n"); // Terminator = <CR/LF^EOI>/

    viPrintf(vi, "XE\n");

    char buf[102800];

    viScanf(vi, "%s", &buf);
    //       cout << deltaT << "\n";

    // Write the data to output
    string s = buf;
    std::stringstream ss(s);
    std::string item;
    vector<string> elems;
    while (std::getline(ss, item, ',')) {
```

```
  elems.push_back(item);
}

// Write the data to appropriate file

string tmp = testName;
int nameLength = tmp.length();

// make sure that the files are in a nice order
if (nameLength < 2)
{
  //os  <<  "L:/MEC  107  Data/STDP/"  <<  testID  <<  "/1006301/Wafer10/"  <<
   dieNum << "/" << devNum << "/Sweep_" << "0" << testName << ".csv";
  os  <<  rootF  <<  dieNum  <<  "/"  <<  devNum  <<  "/"  <<  devNum  <<  "0_"  <<
   testName<< ".csv";
}
else
{
  //os  <<  "L:/MEC  107  Data/STDP/"  <<  testID  <<  "/1006301/Wafer10/"  <<
   dieNum << "/" << devNum << "/Sweep_" << testName << ".csv";
  os  <<  rootF  <<  dieNum  <<  "/"  <<  devNum  <<  "/"  <<  devNum  <<  "_"  <<
   testName << ".csv";
}
//os << "L:/MEC 107 Data/STDP/" << testID << "/1006301/Wafer10/" << dieNum
   << "/" << devNum << "summary.csv";

string f_name = os.str();

os.str("");
os.clear();
double R_avg = 0;

FILE *fp = fopen(f_name.c_str(), "w");

int cnt = 1;
fprintf(fp, "V,I,R,G\n");

for (int i = 0; i < elems.size() - 1; i += 2)
{
  double vol = stod(elems[i + 1]); // Source data is returned after the
   force data
  double cur = stod(elems[i]);
  while (cur == 0)
  {
    if (elems.size() > (i + 2 * cnt)) cur = stod(elems[i + 2 * cnt]);
    else if ((i - 2 * cnt) >= 0) cur = stod(elems[i - 2 * cnt]);
    cnt++;
  }
  double res = abs(vol / cur);
  // Check for indeterminite or infinite values.
  if (isinf(res)) cout << "Infinite Value Supressed\n";
  if (isnan(res))   cout << "indeterminate value supressed";
  if (res == 0)res = 1;
  R_avg += isinf(res) || isnan(res) ? 1 : res;
  //fp << vol << "," << cur << "," << abs(res) << "\n";
  fprintf(fp, "%f,%e,%f,%e\n", vol, cur, res, 1 / res);
  //cout << vol << "\t" << cur << "\t" << res << "\n";
```

```
      //fp << elems[i + 1] << "," << elems[i] << "\n";
      //cout << elems[i + 1] << "\t" << elems[i] << "\n";
    }
    R_avg /= elems.size() - 2;
    R_avg *= 2;
    cout << "\tLast\tI: \t" << elems[elems.size() - 2] << "\n";
    cout << "\tAverage\tR: \t" << R_avg << "\n";
    fprintf(fp, ",R_avg,%f\n", R_avg);
    //fp.close();
    fclose(fp);
    //dcnum++;
    //cout << buf;

    viPrintf(vi, "CL 2,3\n");

    viClose(vi);
    viClose(defaultRM);
    return abs(R_avg);
}


//ColdT dcsweep.
double  dcSweep3(double  amplitude,  double  compliance,  int  pts,  string
      testName){

    ViSession defaultRM, vi;
    viOpenDefaultRM(&defaultRM);
    viOpen(defaultRM, "GPIB0::17::INSTR", VI_NULL, VI_NULL, &vi);

    viPrintf(vi, "*RST\n");

    //Enable slots 2 and 3
    viPrintf(vi, "CN 2,3\n");

    //Set slot 2 to sweep measure mode (mode=2) (Vtop)
    //Params: mode(sweep),chnum
    //MM works off of slots;
    viPrintf(vi, "MM 2,2\n");



    // Set slot 3 (SMU 1) sweep parameters
    char dcString[100];

    if (abs(amplitude) <= 500e-3)
    {
      //Sets ADC Integration settings.
      //Syntax: AIT type,mode[,N]
      //Type = 1: High-Resolution A/D
      //Mode = 1: Manual
      //N = Number of averages.
      viPrintf(vi, "AIT 1,1,100\n"); //High-Res A/Ds average 20 times.
      //Specifies ADC type for each channel.
      //Syntax: AAD chnum[, type]
      viPrintf(vi, "AAD 2,1\n"); //Sets slot 2 to High Res
      viPrintf(vi, "AAD 3,1\n"); //Sets slot 3 to High Res
```

```
  //0.5V Limited autoranging
  sprintf(dcString, "WV 2,1,5,0,%f,%i,%e\n", amplitude, pts, compliance);
}
else if (abs(amplitude) <= 5)
{
  //Sets ADC Integration settings.
  //Syntax: AIT type,mode[,N]
  //Type = 1: High-Resolution A/D
  //Mode = 1: Manual
  //N = Number of averages.
  viPrintf(vi, "AIT 1,1,50\n"); //High-Res A/Ds average 50 times.
  //Specifies ADC type for each channel.
  //Syntax: AAD chnum[, type]
  viPrintf(vi, "AAD 2,1\n"); //Sets slot 2 to High Res
  viPrintf(vi, "AAD 3,1\n"); //Sets slot 3 to High Res
  //5V limited autoranging
  sprintf(dcString, "WV 2,1,50,0,%f,%i,%e\n", amplitude, pts, compliance);
}
else
{
  //Sets ADC Integration settings.
  //Syntax: AIT type,mode[,N]
  //Type = 1: High-Resolution A/D
  //Mode = 1: Manual
  //N = Number of averages.
  viPrintf(vi, "AIT 1,1,50\n"); //High-Res A/Ds average 50 times.
  //Specifies ADC type for each channel.
  //Syntax: AAD chnum[, type]
  viPrintf(vi, "AAD 2,1\n"); //Sets slot 2 to High Res
  viPrintf(vi, "AAD 3,1\n"); //Sets slot 3 to High Res
  sprintf(dcString,    "WV    2,1,200,0,%f,%i,%e\n",    amplitude,    pts,
   compliance);
}

//cout << dcString;
ostringstream os;
//os << "WV 2,1,0,0," << amplitude << ",51," << compliance << "\n";
//string sweepParam = os.str();
string sweepParam = dcString;
// Params: chnum,mode(linear),range(auto),start,stop,step,icomp
viPrintf(vi,  (char*)sweepParam.c_str());  //  A  little  magic  necessary  to
   turn string into Char*

os.str("");
os.clear();

// Force 0V at slot3 (SMU 2) with auto-ranging and 100mA current limit.
viPrintf(vi, "DV 3,0,0,0.1\n");

// Set format to return 12 digits with a header, and return the sourcing
// data
// Params: format, mode
viPrintf(vi, "FMT 2,1\n"); // Terminator = <CR/LF^EOI>/

viPrintf(vi, "XE\n");

char buf[102800];
```

```
viScanf(vi, "%s", &buf);
//       cout << deltaT << "\n";

// Write the data to output
string s = buf;
std::stringstream ss(s);
std::string item;
vector<string> elems;
while (std::getline(ss, item, ',')) {
  elems.push_back(item);
}

// Write the data to appropriate file

string tmp = testName;
int nameLength = tmp.length();

// make sure that the files are in a nice order
os  <<  rootF  <<  dieNum  <<  "_"  <<  devNum  <<  "_"  <<  testName  <<  "_"  <<
   temperature << ".csv";

string f_name = os.str();

os.str("");
os.clear();
double R_avg = 0;

FILE *fp = fopen(f_name.c_str(), "w");

fprintf(fp, "V,I,R,G\n");
int cnt = 1;

for (int i = 0; i < elems.size() - 1; i += 2)
{
  double  vol  =  stod(elems[i  +  1]);  //  Source  data  is  returned  after  the
   force data
  double cur = stod(elems[i]);
  while (cur == 0)
  {
    if(elems.size() > (i + 2*cnt)) cur = stod(elems[i + 2*cnt]);
    else if ((i - 2 * cnt) >= 0) cur = stod(elems[i - 2 * cnt]);
    cnt++;
  }
  double res = abs(vol / cur);
  // Check for indeterminite or infinite values.
  if (isinf(res)) cout << "Infinite Value Supressed\n";
  if (isnan(res))   cout << "indeterminate value supressed";
  if (res == 0) res = 1;

  R_avg += isinf(res) || isnan(res) ? 1 : res;
  //fp << vol << "," << cur << "," << abs(res) << "\n";
  fprintf(fp, "%f,%e,%f,%e\n", vol, cur, res, 1 / res);
  //cout << vol << "\t" << cur << "\t" << res << "\n";

  //fp << elems[i + 1] << "," << elems[i] << "\n";
  //cout << elems[i + 1] << "\t" << elems[i] << "\n";
```

```
   }
   R_avg /= elems.size() - 2;
   R_avg *= 2;
   cout << "\tLast\tI: \t" << elems[elems.size() - 2] << "\n";
   cout << "\tAverage\tR: \t" << R_avg << "\n";
   fprintf(fp, ",R_avg,%f\n", R_avg);
   //fp.close();
   fclose(fp);
   dcnum++;
   //cout << buf;

   viPrintf(vi, "CL 2,3\n");

   viClose(vi);
   viClose(defaultRM);
   return abs(R_avg);
}



double  dcSweep4(double  amplitude,  double  compliance,  int  pts,  string
      testName){

   ViSession defaultRM, vi;
   viOpenDefaultRM(&defaultRM);
   viOpen(defaultRM, "GPIB0::17::INSTR", VI_NULL, VI_NULL, &vi);

   viPrintf(vi, "*RST\n");

   //Enable slots 2 and 3
   viPrintf(vi, "CN 2,3\n");

   //Set slot 2 to sweep measure mode (mode=2) (Vtop)
   //Params: mode(sweep),chnum
   //MM works off of slots;
   viPrintf(vi, "MM 2,2\n");
   ostringstream dcNum;

   dcNum << setfill('0') << setw(3) << dcnum;

   // Set slot 3 (SMU 1) sweep parameters
   char dcString[100];

   if (abs(amplitude) <= 500e-3)
   {
     //Sets ADC Integration settings.
     //Syntax: AIT type,mode[,N]
     //Type = 1: High-Resolution A/D
     //Mode = 1: Manual
     //N = Number of averages.
     viPrintf(vi, "AIT 1,1,20\n"); //High-Res A/Ds average 20 times.
     //Specifies ADC type for each channel.
     //Syntax: AAD chnum[, type]
     viPrintf(vi, "AAD 2,1\n"); //Sets slot 2 to High Res
     viPrintf(vi, "AAD 3,1\n"); //Sets slot 3 to High Res
     //0.5V Limited autoranging
     sprintf(dcString, "WV 2,1,5,0,%f,%i,%e\n", amplitude, pts, compliance);
```

```
}
else if (abs(amplitude) <= 5)
{
  //Sets ADC Integration settings.
  //Syntax: AIT type,mode[,N]
  //Type = 1: High-Resolution A/D
  //Mode = 1: Manual
  //N = Number of averages.
  viPrintf(vi, "AIT 1,1,50\n"); //High-Res A/Ds average 50 times.
  //Specifies ADC type for each channel.
  //Syntax: AAD chnum[, type]
  viPrintf(vi, "AAD 2,1\n"); //Sets slot 2 to High Res
  viPrintf(vi, "AAD 3,1\n"); //Sets slot 3 to High Res
  //5V limited autoranging
  sprintf(dcString, "WV 2,1,50,0,%f,%i,%e\n", amplitude, pts, compliance);
}
else
{
  //Sets ADC Integration settings.
  //Syntax: AIT type,mode[,N]
  //Type = 1: High-Resolution A/D
  //Mode = 1: Manual
  //N = Number of averages.
  viPrintf(vi, "AIT 1,1,50\n"); //High-Res A/Ds average 50 times.
  //Specifies ADC type for each channel.
  //Syntax: AAD chnum[, type]
  viPrintf(vi, "AAD 2,1\n"); //Sets slot 2 to High Res
  viPrintf(vi, "AAD 3,1\n"); //Sets slot 3 to High Res
  sprintf(dcString,    "WV    2,1,200,0,%f,%i,%e\n",    amplitude,    pts,
   compliance);
}

//cout << dcString;
ostringstream os;
//os << "WV 2,1,0,0," << amplitude << ",51," << compliance << "\n";
//string sweepParam = os.str();
string sweepParam = dcString;
// Params: chnum,mode(linear),range(auto),start,stop,step,icomp
viPrintf(vi,  (char*)sweepParam.c_str());  //  A  little  magic  necessary  to
   turn string into Char*

os.str("");
os.clear();

// Force 0V at slot3 (SMU 2) with auto-ranging and 100mA current limit.
viPrintf(vi, "DV 3,0,0,0.1\n");

// Set format to return 12 digits with a header, and return the sourcing
// data
// Params: format, mode
viPrintf(vi, "FMT 2,1\n"); // Terminator = <CR/LF^EOI>/

viPrintf(vi, "XE\n");

char buf[102800];

viScanf(vi, "%s", &buf);
```

```
//       cout << deltaT << "\n";

// Write the data to output
string s = buf;
std::stringstream ss(s);
std::string item;
vector<string> elems;
while (std::getline(ss, item, ',')) {
  elems.push_back(item);
}

// Write the data to appropriate file

string tmp = testName;
int nameLength = tmp.length();

// make sure that the files are in a nice order
//os << rootF << dieNum << "_" << devNum << "_" << testName << "_" <<
   dcNum.str() << ".csv"; //ColdT name

//Old Version 11AM 1-28
ostringstream fileNumber;
fileNumber << setfill('0') << setw(3) << fnum;

//os << rootF << dieNum << "/" << devNum << "/" << devNum << "_" <<
   testName << "_" << fileNumber.str() << ".csv";

//New Version
os << rootF << dieNum << "/" << devNum << "_" << testName <<  "_" <<
   fileNumber.str() << ".csv";

string f_name = os.str();

os.str("");
os.clear();
double R_avg = 0;

FILE *fp = fopen(f_name.c_str(), "w");

fprintf(fp, "V,I,R,G\n");
int cnt = 1;

//Often, the first 5 points or so will have the SMU in a different range
   than is required.
//For many-point sweeps, tossing out these points gives a more accurate
   measurement.
int firstAvgToss = 10;

if (pts > 50) firstAvgToss = 10;
else firstAvgToss = 0;



for (int i = 0; i < elems.size() - 1; i += 2)
{
  double vol = stod(elems[i + 1]); // Source data is returned after the
   force data
```

```
      double cur = stod(elems[i]);
      while (cur == 0)
      {
        if (elems.size() >(i + 2 * cnt)) cur = stod(elems[i + 2 * cnt]);
        else if ((i - 2 * cnt) >= 0) cur = stod(elems[i - 2 * cnt]);
        cnt++;
      }
      double res = abs(vol / cur);
      // Check for indeterminite or infinite values.
      if (isinf(res)) cout << "Infinite Value Supressed\n";
      if (isnan(res))   cout << "indeterminate value supressed";
      if (res == 0) res = 1;

      if(i>firstAvgToss+1) R_avg += isinf(res) || isnan(res) ? 1 : res;
      //fp << vol << "," << cur << "," << abs(res) << "\n";
      fprintf(fp, "%f,%e,%f,%e\n", vol, cur, res, 1 / res);
      //cout << vol << "\t" << cur << "\t" << res << "\n";

      //fp << elems[i + 1] << "," << elems[i] << "\n";
      //cout << elems[i + 1] << "\t" << elems[i] << "\n";
    }

    R_avg /= elems.size() - 2 + firstAvgToss;
    R_avg *= 2;
    cout << "\tLast\tI: \t" << elems[elems.size() - 2] << "\n";
    cout << "\tAverage\tR: \t" << R_avg << "\n";
    fprintf(fp, ",R_avg,%f\n", R_avg);
    //fp.close();
    fclose(fp);
    dcnum++;
    //cout << buf;
    fileNumber.str("");
    fileNumber.clear();
    fnum++;
    viPrintf(vi, "CL 2,3\n");

    viClose(vi);
    viClose(defaultRM);
    return abs(R_avg);
}

double  dcSweep(double  amplitude,  double  compliance,  int  pts,  string
      testName){

    ViSession defaultRM, vi;
    viOpenDefaultRM(&defaultRM);
    viOpen(defaultRM, "GPIB0::17::INSTR", VI_NULL, VI_NULL, &vi);

    viPrintf(vi, "*RST\n");

    viPrintf(vi, "CN 2,3\n");

    // Set ch2 to sweep measure mode (mode=2)
    // Params: mode(sweep),chnum
    //MM works off of slots;
    viPrintf(vi, "MM 2,2\n");
```

```
// Set slot 3 (SMU 1) sweep parameters
char dcString[100];
sprintf(dcString, "WV 2,1,200,0,%f,%i,%e\n", amplitude, pts, compliance);
//cout << dcString;
ostringstream os;
//os << "WV 2,1,0,0," << amplitude << ",51," << compliance << "\n";
//string sweepParam = os.str();
string sweepParam = dcString;
// Params: chnum,mode(linear),range(auto),start,stop,step,icomp
viPrintf(vi,  (char*)sweepParam.c_str()); //  A  little  magic  necessary  to
   turn string into Char*

os.str("");
os.clear();

// Force 0V at slot3 (SMU 2) with auto-ranging and 100mA current limit.
viPrintf(vi, "DV 3,0,0,0.1\n");

// Set format to return 12 digits with a header, and return the sourcing
// data
// Params: format, mode
viPrintf(vi, "FMT 2,1\n"); // Terminator = <CR/LF^EOI>/

viPrintf(vi, "XE\n");

char buf[102800];

viScanf(vi, "%s", &buf);
//       cout << deltaT << "\n";

// Write the data to output
string s = buf;
std::stringstream ss(s);
std::string item;
vector<string> elems;
while (std::getline(ss, item, ',')) {
  elems.push_back(item);
}

// Write the data to appropriate file

string tmp = testName;
int nameLength = tmp.length();

// make sure that the files are in a nice order
if (nameLength < 2)
{
  //os  <<  "L:/MEC  107  Data/STDP/"  <<  testID  <<  "/1006301/Wafer10/"  <<
   dieNum << "/" << devNum << "/Sweep_" << "0" << testName << ".csv";
  os << rootF << dieNum << "/" << devNum << "/" << devNum << "_sweep_" <<
   "0" << testName << "_"<< dcnum << ".csv";
}
else
{
  //os  <<  "L:/MEC  107  Data/STDP/"  <<  testID  <<  "/1006301/Wafer10/"  <<
   dieNum << "/" << devNum << "/Sweep_" << testName << ".csv";
```

```
      os << rootF << dieNum << "/" << devNum << "/" << devNum << "_sweep_" <<
        testName << "_" << dcnum << ".csv";
    }
    //os << "L:/MEC 107 Data/STDP/" << testID << "/1006301/Wafer10/" << dieNum
       << "/" << devNum << "summary.csv";

    string f_name = os.str();

    os.str("");
    os.clear();
    double R_avg = 0;

    FILE *fp = fopen(f_name.c_str(), "w");

    fprintf(fp, "V,I,R,G\n");
    int cnt = 1;
    for (int i = 0; i < elems.size() - 1; i += 2)
    {
      double vol = stod(elems[i + 1]); // Source data is returned after the
        force data
      double cur = stod(elems[i]);
      while (cur == 0)
      {
        if (elems.size() > (i + 2 * cnt)) cur = stod(elems[i + 2 * cnt]);
        else if ((i - 2 * cnt) >= 0) cur = stod(elems[i - 2 * cnt]);
        cnt++;
      }
      double res = abs(vol / cur);
      // Check for indeterminite or infinite values.
      if (isinf(res)) cout << "Infinite Value Supressed\n";
      if (isnan(res))   cout << "indeterminate value supressed";
      if (res == 0)res = 1;
      R_avg += isinf(res) || isnan(res) ? 1 : res;
      //fp << vol << "," << cur << "," << abs(res) << "\n";
      fprintf(fp, "%f,%e,%f,%e\n", vol, cur, res, 1 / res);
      //cout << vol << "\t" << cur << "\t" << res << "\n";

      //fp << elems[i + 1] << "," << elems[i] << "\n";
      //cout << elems[i + 1] << "\t" << elems[i] << "\n";
    }
    R_avg /= elems.size()-2;
    R_avg *= 2;
    cout << "\tLast\tI: \t" << elems[elems.size() - 2] << "\n";
    cout << "\tAverage\tR: \t" << R_avg << "\n";
    fprintf(fp, ",R_avg,%f\n", R_avg);
    //fp.close();
    fclose(fp);
    dcnum++;
    //cout << buf;

    viPrintf(vi, "CL 2,3\n");

    viClose(vi);
    viClose(defaultRM);
    return abs(R_avg);
}
```

```
void  resetTests(int  numTests,  double  riseTime,  double  fallTime,  double
      widthAtHalfMax, double tail, double posAmplitude, double negAmplitude,
      double writeScale)
{
   vector<int> testNums;
   vector<double> deltaGs;
   double deltaG;
   double maxG;
   double thisG = 1 / dcSweep(20E-3, 10E-3, 51, "Init"); // Get an initial
      conductance reading
   maxG = thisG;
   double previousG;

   double deltaT;
   cout << "Enter a spike timing interval to test: ";
   cin >> deltaT;
   int k = 1;
   int m = 2;

   for (int i = 1; i <= numTests; i++) {

     testNums.push_back(i);

     createSpikes(riseTime,  fallTime,  widthAtHalfMax,  tail,  posAmplitude,
      negAmplitude, deltaT, writeScale, to_string(k));
     previousG = thisG;
     thisG = 1 / dcSweep(20E-3, 10E-3, 51, to_string(k));
     maxG = thisG > maxG ? thisG : maxG;

     deltaG = (thisG - previousG);
     deltaGs.push_back(deltaG); // change in conductance calculation

     createSpikes(riseTime,  fallTime,  widthAtHalfMax,  tail,  posAmplitude,
      negAmplitude, 12E-3, writeScale, to_string(m));
     previousG = thisG;
     thisG = 1 / dcSweep(20E-3, 10E-3, 51, to_string(m));
     k += 2;
     m += 2;
   }
   double scaleFactor = 100 / maxG;

   transform(deltaGs.begin(),         deltaGs.end(),         deltaGs.begin(),
      std::bind1st(multiplies<double>(), scaleFactor));

   ostringstream summary_filename;
   //summary_filename   <<   "L:/MEC   107   Data/STDP/"   <<   testID   <<
      "/1006301/Wafer10/"  <<  dieNum  <<  "/"  <<  devNum  <<  "/"  <<  devNum  <<
      "_summary.csv";
   //summary_filename  <<  "C:/Users/koltondrake/Documents/STDP/STDP_data/"  <<
      testID  <<  "/1006301/Wafer10/"  <<  dieNum  <<  "/"  <<  devNum  <<  "/"  <<
      devNum << "_summary.csv";
   summary_filename << rootF << dieNum << "/" << devNum << "/" << devNum <<
      "_summary.csv";
   string fname = summary_filename.str();
   ofstream fp;
   fp.open(fname);
   fp << "Test #,dG\n";
   for (int i = 0; i < testNums.size(); i++)
   {
     fp << testNums[i] << "," << deltaGs[i] << "\n";
   }
   fp.close();
   summary_filename.str("");
   summary_filename.clear();
}

// Make sure that if the window is closed or the process is stopped the VISA
      resources
// are shut down
BOOL CtrlHandler(DWORD)
{
   //MessageBox(NULL,  "Program  closed",  "Message",  MB_ICONEXCLAMATION  |
      MB_OK);
   WGFMU_closeSession();
   viClose(vi);
   viClose(defaultRM);
   FILE *f = fopen("C:/Users/koltondrake/Desktop/dump.txt", "w");
   fprintf(f, "Program Didn't End Well");
   exit(-1);
}


int introQuery()
{
   SetConsoleCtrlHandler((PHANDLER_ROUTINE)&CtrlHandler, TRUE);

   try{
     ViSession defaultRM, vi;
     viOpenDefaultRM(&defaultRM);
     //viOpen(defaultRM, "GPIB0::17::INSTR", VI_NULL, VI_NULL, &vi);
     viOpen(defaultRM, "GPIB0::17::INSTR", VI_NULL, VI_NULL, &vi);
     viPrintf(vi, "*RST\n");
     viClose(vi);
     viClose(defaultRM);

     int ch1 = 101;
     int ch2 = 102;
     string shouldCondition;
     cout << "What is the test identifier for this test?\n";
     cin >> testID;
     cout << "DIE NUMBER (in the form DieXX, where XX is the number)\n";
     cin >> dieNum;
     cout << "Device Number (in the form DevXX, where XX is the number)\n";
     cin >> devNum;
     cout << "Would you like to condition the device before running learning
      spikes? Y/N\n";
     cin >> shouldCondition;
     //                                  //transform(shouldCondition.begin(),
      shouldCondition.end(),shouldCondition.begin(),::tolower);
     // Set the full width half max parameters (for the positive going spike)
     double widthAtHalfMax = 10E-3;
     double riseTime = 9E-3;
     double fallTime = 9E-3;
     double posAmplitude = 0.3;
```

```
      double negAmplitude = -.2;
      double tail = 22E-3;

      // Set the maximum deltaT that you want to test.
      double maxDT = 40E-3;

      // Set the number of tests to run (this will be used to sample at even
       intevals between the min
      // and max deltaT values). Remember that these tests are double sided,
       so both the positive and
      // negative side will be run, resulting in twice as many tests as you
       say. The endpoints of your
      // test range will always be run, so take this into account if you'd
       like all of the intermediate
      // test numbers to be 'nice' numbers.
      int numTests = 5; // You always probably mean to do one more than you
       think (think ends)

      //WGFMU_openSession("GPIB0::17::INSTR");
      WGFMU_openSession("GPIB0::17::INSTR");
      if (shouldCondition == "y")
      {
        double R1 = dcSweep(20E-3, 10E-3, 51, "R1");
        double W1 = dcSweep(2, 1E-6, 51, "W1");
        double R2 = dcSweep(20E-3, 10E-3, 51, "R2");
        double E1 = dcSweep(-1, 20E-6, 51, "E1");
        double R3 = dcSweep(20E-3, 10E-3, 51, "R3");
        double W2 = dcSweep(2, 1E-6, 51, "W2");
        double R4 = dcSweep(20E-3, 10E-3, 51, "R4");
      }
      string tnum = "test0";
      //             createSpikes(riseTime,   fallTime,   widthAtHalfMax,   tail,
       posAmplitude, negAmplitude,40E-3,1.0,tnum );
      runStdpSuite(maxDT, numTests, riseTime, fallTime, widthAtHalfMax, tail,
       posAmplitude, negAmplitude);
      //       resetTests(numTests, riseTime, fallTime, widthAtHalfMax, tail,
       posAmplitude, negAmplitude,1.0);
      dcSweep(2, 20E-6, 51, "conditioning");
      //dcSweep(20E-3, 10E-3, "read");
      WGFMU_initialize(); //WGFMU_disconnect(101);
      WGFMU_closeSession();
    }
    catch (...) {
      WGFMU_closeSession();
      viClose(vi);
      viClose(defaultRM);
      FILE *f = fopen("C:/Users/koltondrake/Desktop/dump.txt", "w");
      fprintf(f, "Program Didn't End Well");
      exit(-1);
    }
}

int csvparse(string cmd,double amplitude, int dt)
{
    string line, filename = "C:/Users/koltondrake/Documents/STDP/stdp_A1.csv";
    if (cmd != "w"||cmd!="e" )
    {
```

```
  filename = "C:/Users/koltondrake/Documents/STDP/"+cmd+".csv";
}
vector<string> row;
double amplitudeL = amplitude;
//if (cmd == "w" && amplitudeL<0) amplitudeL *= -1;
//if (cmd == "e" && amplitudeL>0) amplitudeL *= -1;
curAmp = amplitudeL;
ifstream in(filename);
if (in.fail())  { cout << "File not found" << endl; return 0; }
int rowCount = 0;
string::size_type sz;
double d;
testWave.waveData.clear();
testWave2.waveData.clear();


while (getline(in, line) && in.good())
{
  csvline_populate(row, line, ',');
  rowCount++;

  if (rowCount == 2)
  {
    int leng = row.size();
    for (int i = 0; i < leng; i++)
    {
      //cout << row[i] << "\t";
      try
      {
         d = stod(row[i], &sz);
      }
      catch (...)
      {
         //cout << "oops"; //we can't convert the value to a double.
         //break;
      }
      if (i == 2)
      {
         testWave.freq = d;
         testWave2.freq = d;
      }
      if (i == 3)
      {
         testWave.amp = d;
         testWave2.amp = d;
      }
      if (i == 4)
      {
         testWave.offset = d;
         testWave2.offset = d;
      }
      if (i == 5)
      {
         testWave.length = (int)d;
         testWave2.length = (int)d;
      }
    }
```

```
      }


      int leng = row.size();
      if (rowCount >= 2)
      {

        for (int i = 0; i < leng; i++)
        {
          //cout << row[i] << "\t";
          try
          {
             d = stod(row[i], &sz);
          }
          catch (...)
          {
             //cout << "oops"; //we can't convert the value to a double.
             break;
          }
          //if (i == 0)testWave.waveData.push_back(d*testWave.amp);
          if (i == 0)
          {
             testWave.waveData.push_back(d*amplitudeL);
             testWave2.waveData.push_back(d*amplitudeL);
          }
        }
        //cout << endl;
      }
    }
    in.close();
    vector<double>::iterator it;
    if (cmd == "w")
    {
      it = testWave.waveData.begin();
      for (int i = 0; i < dt; i++)
      {
        it = testWave.waveData.insert(it, 0);
      }
    }
    if (cmd == "e")
    {
      it = testWave2.waveData.begin();
      for (int i = 0; i < dt; i++)
      {
        it = testWave2.waveData.insert(it, 0);
      }
    }
    return 0;
}

int csvparse1(string cmd, double wAmp, double eAmp)
{
    string line;
    string filename = "C:/Users/koltondrake/Documents/STDP/" + cmd + ".csv";
    vector<string> row;

    ifstream in(filename);
```

```
if (in.fail())  { cout << "File not found" << endl; return 0; }
int rowCount = 0;
string::size_type sz;
double d;
testWave.waveData.clear();
testWave2.waveData.clear();


while (getline(in, line) && in.good())
{
  csvline_populate(row, line, ',');
  rowCount++;

  if (rowCount == 2)
  {
    int leng = row.size();
    for (int i = 0; i < leng; i++)
    {
      //cout << row[i] << "\t";
      try
      {
         d = stod(row[i], &sz);
      }
      catch (...)
      {
         //cout << "oops"; //we can't convert the value to a double.
         //break;
      }
      if (i == 2)
      {
         testWave.freq = d;
         testWave2.freq = d;
      }
      if (i == 3)
      {
         testWave.amp = d;
         testWave2.amp = d;
      }
      if (i == 4)
      {
         testWave.offset = d;
         testWave2.offset = d;
      }
      if (i == 5)
      {
         testWave.length = (int)d;
         testWave2.length = (int)d;
      }
    }
  }


  int leng = row.size();
  if (rowCount >= 2)
  {

    for (int i = 0; i < leng; i++)
```

```
       {
         //cout << row[i] << "\t";
         try
         {
            d = stod(row[i], &sz);
         }
         catch (...)
         {
            //cout << "oops"; //we can't convert the value to a double.
            break;
         }
         //if (i == 0)testWave.waveData.push_back(d*testWave.amp);
         if (i == 0)
         {
            if (d == 0.05) //We see a read
            {
              testWave.waveData.push_back(0.2);
              testWave2.waveData.push_back(0);
            }
            else if (d > 0)
            {
              testWave.waveData.push_back(d*wAmp); //Positive voltages go to
      Channel 1
              testWave2.waveData.push_back(0);  //Negative  voltages  go  to
      Channel 2 (as positive voltages).
            }
            else if (d < 0)
            {
              testWave.waveData.push_back(d*eAmp); //Negative voltages go to
      Channel 2 (as positive voltages).
              testWave2.waveData.push_back(0);
            }
            else if (d == 0)
            {
              testWave.waveData.push_back(0);
              testWave2.waveData.push_back(0);
            }
         }
       }
       //cout << endl;
     }
   }
   in.close();
   return 0;
}



/**
This csvparse is for taking a csv file and placing negative amplitudes on
      Channel 2 and positive amplitudes on Channel 1.
*/
int csvparse2(string cmd, double wAmp, double eAmp)
{
   string line;
   string filename = "C:/Users/koltondrake/Documents/STDP/" + cmd + ".csv";
   vector<string> row;
```

```
ifstream in(filename);
if (in.fail())  { cout << "File not found" << endl; return 0; }
int rowCount = 0;
string::size_type sz;
double d;
testWave.waveData.clear();
testWave2.waveData.clear();


while (getline(in, line) && in.good())
{
  csvline_populate(row, line, ',');
  rowCount++;

  if (rowCount == 2)
  {
    int leng = row.size();
    for (int i = 0; i < leng; i++)
    {
      //cout << row[i] << "\t";
      try
      {
         d = stod(row[i], &sz);
      }
      catch (...)
      {
         //cout << "oops"; //we can't convert the value to a double.
         //break;
      }
      if (i == 2)
      {
         testWave.freq = d;
         testWave2.freq = d;
      }
      if (i == 3)
      {
         testWave.amp = d;
         testWave2.amp = d;
      }
      if (i == 4)
      {
         testWave.offset = d;
         testWave2.offset = d;
      }
      if (i == 5)
      {
         testWave.length = (int)d;
         testWave2.length = (int)d;
      }
    }
  }


  int leng = row.size();
  if (rowCount >= 2)
  {
      for (int i = 0; i < leng; i++)
      {
        //cout << row[i] << "\t";
        try
        {
           d = stod(row[i], &sz);
        }
        catch (...)
        {
           //cout << "oops"; //we can't convert the value to a double.
           break;
        }
        //if (i == 0)testWave.waveData.push_back(d*testWave.amp);
        if (i == 0)
        {
           if (d == 0.05) //We see a read
           {
             testWave.waveData.push_back(0.2);
             testWave2.waveData.push_back(0);
           }
           else if (d > 0)
           {
             testWave.waveData.push_back(d*wAmp); //Positive voltages go to
      Channel 1
             testWave2.waveData.push_back(0);  //Negative  voltages  go  to
      Channel 2 (as positive voltages).
           }
           else if (d < 0)
           {
             testWave.waveData.push_back(0);
             testWave2.waveData.push_back(d*eAmp*-1); //Negative voltages go
      to Channel 2 (as positive voltages).
           }
           else if (d == 0)
           {
             testWave.waveData.push_back(0);
             testWave2.waveData.push_back(0);
           }
        }
      }
      //cout << endl;
    }
  }
  in.close();
  return 0;
}

/**
This  csvparse  is  for  taking  a  csv  file  and  placing  negative  amplitudes  on
      Channel 2 and positive amplitudes on Channel 1.
*/
int csvparse3(string cmd, double wAmp, double eAmp)
{
  string line,line2;
  string   filename   =   "C:/Users/koltondrake/Documents/STDP/"   +   cmd   +
      "_ch1.csv";
```

```
string   filename2   =   "C:/Users/koltondrake/Documents/STDP/"   +   cmd   +
   "_ch2.csv";
vector<string> row,row2;

ifstream in(filename);
ifstream in2(filename2);
if (in.fail()||in2.fail())  { cout << "File not found" << endl; return 0;
   }
int rowCount = 0;
int rowCount2 = 0;

string::size_type sz;
double d;


testWave.waveData.clear();
testWave.rawData.clear();

testWave2.waveData.clear();
testWave2.rawData.clear();

while (getline(in, line) && in.good())
{
  csvline_populate(row, line, ',');
  rowCount++;

  if (rowCount == 2)
  {
    int leng = row.size();
    for (int i = 0; i < leng; i++)
    {
      //cout << row[i] << "\t";
      try
      {
         d = stod(row[i], &sz);
      }
      catch (...)
      {
         //cout << "oops"; //we can't convert the value to a double.
         //break;
      }
      if (i == 2)
      {
         testWave.freq = d;
         //testWave2.freq = d;
      }
      if (i == 3)
      {
         testWave.amp = d;
         //testWave2.amp = d;
      }
      if (i == 4)
      {
         testWave.offset = d;
         //testWave2.offset = d;
      }
      if (i == 5)
```

```
        {
           testWave.length = (int)d;
           //testWave2.length = (int)d;
        }
      }
    }


    int leng = row.size();
    if (rowCount >= 2)
    {

      for (int i = 0; i < leng; i++)
      {
        //cout << row[i] << "\t";
        try
        {
           d = stod(row[i], &sz);
        }
        catch (...)
        {
           //cout << "oops"; //we can't convert the value to a double.
           break;
        }
        //if (i == 0)testWave.waveData.push_back(d*testWave.amp);
        if (i == 0)
        {
           testWave.rawData.push_back(d);
           if (d == 0.05) //We see a read
           {
             testWave.waveData.push_back(0.05);
             //testWave2.waveData.push_back(0);
           }
           else if (d > 0)
           {
             testWave.waveData.push_back(d*wAmp);  //Positive  voltages  go  to
     Channel 1
             //testWave2.waveData.push_back(0);  //Negative  voltages  go  to
     Channel 2 (as positive voltages).
           }
           else if (d < 0)
           {
             testWave.waveData.push_back(d*eAmp);
             //testWave2.waveData.push_back(d*eAmp*-1);  //Negative  voltages
     go to Channel 2 (as positive voltages).
           }
           else if (d == 0)
           {
             testWave.waveData.push_back(0);
             //testWave2.waveData.push_back(0);
           }
        }
      }
      //cout << endl;
    }
  }
  in.close();
```

```
while (getline(in2, line2) && in2.good())
{
  csvline_populate(row2, line2, ',');
  rowCount2++;

  if (rowCount2 == 2)
  {
    int leng = row2.size();
    for (int i = 0; i < leng; i++)
    {
      //cout << row[i] << "\t";
      try
      {
         d = stod(row2[i], &sz);
      }
      catch (...)
      {
         //cout << "oops"; //we can't convert the value to a double.
         //break;
      }
      if (i == 2)
      {
         //testWave.freq = d;
         testWave2.freq = d;
      }
      if (i == 3)
      {
         //testWave.amp = d;
         testWave2.amp = d;
      }
      if (i == 4)
      {
         //testWave.offset = d;
         testWave2.offset = d;
      }
      if (i == 5)
      {
         //testWave.length = (int)d;
         testWave2.length = (int)d;
      }
    }
  }


  int leng = row2.size();
  if (rowCount2 >= 2)
  {

    for (int i = 0; i < leng; i++)
    {
      //cout << row[i] << "\t";
      try
      {
         d = stod(row2[i], &sz);
      }
      catch (...)
```

```
         {
            //cout << "oops"; //we can't convert the value to a double.
            break;
         }
         if (i == 0)
         {
            testWave2.rawData.push_back(d);
            if (d == 0.05) //We see a read
            {
              testWave2.waveData.push_back(0.05);

              //testWave2.waveData.push_back(0);
            }
            else if (d > 0)
            {
              testWave2.waveData.push_back(d*wAmp); //Positive voltages go to
      Channel 1

              //testWave2.waveData.push_back(0);  //Negative  voltages  go  to
      Channel 2 (as positive voltages).
            }
            else if (d < 0)
            {
              testWave2.waveData.push_back(d*eAmp);

              //testWave2.waveData.push_back(d*eAmp*-1);  //Negative  voltages
      go to Channel 2 (as positive voltages).
            }
            else if (d == 0)
            {
              testWave2.waveData.push_back(0);

              //testWave2.waveData.push_back(0);
            }
         }
       }
     }
   }
   in2.close();

   return 0;
}

/**
This csvparse is for reading in the "Read" bumps.
*/
int csvparseRead(string cmd, double wAmp, double eAmp)
{
   string line, line2;
   string   filename   =   "C:/Users/koltondrake/Documents/STDP/"   +   cmd   +
      "_ch1.csv";
   string   filename2   =   "C:/Users/koltondrake/Documents/STDP/"   +   cmd   +
      "_ch2.csv";
   vector<string> row, row2;

   ifstream in(filename);
   ifstream in2(filename2);
```

```
if (in.fail() || in2.fail())  { cout << "File not found" << endl; return
   0; }
int rowCount = 0;
int rowCount2 = 0;

string::size_type sz;
double d;


testWave3.waveData.clear();
testWave3.rawData.clear();

testWave4.waveData.clear();
testWave4.rawData.clear();

while (getline(in, line) && in.good())
{
  csvline_populate(row, line, ',');
  rowCount++;

  if (rowCount == 2)
  {
    int leng = row.size();
    for (int i = 0; i < leng; i++)
    {
      //cout << row[i] << "\t";
      try
      {
         d = stod(row[i], &sz);
      }
      catch (...)
      {
         //cout << "oops"; //we can't convert the value to a double.
         //break;
      }
      if (i == 2)
      {
         testWave3.freq = d;
         //testWave2.freq = d;
      }
      if (i == 3)
      {
         testWave3.amp = d;
         //testWave2.amp = d;
      }
      if (i == 4)
      {
         testWave3.offset = d;
         //testWave2.offset = d;
      }
      if (i == 5)
      {
         testWave3.length = (int)d;
         //testWave2.length = (int)d;
      }
    }
  }
  int leng = row.size();
  if (rowCount >= 2)
  {

    for (int i = 0; i < leng; i++)
    {
      //cout << row[i] << "\t";
      try
      {
         d = stod(row[i], &sz);
      }
      catch (...)
      {
         //cout << "oops"; //we can't convert the value to a double.
         break;
      }
      //if (i == 0)testWave.waveData.push_back(d*testWave.amp);
      if (i == 0)
      {
         testWave.rawData.push_back(d);
         if (d == 0.02) //We see a read
         {
           testWave3.waveData.push_back(0.02);
           //testWave2.waveData.push_back(0);
         }
         else if (d > 0)
         {
           testWave3.waveData.push_back(d*wAmp); //Positive voltages go to
   Channel 1
           //testWave2.waveData.push_back(0);  //Negative  voltages  go  to
   Channel 2 (as positive voltages).
         }
         else if (d < 0)
         {
           testWave3.waveData.push_back(d*eAmp);
           //testWave2.waveData.push_back(d*eAmp*-1);  //Negative  voltages
   go to Channel 2 (as positive voltages).
         }
         else if (d == 0)
         {
           testWave3.waveData.push_back(0);
           //testWave2.waveData.push_back(0);
         }
      }
    }
    //cout << endl;
  }
}
in.close();

while (getline(in2, line2) && in2.good())
{
  csvline_populate(row2, line2, ',');
  rowCount2++;
```

```
if (rowCount2 == 2)
{
  int leng = row2.size();
  for (int i = 0; i < leng; i++)
  {
    //cout << row[i] << "\t";
    try
    {
       d = stod(row2[i], &sz);
    }
    catch (...)
    {
       //cout << "oops"; //we can't convert the value to a double.
       //break;
    }
    if (i == 2)
    {
       //testWave.freq = d;
       testWave4.freq = d;
    }
    if (i == 3)
    {
       //testWave.amp = d;
       testWave4.amp = d;
    }
    if (i == 4)
    {
       //testWave.offset = d;
       testWave4.offset = d;
    }
    if (i == 5)
    {
       //testWave.length = (int)d;
       testWave4.length = (int)d;
    }
  }
}


int leng = row2.size();
if (rowCount2 >= 2)
{

  for (int i = 0; i < leng; i++)
  {
    //cout << row[i] << "\t";
    try
    {
       d = stod(row2[i], &sz);
    }
    catch (...)
    {
       //cout << "oops"; //we can't convert the value to a double.
       break;
    }
    if (i == 0)
    {
         testWave4.rawData.push_back(d);
         if (d == 0.05) //We see a read
         {
           testWave4.waveData.push_back(0.05);

           //testWave2.waveData.push_back(0);
         }
         else if (d > 0)
         {
           testWave4.waveData.push_back(d*wAmp); //Positive voltages go to
   Channel 1

           //testWave2.waveData.push_back(0);  //Negative  voltages  go  to
   Channel 2 (as positive voltages).
         }
         else if (d < 0)
         {
           testWave4.waveData.push_back(d*eAmp);

           //testWave2.waveData.push_back(d*eAmp*-1);  //Negative  voltages
   go to Channel 2 (as positive voltages).
         }
         else if (d == 0)
         {
           testWave4.waveData.push_back(0);

           //testWave2.waveData.push_back(0);
         }
      }
    }
  }
 }
 in2.close();

 return 0;
}

// Reads in a single line from a waveform file.

void  csvline_populate(vector<string>  &record,  const  string&  line,  char
   delimiter)
{
 int linepos = 0;
 int inquotes = false;
 char c;
 int i;
 int linemax = line.length();
 string curstring;
 record.clear();

 while (line[linepos] != 0 && linepos < linemax)
 {

   c = line[linepos];

   if (!inquotes && curstring.length() == 0 && c == '"')
   {
        //beginquotechar
        inquotes = true;
      }
      else if (inquotes && c == '"')
      {
        //quotechar
        if ((linepos + 1 <linemax) && (line[linepos + 1] == '"'))
        {
          //encountered 2 double quotes in a row (resolves to 1 double quote)
          curstring.push_back(c);
          linepos++;
        }
        else
        {
          //endquotechar
          inquotes = false;
        }
      }
      else if (!inquotes && c == delimiter)
      {
        //end of field
        record.push_back(curstring);
        curstring = "";
      }
      else if (!inquotes && (c == '\r' || c == '\n'))
      {
        record.push_back(curstring);
        return;
      }
      else
      {
        curstring.push_back(c);
      }
      linepos++;
    }
    record.push_back(curstring);
    return;
}

void dcsweepQuery()
{
    string testNumberStr;
    cout << "DIE NUMBER (in the form DieXX, where XX is the number):> ";
    cin >> dieNum;
    cout << "Device Number (in the form DevXX, where XX is the number):> ";
    cin >> devNum;
    cout << "Test Number:> ";
    cin >> testNumberStr;
    fnum = stoi(testNumberStr);
    cout << "Do you want to condition?> ";
    string ans;
    cin >> ans;

    if (ans == "y")
    {
      condition();
    }
```

```
}


void dSee(string ampl, string comp, string points)
{
   size_t size = 0;

   if (ampl.find_first_of("m", size) !=
      string::npos)ampl.replace(ampl.find_first_of("m", size), 1, "e-3");
   else if (ampl.find_first_of("u", size) !=
      string::npos)ampl.replace(ampl.find_first_of("u", size), 1, "e-6");
   else if (ampl.find_first_of("n", size) !=
      string::npos)ampl.replace(ampl.find_first_of("n", size), 1, "e-9");

   if (comp.find_first_of("m", size) !=
      string::npos)comp.replace(comp.find_first_of("m", size), 1, "e-3");
   else if (comp.find_first_of("u", size) !=
      string::npos)comp.replace(comp.find_first_of("u", size), 1, "e-6");
   else if (comp.find_first_of("n", size) !=
      string::npos)comp.replace(comp.find_first_of("n", size), 1, "e-9");


   double amplitude = stod(ampl);
   double compliance = stod(comp);
   double numpts = stod(points);

   dcSweep4(amplitude, compliance, numpts, "DC");


}

// Simple DC resistance read up to 20 mV with a 10 mA compliance

void res(string points)
{
   double numpts = stod(points);
   dcSweep4(20E-3, 10E-3, numpts, "R");
}

// Allows the user to call “condition” from the menu. This function is edited
// for specific experiments.

void condition()
{
   dSee("1", "10u", "200");
   //dcSweep(20E-3, 10E-3, 51, "Read");
   //dcSweep(0.5, 100E-9, 201,  "Write");
   //dcSweep(20E-3, 10E-3, 51, "Read");
   /*
   dcSweep(20E-3, 10E-3, 51, "Read");
   dcSweep(-1, 100E-3, 201, "Erase");
   dcSweep(20E-3, 10E-3, 51, "Read");
   */
   //dcSweep(0.75, 1E-7, "Write_100n");
   //dcSweep(20E-3, 10E-3, "Read");
   //dcSweep(1.0, 1E-8, "Write_10n");
   //dcSweep(20E-3, 10E-3, "Read");
   //dcSweep(20E-3, 10E-3, ".02");
   //dcSweep(0.6, 10E-6, ".60");
   //dcSweep(20E-3, 10E-3, ".02");
   //dcSweep(-0.6, 10E-3, "-.60");
   //dcSweep(20E-3, 10E-3, ".02");
   //dcSweep(0.6, 10E-6, ".60");
   //dcSweep(20E-3, 10E-3, ".02");
}

// Starts the session with the WGFMU and begins the console application.

int main()
{
   dcsweepQuery();
   WGFMU_openSession("GPIB0::17::INSTR");
   wgfmu_arb();
   //introQuery();
}
```