

# Project Final Report

## Flex Glove: Simon Says

Spring 2011

### ***Project Abstract***

In this project, I have recreated the classic game of Simon Says using the ZNEO microprocessor. The game incorporates sound, light, and finger movements to achieve an immersive experience by employing LEDs, a speaker, and flex sensors. The flex sensors are the single input sensors and the mechanism for a player to interact with the game. The 5 flex sensors are attached to a glove creating a single Flex Glove that can detect individual finger flexes. Like a button input, a finger flex is either flexed (pressed) or unflexed (unpressed) depending on the degree of deflection on the flex sensor. A flex is joined by a corresponding sound and LED color. With five fingers, the glove can send combinations of the 5 inputs to the game. These inputs are used to repeat *item* sequences presented by the computer, Simon. As Simon introduces an additional item in the sequence to the player in a given round, the player must use the Flex Glove to repeat the entire sequence. The object of the game is to achieve the highest score possible by going the most rounds with Simon.



## **Status**

The project has been completed successfully, and my goal of integrating the various hardware technologies has been met. The flex sensors were highly responsive to user finger movements, and I successfully created a two position input for each finger. Finger flexes cause the corresponding speaker tone to play and LED color to display. The individual sensors when combined with the software create an exciting and fun user experience during the game.

## **The Game**

The game will begin when a player flexes, then releases fingers in order from his thumb to his pinky. A “start game” melody will play after the sequence has been entered. Shortly after the “start game” melody, the game will commence with the computer introducing one new ordered item in each round. To advance to the next round, a player must correctly input the entire given sequence up to the last added ordered item. In each round, the sequence that Simon plays gets quicker. The final score is the round in which the player fails to correctly repeat the sequence.

Because the game requires the special sequence in order to start, a player is free to play with the flex glove while the game is not active. This allows the player to experiment with how the fingers can affect the LEDs and speaker.

## **Hardware Specification**

### **Materials**

- ZNEO
- 5 different LEDs (can use tri-color LEDs, or other LEDs)



- Speaker

## Project Final Report



- 5 Flex Sensors (10 K Ohms each)



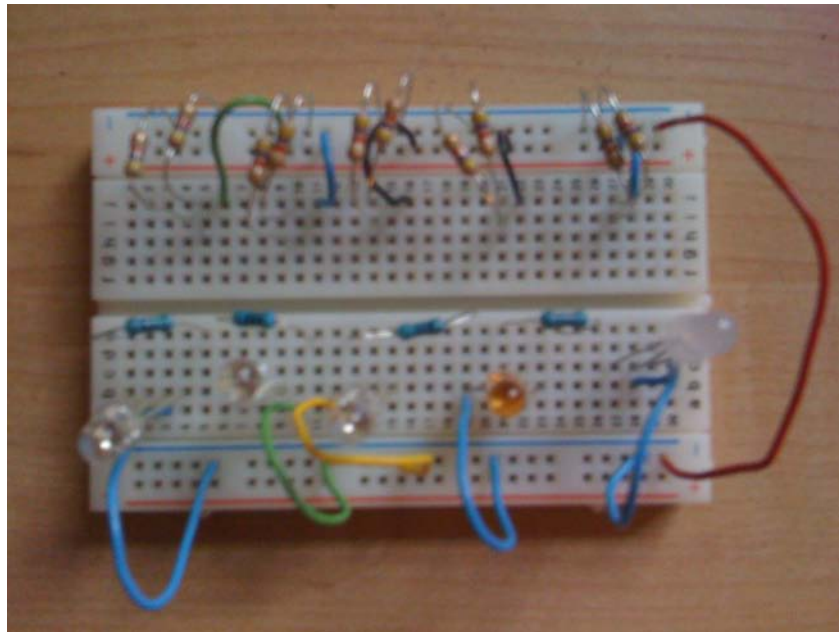
- 10 4.7 K Ohm Resistors
- 5 200 Ohm Resistors
- 1 Old Glove that you don't want
- Black tubing to encapsulate flex sensors (optional)
- Lots of wires



- Bread board
- LED Array (on ZNEO)

## Hardware Setup

1. Connect LEDs to the bread board. LED setup can vary. Each LED should produce a different color and should take a voltage input that is either high or low. The voltage input should come from a GPIO pin and should go through a 100 - 200 Ohm resistor depending on the color you want. The pins I chose are PD0-PD4. For tricolor LEDs with 4 leads, the longer lead is the voltage input. The color can be selected by connecting one or the three remaining leads to ground.
2. Solder the flex sensors to 3-5 foot long wires
3. Connect the soldered flex sensor wires to the bread board. For each flex sensor, you will need two 4.7 K Ohm resistors in parallel. The input voltage needs to be 5V. You will need to set up a voltage divide, which is why the additional resistors are needed. This assumes that the internal reference of the ZNEO is 2V. The output voltage should connect to 5 analog inputs of the ZNEO. I have chosen the AF of pins PB0 - PB4
4. Attach the flex sensors to an old glove
5. Connect the speaker to PC0 and PC1. The T1OUT AF of PC1 will be used.



## Software Specification

The game is driven by a single GameUpdate function that runs in an interrupt. It is this function that is responsible for all game play and sensor input and output. The actual state of the game itself is stored in various global variables. Examples of these global variables are, though not limited to, (1) the current item

that the player must repeat (2) the current frequency of the speaker (3) which fingers are flexed (4) what string is to be displayed on the LED array (5) whether it's the computer's or player's turn, etc.

## Hardware Drivers

### Speaker Driver

The speaker driver uses timer1 to interact with the speaker via the T1OUT pin. To create any given tone, the driver uses the timer to send voltage pulses at desired frequencies to the speaker. The driver can control the duration of a tone using *timeRemaining* global variable that represents the number of milliseconds remaining to play the current tone. In each game update, if that value is greater than zero, the *timeRemaining* is decremented by one. If the value is zero, the driver deactivates the timer. The programmer is given an *init* function to initialize the speaker and various functions to set the frequency to play until deactivated, or to play a frequency for a desired amount of time. The simple API can produce melodies like the start and end game melodies. It can also produce unique tones that can be used to correspond with different finger flexes.

Because the driver API allows the programmer to easily set the speaker frequency, it is trivial to produce combined tones for combined flex inputs. In other words, the driver can make it sound like two tones are playing at the same exact time using just one speaker. This is achieved by averages the desired frequencies to produce a new combined frequency that can correlate to flex inputs up to five fingers at the same time, each variation producing a different, unique tone.

### LED Array Driver

The LED array driver manipulates GPIO E and G ports to produce text on the LED array. It has an *init* function and a single API function for a programmer to set text on the array. If the text does not fit the 4 character array, then the text automatically scrolls. The LED array is used to convey messages to the player in the Simon Says game. The function to display text to the array is in the main while loop to ensure that the LEDs will not blink noticeably, considering the game is updating every millisecond.

### Flex Sensors Driver

The flex sensor driver provides an *init* function and various API to check on the current flex of the given sensor. The driver converts analog voltage input to digital input by leveraging the ZNEO analog to digital converter. The driver has two important API calls: (1) *setFlexThresholds* and (2) *checkFlexsor*. The *setFlexThresholds* is used to determine how much a sensor must be flexed in order to register a flex with the program. When called, the function sets a threshold for a given sensor based on the current flex of the sensor. This assumes that the current flex is a flex sensor in a non-flex state. The threshold is

set at a 25% negative change in output voltage from the baseline, non-flex voltage. When the ZNEO starts, the player is asked to just relax his/her hand in the flex glove to produce the baseline flex. The position of the fingers at the 3 second point are considered to be the baseline positions. For example, if the baseline is 400 for a given finger, then the threshold of that finger is set at 300. This provides for a reasonable and deliberate flex to trigger the game sensors.

The *checkFlexor* function is called with each game update. The function iterates through all flex sensors to determine if they are flexed or not. It also employs a software debouncer that is used to prevent for false positive flexes. The debouncer basically makes sure that a finger is down for a certain period of time before it is considered flexed, and similarly for when it is unflexed.

### LED Driver

The LED driver provides for a simple *init* function, and accompanying on and off functions. The driver uses the GPIO D port to send voltages to the LEDs.

## **Software Design**

### Interrupts

I implement a single interrupt tied to a 1 millisecond timer. This interrupt is responsible for updating the state of the game. I decided to use only one interrupt to simplify the overall design, and to prevent potential timing issues that might arise as the result of having more than one interrupt. All program execution runs through the single interrupt with exception to the LED array update which runs in the main while loop. This essentially gives precedence to the game updates and allows the LED array to look bright and smooth in transitions even when the game is being played.

### Anchors: LEDs and Sounds

Throughout the code, references to “anchors” exist. An anchor is a fancy way of saying that I want to affect multiple sensors at the same time by having the sensors “anchored” to a single function. Because I want the LEDs and sounds to function simultaneously, in software I coupled their functionality into an anchor. For example, if I want LED 1 and tone 1 to become active when flex sensor 1 goes active, I simply call *setAnchor(1)*. This abstraction simplifies coding. Rather than having to code each LED and sound separately whenever a flex occurs, now I just call anchor functions. If I were to ever expand the game to include an additional sensor, all I would have to do is tie that sensor into the anchor functions.

### Debugging via the UART

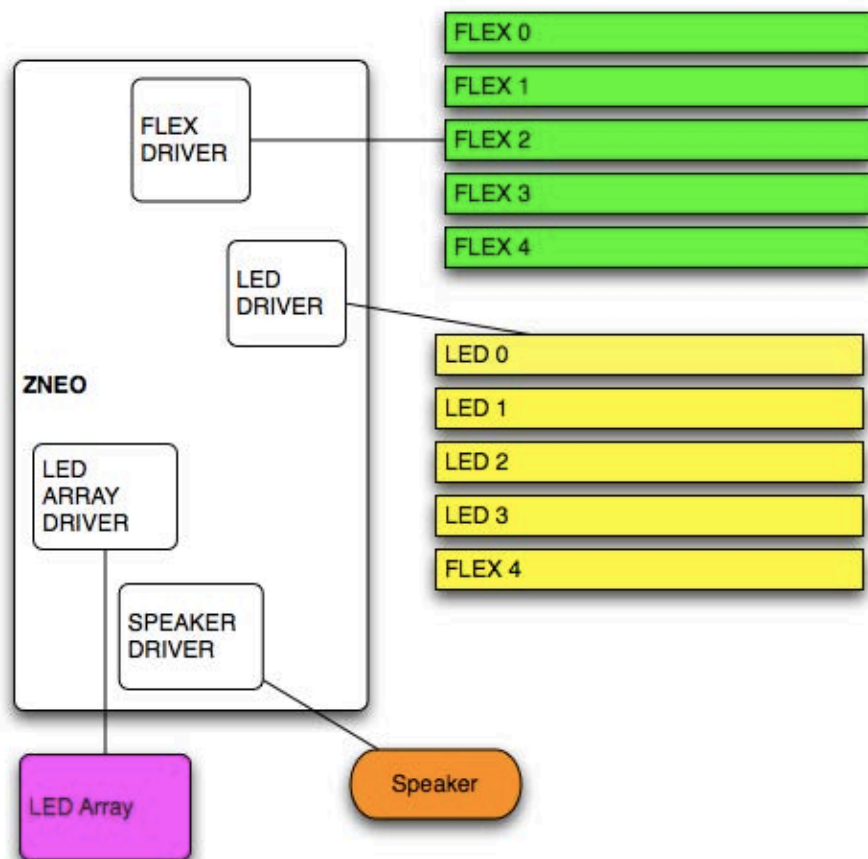
The game uses a UART to connect to a serial interface program such as putty. This provides an effective means of debugging the program.

### Timing

In a game like Simon Says, timing is everything. Depending on various conditions, there is usually at least one global variable counting down to zero, where something must happen within that time. All timing functionality stems from a global counter of the number of milliseconds that have passed since program start. This counter is incremented with each interrupt. During game play, the first timer is the time limit placed on the player response action. This is set at 4 seconds. If the player does not make a flex in that time, the player loses. Another timer occurs when Simon plays back notes. Between each note, Simon will pause for a period of time  $n$ . In each round,  $n$  becomes 10% less than the previous time. This means that as the rounds progress, not only does the player have more items to remember, but Simon is also playing those items faster.

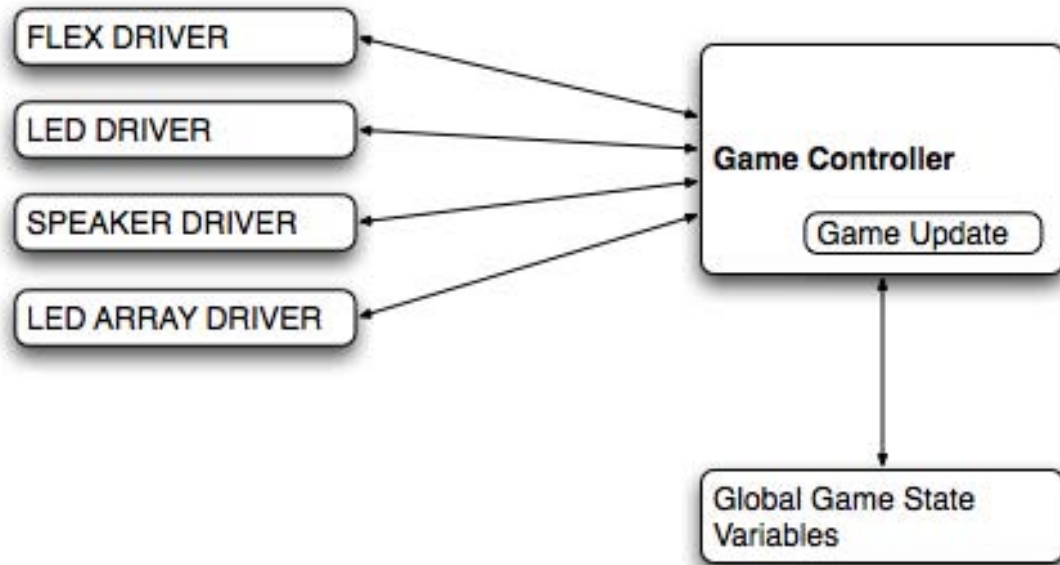
### ***Implementation & Construction***

#### Hardware Block Diagram



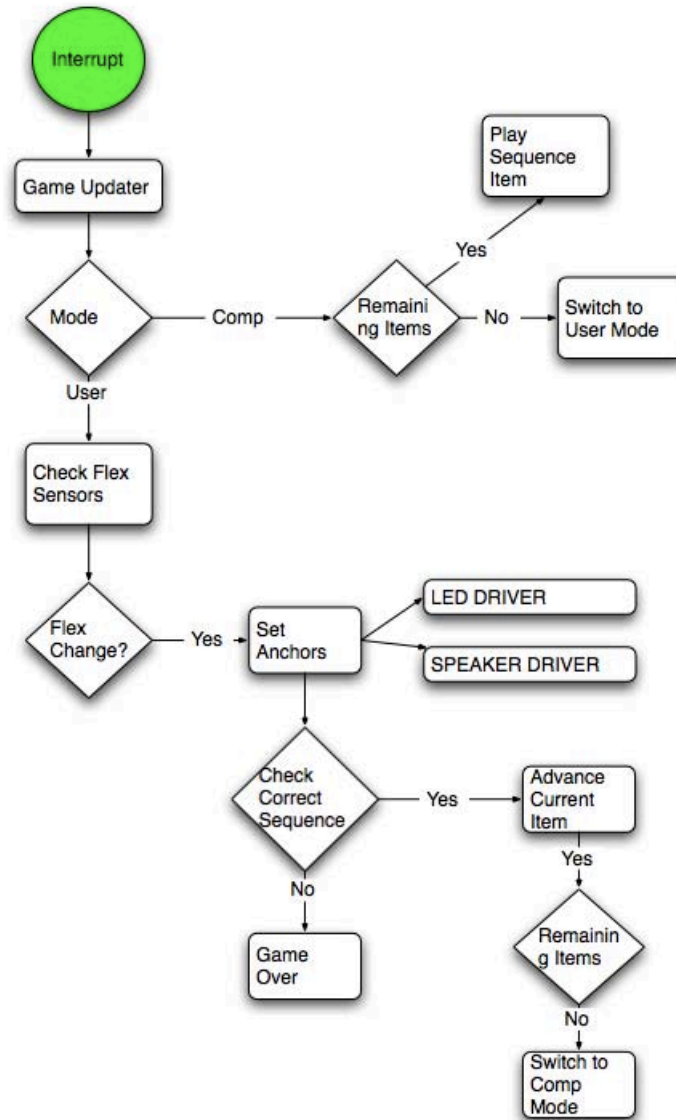
# Project Final Report

## Software Block Diagram



## State Diagram

# Project Final Report



## Milestones

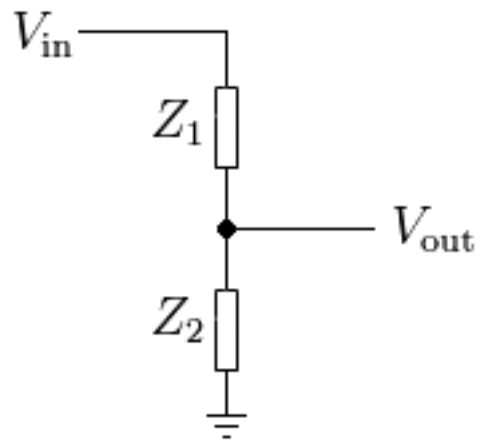
1. Project Design and Planning
2. Game Programming
3. Game audio
4. Flex Sensors Driver and Hardware
5. LED setup
6. Hardware/Software Integration
7. Testing

## Circuits

Spring 2011

## Project Final Report

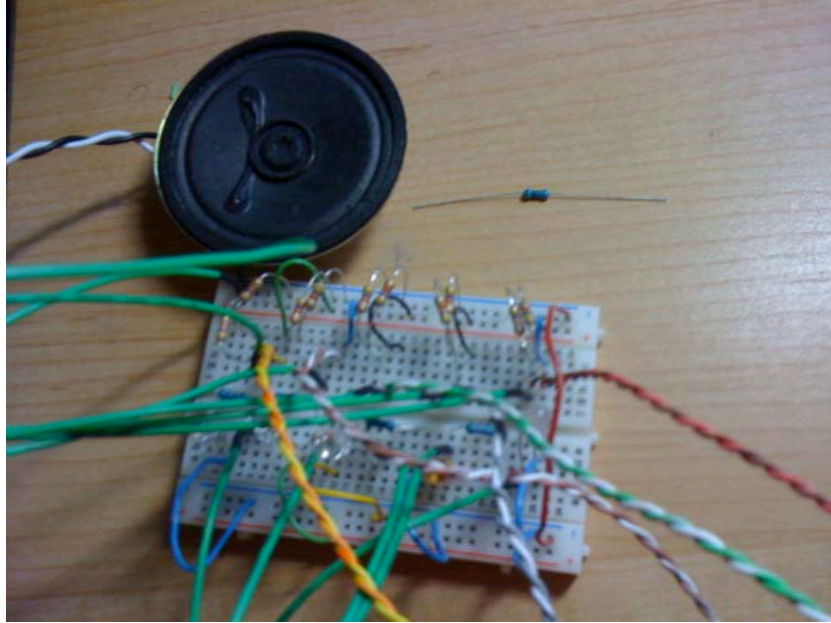
Voltage Divider for flex sensor where  $Z_1$  is a flex sensor and  $Z_2$  is the other resistor



Photos



# Project Final Report





## ***Retrospective***

### **Future Improvements**

The project really exceeded my expectations, however if I had more time, there were three more features that I would have liked to pursue. (1) I really wanted to make this a two player game so that two people could compete against each other in real time. Each player would have a glove and a point system would be devised that rewarded playback finish time. The goal would be extended to not only repeat the sequence correctly, but to also do it quicker than your opponent. (2). Have simon provide combined items in a sequence, such that a player would have to flex certain fingers at the same time. This in some ways could be used to teach someone how to play guitar chords by having those as the items. The sound that would be heard would also correspond to the chord inputted. (3) Extend the Flex Glove to different games or capabilities. I really wanted to have the glove virtually type, however that would have required a lot more functionality, especially with establishing more flex points for each finger.

### **Lessons Learned**

There are plenty of lessons learned from this project and the class, but if I could focus on just one specific lesson, I think that would be the importance of modularizing hardware functionality with software. In this project, I used LEDs, flex sensors, a speaker, and an LED array. All of these hardware components make up the game that I created. I would had never been able to complete the project if I tried to do everything at once. Rather, I built drivers for each hardware component separately. These drivers abstracted all the difficult hardware

interactions, and provided functions that were both simple and reliable. For instance, the LED array driver was a single function `SetDisplayText(String)`, and that was it. I could add any string that I wanted, and I didn't have to think about all the complexity involved with turning on and off individual LEDs. Another example would be the flex sensors. The de-bouncer that I used for the flex sensors was actually the same de-bouncer that I wrote for the buttons. Because I was able to make it generic and modular in the first place, it was easy to port the function to another sensor. The quality of the drivers translated into easy game development, which in turn, translated into a quality game.

## Attachments

1. Flex Sensor datasheet: [http://www.sparkfun.com/datasheets/Sensors/Flex/FLEXSENSOR\(REVA1\).pdf](http://www.sparkfun.com/datasheets/Sensors/Flex/FLEXSENSOR(REVA1).pdf) or `flex_sensor_datasheet.pdf`
2. Tri-Color LED datasheet: [http://cgi.ebay.com/100-pcs-5mm-RGB-LED-4000mcd-Common-Cathode-Free-R-/250787208440?pt=LH\\_DefaultDomain\\_0&hash=item3a64151cf8](http://cgi.ebay.com/100-pcs-5mm-RGB-LED-4000mcd-Common-Cathode-Free-R-/250787208440?pt=LH_DefaultDomain_0&hash=item3a64151cf8)