**CPEN 291 Mini-Project Report**

**A. Group info**

Lab section: *L2B* Group #: B\_G11 Group’s Lab Bench #s: 11 & 12

Student names:

|  |  |
| --- | --- |
| **Sanjeev Krishnan** | **Parsa Riahi** |
| **Arnold Ying** | **Amir Ali Barkam** |
| **Stella Wang** | **Rain Zhang** |
| **Manek Gujral** |  |

**B. Technical documentation for the main functionality**

* What the six dance moves are, and how they are different

1. Waddle: The robot flaps its feet up and down similar to how a penguin waddles
2. Pop-Step: The robot rotates its feet inwards and outwards in a pop and step sort of fashion, hence the name.
3. Ballerina: The robot rotates its legs and performs a tip toe sort of action. The robot then proceeds to rotate and kick at the end.
4. High-Knees: The robot brings its legs up and down one by one, similar to a high knees exercise.
5. Excite: This move performs several kicks back to back, switching between the right and left legs.
6. Shuffle: The robot performs a simple shuffle by turning its legs left and right several times.

* How we implemented each of the six robot moves

1. Waddle: Waddle is performed by rotating the bottom motors down towards the inside of the robot, alternating between left and right.
2. Pop-Step: The pop step rotates the leg motors all the way to the outside and back all the way to the inside and resets to the starting position and switches to the other legs, performing the same motion. The process is repeated several times.
3. Ballerina: Ballerina starts by rotating one leg motor to the outside and back to the reset position, then rotates the foot motor of the same side down towards the outside of the robot, thus pushing that legs up. Then the motor is turned back down and the leg motor is suddenly turned to the outside and its foot motor performs the waddle. The leg then returns to the reset position. This is then repeated for the other leg.
4. High-Knees: The robots feet motors alternate between pressing down towards the outside of the robot on one side and onto the other. This is repeated several times.
5. Excite: Excite performs the sudden leg turn to the outside, and the waddle up, and back to the reset position, alternating between legs.
6. Shuffle: Shuffle turns the legs motors together to the same side and back, and then to the other side and back. This is repeated several times.

* The six songs we orchestrated

1. Anthem: A powerful moving anthem
2. Mario: Super Mario Bros Theme Song
3. Stranger Things: Stranger Things Theme Song
4. All Star: All-Star by Smash Mouth Chorus
5. Tetris: Class Tetris Game Theme Song
6. Fortnite: Fortnite Default Dance Song

* What info the LCD displays
  + We use a state machine to loop through the following states:
    - loading screen
    - The LDC display shows a welcome screen and prompts for user input upon powering on the robot
    - Course info screen
    - Enter passcode screen
* if the password is wrong the screen will show this and let you try again
* if the password is right it goes to the main menu
  + - Main menu, consisting of 5 options to choose from the keypad:
* Default Menu (1), Dance Menu (2), Song Menu (3), About (4), Exit (5)
  + - Default Menu: displays an animation on LCD before showcasing all the dance moves. LCD also shows names of dance moves
    - Dance Menu: Choose between six dance moves with the six buttons, respectively: Waddle, Pop-step, Ballerina, High-Knees, Excite, Shuffle to keypad 1-6. Returns to the Main Menu once done or an interrupt is triggered.
    - Song Menu: Choose between six songs with the six buttons, respectively: Anthem, Mario, Stranger-Things, All-Star, Tetris, and Fortnite to keypad 1-6. Returns to the Main Menu once done or an interrupt is triggered.
    - About screen showing system information and general info about the bipedal dancing robot returns to Main Menu
    - Exit screen terminates the program
* how the LCD has been mounted/attached for optimal viewing
  + The display faces upwards for easy viewing from all sides, unobstructed by wiring and other circuit clutter

**C. Technical documentation for the additional functionality**

* What the additional functionalities are
  + synchronized music for dance moves with the Piezo Buzzer
  + Passcode detection, system diagnostics, and an interactive UI with the keypad
  + Wall detection and motion sensing to abort task with the ultrasonic sensor
  + lights to indicate button clicks, dance moves and supplemental functionalities with the RGB module
  + Dancing animation to accompany the dance moves
* Include the list of the additional components you used
  + Ultrasonic sonar sensor
  + Piezo buzzer
  + Membrane Keypad
  + NOT and NAND gate chips to make a 2 to 3 decoder for the columns of the keypad (2 to 4 with only 3 states used)
  + Protoboard to cleanup the servo wiring
  + RGB LED circuit controlled by NPN transistors to increase brightness
* The hardware implementation
  + Breadboard x2
    - The main breadboard holds the itsy bitsy, the TFT LCD, the RGB module amplified circuit, the Piezo buzzer, and hosts connections from all components on both breadboards and the protoboard to the itsy bitsy pins, a full pin diagram can be seen on the fritzing image.
    - The secondary breadboard is attached back to back with the main breadboard facing downward on the robot, hiding extensive wiring from view. It holds the protoboard in place as well as hosting the gate array corresponding to a 2 to 4 decoder.
  + Soldering
    - the itsy bitsy, TFT LCD, RGB module, and protoboard were all soldered through-hole with the chisel iron at 3600C.
  + Protoboard
    - One unified channel for servo VCC and one for servo GND.
    - PWM signals and the unified VCC and GND are connected to the main breadboard, with PWM for the four servos going to Itsy Bitsy D10-13.
    - the protoboard is held in place to the second breadboard by a couple of small connectors soldered on a single pin each so that the board does not move.
  + Cleanup and Aesthetic Design
    - The keypad was superglued to the side and underside of the robot body casing to make clicking the buttons a more haptic and easy process for the user. Other components are taped or fastened in place for structural rigidity.
    - The keypad and servo wires are all fed through the base of the robot casing, hidden from view to enhance the overall aesthetic of the robot.
    - Most visible m2m connector wires were replaced with manually stripped wires that are more flush to the casing and make the circuit easier to follow visually.
  + RGB Module
    - Reverse logic for LEDs meaning the three LEDs have a shared 5V pin, with separate ground connections
    - Each color pin connects to a transistor which steps up the current of each LED separately
    - If the GPIO pin for an LED is pulled up, then the LED turns on since the collector and emitter are then at the same voltage. If the pin is pulled down, then the 5V load is taken across the transistor and there is no current through the LEDs, causing them to turn off.
* The software implementation
  + FSM
    - The finite state machine is used to run the GUI. There are eight states: Loading, Passcode, Home, Dance, Music, About, Exit, Request and Default.
    - Each state has a function for the GUI, and when there is an input from the sensor(Keypad or Ultrasonic) it changes state.
    - Loading state - a loading screen with an animation
    - Passcode state - a welcome screen and prompts for a password upon powering on the robot
* if the password is wrong the screen will show this and let you try again
* if the password is right it goes to the main menu
  + - Home state - consists of 5 options to choose from the keypad:
* (1)Default, (2)Dance, (3)Song, (4)About, (5)Exit
  + - Default state - Plays all the dance moves
    - Dance state - Choose between six dance moves with the six buttons, respectively: Waddle, Pop-step, Ballerina, High-Knees, Excite, Shuffle to keypad 1-6. Returns to the Main Menu once done or an interrupt is triggered.
    - Song state - Choose between six songs with the six buttons, respectively: Anthem, Mario, Stranger-Things, All-Star, Tetris, and Fortnite to keypad 1-6. Returns to the Main Menu once done or an interrupt is triggered.
    - Request state - asks the user if they want to (1) Play Again, (2) Dance, (3) Play Music or (4) return to home
    - About state - shows system information and general info about the bipedal dancing robot returns to Main Menu
    - Exit state - terminates the program
  + Interrupt
    - Since circuit python does not support interrupt service routine, we had to create an interrupt method that breaks out of loops when a sensor is triggered.  Note: An ISR was not implemented.
  + Ultrasonic Sensor
    - Using the adafruit library we were able to obtain distance values from the sonar. We then stop the dance moves if the distance value is below a certain threshold. Furthermore, we included a try-except block for cases where the sonar returns a ‘Runtime error’ when it is not able to detect a distance.
  + Piezo Buzzer
    - Using the pulse-io library to input the frequency that the piezo should buzz at and the sleep function as a controller for the duration of each note.
  + Keypad
    - Since the keypad relies on a 0V signal through the input (columns) to be shorted to the outputs (rows) we simply rotate which column received the low signal whenever checking for a button press, and then associate to the corresponding row.
    - Since we used a 2 to 3 decoder to activate the columns in an effort to reduce the pins used, the logic to check each column, in order, was a combination of the two outputs from the Itsy Bitsy (ie. A & !B for column 1, B & !A for column two, etc.) rotated every 0.1 seconds.
    - This logic carried forward in checking for a password on the keypad, storing the order of sequential button clicks and evaluating it against a given passcode (which we set to 1234 for simplicity).
  + RGB LED circuit
    - Since the LEDs were controlled by the GPIO pins on the negative end of the diode, a reverse logic applied for turning on the LEDs
    - We simplified the code by simply setting the LEDs to a certain color by the name of the color, not its RGB value. This is due to the 3 dictionaries storing the r, g, and b values for different colors that were referenced in our code.

**D. Test and evaluations**

Before integrating any hardware or software implementation to our robot, we conducted testing on each component. For example, we began with testing our Itsy Bitsy chip (checking that it connects and powers properly through extensive continuity tests and simple USB driver connection testing).

In general, after testing a component rigorously by running various inputs and output controls both on the hardware and software side, we felt comfortable adding it to the overall implementation through our process of continuous integration.

We then moved on to test the second most essential part of the robot, the legs/ servo motors, as well as continuity testing the proto board that controls the servos. One of our main challenges early on was determining the effective angle that corresponded to 90 degrees for each servo before installing the 3-D printed components, which was also a different value for each servo due to the prolonged use of some of the servos wearing down their gears. Once the chip and legs were working together smoothly, we followed the same process where we tested code for one specific hardware component individually before integrating it, running agile tests and rapid prototyping.

At one point after making the protoboard, the whole circuit stopped working even though the chip would still turn on. After a group effort of diagnosing the issues of failed connections to the computer we determined the device driver was no longer working, and further debugging showed the issue to be caused by a very small solder bridge on the protoboard connecting the 5V plane straight to one of the PWM enabled outputs, and rendering the chip useless. After resoldering and ensuring the quality of a fixed protoboard control circuit for the servos, we connected a new Itsy Bitsy and we were back on track. We learned here the importance of extensive hardware testing before running a program.

With the LCD we encountered a situation where it would suddenly turn off and we were able to determine that there existed a loose connection inside the rails of the breadboard which caused occasional open circuits. Thus the location of the LCD was changed to make use of the newer pins of the breadboard. To ensure it worked, we ran a simple set of standard programs for image display onto the LCD and then went on creating our state machine to display many different sets of images and menu options. This along with the servo alignment issue made us aware of the limitation of the components we use and how to work with and around them.

There was a litany of times in which the software either wasn’t compiling, wasn’t performing correctly, or crashed unexpectedly. The debugging process of these issues always followed a gradual escalation approach for systematic efficiency. This process began with checking if all the necessary libraries were installed, then code debugging individually or in pairs, some time spent rubber ducking the issue, then usually landing at the solution or a definitive reason as to the issue we had. When implementing solutions, sometimes it was as simple as adding a missing colon, but other times required completely new approaches to problems due to the constraints we faced. One example of this was the process of synchronizing the music with the dance moves. We tried to do this best with a threading approach but soon came to realize that CircuitPython did not permit such things, and then had to resort to assigning individual notes to each motion to give the impression of seamless integration. We resorted to a similar sequential program for integrated LED control as well, having many small delays to give enough time to view the LED as on before it was shut off. Here we learned much about the tradeoffs in user-friendliness versus performance in running a single thread program.

**E. Conclusions and Reflections**

This mini project taught us many lesson about the software development process, how to best manage and delegate time in a group setting, and many neat tricks and skills to implement a bipedal dancing robot with a haptic user interface that syncs its dance moves to music and animations while detecting obstacles and obstructions.

From the start of the project, we adopted a very efficient approach to group collaboration and idea generation which allowed us to have more time to clean up our final project and present a robot with more robust core and additional functionalities. This is best explained in Appendix A. We were able to identify key constraints in our design and in the technology we were working with, and find compromises and solutions that allowed us to maintain all the functionality we needed without sacrificing response time, whether it be a software synchronization issue, a limitation in the hardware, or anywhere in between. When we ran into issues or conflicts, we were able to discuss and narrow down the source of the issues, and promptly fix them, with each team member naturally taking the role of project manager and devoted debugger whenever needed of their own accord.

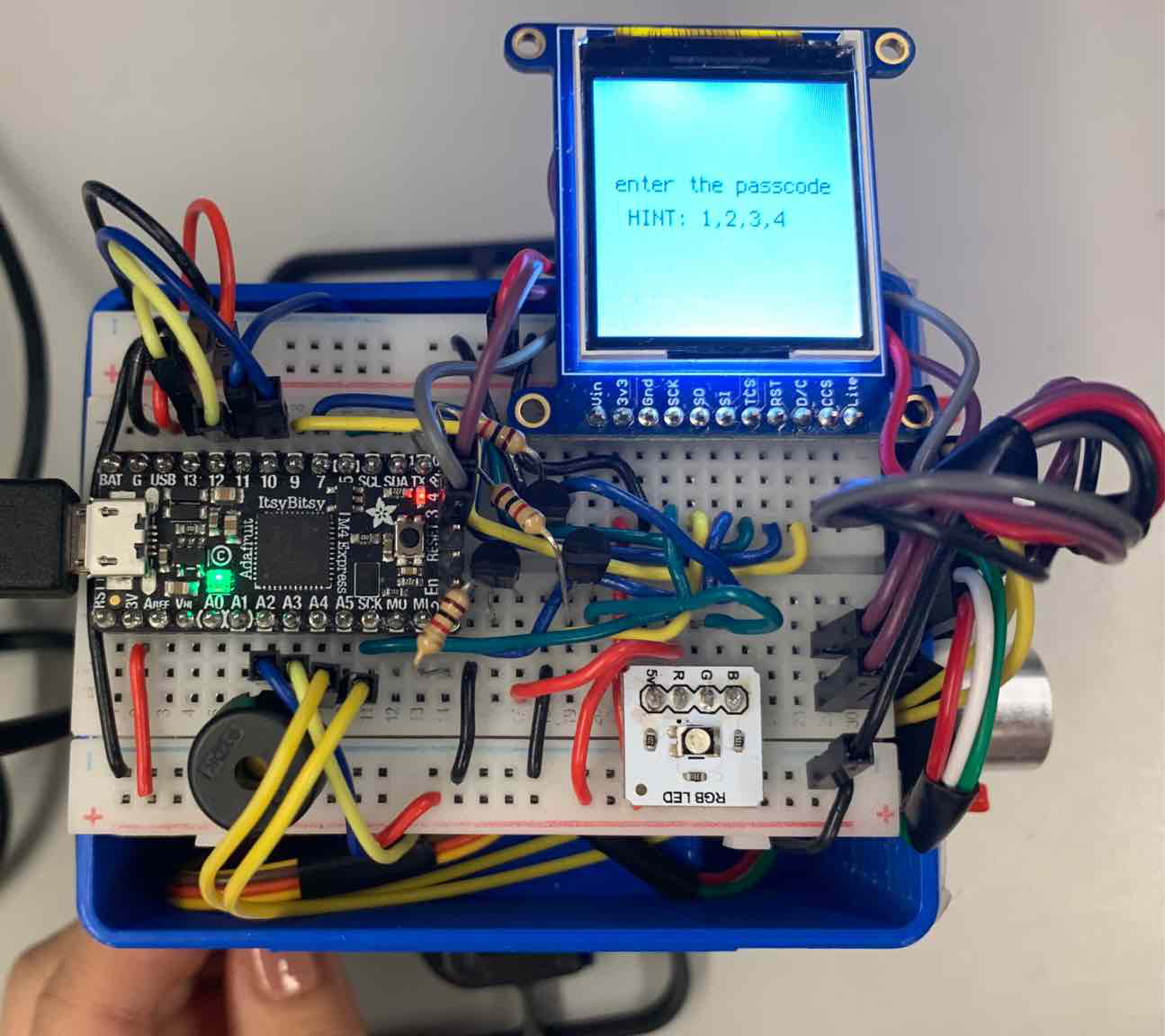
Some cool concepts we learned were how to use CircuitPython to create autonomy in a device, program synchronization versus parallelism, hardware asset management, and how to implement functionality for multiple different devices and modes in unison. Some tricks we learned were how to improve our system’s perceived latency through more fine grained modularity in our code, how to write songs as array of frequencies with delays, how to maximize the number of signal we can send with a finite number of output pins, how to implement a finite state machine, and a plethora of niche functionalities and capabilities of the different CircuitPython libraries and hardware capabilities, from animations to complex motor combinations to emulate dances.

In short, this project served as a very useful foundation for a vast array of technical skills that will prepare us for not only our future projects but also for work as Computer Engineers in our future careers, giving us great freedom in the design process while assisting our team in understanding new concepts whenever we needed it. We learned many new skills and developed team relations that we will continue forward into greater projects with greater challenges and with even greater results!

**F. References and bibliography**

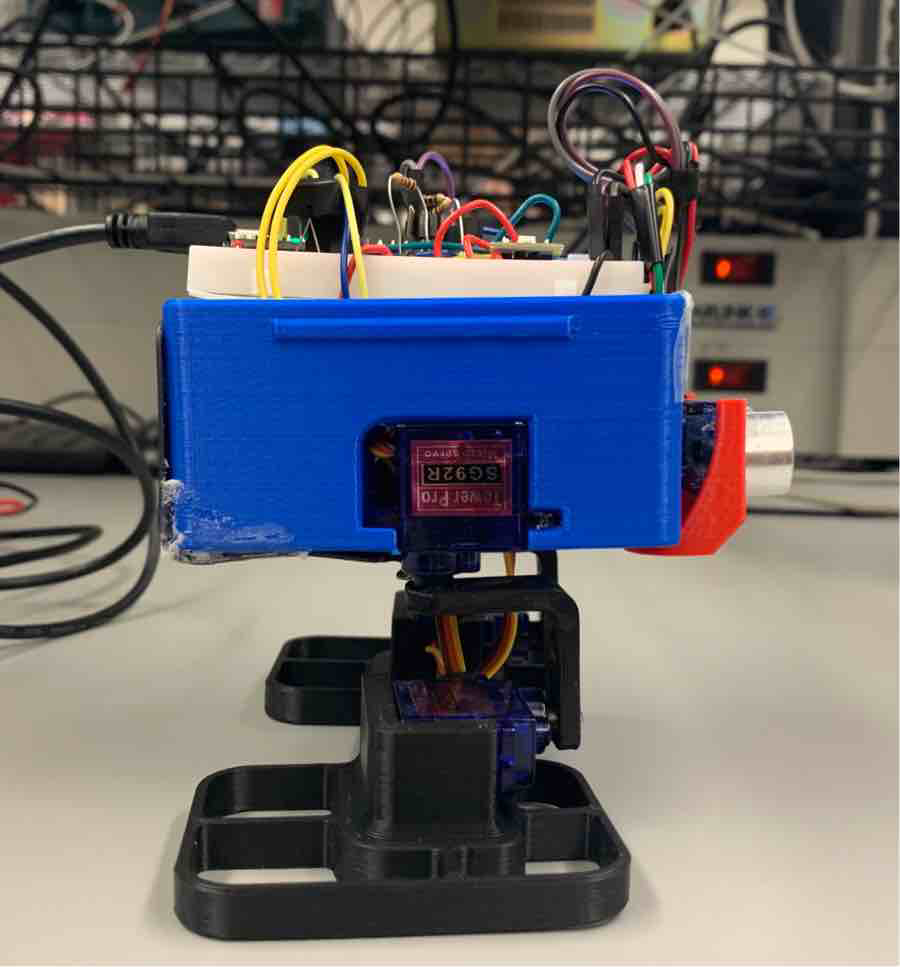
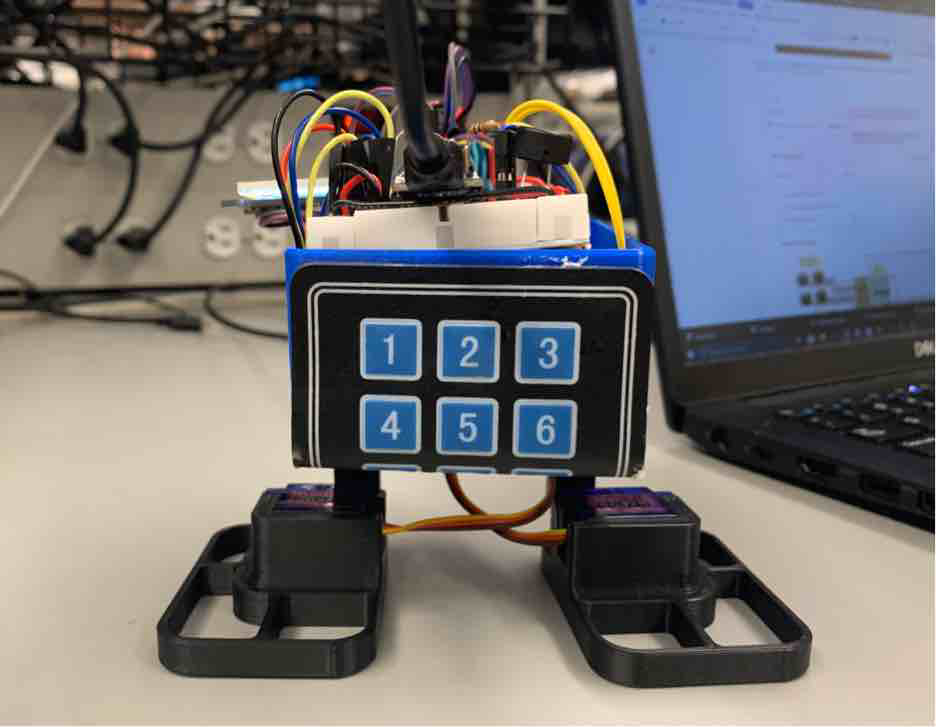
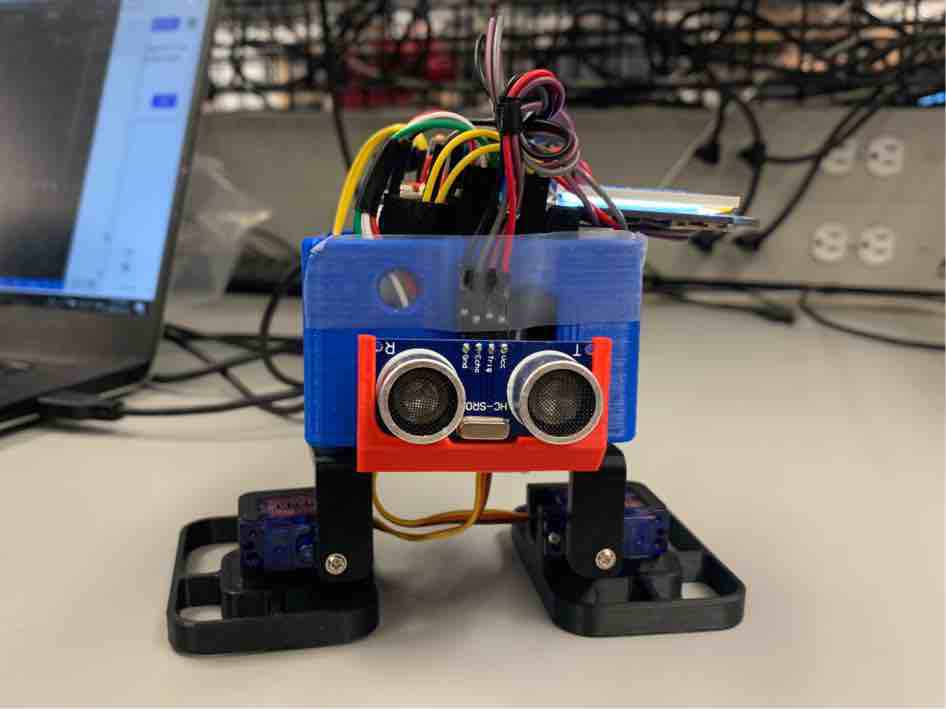
|  |  |
| --- | --- |
| **Filename** | **Description** |
| GUI.py | the main file with all our code |
| mp1.fzz | the fritzing file for the wiring of all the components of our robot |
| Readme | Description |

**Appendix A – Robot pictures**

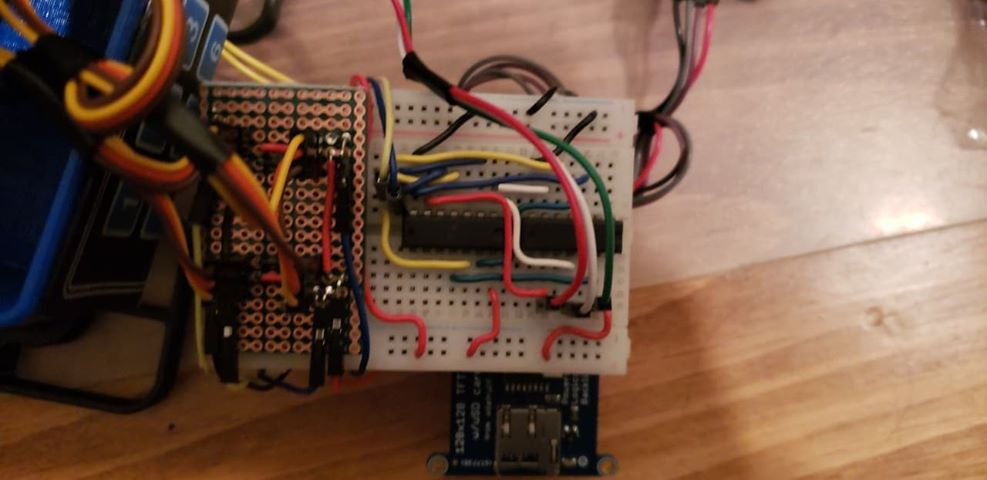


**A-1.** Top view of robot, breadboard and LCD display.

Here is the full view of the top of the robot, displaying the Itsy Bitsy, LCD, RGB LED, Piezo Buzzer, and Ultrasonic sensor. The m2f wires are minimized as much as possible, but we could not get a suitable replacement to clean them up as we could with the male to male wires. The purple and grey wires connect the logic pins of the sonar to the chip. Underneath is the logic and power wires for the LCD, somewhat interwoven with the LED circuits due to the limited space existing on the breadboard. On the bottom left side are the logic wires that drive the keypad, which also passes through a decoder on the second breadboard onto the green, white, and red wires on the bottom of the photo. The wires on the top left to connect to the protoboard for the servos attached to the secondary breadboard hidden under the main one. Every single possible GPIO pin on the Itsy Bitsy was used except a single analog pin.

**A-2.** Side view of robot, servo motors; **A-3.** Back view of robot, keypad; **A-4.** Front view of the robot



**A-5.** View of hidden breadboard underneath the main breadboard. Holds servo control protoboard on the left and the 2 to 3 decoder on the right.

**Appendix B - Code**

# ------------------------------------------------------------------------------------------------------#

# Authors: Manek, Sanjeev, Parsa, Amir, Stella, Arnold, Rain

#

# Function: Dancing Robot

#

# Date: 10/02/2020

# ------------------------------------------------------------------------------------------------------#

# import all the libraries that we need

import time

import sys

import board

import displayio

import terminalio

import label

from adafruit\_st7735r import ST7735R

import digitalio

import adafruit\_matrixkeypad

import pulseio

import servo

from analogio import AnalogIn

import adafruit\_hcsr04

# ------------------------------------------------------------------------------------------------------#

#

# Sonar code

#

# ------------------------------------------------------------------------------------------------------#

# initialize sonar with adafruit\_hcsr04 library

# trigger pin at D4 and echo pin at D3

sonar = adafruit\_hcsr04.HCSR04(trigger\_pin=board.D4, echo\_pin=board.D3)

# define the checkSonar functions that takes in threshold value

# funciton returns true if distance detected by sonar is less than threshold

def checkSonar(threshold):

   try:

      distance = sonar.distance

      return distance < threshold

   except RuntimeError:

      return False

# custom error for checking if robot is too close to object during dance

class TooCloseError(Exception):

   pass

# ------------------------------------------------------------------------------------------------------#

#

# Keypad code

#

# ------------------------------------------------------------------------------------------------------#

# setting up rows and cols of keypad output to its corresponding pins on the itsybitsy

# board A4 to keypad row 0 (1, 2, 3)

row0 = digitalio.DigitalInOut(board.A4)

row0.direction = digitalio.Direction.INPUT

row0.pull = digitalio.Pull.UP

# board A5 to keypad row 1 (4, 5, 6)

row1 = digitalio.DigitalInOut(board.A5)

row1.direction = digitalio.Direction.INPUT

row1.pull = digitalio.Pull.UP

# board A2 to keypad output 1 (2:3 decoder)

out1 = digitalio.DigitalInOut(board.A2)

out1.direction = digitalio.Direction.OUTPUT

out1.value = False

# board A2 to keypad output 2 (2:3 decoder)

out2 = digitalio.DigitalInOut(board.A3)

out2.direction = digitalio.Direction.OUTPUT

out2.value = False

# create a 2D array representing the keys

keys = ((1, 2, 3),

       (4, 5, 6))

# define keypadDecode function that iterates through the different combination of output 1,2

# function returns the key pressed (1 - 6), if no key is pressed, function returns 0

def keypadDecode():

   key = 0

   for i in range(1, 4):

      time.sleep(.1)

      if i == 1:

         out2.value = True

         out1.value = False

      if i == 2:

         out1.value = True

         out2.value = False

      if i == 3:

         out1.value = True

         out2.value = True

      key = keypadHelper(i)

      if key != 0:

         setColor(**"white"**)

         time.sleep(0.1)

         setColor(**"off"**)

         return key

   return key

# define helper function to help decode which row the key pressed is at

# function takes column number as parameter and returns the key pressed, returns 0 if no key is pressed

def keypadHelper(col):

   if not row0.value:

      return col

   if not row1.value:

      return col + 3

   return 0

# define checkPass function to read the input from the keypad

# blocks indefinitely until a password is entered, returns true if password entered is matched, false otherwise

def checkPass():

   seq = []

   pwd = [1, 2, 3, 4]

   i = 0

   while True:

      keys = keypadDecode()

      if keys:

         seq.append(keys)

         i = i + 1

         time.sleep(0.4)

      if i >= 4:

         if seq == pwd:

            seq = []

            i = 0

            return True

         else:

            seq = []

            return False

      time.sleep(0.1)

# define the interrupt function that checks both the keypad for user input and sonar for distance

# function returns true and flashes RBG red 3 times if any key is pressed or the distance detected by sonar is less than 5cm

# otherwise, function returns false

def interrupt():

   keys = 0

   keys = keypadDecode()

   if keys != 0 or checkSonar(5):

      flashRed()

      return True

   else:

      return False

# define the flashRed helper function

# it calls the setColor function 6 times to flash the RBG on and off 3 times

# 0.2 second delay between flashes

def flashRed():

   setColor(**"red"**)

   time.sleep(0.1)

   setColor(**"off"**)

   time.sleep(0.1)

   setColor(**"red"**)

   time.sleep(0.1)

   setColor(**"off"**)

   time.sleep(0.1)

   setColor(**"red"**)

   time.sleep(0.1)

   setColor(**"off"**)

   time.sleep(0.1)

# ------------------------------------------------------------------------------------------------------#

#

# dance code

#

# ------------------------------------------------------------------------------------------------------#

# piezo buzzer setup

# board A1 to piezo buzzer

piezo = pulseio.PWMOut(board.A1, duty\_cycle=0, frequency=440, variable\_frequency=True)

# servo setup

# board D10 to left leg servo motor

pwm1 = pulseio.PWMOut(board.D10, frequency=50)

legL = servo.Servo(pwm1)

# board D11 to right leg servo motor

pwm2 = pulseio.PWMOut(board.D11, frequency=50)

legR = servo.Servo(pwm2)

# board D12 to right foot servo motor

pwm3 = pulseio.PWMOut(board.D12, frequency=50)

footR = servo.Servo(pwm3)

# board D13 to left foot servo motor

pwm4 = pulseio.PWMOut(board.D13, frequency=50)

footL = servo.Servo(pwm4)

# music variable to control whether the robot dance move plays music or not

music = 1

# demo variable to control whether we check sonar distance during the dance moves

demo = 0

# frequency lists for the six songs

ANTHEM = [392, 523, 392, 440, 494, 330, 330,

         440, 392, 349, 392, 262, 262,

         294, 294, 330, 349, 349, 392, 440, 494, 523, 587,

         659, 587, 523, 587, 494, 392,

         523, 494, 440, 494, 330, 330,

         440, 392, 349, 392, 262, 262,

         523, 494, 440, 392, 494, 523, 587,

         659, 587, 523, 494, 523, 587, 392, 392, 494, 523, 587,

         523, 494, 440, 392, 440, 494, 330, 330, 392, 440, 494,

         523, 440, 494, 523, 440, 494, 523, 440, 523, 698,

         698, 659, 587, 523, 587, 659, 523, 523,

         587, 523, 494, 440, 494, 523, 440, 440,

         523, 494, 440, 392, 262, 392, 440, 494, 523]

MARIO = [659, 659, 659, 523, 659, 784, 392, 523, 392, 330, 440, 494, 466, 440, 392, 659, 784, 880, 698, 784, 659,

        523, 587, 494, 523, 392, 330, 440, 494, 466, 440, 392, 659, 784, 880, 698, 784, 659, 523, 587, 494]

STRANGER = [131, 165, 196, 247, 262, 247, 196, 165]

ALLSTAR = [466, 369, 369, 311, 369, 369, 369, 311, 369, 369, 369, 466, 466, 369, 369, 311, 369, 369, 369, 311,

          369, 369, 369, 466, 369, 466, 554, 494, 554, 622, 740, 831, 740, 369, 369, 415, 369, 466, 415, 415,

          369, 415, 311]

TETRIS = [659, 494, 523, 587, 659, 587, 523, 494, 440, 440, 523, 659, 587, 523, 494, 494, 494, 523, 587, 523,

         494, 494, 494, 523, 587, 659, 523, 440, 440, 587, 587, 698, 880, 784, 698, 659, 659, 523, 659, 587,

         523, 494, 494, 523, 587, 659, 523, 440, 440, 659, 494, 523, 587, 659, 587, 523, 494, 440, 440, 523,

         659, 587, 523, 494, 494, 523, 587, 659, 523, 440, 440, 587, 587, 698, 880, 784, 698, 659, 659, 523,

         659, 587, 523, 587, 659, 523, 440, 440]

FNITE = [349, 415, 466, 466, 415, 349, 415, 466, 466, 415, 349, 311, 349, 466, 415, 349, 311, 349]

# define buzzer\_off function, takes no input and turns piezo buzzer off (duty cycle 0)

def buzzer\_off():

   piezo.duty\_cycle = 0  # Off

# define buzzer\_on function, takes no input and turns piezo buzzer on (duty cycle 50%)

def buzzer\_on():

   piezo.duty\_cycle = 65536 // 2  # On 50%

# define playNote helper function, plays a notes according to the input parameters

# it makes the piezo buzzer play a note at 'freq' frequency and sleeps for 'delay' seconds

def playNote(freq, delay):

   piezo.frequency = freq

   piezo.duty\_cycle = 65536 // 2  # On 50%

   time.sleep(delay)  # On

############################

# basic move functions

# define rotate function, rotates 'limb' according to the input parameters

# rotates the selected 'limb' fro 'min' angle to 'max' angle with 'step' increments

def rotate(limb, min, max, step, start, song):

   # the function raises an exception and returns premptively if:

   #   - the sonar is detecting objects less than 5 cm AND

   #   - the demo variable is set to false (not demoing)

   if checkSonar(5) and (not demo):

      raise TooCloseError

      return start

   # start variable is to adjust the offset to the song that we are playing along with the movement

   i = start

   for x in range(min, max + step, step):

      limb.angle = x

      # function plays 'song' if the music variable is set to 1

      if music == 1:

         playNote(song[i % len(song)], 0.3)

      else:

         time.sleep(0.15)

      i += 1

   return i

# define double\_rotate function, rotates two limbs 'limb1' and 'limb2' according to the input parameters

# similar to rotate's functionality but moves two limbs at once

def double\_rotate(limb1, limb2, min, max, step, start, song):

   # the function raises an exception and returns premptively if:

   #   - the sonar is detecting objects less than 5 cm

   if checkSonar(5):

      raise TooCloseError

      return start

   i = start

   for x in range(min, max + step, step):

      limb1.angle = x

      limb2.angle = x

      if music == 1:

         playNote(song[i % len(song)], 0.3)

      else:

         time.sleep(0.15)

      i += 1

   return i

############################

# basic dance functions

# define tapFoot function, moves the left or right ankle up and down, plays the input song if music is set to 1

# calls the rotate function twice

def tapFoot(start, song, limb):

   if limb == footL:

      start = rotate(footL, 90, 60, -10, start, song)

      start = rotate(footL, 60, 90, 10, start, song)

   else:

      start = rotate(footR, 100, 130, 10, start, song)

      start = rotate(footR, 130, 100, -10, start, song)

   return start

# define kick function, moves the left or right leg outwards, then moves the left or right ankle up and down,

# then moves the left or right leg back inwards, plays the input song while moving if music is set to 1

# calls the rotate function twice, directly modifies left or right legs angle

def kick(start, song, limb):

   if limb == legR:

      limb.angle = 160

      start = rotate(footR, 100, 60, -10, start, song)

      start = rotate(footR, 60, 100, 10, start, song)

      limb.angle = 90

   else:

      limb.angle = 20

      start = rotate(footL, 90, 130, 10, start, song)

      start = rotate(footL, 130, 90, -10, start, song)

      limb.angle = 90

   return start

# define footIn function, moves the left or right leg inwards then back outwards, plays the input song while moving if music is set to 1

# calls the rotate function twice

def footIn(start, song, limb):

   if limb == legR:

      start = rotate(limb, 90, 10, -10, start, song)

      start = rotate(limb, 10, 90, 10, start, song)

   else:

      start = rotate(limb, 90, 170, 10, start, song)

      start = rotate(limb, 170, 90, -10, start, song)

   return start

# define footOut function, moves the left or right leg outwards then back inwards, plays the input song while moving if music is set to 1

# calls the rotate function twice

def footOut(start, song, limb):

   if limb == legR:

      start = rotate(limb, 90, 160, 10, start, song)

      start = rotate(limb, 160, 90, -10, start, song)

   else:

      start = rotate(legL, 90, 20, -10, start, song)

      start = rotate(legL, 20, 90, 10, start, song)

   return start

# define wiggle function, moves the left foot up, then right foot up, then left foot down, then right food down,

# plays the input song while moving if music is set to 1

# calls the rotate function four times

def wiggle(start, song):

   start = rotate(footL, 90, 130, 10, start, song)

   start = rotate(footR, 100, 60, -10, start, song)

   start = rotate(footL, 130, 90, -10, start, song)

   start = rotate(footR, 60, 100, 10, start, song)

   return start

# define shuffle function, moves the left and right leg at the same time left and right fully

# plays the input song while moving if music is set to 1

# calls the double\_rotate function 12 times

def shuffle(start, song):

   for angle in range(90, 30, -15):  # 0 - 180 degrees, 5 degrees at a time.

      start = double\_rotate(legL, legR, angle, angle, -15, start, song)

   for angle in range(30, 90, 15):  # 180 - 0 degrees, 5 degrees at a time.

      start = double\_rotate(legL, legR, angle, angle, 15, start, song)

   for angle in range(90, 120, 15):  # 0 - 180 degrees, 5 degrees at a time.

      start = double\_rotate(legL, legR, angle, angle, -15, start, song)

   for angle in range(120, 90, -15):  # 180 - 0 degrees, 5 degrees at a time.

      start = double\_rotate(legL, legR, angle, angle, 15, start, song)

   return start

# define reset\_servo function, resets all the limb servo motors back to its default position

# sleeps 0.1 second after each reset to give the motor time to move

def reset\_servo():

   footR.angle = 97

   time.sleep(0.1)

   footL.angle = 92

   time.sleep(0.1)

   legR.angle = 90

   time.sleep(0.1)

   legL.angle = 90

   time.sleep(0.1)

# 6 dance moves created as a combination of the smaller moves above

#   - each dance calls reset\_servo before and after the dance moves

#   - they also turn the piezo buzzer on if the music is set to 1

#   - the dance functions also checks for the exception raised in the

#     rotate/double\_rotate functions, stops and calls flashRed functions once exception raised.

# the Waddle dance

def dance1():

   reset\_servo()

   if music:

      buzzer\_on()

   start = 0

   for i in range(3):

      try:

         start = wiggle(start, STRANGER)

      except TooCloseError:

         flashRed()

         break

   buzzer\_off()

   reset\_servo()

# the Pop-Step dance

def dance2():

   reset\_servo()

   if music:

      buzzer\_on()

   start = 0

   for i in range(2):

      try:

         start = footOut(start, MARIO, legL)

         start = footIn(start, MARIO, legL)

         start = footOut(start, MARIO, legR)

         start = footIn(start, MARIO, legR)

      except TooCloseError:

         flashRed()

         break

   buzzer\_off()

   reset\_servo()

# the Ballerina dance

def dance3():

   reset\_servo()

   if music:

      buzzer\_on()

   start = 0

   for i in range(2):

      try:

         start = footOut(start, ANTHEM, legL)

         start = tapFoot(start, ANTHEM, footL)

         start = kick(start, ANTHEM, legL)

         start = footOut(start, ANTHEM, legR)

         start = tapFoot(start, ANTHEM, footR)

         start = kick(start, ANTHEM, legR)

      except TooCloseError:

         flashRed()

         break

   buzzer\_off()

   reset\_servo()

# the High-Knees dance

def dance4():

   reset\_servo()

   if music:

      buzzer\_on()

   start = 0

   for j in range(3):

      try:

         start = tapFoot(start, TETRIS, footL)

      except TooCloseError:

         flashRed()

         break

   for j in range(3):

      try:

         start = tapFoot(start, TETRIS, footR)

      except TooCloseError:

         flashRed()

         break

   buzzer\_off()

   reset\_servo()

# the Excite dance

def dance5():

   reset\_servo()

   if music:

      buzzer\_on()

   start = 0

   for i in range(3):

      try:

         start = kick(start, ALLSTAR, legL)

         start = kick(start, ALLSTAR, legR)

      except TooCloseError:

         flashRed()

         break

   buzzer\_off()

   reset\_servo()

# the Shuffle dance

def dance6():

   reset\_servo()

   if music:

      buzzer\_on()

   start = 0

   for i in range(3):

      try:

         start = shuffle(start, FNITE)

      except TooCloseError:

         flashRed()

         break

   buzzer\_off()

   reset\_servo()

# ------------------------------------------------------------------------------------------------------#

#

# songs code

#

# ------------------------------------------------------------------------------------------------------#

# define play\_song function for playing song when given song frequencies as input

def play\_song(song):

   timeout = time.time() + 10

   while True:

      # function exits if the song is longer than 10 seconds

      if time.time() > timeout:

         break

      temp = False

      for f in song:

         # calls the interrupt function to see if anything

         # has called for an interrupt

         if interrupt():

            temp = True

            break

         piezo.frequency = f

         piezo.duty\_cycle = 65536 // 2  # On 50%

         time.sleep(0.3)  # On for 1/4 second

      piezo.duty\_cycle = 0  # Off

      if temp != False:

         break

      time.sleep(0.3)

# ------------------------------------------------------------------------------------------------------#

#

# display code

#

# ------------------------------------------------------------------------------------------------------#

# define reset function that resets the display when we want to change the state of the FSM

def reset():

   displayio.release\_displays()

   spi = board.SPI()

   tft\_cs = board.D5

   tft\_dc = board.D9

   display\_bus = displayio.FourWire(spi, command=tft\_dc, chip\_select=tft\_cs, reset=board.D7)

   global display

   display = ST7735R(display\_bus, width=128, height=128, colstart=2, rowstart=1)

   global splash

   splash = displayio.Group(max\_size=100)

   display.show(splash)

   color\_bitmap = displayio.Bitmap(128, 128, 1)

   color\_palette = displayio.Palette(1)

   color\_palette[0] = 0xFFFFFF  # White

   bg\_white = displayio.TileGrid(color\_bitmap, pixel\_shader=color\_palette, x=0, y=0)

   splash.append(bg\_white)

# method to upload picture onto the LCD

def ShowPic(string, timein):

   with open(string, **"rb"**) as bitmap\_file:

      # Setup the file as the bitmap data source

      bitmap = displayio.OnDiskBitmap(bitmap\_file)

      # Create a TileGrid to hold the bitmap

      tile\_grid = displayio.TileGrid(bitmap, pixel\_shader=displayio.ColorConverter())

      # Create a Group to hold the TileGrid

      group = displayio.Group()

      # Add the TileGrid to the Group

      group.append(tile\_grid)

      # Add the Group to the Display

      display.show(group)

      # Loop forever so you can enjoy your image

      for i in range(timein):

         pass

         time.sleep(0.1)

# define textshow function that shows time dependent text (shows for 'timein' seconds)

# functions prints "textin" to the LCD and sets the background color of the LCD to "bgcolor"

# the position of the text is determined by xc, yc and the function blocks for 'timein' seconds

def textshow(textin, bgcolor, xc, yc, timein):

   text\_area = label.Label(terminalio.FONT, text=textin, color=bgcolor)

   text\_area.x = xc

   text\_area.y = yc

   splash.append(text\_area)

   for i in range(timein):

      pass

      time.sleep(1)

# define textout function that shows time independent text

# same implementation to textshow but does not block after displaying text

def textout(textin, bgcolor, xc, yc):

   text\_area = label.Label(terminalio.FONT, text=textin, color=bgcolor)

   text\_area.x = xc

   text\_area.y = yc

   splash.append(text\_area)

# ------------------------------------------------------------------------------------------------------#

#

# RGB module code

#

# ------------------------------------------------------------------------------------------------------#

# reverse logic on the rgb pins so that a pull up resistor turns the led off and the pull down turns it on

# board D2 to red color

red = digitalio.DigitalInOut(board.D2)

red.direction = digitalio.Direction.INPUT

red.pull = digitalio.Pull.DOWN

# board D1 to green color

green = digitalio.DigitalInOut(board.D1)

green.direction = digitalio.Direction.INPUT

green.pull = digitalio.Pull.DOWN

# board D0 to blue color

blue = digitalio.DigitalInOut(board.D0)

blue.direction = digitalio.Direction.INPUT

blue.pull = digitalio.Pull.DOWN

# set of basic digital colour values to be set on demand in 3 dictionaries, one for each pin

dictRed = {**"red"**: digitalio.Pull.UP, **'cyan'**: digitalio.Pull.DOWN, **"yellow"**: digitalio.Pull.UP,

**"green"**: digitalio.Pull.DOWN, **'blue'**: digitalio.Pull.DOWN, **'magenta'**: digitalio.Pull.UP,

**'white'**: digitalio.Pull.UP, **'off'**: digitalio.Pull.DOWN}

dictGreen = {**"red"**: digitalio.Pull.DOWN, **'cyan'**: digitalio.Pull.UP, **"yellow"**: digitalio.Pull.UP,

**"green"**: digitalio.Pull.UP, **'blue'**: digitalio.Pull.DOWN, **'magenta'**: digitalio.Pull.DOWN,

**'white'**: digitalio.Pull.UP, **'off'**: digitalio.Pull.DOWN}

dictBlue = {**"red"**: digitalio.Pull.DOWN, **'cyan'**: digitalio.Pull.UP, **"yellow"**: digitalio.Pull.DOWN,

**"green"**: digitalio.Pull.DOWN, **'blue'**: digitalio.Pull.UP, **'magenta'**: digitalio.Pull.UP,

**'white'**: digitalio.Pull.UP, **'off'**: digitalio.Pull.DOWN}

# define function setColor that takes an input string and changes the RBG to the specified color

# if the string color is defined in the dictionary declared above changes the led color to one defined in the dictionary

def setColor(color):

   red.pull = dictRed[color]

   green.pull = dictGreen[color]

   blue.pull = dictBlue[color]

# define the anim to play the animation on the LCD

# calls the ShowPic function

def anim(time):

   for i in range(time):

      ShowPic(**"\dance-0.bmp"**, 0.1)

      ShowPic(**"\dance-1.bmp"**, 0.1)

      ShowPic(**"\dance-2.bmp"**, 0.1)

      ShowPic(**"\dance-3.bmp"**, 0.1)

      ShowPic(**"\dance-4.bmp"**, 0.1)

      ShowPic(**"\dance-5.bmp"**, 0.1)

      ShowPic(**"\dance-6.bmp"**, 0.1)

      ShowPic(**"\dance-7.bmp"**, 0.1)

      ShowPic(**"\dance-8.bmp"**, 0.1)

      ShowPic(**"\dance-9.bmp"**, 0.1)

      ShowPic(**"\dance-10.bmp"**, 0.1)

      ShowPic(**"\dance-11.bmp"**, 0.1)

      ShowPic(**"\dance-12.bmp"**, 0.1)

      ShowPic(**"\dance-13.bmp"**, 0.1)

      ShowPic(**"\dance-14.bmp"**, 0.1)

      ShowPic(**"\dance-15.bmp"**, 0.1)

      ShowPic(**"\dance-16.bmp"**, 0.1)

      ShowPic(**"\dance-17.bmp"**, 0.1)

      ShowPic(**"\dance-18.bmp"**, 0.1)

      ShowPic(**"\dance-19.bmp"**, 0.1)

      ShowPic(**"\dance-20.bmp"**, 0.1)

# define the animRev to play the animation in reverse on the LCD

# calls the ShowPic function

def animRev(time):

   for i in range(time):

      ShowPic(**"\dance-20.bmp"**, 0.1)

      ShowPic(**"\dance-19.bmp"**, 0.1)

      ShowPic(**"\dance-18.bmp"**, 0.1)

      ShowPic(**"\dance-17.bmp"**, 0.1)

      ShowPic(**"\dance-16.bmp"**, 0.1)

      ShowPic(**"\dance-15.bmp"**, 0.1)

      ShowPic(**"\dance-14.bmp"**, 0.1)

      ShowPic(**"\dance-13.bmp"**, 0.1)

      ShowPic(**"\dance-12.bmp"**, 0.1)

      ShowPic(**"\dance-11.bmp"**, 0.1)

      ShowPic(**"\dance-10.bmp"**, 0.1)

      ShowPic(**"\dance-9.bmp"**, 0.1)

      ShowPic(**"\dance-8.bmp"**, 0.1)

      ShowPic(**"\dance-7.bmp"**, 0.1)

      ShowPic(**"\dance-6.bmp"**, 0.1)

      ShowPic(**"\dance-5.bmp"**, 0.1)

      ShowPic(**"\dance-4.bmp"**, 0.1)

      ShowPic(**"\dance-3.bmp"**, 0.1)

      ShowPic(**"\dance-2.bmp"**, 0.1)

      ShowPic(**"\dance-1.bmp"**, 0.1)

      ShowPic(**"\dance-0.bmp"**, 0.1)

# ------------------------------------------------------------------------------------------------------#

#

# gui code

#

# ------------------------------------------------------------------------------------------------------#

# define the states of the GUI

LOADING = 0

PASSCODE = 1

HOME = 2

DANCE = 3

MUSIC = 4

ABOUT = 5

EXIT = 6

REQUEST = 7

DEFAULT = 8

# initial default state is loading

state = LOADING

# keeps checking the FSM state and update the current state according to the inputs

while True:

   # if state is loading, it goes to passcode

   if state == LOADING:

      splash = displayio.Group(max\_size=100)

      reset()

      reset\_servo()

      # display "Loading..." meassage on the LCD

      string1 = **"Loading"**

textshow(string1, 0x000000, 30, 64, 0.0001)

      string2 = **"..."**

i = 0

      x = 72

      while (i < 3):

         textshow(string2[i], 0x000000, x, 64, 0.0001)

         i += 1

         x += 6

      # display the robot picture on the LCD

      reset()

      ShowPic(**"\Robot.bmp"**, 2)

      time.sleep(1)

      reset()

      # display the "Welcome" text on the LCD

      textshow(**"Welcome"**, 0x000000, 45, 64, 0.1)

      reset()

      # proceed to passcode state for the FSM to request for passcode

      state = PASSCODE

   # if state is passcode, checks the passcode, if correct goes to home, else back to passcode

   if state == PASSCODE:

      # display "enter the passcode", runs checkPass function

      textout(**"enter the passcode** \n **HINT: 1,2,3,4"**, 0x000000, 10, 60)

      boolean = False

      boolean = checkPass()

      # if passcode is correct the FSM proceed to the home state

      if boolean:

         boolean = False

         state = HOME

         reset()

      # if passcode is incorrect display "wrong passcode" and then keeps fetching passcode

      else:

         reset()

         textshow(**"wrong passcode"**, 0x000000, 20, 60, 2)

         # time.sleep(1)

         state = PASSCODE

         reset()

   # if state is home, checks keypad and goes to coressponding state

   elif state == HOME:

      setColor(**'off'**)

      # display the home menu on the screen

      textout(**"Press a key:** \n **1) Demo** \n **2) Dance** \n **3) Music** \n **4) About** \n **5) Exit "**, 0x000000, 10, 60)

      keys = 0

      # keeps checking the keypad for a input, blocks indefinitely until user inputs

      while keys == 0:

         keys = keypadDecode()

      # set the next state of the FSM according to the user input

      if keys == 1:

         state = DEFAULT

         reset()

      elif keys == 2:

         state = DANCE

         reset()

      elif keys == 3:

         state = MUSIC

         reset()

      elif keys == 4:

         state = ABOUT

         reset()

      elif keys == 5:

         state = EXIT

         reset()

      else:

         state = HOME

         reset()

   # if state is dance, plays the dance move coressponding to the keypad number pressed

   elif state == DANCE:

      # display the dance menu on the screen

      textout(**"Press a key:** \n **1) Waddle** \n **2) Pop-Step** \n **3) Ballerina** \n **4) High-Knees** \n **5) Excite** \n **6) Shuffle"**,

             0x000000, 10, 60)

      keys = 0

      # keeps checking the keypad for a input,

      # blocks indefinitely until user inputs or until sonar detects an object less than 5cm

      while keys == 0 and not checkSonar(5):

         keys = keypadDecode()

      # call the corresponding dance function depending on user input then move the FSM to request state

      # if no user input, check sonar for distance less than 5, if such, return FSM to home state.

      if keys == 1:

         reset()

         textout(**"Waddling"**, 0x000000, 41, 48)

         textout(**"Press any Button"**, 0x000000, 17, 64)

         textout(**"to return"**, 0x000000, 35, 80)

         setColor(**'green'**)

         time.sleep(2)

         reset()

         anim(1)

         dance1()

         animRev(1)

         setColor(**'off'**)

         state = REQUEST

         test = dance1

         reset()

      elif keys == 2:

         reset()

         textout(**"Pop-stepping"**, 0x000000, 34, 48)

         textout(**"Press any Button"**, 0x000000, 17, 64)

         textout(**"to return"**, 0x000000, 35, 80)

         setColor(**'green'**)

         time.sleep(2)

         reset()

         anim(1)

         dance2()

         animRev(1)

         setColor(**'off'**)

         state = REQUEST

         test = dance2

         reset()

      elif keys == 3:

         reset()

         textout(**"Balleting"**, 0x000000, 40, 48)

         textout(**"Press any Button"**, 0x000000, 17, 64)

         textout(**"to return"**, 0x000000, 35, 80)

         setColor(**'green'**)

         time.sleep(2)

         reset()

         anim(1)

         dance3()

         animRev(1)

         setColor(**'off'**)

         state = REQUEST

         test = dance3

         reset()

      elif keys == 4:

         reset()

         textout(**"High-knees"**, 0x000000, 36, 48)

         textout(**"Press any Button"**, 0x000000, 17, 64)

         textout(**"to return"**, 0x000000, 35, 80)

         setColor(**'green'**)

         time.sleep(2)

         reset()

         anim(1)

         dance4()

         animRev(1)

         setColor(**'off'**)

         state = REQUEST

         test = dance4

         reset()

      elif keys == 5:

         reset()

         textout(**"I'm Excited!"**, 0x000000, 33, 48)

         textout(**"Press any Button"**, 0x000000, 17, 64)

         textout(**"to return"**, 0x000000, 35, 80)

         setColor(**'green'**)

         time.sleep(2)

         reset()

         anim(1)

         dance5()

         animRev(1)

         setColor(**'off'**)

         state = REQUEST

         test = dance5

         reset()

      elif keys == 6:

         reset()

         textout(**"Everyday I'm** \n **Shuffling"**, 0x000000, 30, 45)

         textout(**"Press any Button"**, 0x000000, 17, 68)

         textout(**"to return"**, 0x000000, 35, 82)

         setColor(**'green'**)

         time.sleep(2)

         reset()

         anim(1)

         dance6()

         animRev(1)

         setColor(**'off'**)

         state = REQUEST

         test = dance6

         reset()

      elif checkSonar(5):

         setColor(**'red'**)

         time.sleep(0.1)

         state = HOME

         setColor(**'off'**)

         reset()

      # if no user input / sonar detection, stay in dance state

      else:

         state = DANCE

   # if state is about it displays info about robot and returns

   elif state == ABOUT:

      textshow(**"About:** \n **Dancing Robot GUI"**, 0x000000, 10, 24, 5)

      textshow(**"press any button** \n **to return"**, 0x000000, 20, 64, 5)

      keys = 0

      # keeps checking the keypad for a input,

      # blocks indefinitely until user inputs or until sonar detects an object less than 5cm

      while keys == 0 and not checkSonar(5):

         keys = keypadDecode()

      if keys == 1:

         state = HOME

         reset()

      elif keys == 2:

         state = HOME

         reset()

      elif keys == 3:

         state = HOME

         reset()

      elif keys == 4:

         state = HOME

         reset()

      elif keys == 5:

         state = HOME

         reset()

      elif keys == 6:

         state = HOME

         reset()

      elif checkSonar(5):

         setColor(**'red'**)

         time.sleep(0.1)

         state = HOME

         setColor(**'off'**)

         reset()

      else:

         state = ABOUT

         reset()

   # if state is exit it quits the program

   elif state == EXIT:

      textshow(**"Exiting....."**, 0x000000, 30, 64, 3)

      time.sleep(0.5)

      # state =  PASSCODE

      sys.exit()

      # reset()

   # if the state is request it goes to corresponding state according to keypad number pressed

   elif state == REQUEST:

      textout(**"Press a key:** \n **1) Play Again** \n **2) Dance** \n **3) Play Music** \n **4) Home"**, 0x000000, 15, 60)

      keys = 0

      while keys == 0:

         keys = keypadDecode()

      if keys == 1:

         test()

         state = REQUEST

         reset()

      if keys == 2:

         state = DANCE

         reset()

      elif keys == 3:

         state = MUSIC

         reset()

      elif keys == 4:

         state = HOME

         reset()

      else:

         state = REQUEST

   # if state is music it goes to the song according to the keypad pressed

   elif state == MUSIC:

      textout(**"Press a key:** \n **1) Anthem** \n **2) Mario** \n **3) Stranger** \n **4) All-Star** \n **5) Tetris** \n **6) Fortnite"**, 0x000000,

             10, 60)

      keys = 0

      while keys == 0 and not checkSonar(5):

         keys = keypadDecode()

      if keys == 1:

         reset()

         setColor(**'cyan'**)

         textout(**"Playing Anthem"**, 0x000000, 21, 48)

         textout(**"Press any Button"**, 0x000000, 17, 64)

         textout(**"to return"**, 0x000000, 35, 80)

         play\_song(ANTHEM)

         setColor(**'off'**)

         state = REQUEST

         test = lambda: play\_song(ANTHEM)

         reset()

      elif keys == 2:

         reset()

         setColor(**'cyan'**)

         textout(**"Playing Mario"**, 0x000000, 22, 48)

         textout(**"Press any Button"**, 0x000000, 17, 64)

         textout(**"to return"**, 0x000000, 35, 80)

         play\_song(MARIO)

         setColor(**'off'**)

         state = REQUEST

         test = lambda: play\_song(MARIO)

         reset()

      elif keys == 3:

         reset()

         setColor(**'cyan'**)

         textout(**"Playing Stranger"**, 0x000000, 20, 48)

         textout(**"Press any Button"**, 0x000000, 17, 64)

         textout(**"to return"**, 0x000000, 35, 80)

         play\_song(STRANGER)

         setColor(**'off'**)

         state = REQUEST

         test = lambda: play\_song(STRANGER)

         reset()

      elif keys == 4:

         reset()

         setColor(**'cyan'**)

         textout(**"Playing All-Star"**, 0x000000, 20, 48)

         textout(**"Press any Button"**, 0x000000, 17, 64)

         textout(**"to return"**, 0x000000, 35, 80)

         play\_song(ALLSTAR)

         setColor(**'off'**)

         state = REQUEST

         test = lambda: play\_song(ALLSTAR)

         reset()

      elif keys == 5:

         reset()

         setColor(**'cyan'**)

         textout(**"Playing Tetris"**, 0x000000, 21, 48)

         textout(**"Press any Button"**, 0x000000, 17, 64)

         textout(**"to return"**, 0x000000, 35, 80)

         play\_song(TETRIS)

         setColor(**'off'**)

         state = REQUEST

         test = lambda: play\_song(TETRIS)

         reset()

      elif keys == 6:

         reset()

         setColor(**'cyan'**)

         textout(**"Playing Fortnite"**, 0x000000, 20, 48)

         textout(**"Press any Button"**, 0x000000, 17, 64)

         textout(**"to return"**, 0x000000, 35, 80)

         play\_song(FNITE)

         setColor(**'off'**)

         state = REQUEST

         test = lambda: play\_song(FNITE)

         reset()

      elif checkSonar(5):

         setColor(**'red'**)

         time.sleep(0.1)

         state = HOME

         setColor(**'off'**)

         reset()

      else:

         state = MUSIC

   # default state required for the project.

   elif state == DEFAULT:

      # display not set but can be change only after each song

      music = 0

      demo = 1

      time.sleep(0.5)

      reset()

      textout(**"Demo Mode"**, 0x000000, 35, 48)

      setColor(**'green'**)

      time.sleep(2)

      anim(2)

      reset()

      textout(**"Waddle"**, 0x000000, 42, 48)

      dance1()

      setColor(**'cyan'**)

      reset()

      textout(**"Pop-Step"**, 0x000000, 40, 48)

      dance2()

      setColor(**'blue'**)

      reset()

      textout(**"Ballerina"**, 0x000000, 39, 48)

      dance3()

      setColor(**'magenta'**)

      reset()

      textout(**"High-Knees"**, 0x000000, 35, 48)

      dance4()

      setColor(**'red'**)

      reset()

      textout(**"Excite"**, 0x000000, 45, 48)

      dance5()

      setColor(**'yellow'**)

      reset()

      textout(**"Shuffle"**, 0x000000, 43, 48)

      dance6()

      setColor(**'white'**)

      music = 1

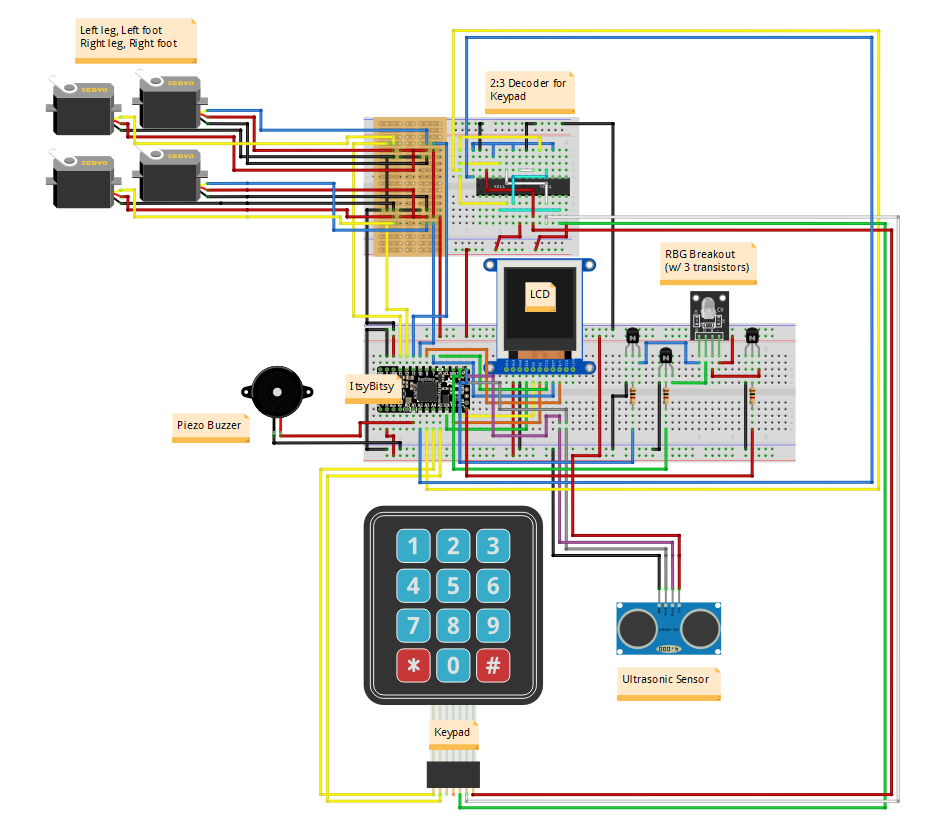
      demo = 0

      state = HOME

      animRev(2)

      reset()

**Appendix C - Fritzing**



Here is our fritzing file. The actual implementation of the RBG Breakout circuit along with 3 transistors is at the center of the bottom breadboard within the wires of LCD. The RBG Breakout circuit here is displayed on the right side of the bottom breadboard for clarity issues. Components such as Keypad, Ultrasonic Sensor and the Breadboards are put compactly inside the 3D printed robot chassis rather than separate as shown in this fritzing file.

**Appendix D - GitHub**

Parsa Riahi did the soldering and a majority of the hardware components wiring and cleanup. Often times group members would work off of other group members’ laptops for convenience and did not change the git username and email when committing, so some uncertainty is to be expected from the lines committed per team member.

**Appendix E – Complete Component list**

* ItsyBitsy M4 Express
* Color TFT LCD Display
* Piezo Buzzer
* RGB module with unified voltage line
* 4x servo motors
* 3 x 4 numeric matrix Keypad
* 3-D printed casing components
* 7404 NOT gate array
* 7400 NAND gate array
* 2x Breadboards
* Protoboard
* 3x 2N3904 NPN Transistors
* Various wires for inter-component connections
* Various sizes of male to male connectors
* HCSR04 Ultrasonic Range Sensor
* 3x 1k ohm resistors for amplifier circuits

**Appendix F – Answer the following questions:**

Q1 – Teamwork: Explain in detail the methods your group has used to communicate effectively among team members.

To streamline the development process, we clearly delegated tasks to each teammate both through a README on Github as well as in person, wherever possible. We developed a group culture of openness and honesty so that every team member was able to contribute and test their ideas effectively. This also allowed for extensive constructive criticism to occur so that we could rapidly prototype effectively and develop and polish a large breadth of ideas fast into our core additional functionalities and core features.

To maintain a consistent and efficient level of communication, our team used a messenger group chat to stay in touch while not meeting face to face. This was especially practical during the reading break since many of our team members were off-campus at this time, and yet we were still able to coordinate meetings, delegate tasks, and to complete the project during this time.

Another way we ensure that everyone is working equitably for the project is through the analysis of our Github commits in terms of lines added and removed, past peer reviews, and qualitative milestones reached for the hardware and for the report. This allows for a simple diagnosis of everyone’s workload, and gives the opportunity to each teammate to see where they may be falling behind, and thus incentivizing consistent self-improvement for the teammate as well as ensuring we meet deadlines.

Q2 – Design Process for the additional functionalities: Describe clearly the process you used for the following design aspects of your own additional functionalities. Please spend time carefully answering each of them.

1. **Use of process**: Describe your approach to adapt and apply a general design process for any additional feature. What was your approach?

We used an iterative rapid prototyping process to incrementally add new features to our bipedal robot. We began with the base requirements given to us by the lab description, then went on to address the open-ended problems we still saw with the system, adding multiple new features to improve client engagement and to be able to fully visually display the data we received from the RPi.

1. **Constraint identification**: Explain the constraints that you must consider in the design of the additional functionalities.

We know that our main constraint with the robot is the ease of use for the user, as well as the physical hardware constraints for the motors and sensors which can only measure and turn accurately to a certain degree, as well as physical spacing constraints.

To limit the user constraints, our robot has no barrier to entry and has an easy UI to easily visualize all options and provide haptic feedback through the RGB LED when keypad buttons are clicked and to indicate different color-coded states in the FSM.

One constraint we faced here was the way in which we implemented user control to our state machine. To give the appearance of synchronous behavior and instant interrupts we first tried to implement threading and interrupt routines, but neither of these is possible with the CircuitPython platform. Our group was thus given the opportunity to truly explore what made software seem so fluid and smooth at the low-level, and how to simulate the behavior we wanted with parallelism to imitate synchronization and sensor-triggered breaks in the code to emulate interrupt routines.

The main physical constraint is the limited size of the robot casing and number of pins on the Itsy Bitsy, which posed a significant hardware challenge for integrating a large number of components together. We made three significant changes to our design to accommodate this. First we centralized the servo controls off of the breadboard onto a protoboard. Then we reduced the number of pins the keypad used from 7 to 4 by adjusting our design to only use 2 rows of the keypad, as well as creating a 2 to 3 decoder for the row inputs to decrease the number of pins by an additional pin. Finally, these two solutions were both put onto a secondary breadboard, which fastened underneath the main breadboard, away from view so as to be more visually appealing but also working within our size constraint.

1. **Solution generation**: Explain at least two possible alternative additional features that your group rejected due to technical reasons and explain why.

We were considering thread implementations and asynchronous processing to execute servo movements and buzzer beats independently. However, the current implementation of CircuitPython does not have threading enabled as a decision by the developers.

Edge detection was one of the planned additional functionalities of the robot. The robot would stop moving once it detected that its location is near an edge. However, we decided that a more useful and robust motion/wall detection would be more useful, since the sonar’s angle is fixed and we could only choose one of these features and have them integrate effectively.

We were also limited in the number of pins on the Itsy Bitsy M4 board, which made it so that we did not have enough pins to use the bottom two rows of the keypad, as well as making our distance readings from the sonar less accurate since we did not have the physical space to implement a temperature measuring circuit. We also considered using a shift register but this too was costly on pin use. We thus compromised our hardware pins to only use 2 rows of the keypad for our additional functionality and assumed a default temperature of 21oC for the distance measurement.

The last major constraint that we faced was the limited memory on the Itsy Bitsy board, which limited the number of images we were able to display on our LCD and made our animations less fluid. We did a lot of file compressions and decreased the animation frame rate to make the best use of the limited storage capacity.

1. **Solution Assessment**: Explain how you tested and assessed the viability and then correctness of your group’s additional features.
   1. Through step by step testing of our state machine, we went through every possible combination and path in the state machine, looking to see that nothing crashed and that all the additional functionalities work by way of analyzing the RGB LED for the desired behavior we programmed to test the viability of the additional features:
      1. Every button click flashes the LED white
      2. The sonar detecting an object within 5 cm flashes the LED red three times then returns to the previous state machine menu page
      3. Dances turn the LED green, while songs turn it cyan
      4. The demo state rotates through the basic RGB colours along with each different dance.
      5. Any issues or errors flash the LED yellow
   2. The correctness of the additional features were debugged more qualitatively than quantitatively due to the nature of evaluating how correct the servos dance, how the songs sound in sync with the dances, how intuitive the keypad-LCD-LED system works for a user, and how reliable the sonar detects motion and walls.