Music Therapy Gait Assist Device

Electrical Engineering Senior Design 2023

Presented to: Dr. Michael Schafer

By: Patrick Condon, Anna Martelli, Ryan Schenck, Laurynas Zavistanavicius

Table of Contents

1 Introduction

Motor dysfunction results from various diseases and injuries, causing emotional distress, decreasing quality of life, impairing daily activities, and inhibiting agency. There are many medications, therapies, and rehabilitation treatments that vary in effectiveness, carry risks, and have high costs or low accessibility. The goal is to find a treatment method that is non-invasive, low-risk, low-cost, accessible, has proven success, and is widely applicable to the range of people afflicted with motor dysfunction. In Parkinson's Disease patients, the timing of repetitive sequences and movements are particularly affected. The most evident consequence of this deficit is the alteration of gait patterns including loss of rhythm, shorter steps, and slower gait. A cluster of cells called the substantia nigra pars compacta (SNpc) release the neurotransmitter dopamine to neurons of the striatum which project to the basal ganglia. The basal ganglia is connected to the thalamus and motor cortex which imparts control over motor output and learning. Unmedicated Parkinson's patients have a deterioration of the dopaminergic neurons in this region. [1]

Our goal is to utilize the principles of music therapy to create a wearable device that can provide real-time feedback for Parkinson's patients who have experienced a decline in gait control. Music therapy is a well-established technique for treating motor dysfunction across a wide variety of injuries and diseases.[2] The expansive territory of the brain that music activates has been proven useful in neurologic rehabilitation – the brain can rewire itself around damaged areas to create new pathways connecting motor centers. Music functions as an external timekeeper that supports the impaired function of the defective basal ganglia in patients with Parkinson's through the involvement of compensatory networks in the brain. Dopamine is also released when listening to music, which is why music therapy is particularly effective for Parkinson's patients with deteriorated dopamine production.

Regarding motor symptoms, a great number of studies have shown that music as rhythmic auditory stimuli (RAS) can improve gait-related movements, functional mobility, and balance, as well as freezing of gait and falls of gait in Parkinson's Disease. Entrainment is at the root of the principles of music therapy.[3] It is defined as a temporal locking process in which one system’s motion or signal frequency matches or persuades the frequency of another system to adjust to its own. The stronger signal entrains the weaker signal with one adjusting to the other, or if the two are equal, they both adjust to meet in the middle. In the case of music, the rhythmic signal causes the temporal-motor system to adjust and match its frequency.[4] Recent research has shown that modulating music output with patient data could be a promising application of music therapy.[5] This is novel because it would incorporate an aspect of user feedback, giving the patient auditory cues.

To incorporate feedback into the practice of music therapy, we designed a device that will track the gait speed of the user and compare it to the tempo of an audio signal. The wearer attempts to walk at their chosen target pace (denoted in steps per minute) while listening to music of that same tempo (denoted in beats per minute) in their selected genre. Depending on how close they are to the target pace, a pace-adjusted audio file is played. This is done seamlessly, with the music picking up from the spot in the song where the adjustment was made. If the user’s gait is slower than their chosen target cadence, the music speeds up to encourage entrainment with the faster signal. Thus, the feedback is pushing the user to maintain their faster desired tempo. If the user is moving faster than target, the music slows down to entrain the user’s motor function closer to the desired tempo. The goal is for a seamless transition in song position to make the experience as smooth and effective as possible. For Parkinson’s patients, typically the goal is to walk as fast as possible in an attempt to regain functionality. In practice, this means beginning at a slower manageable pace. Then, if the user is successful in matching the target gait, they can push themselves to move up to the next highest option. Once the user reaches a formidable level where they fall behind, the music will play at a faster tempo to encourage a speed increase.

In thinking about how to accomplish this, we noted several important requirements. First and foremost, we had to keep in mind that our target audience is older, likely less technologically experienced, and less physically able. Therefore, simplicity and wearability were paramount. In addition, the device had to be able to interface several different components: a user interface, a gait tracking system, and a music processing and output system. Throughout the design process, we were forced to keep this in mind while selecting components, communication protocols, and programming techniques in order to make this possible.

Our design largely meets these specifications that we set out to accomplish. It is able to accurately take user input, track when a step is taken, analyze the user's gait tempo, compare this tempo to the user's desired pace and music tempo, switch to the corresponding pre-processed audio file, and output the properly paced music. This device was also successfully designed to be a wearable apparatus that the user could easily walk with: it straps onto one's leg without issue, and is not too large as to impact the regularity with which one walks. It does, however, have a few downfalls that affect usability and its streamlined operation. Currently, in order to function at full capacity, the device needs to be connected via USB to a laptop or phone for power. Our original design included a rechargeable 3.7 V LiPo battery housed within our case. We made this choice so that the user could switch the device on and walk around hands-free with all necessary components securely on their leg. The issue occurs, however, when we use our battery power path. The device was unable to reset consistently, and would function with a delay or, occasionally, no output. We increased our battery voltage to 4.8 V and were able to operate using battery power, however this solution was not consistent, again incorporating delays, thus deteriorating the quality of our program.

Additionally, directly after an initial upload the program runs smoothly with virtually no delay: file transitions are accurate. However, after running for a substantive amount of time, the delays between switching music files increase and the music transitions become less accurate. The device will resume playing a few before or after the moment where the song was playing previously. Our last issue involved mounting the components on our board. The I2C communication between the microcontroller and the IMU on our PCB is faulty. This severely impacts the accuracy of our gait tracking capability. To regain this functionality, however, we were able to breakout I2C pins such that we could connect a breakout board with the same IMU to our microcontroller. This fix allowed us to prove the concept, and demonstrate a functional product via laptop power.

2 Detailed System Requirements

2.1 Subsystem 1: Gait Tracking

- Device needs to accurately detect motion via our onboard accelerometer. The accelerometer will record when movement has occurred, and our gait analysis will translate these movements into steps. The steps will be quantified into a running average of steps per minute (SPM).
- The device should be rugged enough to withstand sudden, quick, and sustained motions up to running speed. Accelerometer should not break down with sudden or accelerated movements. Its physical placement and attachment are factors to consider here.

2.2 Subsystem 2: Gait Analysis

- The system must be able to register a series of steps in order to judge the speed of one's gait. In order to do so, the code must specify how much time occurs in between each step. Software uploaded to our microcontroller must do this by communicating with the accelerometer to judge a "step" event, and take note of the timestamp.
- The next step is to translate this raw step data into a walking pace with a simple calculation to extrapolate the total time taken for those 20 steps to the same pace over a minute. Finally, send this step information to the music processing subsystem.
- The gait analysis should be resilient to any button presses (adjusting volume, for example) or music changes while walking.

2.3 Subsystem 3: User Interface

- Interface should focus on user input and control, as output is solely audio.
- The interface should be simple and intuitive - our desired user base is older, and many are coming with limited technology experience. The interface should accurately display gait and music options, but nothing more to confuse the users.

- The interface must be relatively small. The device is intended to be worn on the user's thigh: anything too large could be bulky, heavy, or limiting for one’s range of motion.
- The interface must communicate easily with the microcontroller as base selections are made. The interface must also turn the volume up and down in a prompt manner.

2.4 Subsystem 4: Music Storage

- The device must have a source of external memory that can store the data necessary to present multiple options of music. The method will need enough storage to keep a range of bpm which coincide with desired gait paces. Preferably, there is enough storage to have 5 or more pre-processed tracks for each song to be slowed down or sped up at multiple paces.
- Storage must be self-contained in our final device, so will likely need to be located on our pcb itself.

2.5 Subsystem 5: Music Processing

- The music processing subsystem must first be able to access the processed music from the storage location.
- Music processing must have access to live gait analysis data.
- Music processing must take the live gait speed, compare it to the desired gate, and calculate the speed factor which indicates whether the music should be sped up, slowed down, or kept the same.
- The music tempo must be updated to reflect this speed factor when a deviation from the desired pace has occurred.
- Lastly, the music processing must maintain that the music is pleasant to listen to. This means that it cannot alter the pitch when altering the tempo. This will require significant processing to avoid noticeable pitch changes.

2.6 Subsystem 6: Music Output

- System must have a connection to a loudspeaker. A loudspeaker will be the most viable option for a demonstration to a group of people.
- Audio output must have adjustable volume that can change while the device is in use.

2.7 Subsystem 7: Power and Regulation

- The device must use a rechargeable battery that is not too bulky or heavy to the extent that it makes wearing the device uncomfortable.
- The battery must be able to provide power for the whole system for a reasonable time period. Since this device is not directly connected to the user, worries about power regulation to other medical devices that interface directly with the body (ex. an electrode) are not relevant.
- Power must be provided for all necessary voltage levels (VDD for the microcontroller and any higher voltages needed for the user interface or any other subsystems). This VDD should be 3.3 V for the ESP32 series. Ampacity must also be taken into account - our review of similar methods suggest that 1 A of current will be more than enough for our purposes.

2.8 Subsystem 8: Wearable Components

- The packaging must have ample room for all electronic components. Each component must be able to fit nicely into a casing such that no dangerous wiring is exposed. Additionally, the packaging must be able to disperse heat in the event that the device is active for a long period of time.
- The packaging and attachment mechanisms must be able to withstand sudden movements like rapid step events.
- The attachment mechanism of our system must be comfortable such that a user with motor dysfunction is willing to wear the device when embarking on a walk.
- The packaging must be lightweight so as to not impact the gait of a user.

3 Detailed project description

3.1 System theory of operation

The system turns on when a switch is flipped to open the power path to a lithium ion battery. This power source is regulated to 3.3V which powers the microcontroller and peripherals for each of the subsystems: the display, accelerometer, SD card reader, music amplifier, and speaker. This power delivery path also includes a recharging circuit. While the power switch is flipped off, a power source can be plugged into a USB-C connector port which will then recharge a lithium ion battery.

Once the device is powered, the user interface will turn on. On the interface, there are a series of simple pages that take you through all the necessary selections. The user first selects their desired page of gate, specified in steps per minute (SPM). There are four options: 60, 75, 90, and 105 SPM. The next page allows the user to select the genre of music they want to listen to. Again there are four options: rock, latin, pop, and rap. Finally, the user sets their volume level on a scale of 0 to 100. The user should then strap the device to the outside of their right thigh, and select the start button to begin.

The start button triggers the beginning of the two main functions, gait analysis and music processing/output. The user will know that the device is running as music will begin to play out of the speaker. The system plays a song with a BPM and genre the same as the pace and genre which the user selected. The song will begin at a normal playback speed.

Gait tracking also commences. The microprocessor communicates over I2C protocol with the accelerometer. The accelerometer constantly tracks information such as displacement and acceleration on all three axes as well as rotational acceleration and positioning. The gait tracking is done in intervals of ten steps. The algorithm we developed uses linear acceleration data as well as rotational positioning. The algorithm determines when a step is taken with the user’s right leg and clocks the timestamp.

After ten steps from the right leg, the total time for that interval is used to calculate the average time taken per step. This is converted to an SPM number, which is sent to the music processing system.

The music processing system takes this SPM number and compares it to the desired SPM number selected by the user. A speed factor, the ratio of the desired SPM to the user's pace, is then calculated. It is calculated in this way because we want the user to do the inverse of their error; if they are walking too fast, the speed factor will be below one and the music should then slow down so as to queue the user to slow their gate. This calculated speed factor is then rounded to the closest of five values: 0.8, 0.9, 1.0, 1.1, and 1.2. The music has been preprocessed to play at one of these five tempos. If the speed factor is one, then the song continues to play at the original pace as the user is meeting their goal. If the speed factor is not one, however, the music will then begin to play at the pace indicated by the speed factor. The user can also adjust the volume while the program is running.

This cycle runs continuously until the stop button is pressed or the switch is moved to the off position. If the stop button is pressed, the user will then be brought back to the initial screen where they can select a desired gait.

3.2 System Block diagram

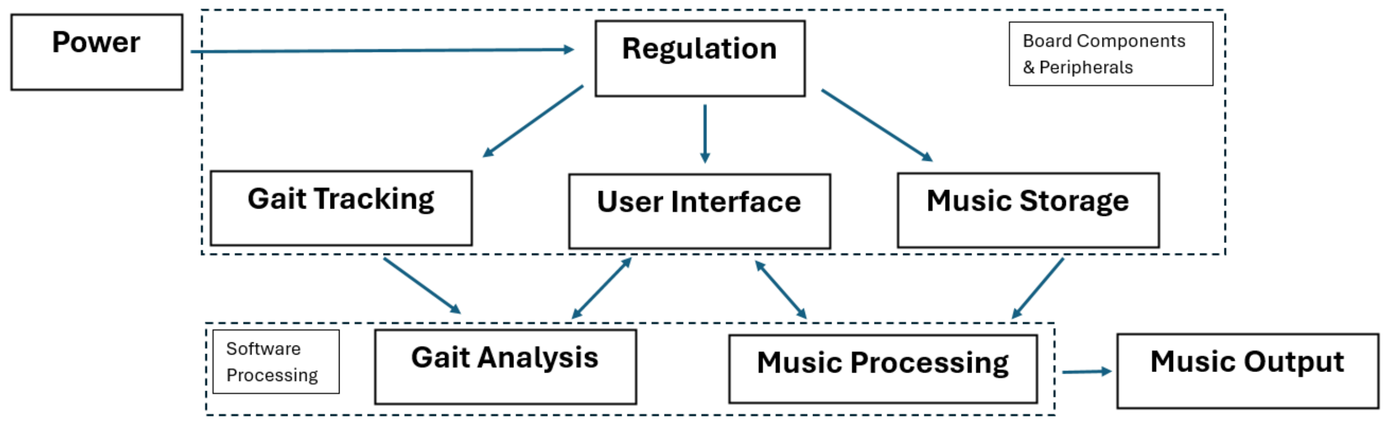


Figure 1. System block diagram.

Our block diagram is divided up into two main physical blocks: components & peripherals that are physically contained on our PCB and software elements contained in our main C program. Our Power subsystem physically feeds into the on-board PCB elements.

3.3 *Gait Tracking*

The gait tracking subsystem has one primary component: the BNO055 nine-axis IMU. The subsystem has the following general requirements:

- Accurately detect motion of the user
- Withstand these movements

The data points required to judge a step event are acquired from the BNO055. The BNO055 is a nine-axis IMU, equipped with a gyroscope, accelerometer, and magnetometer. The BNO055 was chosen as our mechanism for gait tracking because we had referenced the “Head Bangerz” 2023 senior design project which used the same component, and discussed it with Dr. Schafer as a viable option. The subsystem is powered by the main 3.3V power plane. The BNO is connected to the microcontroller via I2C protocol from GPIO pins 8 and 18 wired to the COM0 and COM1 pins on the BNO. The BNO’s COM3 pin is pulled low to set the I2C configuration in “slave” mode and the I2C address to register 0x28.

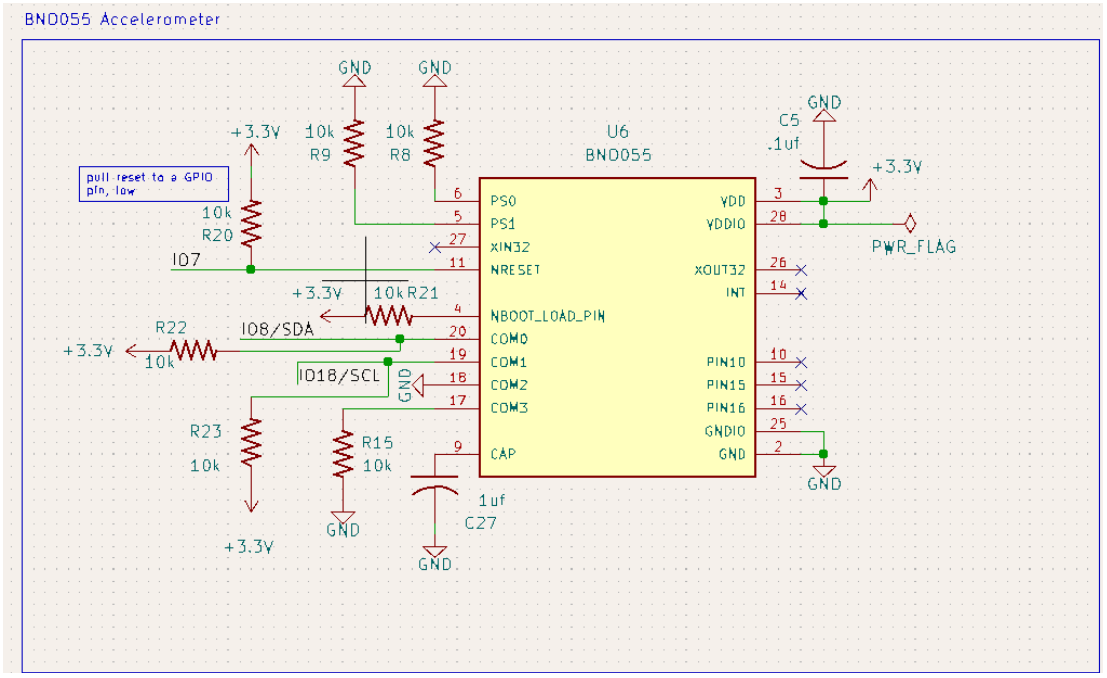


Figure 2. BNO055 subsystem schematic.

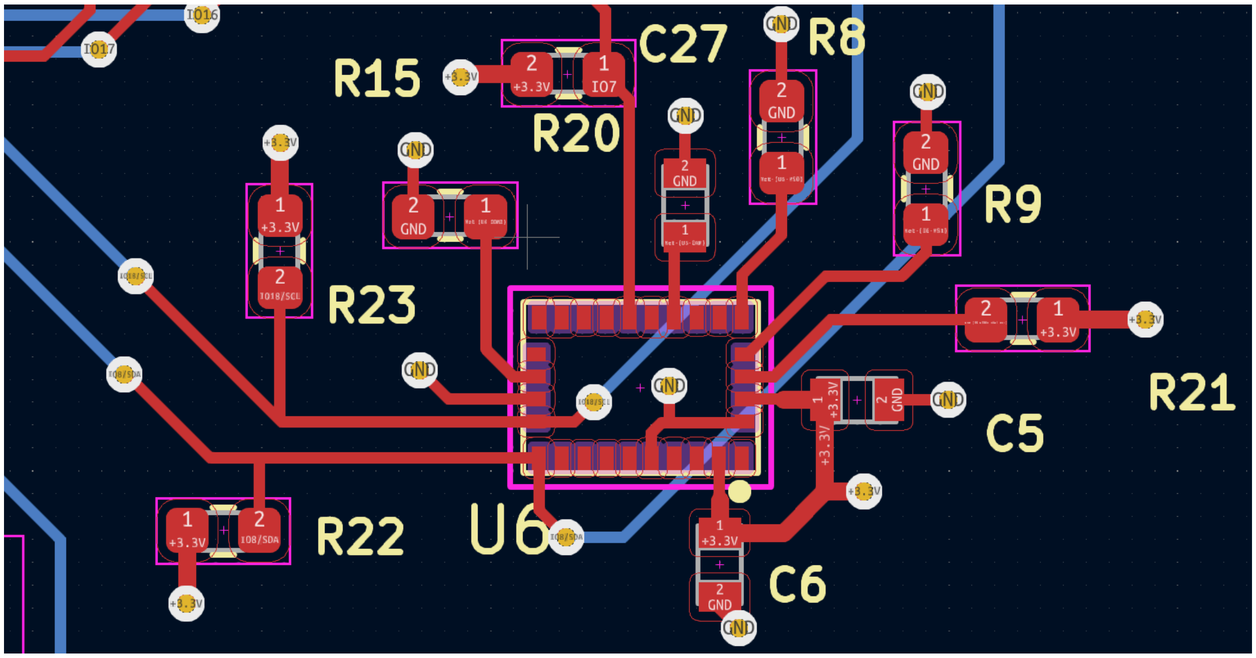


Figure 3. BNO055 subsystem PCB design

The BNO055 is able to monitor a plethora of data points using each of its three components. These data points are constantly monitored whenever the device is powered on. The data points used by the step tracking algorithm are the linear acceleration on the x, y, and z axis, as well as the euler angle in the y plane. When the device is worn, the euler angle in the y plane is 0 degrees when one’s leg is straight down in standing position. The angle increases as the leg hinges forward, or reverts to 359 degrees and decreases when the leg hinges backward. These are the values which the system monitors while the start button has enabled the program.

One issue that was persistent while using the BNO055 was mounting the component for solid connections and consistent I2C communication. We realized too late that placing the ground vias under the part likely raised one side too high for a solid connection. This, in addition to the fact that the part was surface mount, made it nearly impossible to re-solder the part, and despite our best reflowing efforts we were unable to achieve a good connection. Thus, we were forced to use a breakout board for our demo. We would recommend anyone building on this project move the vias and consider using a different IMU that is not surface mount.

3.4 Gait Analysis

Gait analysis is done within Task 1 in the RTOS program which integrates the functionality of each of the device’s subsystems. The general requirements of the subsystem are:

- Register a step event
- Determine the user’s walking pace
- Continuously do these things independent of other tasks

The program is constructed in a loop which only runs if the start button has been pressed. The program continuously runs while the device is being used. At the beginning of each iteration, the code communicates with the BNO055 to acquire 4 data points: linear acceleration along each of the three dimensional axes, and the angle of orientation of one’s leg relative to the plane in which it swings: the euler angle in the y direction. The algorithm then calculates the total linear acceleration using the three axis data points, and stores that value. The algorithm also stores the current euler angle and the previous euler angle (determined at the end of each loop). These three values are used to determine a “step” event:

- The first criterion for a step event is a minimum linear acceleration value. This is to ensure someone is swinging their leg. Through iterative trial and error, as well as referencing similar step detection algorithms, we set this value at 1 m/s^2.
- The second criterion for a step event is a minimum euler angle in the y plane. This is to ensure that someone’s leg is in a forward position. Similarly, through research, trial, and error, we arrived at the value of 10 degrees above parallel.
- The final criterion is that the current euler value is larger than the previous value. This ensures that the leg is actually moving forward, and that we are not double counting steps when the leg is on the downswing.

If these three criteria are met, a “step” is counted. This is done using an if statement. Within this if statement, the step is timestamped relative to the previous step. The program is then delayed momentarily to ensure that a second step is not counted on the same upswing. Then, the number of steps since the last SPM calculation is analyzed. If ten steps have been taken, the timestamps are summed to determine the total amount of time taken to complete 10 steps. Given that only the right leg is being tracked, we complete all the subsequent calculations relative to 20 steps:

- The total time in seconds is divided by 60 to determine the time value in minutes
- SPM is calculated by dividing the total number of steps, 20, by this total time in minutes.
- This SPM number is then saved, and all values are reset.

If the number of steps was not yet ten, then the step counter is simply incremented.

If a step was not registered, this process is skipped entirely and time is incremented by 0.1 seconds.

Lastly, the previous euler angle is updated to the current euler angle. The program then goes back to the top of the loop to retrieve updated values from the BNO055 and restart the process.

This SPM number is stored as a global variable, such that it would update and be accessible to the music processing algorithm.

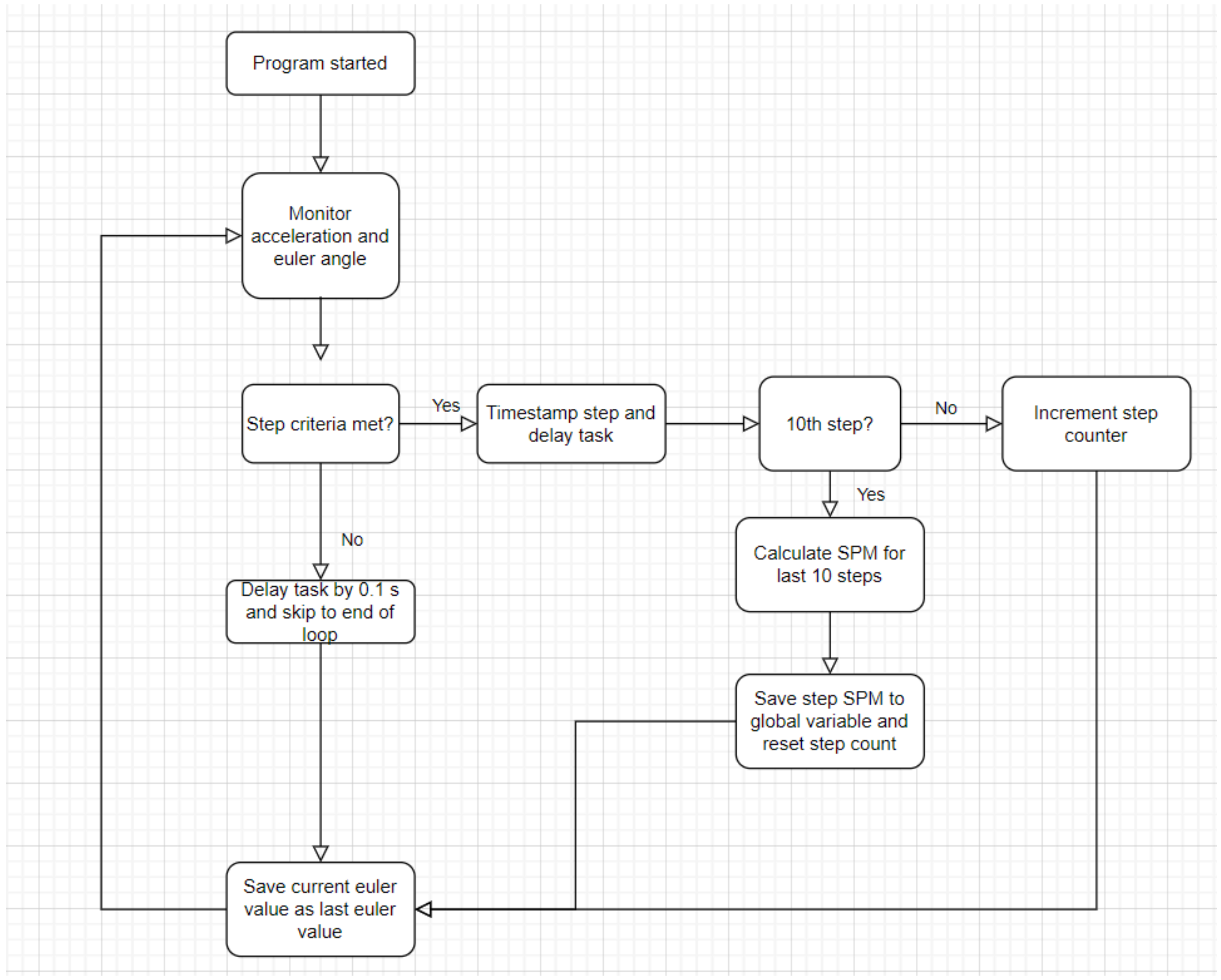

Program started
Monitor acceleration and euler angle
Step criteria met?
Yes
Timestamp step and delay task
10th step?
No
Increment step counter
Yes
No
Calculate SPM for last 10 steps
Delay task by 0.1 s and skip to end of loop
Save step SPM to global variable and reset step count
Save current euler value as last euler value

Figure 4: Gait analysis flow chart

Code: see task1 in the appendix.

3.5 User Interface

The user interface was implemented on the Nextion NX4024T032 3.2-Inch Basic Series HMI Touch Display. The general subsystem requirements were:

- Input driven

- Simple and intuitive
- Small
- Easy to integrate

This Nextion was chosen because of its touchscreen display, preexisting editing software, and size. A touchscreen display was implemented such that the user could cycle through several pages of options without having to use the same set of pushbuttons. The software contained many of the simple building blocks that would be ideal for our older, less technologically experienced target user base. Lastly, the 3.2 inch version was large enough such that it could easily display several options, but still manageable to wear on one's thigh.

The Nextion interfaces with the microcontroller via RX/TX serial communication. The nextion hardware came pre-equipped with a simple 4-pin JST female connector with VDD, GND, RX, and TX pins. Therefore, on our board, we broke out those same pins to a 4 pin JST connector, routing GPIO pins 16 and 17 for RX and TX respectively:

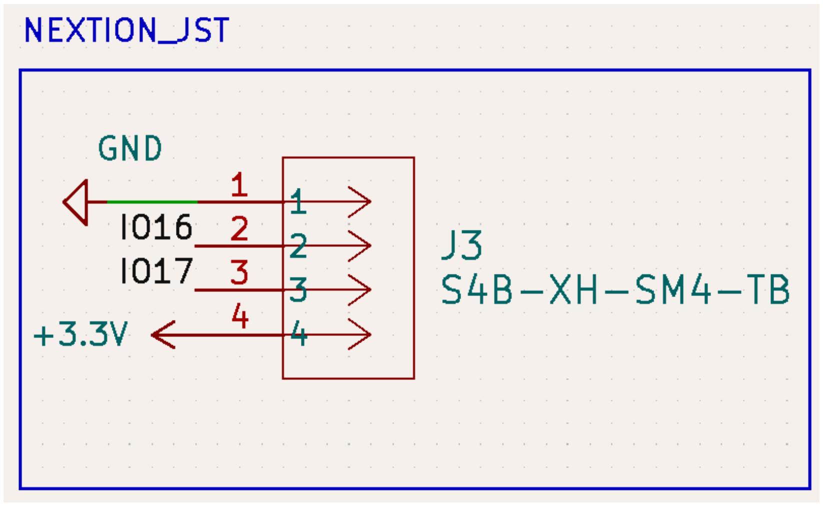


Figure 5: Nextion JST connector on PCB

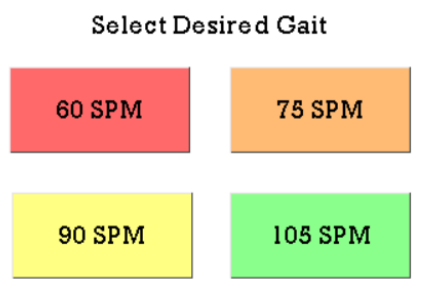
This allowed us to wire the JST directly to the board, only requiring crimping for the female to female JST connector. The graphics were uploaded to the board prior to implementing the program. Therefore, the only information transmitted via the serial communication was registering the selections made on the interface.

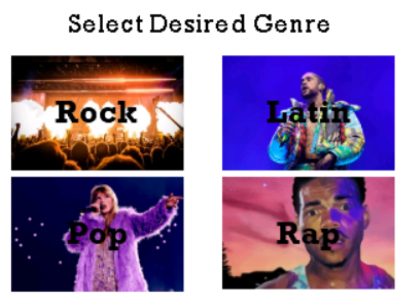
Communication between the microcontroller and user interface was facilitated by the Nextion Software Editor as well as the setup in the RTOS program. Within the software editor, options for selections were uploaded as buttons. For buttons, there is an option to “send component ID” when the button is touched. Selecting this option sends the component ID of the button pressed to the microcontroller, which can then be referenced in the RTOS program.

Within the RTOS program, each button must be initialized by specifying the page it is on, the component ID, and the name. Next, a function was implemented for each button. This dictates what the program should do when that button is pressed. For example, if the button opting for 60 SPM was selected, the function target SPM was set to 60, and the speed component of the name of the wav file was specified to be 60 SPM. These parameters are initialized as variables at the start of the program, and are therefore accessible to each of the tasks within the program. Each of those button functions are then listed in a “listen” function, which indicates that the program should be on alert for those component IDs being sent over serial. Lastly, within setup, each button is initialized so that it can be registered when pressed.

Those function setups are all available through the Nextion library that was added to our program. The library includes numerous header files and other library dependencies. Several of those files had to be modified to make sure that our program and hardware were compatible with the nextion software and hardware. Notably, the Nexhardware.cpp file where you specify the path for Serial communication.

The interface itself looks like the following:





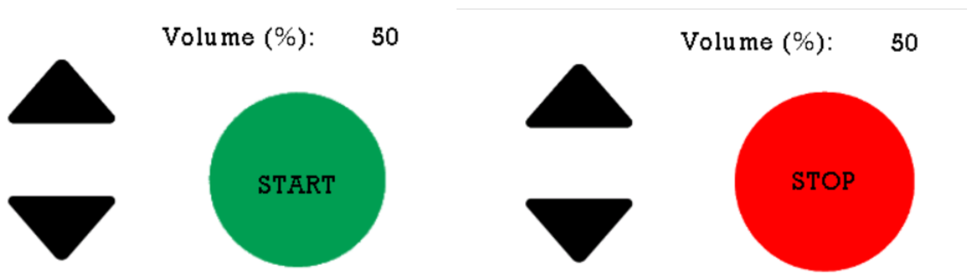


Figure 6. User interface: page 1 (top left), 2 (top right), 3 (bottom left), 4 (bottom right)

This interface design is deliberately simple. It seeks to keep the format consistent between selecting the gait speed and the genre, with four boxes in each corner. The same pattern holds for the start and stop pages. Only the words and color on the start and stop buttons change, leaving those buttons and the volume buttons in the same place.

Each button on each respective page modifies a variable accessible in the program. The gait selection modifies the integer “target_SPM” variable, the genre selection modifies the integer “genre” variable, the volume selection modifies the integer “volume” variable, and the start/stop buttons modify the boolean “start” variable. The first two variables are combined to select the appropriate wav file, the third modifying the voltage of audio output, and the last being the determinant parameter for loops which run tasks 1 and 3.

Code: See appendix

3.6 Music Processing

The music processing was done using the Harmonic Percussive Source Separation Time Scale Modification (HPSS TSM) algorithm by Driedger and Müller [7]. First, the music signal is decomposed into its harmonic and percussive components via horizontal and vertical median filtering of the signal’s STFT, respectively. The harmonic component is stretched using a phase vocoder, whereby an STFT of the signal is taken, the phases of consecutive frequency bins are modified to be continuous across frames, and then the ISTFT is taken with a synthesis hopsize proportional to the time stretch factor. The percussive component is stretched using Overlap-Add (OLA), whereby each analysis frame is Hann-windowed before being resynthesized by the desired stretch factor.

In our original design, our goal was to implement this algorithm on our ESP32 microprocessor so that we could process the music in real time. However, we ran into a number of issues, including the prohibitively long amount of time it would take to manually convert this open-source algorithm from MATLAB to C, and the amount of computing power running this algorithm continuously would consume .In our final design, we chose to preprocess our music. For every song by target SPM and genre, we used the MATLAB implementation of the HPSS TSM algorithm to produce wav files of time-stretched versions of each song at stretch factors of 0.8, 0.9, 1.1, and 1.2. All of the music files were downloaded onto our 32 GB microSD card to be used on our PCB.

The code for switching between playback speeds occurs in the PlayWav() function of the music output subsystem (see 3.8 Music Output for where the switching code fits into the music output process). When the user presses the START button, the start flag becomes TRUE, which initiates the gait tracking process on task 1, core 0 and the playWav process on task 3, core 1. As the data is read from the SD card and sent to the I2S buffer to produce audio output, the global variable speedFactorcurrent is updated every 10 steps the user takes and rounded to 0.8, 0.9, 1, 1.1, or 1.2. If speedFactorcurrent has changed, it will no longer be equal to speedFactorprev, and this will initiate the Switch Wav File process.

As music is read to the I2S buffer, the variable BytesReadSoFarMusicTime (BRSFMT) keeps track of how many bytes have been read from the song normalized to the unstretched version. For example, if 1.5 seconds of music (corresponding to 1.5 s * 44.1 kHz/sample * 2 bytes per sample * 2 channels = 264600 bytes) stretched by a factor of 1.5 have been played, this corresponds to 1 second of unstretched music, or a stretch factor of 1 (176400 bytes). To find the location from which to start reading after the speed factor has changed, BRSFMT is divided by the new speed factor. Continuing the example, if switching to the 0.5 stretch factor file, the jump location would be equal to BRSFMT / 0.5 + 44 = 88244 bytes (the first 44 bytes are part of the wav header and don't contain any of the musical signal). A visualization that these locations all correspond to the same musical event is shown in Figure 7. It is important to round the jump location to the nearest multiple of 2 because all of the samples are 16 bits or 2 bytes. If this value is not rounded in this way, this results in white noise being played 50% of the time after a wav file switch occurs, because the byte values are no longer interpreted correctly. We use a standardized file naming system to simplify the file selection process. Each wav file name consists of three 3-character strings concatenated together. The first string denotes the target SPM, the second string the genre, and the third string the stretch factor applied, followed by the '.wav' file ending.

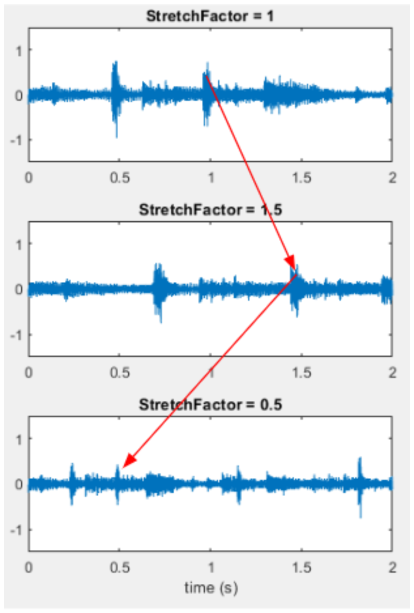


Figure 7: Diagram marking corresponding musical events after applying time stretches of 1.5 and 0.5. All of the marked locations have the same BRSFMT value

When the Switch Wav File process occurs, we close the current wav file being read, change the stretch factor string, and then reconcatenate the strings to select the new wav file from which to read using SwitchWavFileName(). The file pointer location is updated to be in line with the musical time using WavFile.seek(jump_location), where jump_location is computed as described in the paragraph above. speedFactorprev is then set equal to speedFactorcurrent to indicate that the wav file has been switched successfully, and then the music output loop continues as normal to read data from the SD card in the new music file and place it in the I2S buffer.

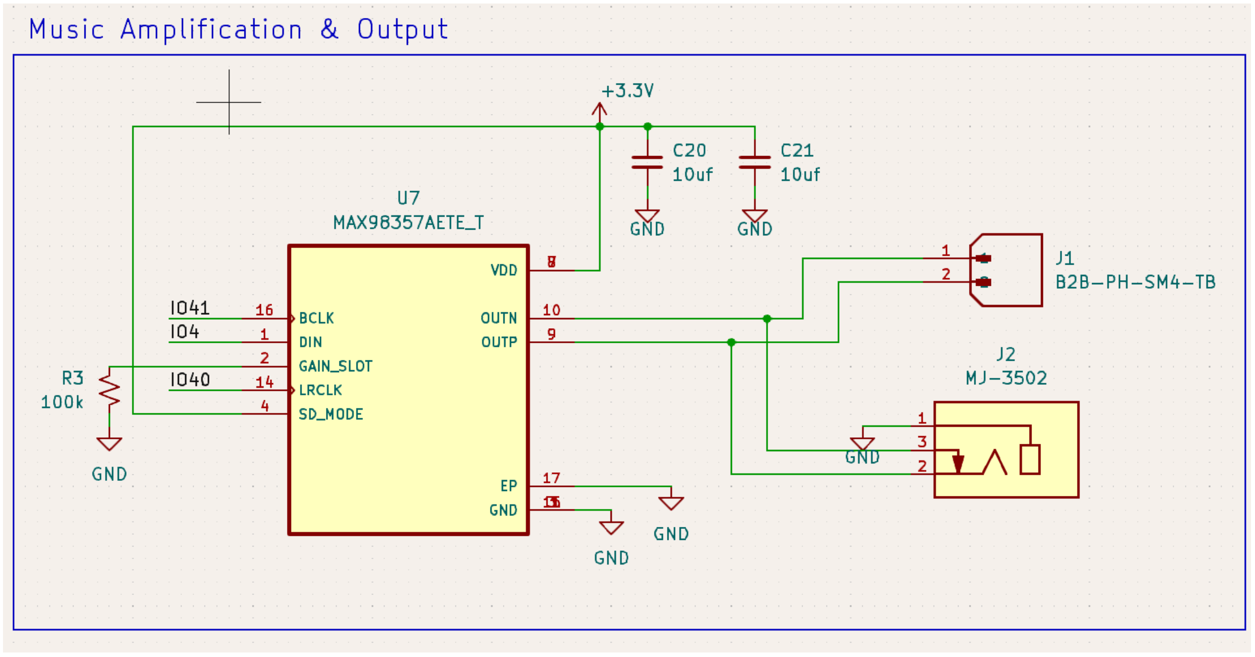


Figure 8: Music Amplification & Output schematic design

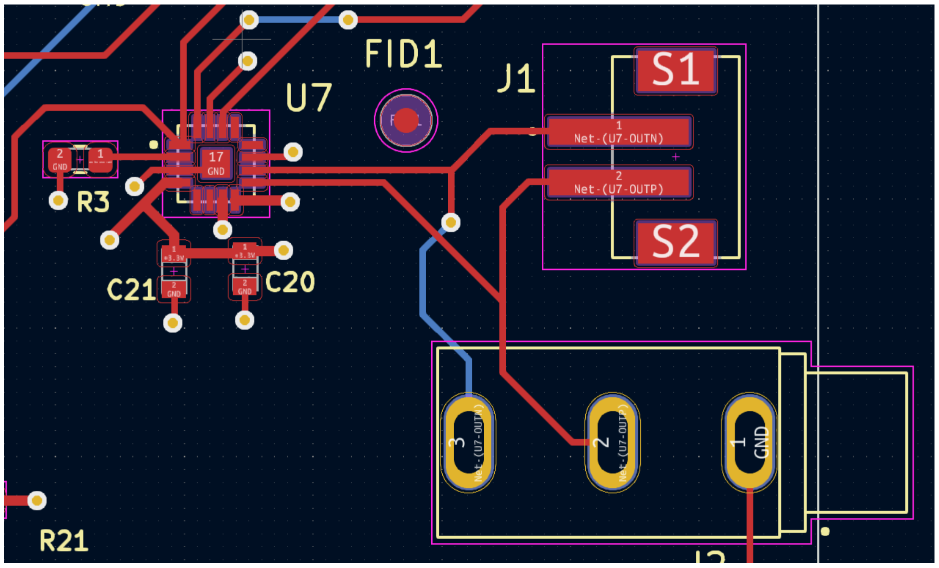


Figure 9: Music Amplification & Output PCB Design

3.7 Music Storage

The number of songs we wanted available to the user required an external memory component in our design. After determining the library of songs we needed for the device, we calculated their storage requirements to be around 4GB. To hold the music files we purchased a microSD card. Since they have a wide array of storage space and the schematic/pcb component would be the same for every variation, we chose to purchase a 32GB microSD card to allow for expansion of the music library if needed. Our schematic for the microSD is shown below:

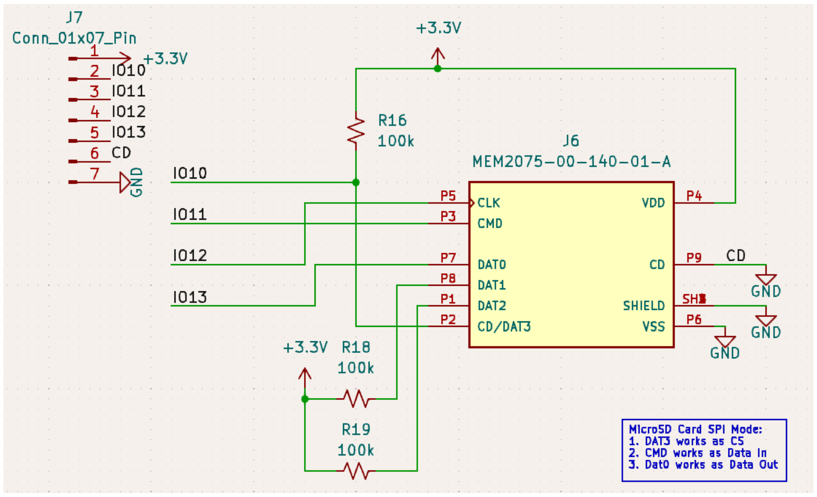


Figure 10: Music Storage subsystem schematic design

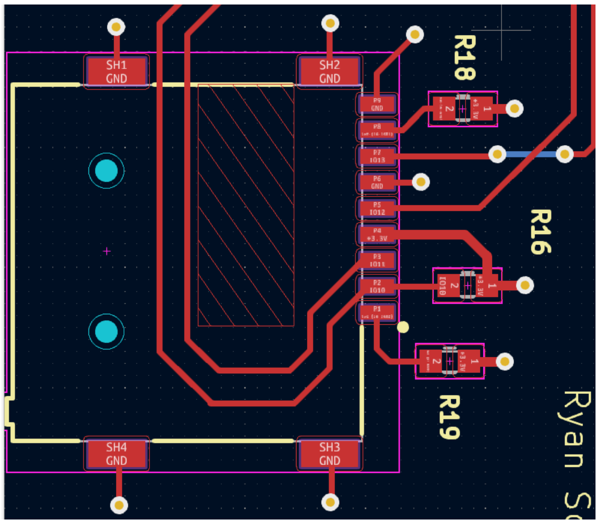


Figure 11: Music Storage PCB Design

We chose to operate the SD card in SPI mode, therefore our connections to the microcontroller consisted of the clock (CLK), chip select (CS), data in, and data out. We found the corresponding SPI pins in the ESP32-S3 and matched them to the pin assignments on the SD card. CD/DAT3 operates as CS in SPI mode; we connected this to FSPICS0/GPIO10 on the microcontroller and pulled it high through a 100kΩ resistor. CMD operates as Data In in SPI mode, so we connected it directly to FSPID/GPIO11 on the microcontroller. CLK connected directly to FSPICLK/GPIO12. Dat0 operates as Data Out in SPI mode, which we connected directly to FSPIQ/GPIO13. VDD was pulled high to 3.3 V and VSS and Shield were connected to ground. CD was connected to ground since we are not using the chip detect functionality, as in our use case the microSD card will not be removed so detecting its presence was not necessary. In SPI mode, DAT1 and DAT2 are not used so we pulled them high through a 100kΩ resistor according to examples from microSD breakout boards, such as the SparkFun Level Shifting microSD Breakout. We chose the MEM2075-00-140-01-A component for the microSD card slot on our PCB because it had the correct number of positions, push in/push out functionality, and the largest quantity available listed on digikey.

As mentioned earlier, to begin testing we used the SparkFun breakout. The pin inputs consisted of power, ground, and straight connections to our microcontroller. This breakout was useful in initial testing, however it included a voltage shifting capability from 5 V, which was not necessary in our case since we use 3.3 V across the board. As mentioned in the music storage system description, our first PCB was not configured as shown in our final schematic. The initial schematic was modeled after the ESP32-S3-korvo-2 development board, which after further review did not align with our needed configuration for operation. Instead, we looked at the SparkFun board, microSD card datasheet, and SD card pin assignments and matched pin configurations accordingly, without the shifting aspect. To test this, we soldered wires onto the pins of a microSD to SD card adapter and breadboarded to our MAX98357A and were able to pull audio files from the microSD and play through our speaker. Then we connected the other breakouts such as the BNO055 and ensured music was output when integrated through the whole system. In our final board iteration we included the breakout pins seen in the top left for possible troubleshooting if needed.

3.8 Music Output

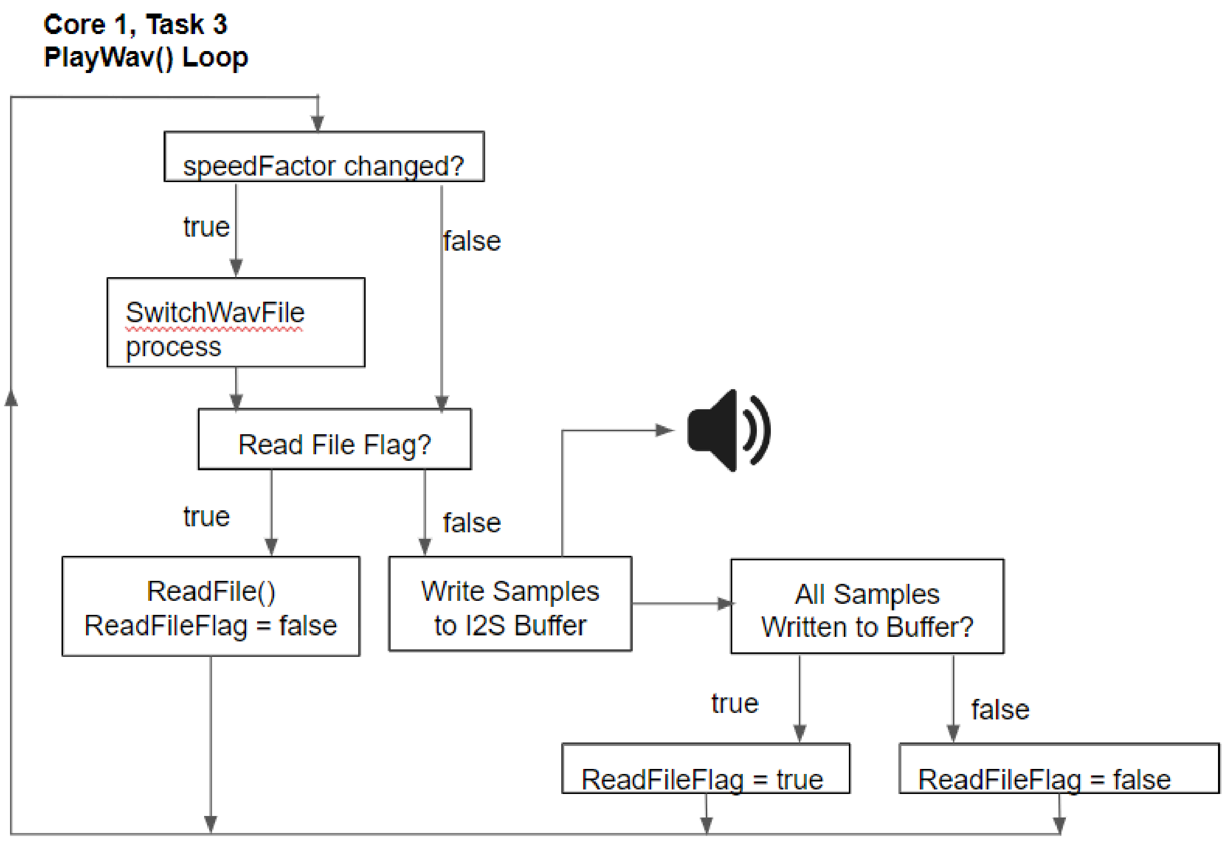
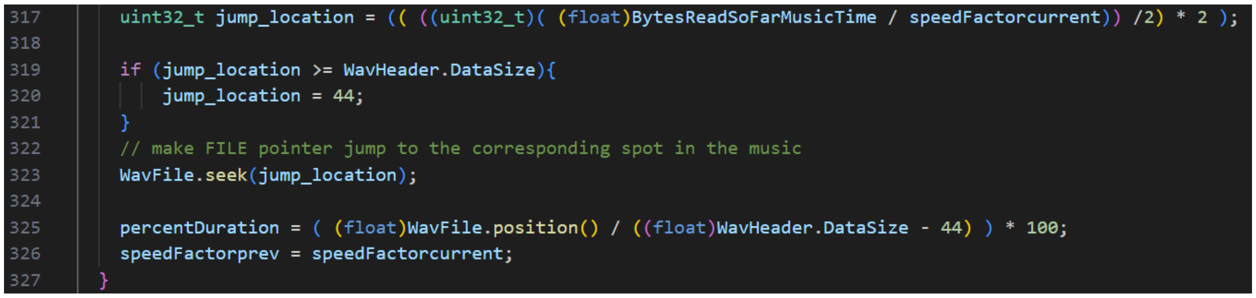


Figure 12: Music Output Block Diagram

The diagram above shows the steps to outputting the preprocessed music. PlayWav() is the name of the function we used to continuously output music using a buffer array. If the output of music processing determines the file needs to be switched, it will open the new file from the SD card and play from the location of the file pointer. The file pointer location is calculated and used in order to have seamless transition between song files, playing the song from the same point it was changed, instead of starting over every time. Within PlayWav(), if the speed of the user has changed, meaning the file needs to change, the program will enter this section of code that updates the pointer.



```
    uint32_t jump_location = (( ((uint32_t)( (float)BytesReadSoFarMusicTime / speedFactorcurrent)) /2) * 2 );

    if (jump_location >= WavHeader.DataSize){
        jump_location = 44;
    }
    // make FILE pointer jump to the corresponding spot in the music
    WavFile.seek(jump_location);

    percentDuration = ( (float)WavFile.position() / ((float)WavHeader.DataSize - 44) ) * 100;
    speedFactorprev = speedFactorcurrent;
  }
```

Figure 13: Section of PlayWav() code for music output

The value of jump_location determines what music will be output. From this, PlayWav() loads in 1024 bytes from the file in the SD card into a buffer. The read pointer outputs this section, then when it reaches the end such that the write pointer equals the read pointer, indicates that the buffer needs to be filled again. This is executed within the ReadFile() function, included in the full code. We output music using the MAX98357A which operates over I2S. This component is a digital pulse-code modulation (PCM) audio amplifier and supports I2S data, utilizing simple operation that operates under the conditions we needed. At first, we considered using a complicated audio codec to output a digital signal, however the MAX983571A coincided more with our schematic and system requirements to output high-quality audio in the most effective/simple method over I2S outputting an analog signal. The schematic is pictured below:

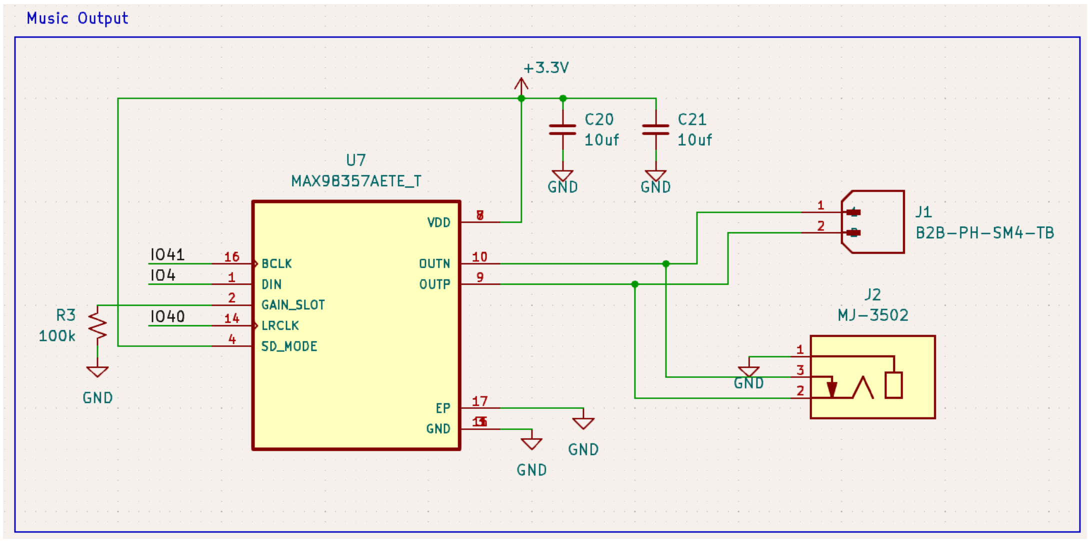


Figure 14. Music Output Schematic

The bit clock (BCLK) is connected to GPIO pin 41, DIN is connected to GPIO pin 4, and left right clock (LRCLK) is connected to GPIO pin 40. These were not constrained by the ESP32-S3 pin layout so we chose pins to coincide with our PCB layout and configured them accordingly in the code. GAIN_SLOT is connected to ground through a 100kΩ resistor because according to the datasheet, this would give us the maximum volume output available at 15dB. Seen within the ReadFile() function in our code, we add functionality to control volume level, with increments of 10 from 0 being no volume to 100 being the max of 15dB. This was done by linearly adjusting the amplitude of the output signal in the read buffer. SD_mode determines the audio channel that is connected to the amplifier output; it is connected to VDD to select the left mode of the stereo input data. The left and right channels within our files match, therefore the output is the original audio data when output through just the left channel. The negative and positive output are connected to our 4Ω, 3W speaker that plays the music. We also included a connection to a TRS audio jack for an optional second output method, which we didn't use for demonstration purposes. By convention, the tip (T) is connected to the positive signal, ring (R) is connected to the negative signal, and the sleeve (S) is connected to ground.

For the I2S configuration, the following parameters were set:

- ESP32 is set as Master mode, sending TX signals
- Sample rate of 44.1kHz
- 16 bits per sample
- RLRL (right left) wav file channel format for stereo output
- I2S communication format
- Interrupt priority 1
- 8 buffers of 1024 bytes each

3.9 Wearable Components

We designed our wearable to be a case that fully contains and encloses all of our subsystems. We wanted our user interface, power USB, power switch, audio jack, and the reset button to be the only things accessible from outside the case. Our PCB was laid out in a way to make all this possible. We then designed our physical case in SolidWorks to be 3D printed, as shown below:

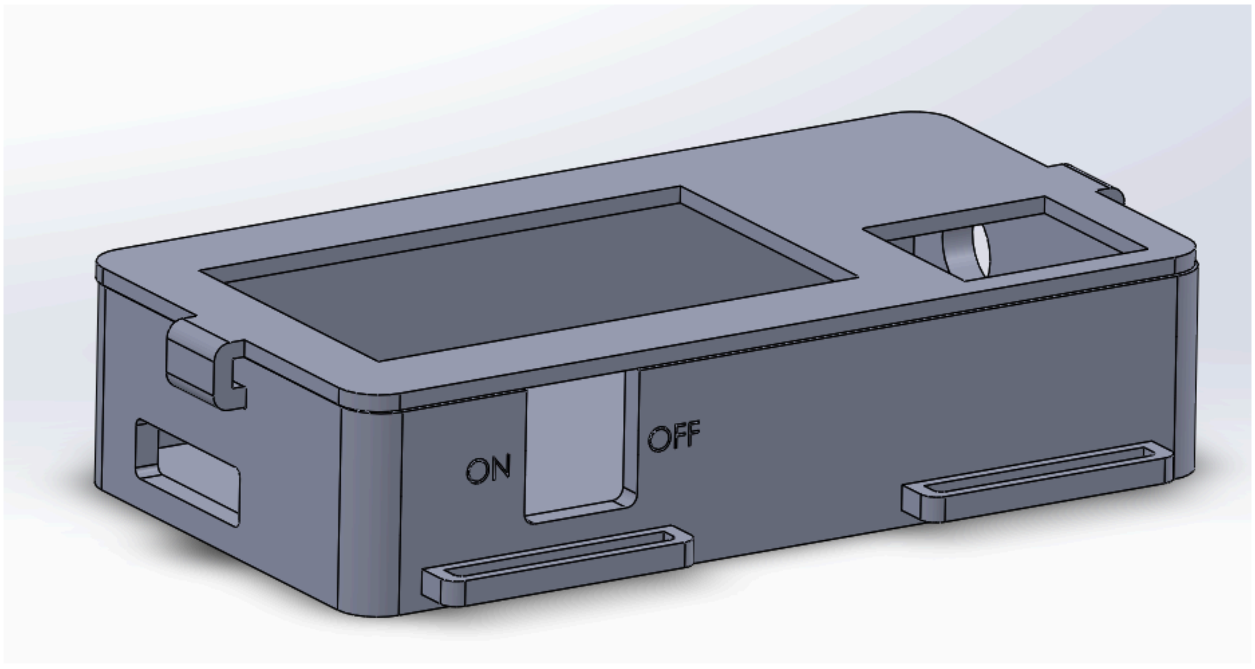


Figure 15: SolidWorks rendering of the case

For this prototype, our greatest considerations were making sure all wiring and components were fully enclosed in the case. It was designed to be lightweight, comfortable, and as small as it could be given our hardware. The case sits at 5.65x3x1.3 in^3. In future iterations, all of these parameters will be improved. Spaces accessible to the user are shown as the three visible holes on the sides of the case: the on/off switch, hole for the reset button to be accessible, and circular hole opposite that for the audio jack. The two spaces on the top of the case are for the Nextion display and the speaker.

The holes on the side, which are mirrored on the opposite side as well, are used to hold the straps. The straps, which are lightweight, flexible, and strong, can be buckled around the users right leg. Given the length of the case and the intent to hold the device tight to the leg in order to more accurately track the movement of the user, we decided to use two straps rather than one. This made for a sturdier attachment, but a slightly bulkier device.

3.10 Interfaces

Our interfaces between subsystems are best described in the context of our microcontroller.

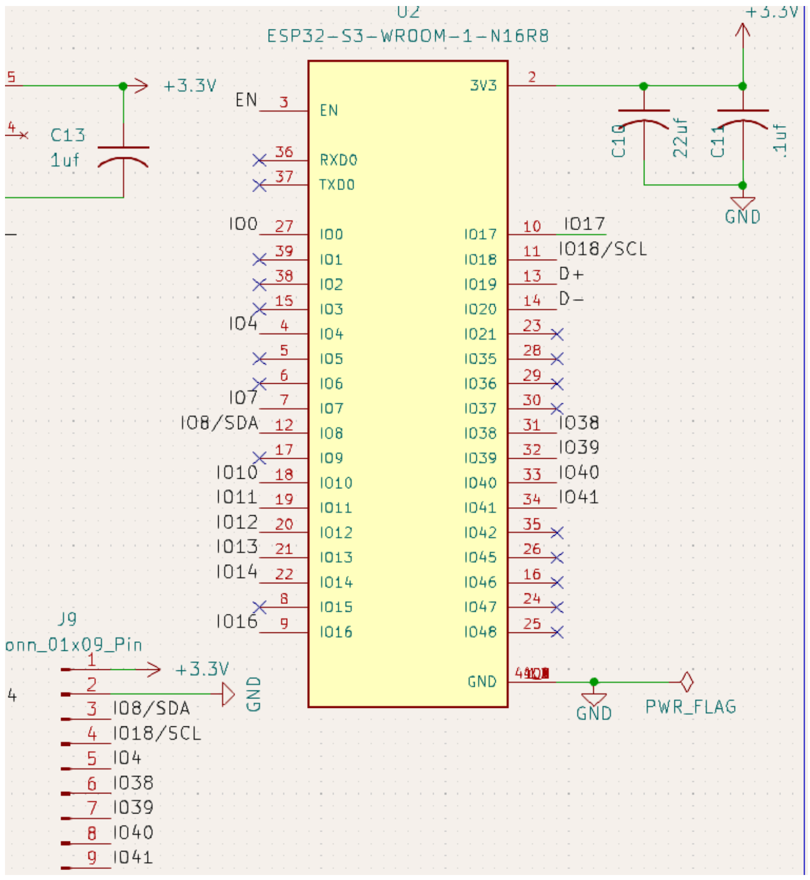


Figure 16: ESP32 Interface Connections

The Nextion display (user interface subsystem) interacts with the program and the rest of the subsystems via serial Rx/Tx communication. This communication takes place using GPIO pins 16 and 17. The BNO055 (gait tracking subsystem) utilizes I2C communication protocol via GPIO pins 8 and 18. Our music processing subsystem uses I2S communication to transmit music from the SD card to the microcontroller and then to the music amplification and stretching MAX98357. The MAX uses GPIO pins 4, 40, and 41. Our SD card uses GPIO pins 10-13 to transmit the music to the ESP32, operating in the I2S protocol.

4 System Integration Testing

4.1 Describe how the integrated set of subsystems was tested.

Our subsystems were all individually tested and debugged using the following breakout boards:

BNO055 Gait tracking & analysis: https://www.adafruit.com/product/2472

MAX98537 Amplifier Music Processing: https://www.adafruit.com/product/3006

MicroSD card breakout:
https://www.digikey.com/en/products/detail/adafruit-industries-llc/254/5761230

To integrate our subsystems, we used all three breakout boards on a single breadboard, powered from one ESP32-S3-WROOM-1 microcontroller. The hardware was connected via dupont connectors as shown below:

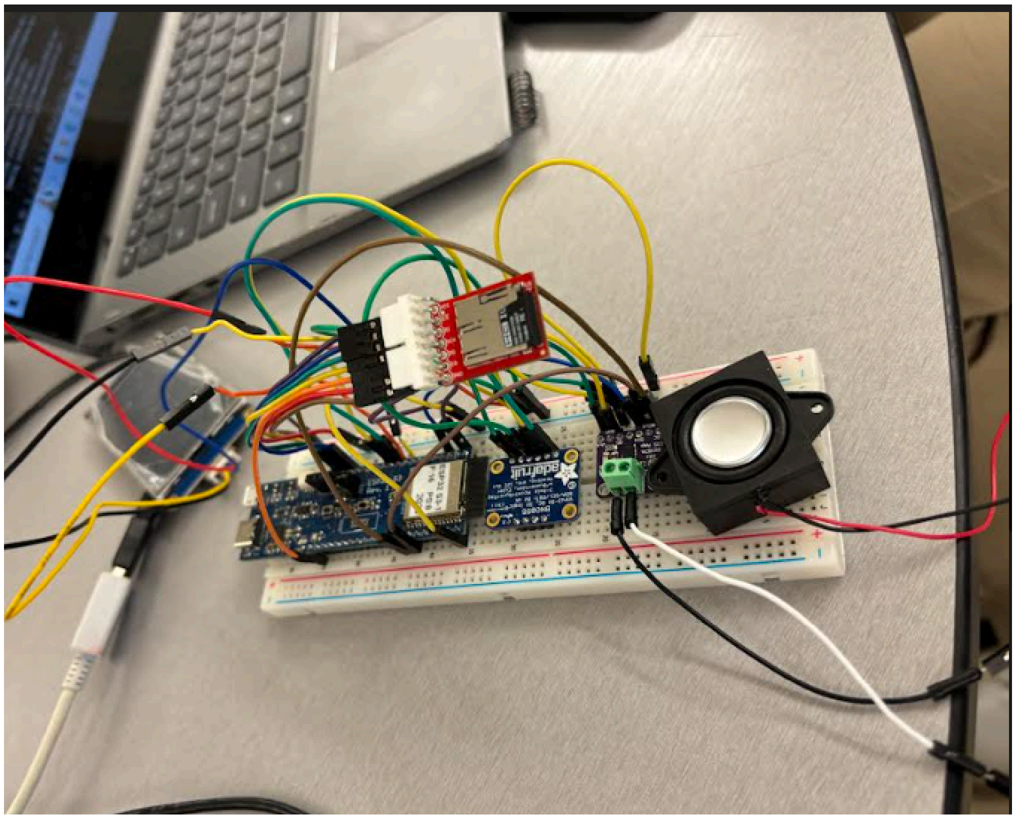


Figure 17: Integrated Subsystems

Our software, previously written as separate C files for 1) Gait analysis, 2) Music Processing & Output and 3) Nextion User Interface were combined into one C program (with all associated libraries and dependencies combined into one workspace). This was accomplished by using a real-time operating system built into VSCode, FreeRTOS. The Gait Analysis program was given task priority of 1, so that if there is a conflict for memory on the microcontroller, reading the user’s steps will take highest priority. Music processing was assigned priority 2 and placed on the second core of our ESP32, due to the processing power it required. Our user interface subsystem was given lowest priority in RTOS, and it was placed on the first core. By integrating our hardware & software in this way, we were able to accurately test the functionality of our subsystems working together in the following ways:

Subsystem 1: Gait Tracking: Steps were displayed on the serial monitor each time the program determined that one had occurred.

Subsystem 2: Gait Analysis: A rolling average of the steps per minute of the previous 10 steps was displayed to the serial monitor. Gait analysis & tracking were both rigorously tested with respect to manual timing prior to the systems integration and required little debugging.

Subsystem 3: User Interface: Button presses were tested via a line of acknowledgement on our serial monitor. In addition, our entire program required the successful registration of the target SPM, genre, and start buttons for the music to start playing.

Subsystem 4: Music Storage: Ensuring that our program played the audio file as stored on our SD Card.

Subsystem 5: Music Processing: We physically simulated steps being taken to test if the music was stretched or compressed in the way our music processing algorithm was designed to, listening for the variations in real time.

Subsystem 6: Music Output: Through our speaker for ease of operation.

Subsystem 7: Power and Regulation: Power and regulation, as specified for our final board design, were not able to be physically tested at this stage. Instead, we confirmed our schematic design with Professor Schafer.

4.2 Show how the testing demonstrates that the overall system meets the design requirements

Following this initial systems integration test, we finalized our schematic design and PCB design and ordered our board from JLCPCB. The first board was used to test the functionality of our subsystems. We found that our power path needed modification from buck-boost converters to a regulated power system using the AP2112 3.3V

Regulator. Upon receiving our final board, ordered following testing on our first board, we conducted our final systems integration tests. We found that it met the following overall design requirements in the following ways:

1. Power & Programming - our board powered & programmed via the USB port. A power LED was used to confirm this. Upon programming, we tested the functionality of our subsystems as described further below. However, following this test, our board was tested with our 3.7 V LiPo battery. The board powered, but the program did not run. We deduced that the 3.7 V LiPo battery was not providing enough current to power all of our subsystems, especially the backlight of our user interface and our speaker. We ordered a 4.8 V battery and found that this provided enough power to operate our subsystems somewhat effectively.

2. Gait Tracking & Analysis Functionality - our onboard BNO055 was found to be dysfunctional due to soldering & connectivity issues. We proceeded with our testing by attaching the adafruit BNO055 breakout board to our PCB via breakout pins for power, ground, and I2C pins. The subsystem was demonstrated to work property with the breakout board - steps were read and SPM was determined accurately and consistent with prior testing.

3. Music Processing & Output functionality - 16 different songs were stored in our onboard SD card, each accessed via a different combination of SPM and genre selections on our user interface. Music slowed or sped up to the degree we had designed, and the pointer in our C file jumped to the correct spot in the song with each change, to a reasonable degree of accuracy.

4. Battery Recharging - our recharging circuit was designed for a 3.7 V LiPo battery. Since we did not end up being able to use this battery, this functionality was not met on our PCB. Our 4.8 V Nickel Cadmium battery cannot be charged by the specific part we chose. For this iteration of our prototype, it was charged via an

external battery charger. In future, the battery charging circuit chosen will be able to support a battery of the type (and voltage) that we choose.

5. Wearability - Our device, although slightly bulkier than intended, is easily connected to the user and does not interfere with one's gate. The casing houses all the hardware components, and has several holes for heat dissipation. Lastly, the device was tested at slow and fast walking paces, displaying durability.

Overall, our systems integrated together quite nicely, thanks to efficient planning of PCB design and the use of FreeRTOS to maximize processing power allocation. Our final product works effectively using both 5V USB power from a laptop or smartphone, or our 4.8 V Nickel Cadmium battery.

5 Users Manual/Installation manual

5.1 How to install your product

No installation is required, as our product is a fully pre-packaged and enclosed wearable device. All that is required of the user is a USB-C power source for battery-recharging functionality.

5.2 How to setup your product

Setup for our product is very simple. First, the user selects their desired steps-per-minute from the touchscreen on the front panel of the device. They then choose their desired genre of music, and adjust the intended volume prior to starting the program. Next, strap the device to the user’s right thigh using the two buckle straps already installed, press the START button, and begin walking. The device then begins to output music and track one’s gait within its own contained apparatus. The user does not need to pay attention to the device unless seeking to adjust the volume or stop the program.

5.3 How the user can tell if the product is working

During operation, one can ensure that the product is working properly if two simple criteria are met: 1) Music should be playing from the speaker at all times, once the START button has been pressed. 2) Walking at a tempo varying from the inputted desired SPM should cause the music to audibly change in pace, with the playback sounding either faster or slower than the original song. If the music is varying, then the device is working.

5.4 How the user can troubleshoot the product

From the outside of our enclosed product, there is almost no way to troubleshoot our device. However, two methods that we have found useful when the device fails to function right away upon powering are 1) to trying powering the device on and off via the power switch on the side of the case and 2) to press and hold the reset button (accessible from the side of the case near the USB input port) for approximately 1 second.

6 To-Market Design Changes

The three major to-market design changes that we would like to implement are: Bluetooth connectivity, the ability to upload and/or stream a custom music library, and improvements upon the comfort and aesthetic of our device.

The addition of bluetooth connectivity would allow our users to connect their personal wireless headphones or earbuds to the device. Wireless headphones have become almost ubiquitous due to their ease of use, and we would eventually like to offer a product that is compatible with the style of listening that our audience prefers. We would accomplish this by utilizing the Bluetooth capabilities of the ESP32-S3 microcontroller to directly connect to wireless headphones, as well as connect to the user’s smartphone if desired.

In this day and age, the ability to upload or stream custom music hinges on smartphone connectivity, so we would emphasize getting our Bluetooth operational as quickly and efficiently as possible. We would like to allow our users to choose songs directly from their downloaded files (mp3, itunes library, etc.) or from streaming platforms such as Spotify or Apple Music. These songs could either be pre-processed as they are uploaded (through an algorithm designed to do just that) or, preferably, processed in real time. For this prototype, we were unable to process our songs in real time, but if given more time and resources to perfect our device before releasing it to the market, we are confident that we could integrate this capability. In this way, a user could choose any song from their streaming service of choice and instantaneously hear it stretch/condense in pace corresponding to their gait.

Lastly, a market necessity is making the device more easily and comfortably wearable. In order to get consumers to migrate to a product, the product must be convenient. Our prototype design, although effective, is slightly too big and bulky. It currently sits at 5.65x3x1.3 in^3 which is larger than the size of one's phone: the biggest thing we carry in our pockets. Therefore, it is a slight burden to wear the device. This was indicated by the fact that some people who tried on the device were worried about it slipping down their leg or and tugging on their clothes. We would first look at shrinking our board and getting rid of the speaker in order to alleviate the size requirements that currently exist. Next, we would shrink the external wiring and limit the empty space inside the casing. Finally, we would look at altering the attachment mechanism: either shrinking the straps or changing the mechanism to a clip. Ideally, changing these components would make the hardware much smaller, leaving the determinant to be the size of the screen.

7 Conclusions

Throughout this project, it was our hope to create a working prototype that could harness the rewarding impacts of music therapy on people with motor dysfunction. The passion that the four of us had for the tangible, humanitarian good that such a device could bring about motivated us through difficult stretches of our design process. Overall, we were pleased with how our working prototype turned out. And, despite facing hardware challenges with our gait tracking subsystem, we were able to pivot and present a functional product on demo day.

We believe that this prototype is only a starting point for where this technology could eventually lead. With more efficient PCB design, higher quality parts (from a larger budget) and offering real-time audio processing, many technical aspects of our design could be optimized. In addition, our to-market improvements as described above would change our basic, functional prototype into a product that could be more easily marketed. Still, given the context and parameters of our senior design process, we are pleased to have been able to construct a functional, working prototype. When demonstrating our device to faculty and friends, we all felt a sense of pride in what we have accomplished. We sincerely hope that our prototype, and subsequent documentation, is used as a starting point for future improvements.

8 Appendices

Complete Hardware Documentation

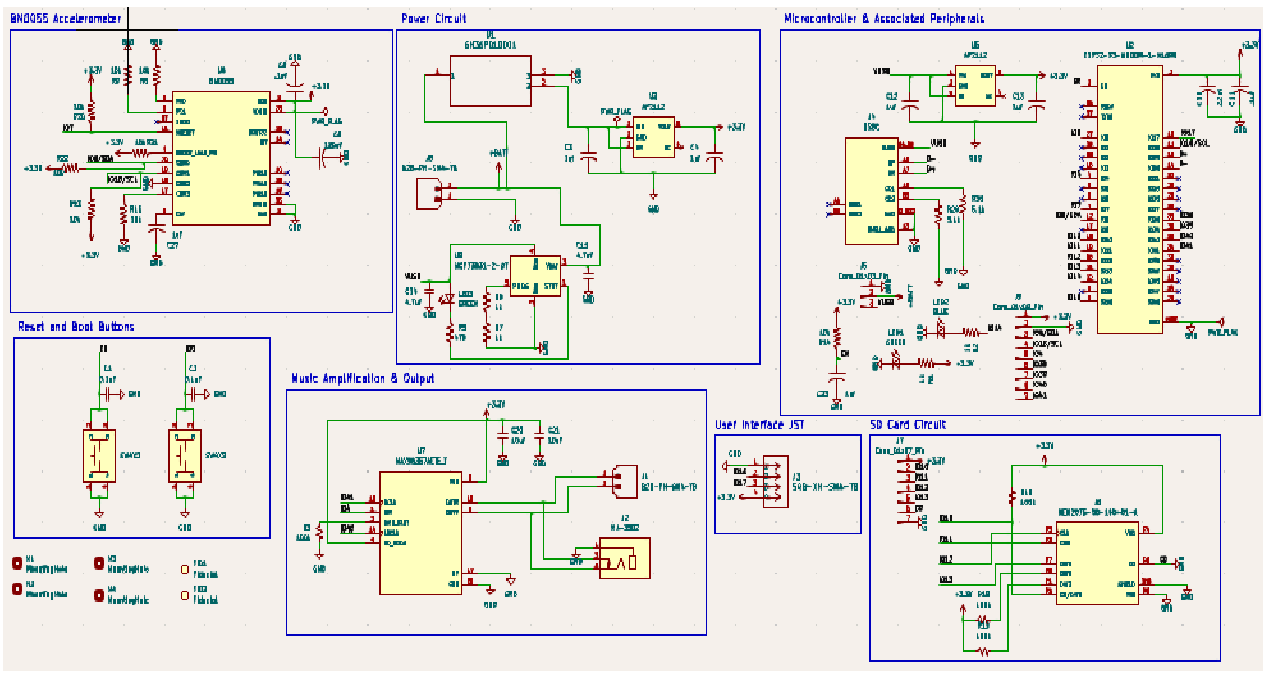


Figure 18: Complete Hardware Schematic Design

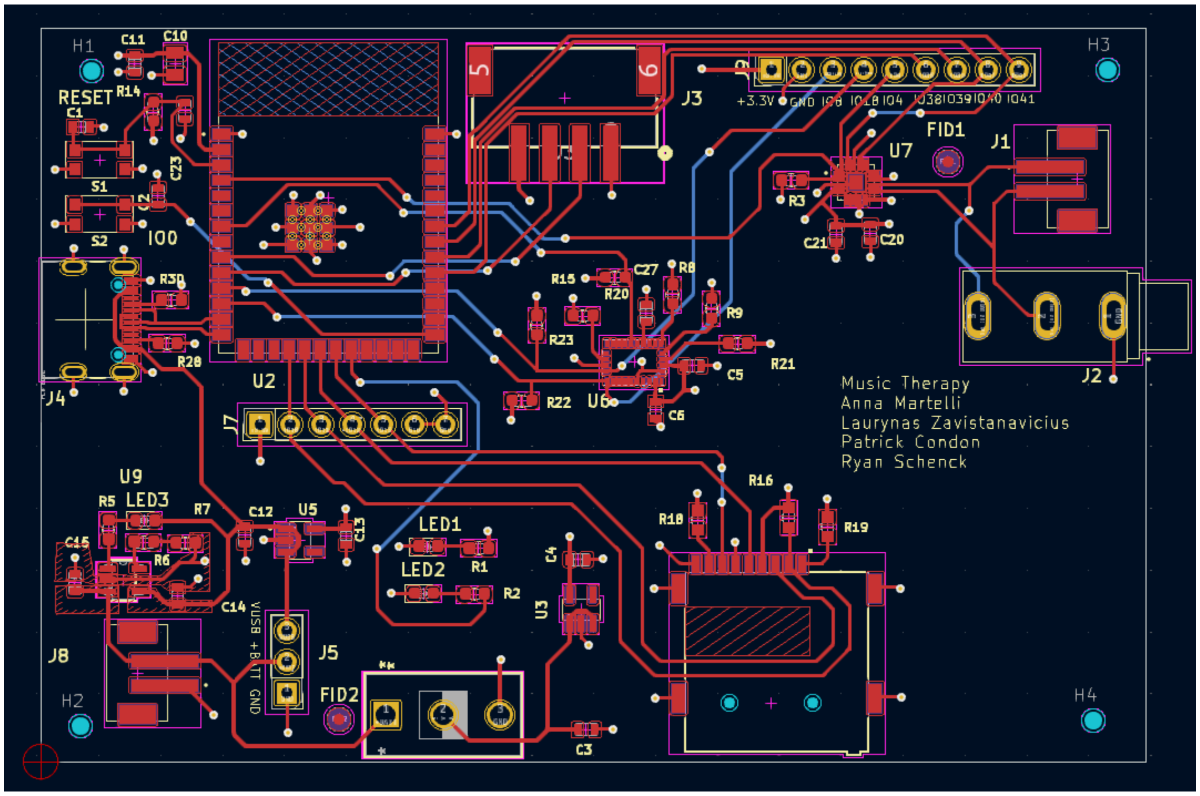


Figure 19: Complete Hardware PCB Design

Datasheet Listings

- BNO055 Accelerometer: https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bno055-ds000.pdf

- ESP32-S3-WROOM-1 N16R8: https://www.espressif.com/sites/default/files/documentation/esp32-s3-wroom-1_wroom-1u_datasheet_en.pdf

- Nextion Basic 3.2'" touchscreen display: https://nextion.tech/datasheets/nx4024t032/

- MAX98357AETE_T Amplifier: https://www.analog.com/en/products/max98357a.html

- SD Card: https://gct.co/files/specs/mem2075-spec.pdf

- Battery Charger: https://ww1.microchip.com/downloads/aemDocuments/documents/APID/ProductDocuments/DataSheets/MCP73831-Family-Data-Sheet-DS20001984H.pdf

Software Listings

BNO055 Github libraries: https://github.com/adafruit/Adafruit_BNO055
The .cpp and .h files should be added to the libraries path in VSCode to ensure functionality of the BNO055.

Nextion Files
Basic instructions to get started:
https://nextion.ca/portfolio-items/nextion-iteadlib-and-esp32-step-by-step/
Libraries: https://github.com/itead/ITEADLIB_Arduino_Nextion

Music Processing Files:
SD library (part of ESP32 standard install):
https://github.com/espressif/arduino-esp32/blob/master/libraries/SD/README.md

Time Stretch Algorithm MATLAB and python:
https://www.audiolabs-erlangen.de/resources/MIR/TSMtoolbox/
https://github.com/meinardmueller/libtsm/tree/master

Final C Program:

```
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BNO055.h>
#include <utility/imumaths.h>
#include <Arduino.h>
#include <SPI.h>
#include <math.h>

#include <Nextion.h>
#include <stdio.h>
#include <stdlib.h>
#include <driver/i2s.h>
#include <SD.h>
```

```
#include <esp_task_wdt.h>

//  Global Variables/objects
static const i2s_port_t i2s_num = I2S_NUM_0; // i2s port number
File WavFile; // File pointer to current music filebeing played
float speedFactorcurrent = 1; // indicates the current ratio between song's
original bpm and gait bpm
float speedFactorprev = 0; // used to determine whether there has been a change
in speedFactor
uint32_t BytesReadSoFarMusicTime = 0; // keeps track of musical progress for
switching todifferent speed versions of song
String gnrstr; // rck, pop, rap, or lat
String targetSPMstr; // 060, 075, 090, or 105
String currentSpeedstr; // 080, 085, 090, 095, 100, 110, or 120
String WavFileName; // Object for root of SD card directory, concatenation of
targetSPMStr, gnrstr, and currentSpeedstr
int volume = 50; // volume on a scale of 0 to 100

boolean start = false;
float current_SPM;
float target_SPM;
int genre;
//--------------------------------------------------------------------------------
----------------------------------------

//    SD Card
          #define SD_CS         10         // SD Card chip select

//    I2S
          #define I2S_DOUT      4          // i2S Data out oin
          #define I2S_BCLK      41         // Bit clock
          #define I2S_LRC       40         // Left/Right clock, also known as
Frame clock or word select
          #define I2S_NUM       0          // i2s port number

// Wav File reading
          #define NUM_BYTES_TO_READ_FROM_FILE 1024    // How many bytes to read
from wav file at a time
```

```
//--------------------------------------------------------------------------------
-----------------------------------------

//--------------------------------------------------------------------------------
-----------------------------------------
// I2S configuration structures

static const i2s_config_t i2s_config = {
    .mode = (i2s_mode_t)(I2S_MODE_MASTER | I2S_MODE_TX),
    .sample_rate = 44100,
    .bits_per_sample = I2S_BITS_PER_SAMPLE_16BIT,
    .channel_format = I2S_CHANNEL_FMT_RIGHT_LEFT, // use stereo wav files
    .communication_format = (i2s_comm_format_t)(I2S_COMM_FORMAT_I2S |
I2S_COMM_FORMAT_I2S_MSB),
    .intr_alloc_flags = ESP_INTR_FLAG_LEVEL1,       // high interrupt priority
    .dma_buf_count = 8,                             // 8 buffers
    .dma_buf_len = 1024,                            // 1K per buffer, so 8K of
buffer space
    .use_apll=0,
    .tx_desc_auto_clear= true,
    .fixed_mclk=-1
};

// These are the physical wiring connections to our I2S decoder board/chip from
the esp32, there are other connections
// required for the chips mentioned at the top (but not to the ESP32), please
visit the page mentioned at the top for
// further information regarding these other connections.

static const i2s_pin_config_t pin_config =
      {
          .bck_io_num = I2S_BCLK,                           // The bit clock
connectiom, goes to pin 27 of ESP32
          .ws_io_num = I2S_LRC,                             // Word select, also
known as word select or left right clock
          .data_out_num = I2S_DOUT,                         // Data out from the
ESP32, connect to DIN on 38357A
          .data_in_num = I2S_PIN_NO_CHANGE                  // we are not
interested in I2S data into the ESP32
```

```
      };

struct WavHeader_Struct
      {
          //   RIFF Section
          char RIFFSectionID[4];      // Letters "RIFF"
          uint32_t Size;              // Size of entire file less 8
          char RiffFormat[4];         // Letters "WAVE"

          //   Format Section
          char FormatSectionID[4];    // letters "fmt"
          uint32_t FormatSize;        // Size of format section less 8
          uint16_t FormatID;          // 1=uncompressed PCM
          uint16_t NumChannels;       // 1=mono,2=stereo
          uint32_t SampleRate;        // 44100, 16000, 8000 etc.
          uint32_t ByteRate;          // =SampleRate * Channels *
(BitsPerSample/8)
          uint16_t BlockAlign;        // =Channels * (BitsPerSample/8)
          uint16_t BitsPerSample;     // 8,16,24 or 32

          // Data Section
          char DataSectionID[4];      // The letters "data"
          uint32_t DataSize;          // Size of the data that follows
      }WavHeader;



//-------------------------------------------------------------------------------
-----------------------------------------

// A boolean test to validate the recieved wavfile data
bool ValidWavData(WavHeader_Struct* Wav)
{

  if(memcmp(Wav->RIFFSectionID,"RIFF",4)!=0)
  {
    // Serial.print("Invalid data - Not RIFF format");
    return false;
  }
```

```
if(memcmp(Wav->RiffFormat,"WAVE",4)!=0)
{
  // Serial.print("Invalid data - Not Wave file");
  return false;
}
if(memcmp(Wav->FormatSectionID,"fmt",3)!=0)
{
  // Serial.print("Invalid data - No format section found");
  return false;
}
if(memcmp(Wav->DataSectionID,"data",4)!=0)
{
  // Serial.print("Invalid data - data section not found");
  return false;
}
if(Wav->FormatID!=1)
{
  // Serial.print("Invalid data - format Id must be 1");
  return false;
}
if(Wav->FormatSize!=16)
{
  // Serial.print("Invalid data - format section size must be 16.");
  return false;
}
if((Wav->NumChannels!=1)&(Wav->NumChannels!=2))
{
  // Serial.print("Invalid data - only mono or stereo permitted.");
  return false;
}
if(Wav->SampleRate>48000)
{
  // Serial.print("Invalid data - Sample rate cannot be greater than 48000");
  return false;
}
if((Wav->BitsPerSample!=8)& (Wav->BitsPerSample!=16))
{
  // Serial.print("Invalid data - Only 8 or 16 bits per sample permitted.");
  return false;
  }
  return true;
}

// Function that clears the memory arrays of the SD Card.
void clearSD()
{
  byte sd = 0;
  digitalWrite(SS, LOW);
  while (sd != 255)
  {
    sd = SPI.transfer(255);
    // Serial.print("sd=");
    // Serial.println(sd);
  }
  digitalWrite(SS, HIGH);
}

void SDCardInit()
{
  SPI.begin();
  delay(10);
  boolean b;
  // *** SD Card **
  b = SD.begin(SS);
  if (!b)
  {
    delay(100);
    clearSD();
    delay(100);
    b = SD.begin(SS);
  }
  if (b)
    ;
    // Serial.println("SD Card started.");
  else
    // Serial.println("SD Card failed to start!");
    // Serial.println();
    pinMode(SD_CS, OUTPUT);
    digitalWrite(SD_CS, LOW);
    delay(100);
    digitalWrite(SD_CS, HIGH); // SD card chips select
    if(!SD.begin(SD_CS))
    {
        // Serial.println("Error talking to SD card!");
        while(true);                  // end program
    }

}


bool FillI2SBuffer(byte* Samples,uint16_t BytesInBuffer)
{
    // Writes bytes to buffer, returns true if all bytes sent else false, keeps
track itself of how many left
    // to write, so just keep calling this routine until returns true to know
they've all been written, then
    // you can re-fill the buffer

    size_t BytesWritten;                        // Returned by the I2S write
routine,
    static uint16_t BufferIdx=0;                // Current pos of buffer to
output next
    uint8_t* DataPtr;                           // Point to next data to send to
I2S
    uint16_t BytesToSend;                       // Number of bytes to send to I2S

    // To make the code eaier to understand I'm using to variables to some
calculations, normally I'd write this calcs
    // directly into the line of code where they belong, but this make it easier
to understand what's happening

    DataPtr=Samples+BufferIdx;                               // Set address to
next byte in buffer to send out
    BytesToSend=BytesInBuffer-BufferIdx;                     // This is amount to
send (total less what we've already sent)
```

```
    i2s_write(i2s_num,DataPtr,BytesToSend,&BytesWritten,1);  // Send the bytes,
wait 1 RTOS tick to complete
    BufferIdx+=BytesWritten;                                 // increasue by
number of bytes actually written

    if(BufferIdx>=BytesInBuffer)
    {
      // sent out all bytes in buffer, reset and return true to indicate this
      BufferIdx=0;
      return true;
    }
    else
      return false;       // Still more data to send to I2S so return false to
indicate this
}

uint16_t ReadFile(byte* Samples)
{
    static uint32_t BytesReadSoFar=0;                   // Number of bytes read
from file so far
    uint16_t BytesToRead;                               // Number of bytes to
read from the file
    uint16_t i;                                         // loop counter
    int16_t SignedSample;                               // Single Signed Sample

    if(BytesReadSoFar+NUM_BYTES_TO_READ_FROM_FILE>WavHeader.DataSize)   // If
next read will go past the end then adjust the
      BytesToRead=WavHeader.DataSize-BytesReadSoFar;                    // amount
to read to whatever is remaining to read
    else
      BytesToRead=NUM_BYTES_TO_READ_FROM_FILE;                          //
Default to max to read

    WavFile.read(Samples,BytesToRead);                  // Read in the bytes from
the file
    BytesReadSoFar+=BytesToRead;
    // Update the total bytes read in so far
    // NEW
    BytesReadSoFarMusicTime+=(uint32_t)( (float)BytesToRead /
speedFactorcurrent);

    if(BytesReadSoFar>=WavHeader.DataSize)              // Have we read in all
the data?
    {
      WavFile.seek(44);                                 // Reset to start of wav
data
      BytesReadSoFar=0;                                 // Clear to no bytes read
in so far
      BytesReadSoFarMusicTime = 0;
    }
    // Change Volume
    for(i=0;i<BytesToRead;i+=2){
        *((int16_t *)(Samples+i))=(*((int16_t *)(Samples+i)))*volume/100;
    }
    //}

    return BytesToRead;                                 // return the number of
bytes read into buffer
}

// switch WavFileName str based on the current speed factor
void switchWavFileName(){
  if(speedFactorcurrent > 1.15){
    currentSpeedstr = String("120");
  }
  else if(speedFactorcurrent > 1.05){
    currentSpeedstr = String("110");
  }
  else if(speedFactorcurrent > 0.95){
    currentSpeedstr = String("100");
  }
  else if(speedFactorcurrent > 0.85){
    currentSpeedstr = String("090");
  }
  else{
    currentSpeedstr = String("080");
  }
  WavFileName = "/" + targetSPMstr + gnrstr + currentSpeedstr + ".wav";
}

void PlayWav()
{
  // Serial.println("Supposed to be playing");
  static bool ReadingFile=true;                       // True if reading file
from SD. false if filling I2S buffer
  static byte Samples[NUM_BYTES_TO_READ_FROM_FILE];   // Memory allocated to
store the data read in from the wav file
  static uint16_t BytesRead;                          // Num bytes actually read
from the wav file which will either be
                                                      //
NUM_BYTES_TO_READ_FROM_FILE or less than this if we are very
                                                      // near the end of the
file. i.e. we can't read beyond the file.
  // New functions
  if (speedFactorcurrent == speedFactorprev){ //if there is no change in
speedFactor, no change needed
  }else{ // otherwise, need to change the wavfile from which music is played and
move the f

    //Serial.print(stretchF
    float percentDuration = ( (float)WavFile.position() /
((float)WavHeader.DataSize - 44) ) * (float)100;
    // Serial.print("Percent duration of song before jump: ");
    // Serial.println(percentDuration);

    WavFile.close();
    switchWavFileName();
    WavFile = SD.open(WavFileName);

      if(WavFile==false){
        // Serial.print("Could not open ");
        // Serial.println(WavFileName);
       }
    WavFile.read((byte *) &WavHeader,44);
    //DumpWAVHeader(&WavHeader);                 // DELETE? ****
    if(ValidWavData(&WavHeader))  {
      i2s_set_sample_rates(i2s_num, WavHeader.SampleRate);
    }

    uint32_t jump_location = (( ((uint32_t)( (float)BytesReadSoFarMusicTime /
speedFactorcurrent)) /2) * 2 );

    if (jump_location >= WavHeader.DataSize){
        jump_location = 44;
    }
    // make FILE pointer jump to the corresponding spot in the music
    WavFile.seek(jump_location);

    percentDuration = ( (float)WavFile.position() / ((float)WavHeader.DataSize -
44) ) * 100;
    // Serial.print("Percent duration of song after jump: ");
    // Serial.println(percentDuration);

    // Serial.print("WavFile.position() = ");
    // Serial.println(WavFile.position());
    // Serial.println(WavFileName);
    // Serial.println("***************** end of music data ****************\n");
    speedFactorprev = speedFactorcurrent;
  }

  if(ReadingFile)                                      // Read next chunk of data
in from file if needed
  {
    BytesRead=ReadFile(Samples);                       // Read data into our
memory buffer, return num bytes read in
    ReadingFile=false;                                 // Switch to sending the
buffer to the I2S
  }
  else
    ReadingFile=FillI2SBuffer(Samples,BytesRead);         // We keep calling this
routine until it returns true, at which point
                                                       // this will swap us back
to Reading the next block of data from the file.
```

```
                                                      // Reading true means it
has managed to push all the data to the I2S
                                                      // Handler, false means
there still more to do and you should call this
                                                      // routine again and again
until it returns true.
}

float speedFactorlist[7] = {0.8, 0.9, 1, 1.1, 1.2};
float closestValue(float x, float *list, int size) {
    if (size == 0) {
        printf("Error: Empty list.\n");
        exit(1);
    }

    float closest = list[0]; // Initialize closest to the first value in the list
    float min_diff = fabs(x - list[0]); // Initialize min_diff to the absolute
difference between x and the first value

    for (int i = 1; i < size; i++) {
        float diff = fabs(x - list[i]); // Calculate the absolute difference
between x and the current value in the list
        if (diff < min_diff) { // If the current difference is smaller than the
minimum difference found so far
            closest = list[i]; // Update closest to the current value
            min_diff = diff; // Update min_diff to the current difference
        }
    }

    return closest; // Return the closest value
}


/*********************************************************************************
**************************************************************/

// Gait tracking parameters
uint16_t BNO055_SAMPLERATE_DELAY_MS = 100;
Adafruit_BNO055 bno = Adafruit_BNO055(55, 0x28, &Wire);
```

```
int steps = 0;
float t = 0;
float times_array[10];
int array_index = 0;
float time_sum = 0;
float time_avg = 0;
float time_sum_mins = 0;
float SPM = 0;

// Here we initilize all Nextion Parameters
// Format is (page number, id, name)

// page0
NexPage p0 = NexPage(0,0,"page0");
NexText p0_t2 = NexText(0,5,"t0");
NexButton p0_b0 = NexButton(0,1,"b0");
NexButton p0_b1 = NexButton(0,2,"b1");
NexButton p0_b2 = NexButton(0,3,"b2");
NexButton p0_b3 = NexButton(0,4,"b3");

// page1
NexPage p1 = NexPage(1,0,"page1");
NexText p1_t2 = NexText(1,5,"t0");
NexButton p1_b4 = NexButton(1,1,"b4");
NexButton p1_b5 = NexButton(1,2,"b5");
NexButton p1_b6 = NexButton(1,3,"b7");
NexButton p1_b7 = NexButton(1,4,"b8");

//page2
NexPage p2 = NexPage(2,0,"page2");
NexText p2_t0 = NexText(2,4,"t0");
NexNumber p2_n1 = NexNumber(2,5,"n0");
NexButton p2_b8 = NexButton(2,3,"b8");
NexButton p2_b9 = NexButton(2,2,"b9");
NexButton p2_b2 = NexButton(2,1,"b2");



//page3
NexPage p3 = NexPage(3,0,"page3");
```

```
NexText p3_t0 = NexText(3,3,"t0");
NexNumber p3_n1 = NexNumber(3,5,"n1");
NexButton p3_b10 = NexButton(3,1,"b10");
NexButton p3_b11 = NexButton(3,2,"b11");
NexButton p3_b2 = NexButton(3,4,"b2");

// Initialize nextion buttons to "listen"
// Allows them to send serial data
NexTouch *nex_listen_list[] = {
  &p0_b0, &p0_b1, &p0_b2, &p0_b3, &p1_b4, &p1_b5, &p1_b6, &p1_b7,
  &p2_b8, &p2_b9, &p2_b2, &p3_b10, &p3_b11, &p3_b2, NULL
};

// Denote what each button does when pressed

// 60 SPM button
void p0_b0_Press(void *ptr) {
  target_SPM = 60;
  targetSPMstr = String("060"); // begin music file selection
  Serial.print("Target SPM: ");
  Serial.println(target_SPM);
}

// 75 SPM button
void p0_b1_Press(void *ptr) {
  target_SPM = 75;
  targetSPMstr = String("075"); // begin music file selection
  //WavFile = SD.open("/cher2.wav");
  Serial.print("Target SPM: ");
  Serial.println(target_SPM);
}

// 90 SPM button
void p0_b2_Press(void *ptr) {
  target_SPM = 90;
  targetSPMstr = String("090"); // begin music file selection
  Serial.print("Target SPM: ");
  Serial.println(target_SPM);
}
```

```
// 105 SPM button
void p0_b3_Press(void *ptr) {
  target_SPM = 105;
  targetSPMstr = String("105"); // begin music file selection
  Serial.print("Target SPM: ");
  Serial.println(target_SPM);
}

// Rock button
void p1_b4_Press(void *ptr) {
  genre = 0; // Rock
  gnrstr = String("rck");
  WavFileName = "/" + targetSPMstr + gnrstr + "100.wav"; // complete music file
selection
  Serial.print("Rock Genre. ID: ");
  Serial.println(genre);
}

// Latin button
void p1_b5_Press(void *ptr) {
  genre = 1; // Latin
  gnrstr = String("lat");
  WavFileName = "/" + targetSPMstr + gnrstr + "100.wav"; // complete music file
selection
  Serial.print("Latin Genre. ID: ");
  Serial.println(genre);
}

// Pop button
void p1_b6_Press(void *ptr) {
  genre = 2; // Pop
  gnrstr = String("pop");
  WavFileName = "/" + targetSPMstr + gnrstr + "100.wav"; // complete music file
selection
  Serial.print("Pop Genre. ID: ");
  Serial.println(genre);
}
```

```
// Rap button
void p1_b7_Press(void *ptr) {
  genre = 3; // Rap
  gnrstr = String("rap");
  WavFileName = "/" + targetSPMstr + gnrstr + "100.wav"; // complete music file
selection
  Serial.print("Rap Genre. ID: ");
  Serial.println(genre);
}

// Volume up button
void p2_b8_Press(void *ptr) {
  volume+=10;
  if(volume>100){
    volume = 100;
  }
  Serial.print("Volume is: ");
  Serial.println(volume);
}

// Volume down button
void p2_b9_Press(void *ptr) {
  volume-=10;
  if(volume<0){
    volume = 0;
  }
  Serial.print("Volume is: ");
  Serial.println(volume);
}

// Start button
void p2_b2_Press(void *ptr) {
  start = true; // this starts tasks 1 (gait tracking) and 3 (music processing)
in RTOS
  // Serial.println("Start is now true");
  // Serial.println(start);
  speedFactorprev = 0; // this forces the music task to reset the song
  Serial.println(WavFileName);
}
```

```
// Volume up button (on stop page)
void p3_b10_Press(void *ptr) {
  volume+=10;
  if(volume>100){
    volume = 100;
  }
  Serial.print("Volume is: ");
  Serial.println(volume);
}

// Volume down button (on stop page)
void p3_b11_Press(void *ptr) {
  volume-=10;
  if(volume<0){
    volume = 0;
  }
  Serial.print("Volume is: ");
  Serial.println(volume);
}

// stop button
void p3_b2_Press(void *ptr) {
  start = false; // this stops tasks 1 (gait tracking) and 3 (music processing)
in RTOS
  WavFile.close(); // close the wavefile
  // Serial.println("Stopped");
  volume = 50; // reset the volume
  // ESP.restart();
}

// Tasks - 1:BNO055 2:Nextion 3:Music processing
void task1( void * parameters){
  esp_task_wdt_init(3000, false);
  for (;;){
    // Serial will not be called until start is pressed - give ample time for
this to occur
    Serial.setTimeout(1000000);
    // Start the program when start button is pressed
    if(start == 1){
      // Initialize variables
      float a_mag = 0; // magnitude of acceleration
      float euler_current = 0; // current euler angle
      float euler_last = 0; // euler angle of the previous reading

      /* Get a new sensor event */
      sensors_event_t orientationData , angVelocityData , linearAccelData,
magnetometerData, accelerometerData, gravityData;
      imu::Vector<3> euler =
      bno.getVector(Adafruit_BNO055::VECTOR_EULER);
      bno.getEvent(&linearAccelData, Adafruit_BNO055::VECTOR_LINEARACCEL);

      // Access linear acceleration values
      float x = linearAccelData.acceleration.x;
      float y = linearAccelData.acceleration.y;
      float z = linearAccelData.acceleration.z;

      euler_current = euler.y(); // determine angle of rotation
      a_mag = pow((pow(x,2) + pow(y, 2) + pow(z,2)), 0.5); // determine magnitude
of linear acceleration

      // conditionally determine if a step has been taken using linear
acceleration and angle of rotation over the past two steps
      if ((a_mag>1) && (euler_current > 10) && euler_current > euler_last)
      {
        // STEP TAKEN - record time at which step was taken & increment step
counter
        steps = steps + 1;
        Serial.print("Step taken: ");
        Serial.print(t);
        Serial.println(" seconds since last step.");
        times_array[array_index] = t;
        t = .7;
        int j;
        vTaskDelay(700/portTICK_PERIOD_MS);  // delay task1 by 0.7 seconds
        if (array_index == 9){ //9
          for (j = 0; j < 10; j++) //10
          {
            time_sum = time_sum + times_array[j];
            time_sum_mins = time_sum/60;
            time_avg = time_sum/20; // 20
            array_index = 0;
            SPM = 20/time_sum_mins; // 20
          }
          Serial.println("\n**************** start of bno data
****************");
          Serial.print("Average step length for the past 20 steps is: "); // 20
          Serial.println(time_avg);
          Serial.print("Total Steps: ");
          Serial.println(steps * 2);
          time_sum = 0;
          time_avg = 0;
          current_SPM = SPM;
          Serial.print("Current SPM:");
          Serial.println(current_SPM);
          Serial.println("**************** End of bno data
****************\n");

          //new
          //stretchFactorprev = stretchFactorcurrent;
          speedFactorcurrent = closestValue( target_SPM/current_SPM,
speedFactorlist, 5);
          Serial.println("\n**************** start of music data
****************");
          Serial.print("speedFactorcurrent = ");
          Serial.println(speedFactorcurrent);
          //end new
        }
        else {
        array_index = array_index + 1;
        }
      }
      else {
        t = t + 0.10;
        vTaskDelay(100 / portTICK_PERIOD_MS);
      }
    euler_last = euler_current;
    }
  }
}

void task2( void * parameters){
  for (;;){
    // task 2 - nexion user interface buttons
    nexLoop(nex_listen_list);
    vTaskDelay(1000 / portTICK_PERIOD_MS);
  }
}

void task3( void * parameters){
  // task 3 - music processing
  esp_task_wdt_init(3000, false);
  for (;;){
    Serial.setTimeout(1000000);
    if(start == 1){
        PlayWav();
    }
  }
}

void setup() {
  // put your setup code here, to run once

  Wire.begin(8,18);
  Serial.begin(115200);
  //delay(1000);

  //while (!Serial) delay(10);  // wait for serial port to open!

  // Serial.println("Orientation Sensor Test"); Serial.println("");

  /* Initialise the sensor */
  if (!bno.begin())
  {
    /* There was a problem detecting the BNO055 ... check your connections */
    // Serial.print("Ooops, no BNO055 detected ... Check your wiring or I2C
ADDR!");
    while (1);
  }

  // NEXTION button setup
  nexInit();
  nexSerial.println("Started"); // Test
  p0_b0.attachPush(p0_b0_Press, &p0_b0);
  p0_b1.attachPush(p0_b1_Press, &p0_b1);
  p0_b2.attachPush(p0_b2_Press, &p0_b2);
  p0_b3.attachPush(p0_b3_Press, &p0_b3);
  p1_b4.attachPush(p1_b4_Press, &p1_b4);
  p1_b5.attachPush(p1_b5_Press, &p1_b5);
  p1_b6.attachPush(p1_b6_Press, &p1_b6);
  p1_b7.attachPush(p1_b7_Press, &p1_b7);
  p2_b8.attachPush(p2_b8_Press, &p2_b8);
  p2_b9.attachPush(p2_b9_Press, &p2_b9);
  p2_b2.attachPush(p2_b2_Press, &p2_b2);
  p3_b10.attachPush(p3_b10_Press, &p3_b10);
  p3_b11.attachPush(p3_b11_Press, &p3_b11);
  p3_b2.attachPush(p3_b2_Press, &p3_b2);

  // SD Card Setup
  SDCardInit();
  WavFile.close();
    i2s_driver_install(i2s_num, &i2s_config, 0, NULL);
    i2s_set_pin(i2s_num, &pin_config);


// INITIALIZE TASKS IN RTOS. 2 Cores used, with Core 1 used exclusively for music
processing
  xTaskCreatePinnedToCore(
    task1,
    "task 1",
    25000, // stack size
    NULL, // parameters
    1, // priority
    NULL, // handle
    0 // core
  );

  xTaskCreatePinnedToCore(
    task2,
    "task 2",
    25000, // stack size
    NULL, // parameters
    3, // priority 3
    NULL, // handle
    0 // core
  );

  xTaskCreatePinnedToCore(
    task3,
    "task 3",
    100000, // stack size
    NULL, // parameters
    2, // priority
    NULL, // handle
    1 // core
  );
  delay(1000);

}

void loop() {
  // Nothing - using RTOS
}
```

Bibliography & Works Cited

[1] S. Latif *et al.*, "Dopamine in Parkinson's disease," *Clinica Chimica Acta*, vol. 522, pp. 114–126, Nov. 2021, doi: 10.1016/j.cca.2021.08.009.
[2] M. J. M. Sotomayor, V. Arufe-Giráldez, G. Ruíz-Rico, and R. Navarro-Patón, "Music Therapy and Parkinson's Disease: A Systematic Review from 2015–2020," *International Journal of Environmental Research and Public Health/International Journal of*

Environmental Research and Public Health, vol. 18, no. 21, p. 11618, Nov. 2021, doi: 10.3390/ijerph182111618.
[3] M. H. Thaut, G. C. McIntosh, and V. Hoemberg, “Neurobiological foundations of neurologic music therapy: rhythmic entrainment and the motor system,” *Frontiers in Psychology*, vol. 5, Feb. 2015, doi: 10.3389/fpsyg.2014.01185.
[4] M. H. Thaut and M. Abiru, “Rhythmic Auditory Stimulation in Rehabilitation of Movement Disorders: A review of Current research,” *Music Perception*, vol. 27, no. 4, pp. 263–269, Apr. 2010, doi: 10.1525/mp.2010.27.4.263.
[5] “Effects of Auditory Rhythm and Music on Gait Disturbances in Parkinson’s Disease,” *Frontiers in Neurology*, vol. 6, no. 234, Nov. 2014, doi: 10.3389/fneur.2015.00234.
[6] A. J. Sihvonen, T. Särkämö, V. Leo, M. Tervaniemi, E. Altenmüller, and S. Soinila, “Music-based interventions in neurological rehabilitation,” *Lancet Neurology*, vol. 16, no. 8, pp. 648–660, Aug. 2017, doi: 10.1016/s1474-4422(17)30168-0.
[7] Jonathan Driedger, Meinard Müller, and Sebastian Ewert, “Improving Time-Scale Modification of Music Signals Using Harmonic-Percussive Separation”, IEEE Signal Processing Letters, 21(1): 105–109, 2014