accompanyBot

Aden Fiol, Rahul Khandelwal, and Nora Wan

Department of Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—For musicians looking to play along with piano harmony, finding piano accompanists can sometimes be a difficult and costly task. While musicians in these situations might be able to turn to cheap MIDI recordings, the digital sound does not match the acoustic quality of a real piano. On the other hand, high-end player pianos are too expensive for the average person's budget. With our accompanyBot, we aim to create a portable system capable of reading sheet music and playing the piano parts.**

***Index Terms*—microprocessor, MOSFET, Optical Music Recognition, piano, power, Raspberry Pi, robot, solenoid, tempo, time signature, XML.**

I. Introduction

This project seeks to improve the playing experience for musicians who are practicing and performing alone. Musicians and singers often rehearse music in ensembles with a pianist to emulate a performance-like setting. This enables the musicians to harmonize with each other and get a better sense of group dynamics and tempo. However, different individuals naturally operate on different schedules and cannot meet with others every time they practice. In these cases, individuals practicing alone would still desire piano backing to keep time in preparation for group rehearsals. Additionally, soloists who are performing a piece that involves piano accompaniment may not always be able to find a piano accompanist who can practice and play with them for their big performance. Furthermore, the cost to hire a professional piano accompanist may be a barrier for amateur musicians who are just starting out their solo careers.

The accompanyBot provides a comprehensive solution to service these goals. From a simple control interface on the user's computer, users will be able to upload a file of sheet music and control our custom hardware that is capable of playing the piano part on a piano. Our portable physical interface that mounts to the piano will allow users to pick up and place accompanyBot on various pianos in different locations.

While there are existing technologies that users may be able to utilize to meet similar needs, they are not without their downsides. On the low-budget side, users might turn to MIDI recordings to play along to during practice. However, the sound quality of digital recordings cannot match the acoustics of physical pianos. For soloists who wish to perform at recitals and in professional settings, MIDI recordings will not suffice for the formal atmosphere. On the other side of the spectrum, high-end player pianos that have actuators embedded within the piano may have the desired sound quality but are also very expensive. Moreover, these player pianos are usually grand pianos. Thus the size and weight of these instruments make it impractical for musicians to move them every time they practice in a different place or perform at different venues.

II. Use-Case Requirements

There are several use-case requirements that our design must meet to ensure user satisfaction.

A. Note Playing Accuracy

Since the users will be utilizing our system to accompany them in performances, the accompanyBot should play the correct notes. This note playing accuracy relies on many different factors, such as the sheet music parser, the note scheduler, and the circuitry being built properly. Constructing and integrating these three major components of our project properly will be integral in ensuring high note playing accuracy.

B. Tempo Variability

Since the accompanyBot will be used for both practice and performance, the user should be able to adjust the tempo of the piano playing so that they can start practicing a piece at a slower pace and work their way up for the final performance.

C. Tempo Accuracy

Due to the importance of timing in music, the user will demand high accuracy with respect to the accompanyBot's playing tempo. This allows the musician to keep time with the piece and stay synchronized if their goal is to eventually play along with others. Thus we have set the requirement for the tempo accuracy to be 100% accurate to the specified BPM from the sheet music or that the user inputs.

D. Low Latency

A pleasant user experience is a priority consideration for the design of our product. Ensuring a fast response between the user interface and the physical system is very important. We have decided to guarantee a response time within 150 ms. According to the NIH, the average human response time to auditory stimuli is around 140-160 ms [1]. Guaranteeing 150 ms will allow for a seamless transition from pressing play to hearing the piano being played. Another area of latency that we must consider is the latency between when the user uploads the sheet music to when the system is ready to play it.

III. Architecture and/or Principle of Operation

In order to operate efficiently and accurately, our system is divided into three distinct subsystems:

1. Local Application
2. Raspberry Pi 4
3. Physical Interface

The block diagram in Fig. 1 depicts the three subsystems and their interconnects.

The Local Application (1) provides the user interface where users insert pdf or png files of high resolution[1] music scores. Additionally, user actions, such as playing or pausing accompanyBot operations, modifying the music playing rate or tempo, and jumping to an alternative region or measure of the piece will be received via the local application and sent to the rest of the system. Once users insert their music score, the app is directly responsible for making use of Audiveris, an open source Optical Music Recognition software, to read the notes from the score into an XML file that can be sent downstream.

The second subsystem is composed of a Raspberry Pi 4 (2). The RPi 4 acts as a middleware controller, taking in raw or transformed data from the user and responding by transmitting signals to control the hardware within the physical interface. The necessity for this separation is for a couple of reasons. Firstly, the Audiveris software is best built and run in a Windows environment which is incompatible with Raspberry Pi's native operating system. The RPi 4 is responsible for running the note scheduling algorithm and ON/OFF signal delivery to the hardware solenoids. Note scheduling will be conducted via the manipulation of data constructed by the music21 library. Specifically, the music21 is used to preprocess the XML received. Alongside handling music notes, the microcontroller accepts requests from the local application corresponding to specific dynamic actions the user makes. These actions flow through the main control function to the GPIO pins of the RPi 4.

The electrical signals sent across the GPIO pins are received by 13 transistors from within the physical interface (3). Low pin signals at the gates of the transistors will put them in cut-off, while high pin signals above the threshold voltages will turn the transistors on. Based on which transistors are on or off, current will flow from the power supply to the corresponding solenoids, thereby pressing down notes on the piano. In the unlikely event that too many transistors are turned on, a circuit breaker will restrict current flow to all solenoids until the power draw settles back down to a safe level.

IV. Design Requirements

A. OMR Parser Accuracy

In order to take advantage of pdf/png inputs, the accompanyBot must be able to parse music scores with high accuracy. We are using OMR software to handle the parsing. Though perfect note parsing would be ideal, there is some room for error as the primary function of accompanyBot is to provide accompaniment. For this reason, our users would find a minimum of 95% accuracy in note parsing to meet their standards for practice or performance. An accurately parsed note is defined as a note whose parsed pitch and duration match the original note.

B. Note Scheduling Accuracy

Once the OMR parser accurately translates the sheet music to an XML file, the note scheduling algorithm should convert that into correct and properly timed signals that are sent to the physical circuitry. This is imperative for the validity of our system since music is all about timing and pitch correctness. Without a high accuracy of note

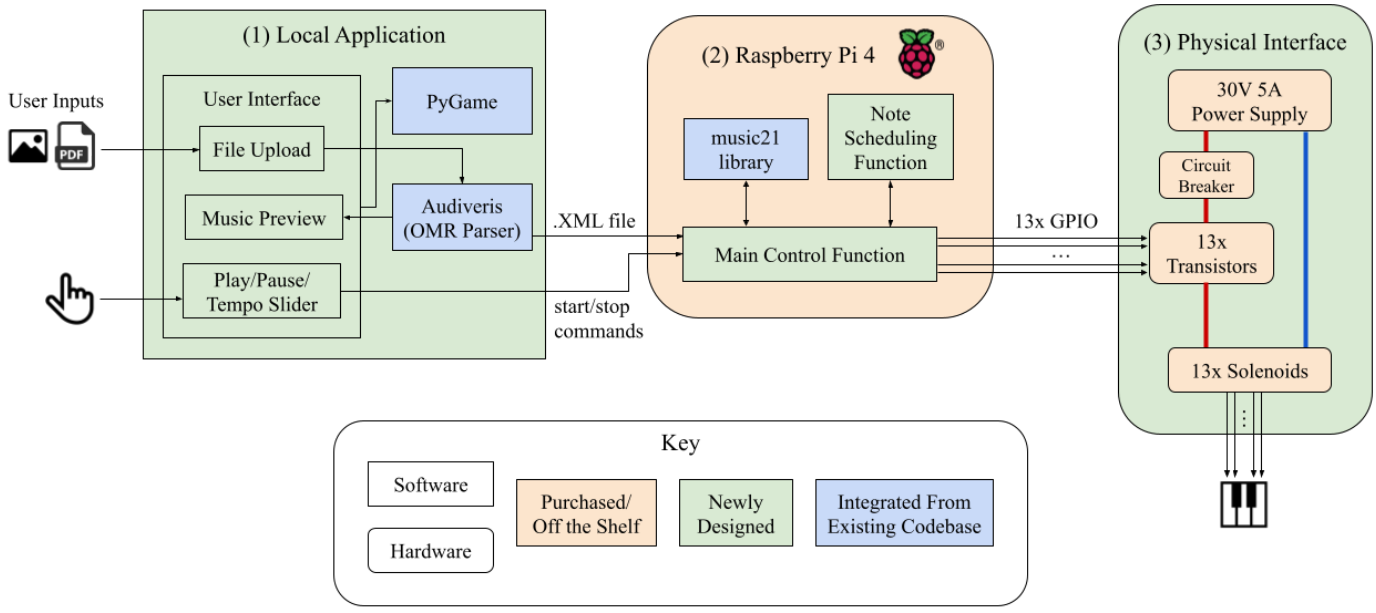

Fig. 1. Block diagram drawing of overall system

scheduling, the users of our product would never be able to practice and perform with our accompanyBot. Since note scheduling accuracy is of significant importance we are setting this requirement to be 100%. Furthermore, this requirement assumes that the tempo and shortest length note fit within the constraints of our system. This requirement is relative to the output of the OMR parser since the note scheduling algorithm has no control over the accuracy of the upstream parsing.

C. Key Press Frequency

For our accompanyBot to meet timing requirements as listed in the use-case requirements section, we need to ensure a quantifiable threshold for the frequency of key presses that the solenoids must achieve. Initially, we set the requirement for the solenoids to reach a metric of six key presses per second. However, based on initial testing of the solenoids and RPi 4, we found that the solenoids were physically limited in the frequency that they are able to push and pull. Since they moved slower than expected, we lowered the key press frequency requirement to four key presses per second. This corresponds to a rate of 240 BPM which we found acceptable for the type of music that a general user would want to play.

D. Power Consumption

Larger chords and simultaneous note presses will demand more electrical work to induce the solenoid motions. Since each of the key pressing solenoids will draw a non-negligible amount of power, we must maintain our system to be safe for the average individual to attach to their piano or keyboard. We found that the average electronic keyboard can draw up to 60 watts of power [2]. For this reason, the hardware component of the robot has a quantitative requirement to never consume more than 60 watts of power. This will ensure users have a safe playing experience while also not overcharging their power bill.

From the data sheets of our solenoids, an activated solenoid requires a max of 12V and 1A. Based on measurements we gathered from initial testing, we were able to operate the solenoid with 10V and 0.65A. However, to provide a buffer, we will use the maximum values for power calculations. Thus, according to (1), the power required to activate one solenoid is (12V)(1A) = 12W. Therefore our system should only support five or fewer solenoids pressed at once.

$$P = IV \tag{1}$$

In cases where the number of notes played at one time exceeds five, which will cause the power to exceed the safety threshold, the least recently played note should be omitted from playing. This restriction will be enforced by the scheduling algorithm that decides which solenoids to activate. In the unlikely case that our scheduling fails, we will also include a circuit breaker that triggers if the power goes over 60W.

V. Design Trade Studies

A. OMR Framework Selection

We opted for Audiveris as opposed to other OMR frameworks primarily due to its high accuracy of identifiable notes and ability to segment multiple parts as opposed to other frameworks available. Initially, we had considered using an openCV and tensorflow based ML model built for Python. This was so that our UI hub application, which will also be built in Python, could directly process the music scores without needing to contact an external application. Early on we tested a pre-trained model developed by researchers in Montreal, Alicante, and Valencia [3]. Overall parsing time for a single page was no more than 10 seconds. However, the model was built to process strictly monophonic scores. Another Python based solution we tested was Oemer [4], which could be easily installed and used via pip and import statements. Oemer met our requirement of processing polyphonic scores but failed to meet our note accuracy and processing time requirements. Initial testing yielded close to 50% of correct notes placed in the XML file structure, and also the time to parse a single page of notes from two staves exceeded a minute.

Despite having a complicated build process, Audiveris proved to be the best open source solution for our project. Testing its OMR engine on multiple page scores yielded an average time of 20 seconds per page, not unreasonable for a user to wait the one time cost. Additionally, it had seamless integration with the music21 Python library, a framework that will be essential for us to schedule notes on the hardware. The traditional usage of Audiveris is through its GUI. However, since we will build out a separate GUI for the accompanyBot user interface, we are making use of the developer command line usage of Audiveris as referenced in its documentation [5]. To further confirm the effectiveness of Audiveris, we played back the MIDI equivalent of XML generated from Chopin and Beethoven scores. Aside from the digitized sound of the player and a few odd notes over the course of a ~3 minute piece, the audio matched the written notes.

The last option for us was to build an OMR engine from scratch. Manually crafting an ML model or framework would have gone out of the scope of our goals. From our understanding, many researchers have spent years developing and improving the models for the machine learning and optical classification of notes. They have not yet reached a complete solution, though a good approximation is achievable. Overall, integrating Audiveris allows us to give better attention to the other subsystems of the accompanyBot.

B. Microcontroller Selection

When choosing the specific microcontroller to act as the mediator between the software and physical circuit of our project, we prioritized three characteristics: compute power, a sufficient number of GPIO pins, and ease of integration with other parts of the project. The three options we considered using were the Arduino Uno, STM32F4, and RPi 4. The former two candidates were considered because

members of the group had used them for previous projects and classes, while the latter was considered due to its solid reputation for being a reliable general-use microprocessor.

Due to the need to parse the XML file into usable data and schedule signals at specific times, we required a component that had significant computing power and memory available to store the data. This led us to choose a microprocessor over a microcontroller like the Arduino Uno that is more specialized for specific tasks. Furthermore, the Arduino Uno did not have as many GPIO pins as the other two options, so we ruled out the Arduino Uno completely.

At this point, we had decided to have the microprocessor directly control the power switching of the solenoids with the GPIO pins, with each GPIO output corresponding to one key. Thus we needed a microprocessor that could support a large number of GPIO outputs in order for us to cover all the required keys. When looking at the STM32F4 versus the RPi 4, we found that the STM32F4 had 80 GPIO pins [6] in total whereas the RPi 4 had 26 [7]. However, we had also decided to limit the scope of the accompanyBot to only play one octave, which required a maximum of 13 pins. Thus, both options were still valid since they had an appropriate amount of pins while also being able to have some extra pins in case they were needed for other purposes.

Therefore the last criterion was crucial in deciding which microprocessor to use for the accompanyBot. Since we are developing the user interface in Python, we felt that having the coding environment on the microprocessor also be in Python would make it easier to handle the communication between them through libraries such as pySerial, which we also have experience with. In addition to this, we found through our initial search that there were many Python packages relating to analyzing music, so we could fall back on them when performing the XML parsing and note scheduling (This is precisely what we did in the end with our choice to utilize the music21 library). Thus, for the reasons specified above, we chose to use an RPi 4 from the class inventory.

C. *Solenoid Selection*

There is a large variety of solenoids on the market. They vary in terms of force, size, type, and electrical requirements. We decided to purchase three different solenoids that varied across each category: a 5N Adafruit Small Push-Pull solenoid that had a 12V requirement, a 25N Adafruit Large Push-Pull solenoid that had a 12V requirement, and a 25N Pull Type Open Frame Solenoid Electromagnet Linear Motion JF-1039.

During our testing of the three solenoids, we noticed major downfalls for two of the solenoids which led to us choosing the 25N Adafruit Large Push-Pull solenoid. The 5N Adafruit Small Push-Pull solenoid struggled in three categories: stroke length, force output, and power consumption per unit of force. Our project needed solenoids that would be able to reach the keys from a reasonable distance away and with enough force to depress the keys while also not consuming too much power. Constructing a mechanism that would hold the weight of all the solenoids and circuitry close enough to the keyboard without falling over would be difficult. Thus, a stroke length that is too small would be difficult to design around. Additionally, from our initial testing, 5N was not enough force to depress an average piano key. Furthermore, it consumed the same amount of power as the large push-pull solenoid, but the power consumption per unit of force was much higher, so for those reasons, the 5N Adafruit solenoid would not work within the constraints of our system. The 25N Pull Type Open Frame Solenoid was lacking in one major area: the fact that it was a pull type rather than a push-pull solenoid. When researching solenoids we were not aware that the pull type solenoid would not be fixed in place. When we received the part, we discovered that the ferromagnetic rod for depressing a piano key was free to fall out of its exterior metal casing. This was not what we desired. Instead, we preferred the 25N Adafruit solenoid since its ferromagnetic rod was fixed and did not fall out of the external metal casing.

Overall, the design shortcomings of two out of the three solenoids led us to conclude that the 25N Adafruit Push-Pull solenoid was the best option for our accompanyBot. We are able to get a lot of force per watt and a high stroke length without the internal rod falling out of the external metal casing, making it the logical option to use. The final factor that solidified our decision was the price point of the 25N Adafruit Large Push-Pull solenoid. Since Adafruit offered a discounted per unit price when buying in bulk, the price of each solenoid was cheaper than other similar solenoids on the market.

VI. System Implementation

Our system can be broken down into three domains: a translation path, notes scheduling path, and execution path depicted in Fig. 2. Subsections *A* and *B* make up the translation path, *C* describes the intermediate path between translation and notes scheduling, *D* goes into depth regarding the RPi operation for scheduling, and *E* lays out our setup and connection of the moving parts and electronics built in-house.

A. *Computer GUI Interface*

The user will have the ability to control the accompanyBot over the GUI software on their computer. The necessity for ease of use and visual appeal is important here, so we will be using the pyGame library in Python to build out this application interface. Fig. 3. illustrates a model of our application's GUI.

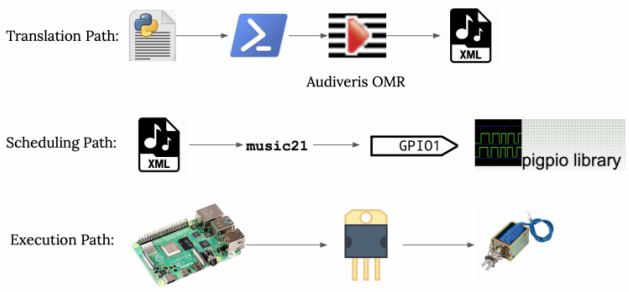

Fig. 2. Visualization of subsystem flows

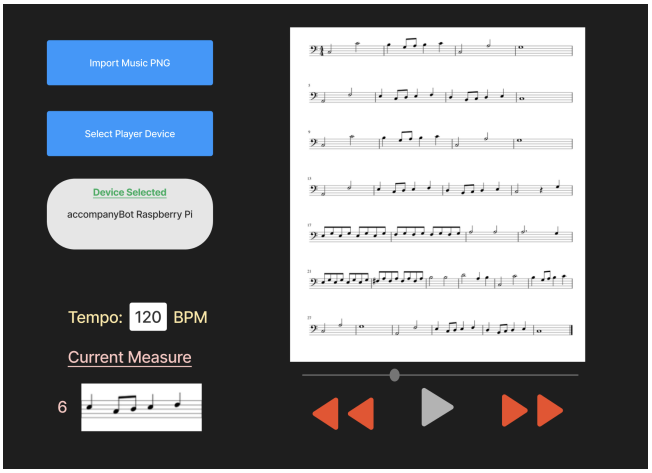

Fig. 3. Model of GUI Application

The user will be able to choose their input sheet music file and output source through buttons labeled “Import Music Score” and “Select Player Device” respectively. The user can manually specify the beats per minute in the tempo window of the application. Additionally, a slider and play button will allow the user to direct the RPi to start or stop playing or change measures. For convenience, a current measure playing slot will show what measure and notes the notes scheduler is currently on.

B. Music Score Reader

When a music score is imported into the hub application, if a corresponding music XML transcription is not found within the application cache then the software will begin a subprocess to execute the Audiveris OMR engine. Due to the limitations of Audiveris dependencies, it must be run only on Windows. Additionally, an install of Java 17 is required to enable Audiveris. For this reason, the hub application will execute a PowerShell script to launch Audiveris and process the music score image into an XML file. This XML will get saved to a desired output location and also be added to the application cache for repeat usage efficiency.

C. Computer to Microprocessor Communication

We will be handling the communication between our laptop and the RPi 4 through a serial USB connection. We believed this was the simplest method of implementation for communication since the user can easily connect the finished product to their computer without having to set up any network connections. We plan to use the pySerial library for transmitting the files and command bytes that will let the RPi 4 know to prepare for a file, start playing, stop playing, or change the tempo. However, if the latency for the file transfer is too high when sending the XML file serially, we may have to switch to a wireless method such as FTP or SFTP.

D. Note Scheduling

The first step of the note scheduling process involves parsing the XML file to extract the key signature, time signature, tempo, and individual notes. To aid in the scheduling task, we have employed the Python library music21 to convert the XML file to a data structure that is more readable and allows for easy access to the musical data. Through the `parse` method of the library, we can extract the essential information needed for scheduling: the clef, tempo, key signature, and an iterable array of notes and their pitches. Fig. 4 illustrates an example of the type of output achievable by using music21. When the snippet of sheet music shown in Fig. 5 is converted to its XML representation, the entire file consists of 184 lines of code. However, after a simple parsing with music21, all the necessary information is condensed into a 20 line Stream object. An important detail to note is that the offset, which is the number in curly braces, indicates the distance in quarter notes that a note or musical feature is located from the beginning of the Stream. The offsets are used by our scheduling algorithm to determine where in the piece a note is played.

The second step in scheduling involves using the data to determine several timing parameters. The general approach to scheduling is to divide up the piece into distinct time intervals starting at fixed time units. The period of each interval should be the duration of the smallest note value since we want to be capable of playing notes of that value one after the other. To find this duration, we first want to calculate the duration of a quarter note (which is the base duration that other notes are in reference to) and then scale it by how many smallest note values are in a quarter note. For example, if the smallest note in a piece is an eighth note, then we would find the duration of an eighth note by finding the duration of a quarter note and multiplying it by 0.5 since an eighth note is half of a quarter note. Equation (2) shows how to calculate the duration of a quarter note in milliseconds. Tempo is the numerical value in beats per minute while the beatToQuarterNoteRatio is the ratio of the beat type to a quarter note. Different tempo instructions are shown in Fig. 6. All three cases would have a tempo of 60 BPM. From top to bottom, the beatToQuarterNoteRatio would be 2, 1, and 0.5 respectively a half note, quarter note, and eighth note.

$$duration = \frac{60{,}000\ ms/min}{Tempo \times beatToQuarterNoteRatio} \quad (2)$$

```
{0.0} <music21.text.TextBox 'C Scale'>
{0.0} <music21.metadata.Metadata object at 0x106fbc210>
{0.0} <music21.stream.Part Piano>
    {0.0} <music21.instrument.Piano 'P1: Piano: Piano'>
    {0.0} <music21.stream.Measure 1 offset=0.0>
        {0.0} <music21.layout.SystemLayout>
        {0.0} <music21.clef.TrebleClef>
        {0.0} <music21.tempo.MetronomeMark larghetto Quarter=60.0>
        {0.0} <music21.key.KeySignature of no sharps or flats>
        {0.0} <music21.meter.TimeSignature 4/4>
        {0.0} <music21.note.Note C>
        {1.0} <music21.note.Note D>
        {2.0} <music21.note.Note E>
        {3.0} <music21.note.Note F>
    {4.0} <music21.stream.Measure 2 offset=4.0>
        {0.0} <music21.note.Note G>
        {1.0} <music21.note.Note A>
        {2.0} <music21.note.Note B>
        {3.0} <music21.note.Note C>
        {4.0} <music21.bar.Barline type=final>
```

Fig. 4. Example of the compact data structure that is produced from parsing the C scale below using music21

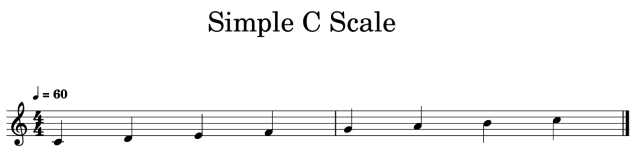

Fig. 5. Quarter note C scale in 4/4 time signature at 60 BPM

At this point in the scheduling process, we will be able to determine if the piece is playable or not. The max limit of four key presses per second from our design requirements means that each note must be at least 250 ms in length. Therefore if the duration of the smallest note value is less than that, we will send a signal back to the computer to notify the user.

If the piece passes the above check, then the scheduling algorithm will continue. The function will iterate through the array of notes and determine which notes are being played at each time unit. Since the time units are based on the smallest note value, they can be found by taking the offset of a note in the Stream and multiplying it by the beatToQuarterNoteRatio of the smallest note value. For each note that is being played, the corresponding GPIO pin will need to be set high. For the rest of the notes, they should be set low. Based on a dictionary mapping of keys to GPIO pins, a bit mask will be generated for which pins are high. The mask will be saved to a dictionary where the key value pairs are time unit and a bit mask. The function will also check that no more than five notes are at one offset. If there are more than five, the scheduling algorithm will choose which note to omit based on previously played notes. After the function finishes, the RPi 4 will return a signal to the computer to let it know that the file has successfully finished scheduling.

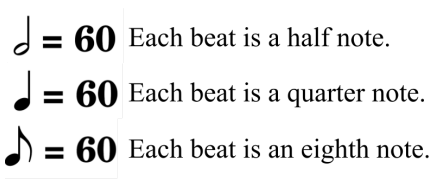

Fig. 6. Examples of different tempo markings

Then, when the RPi receives the signal to start playing from a specific measure, it will set the time unit to the measure number times the number of beats in a measure scaled by the number of the smallest note value in a beat. The function will have a while loop in which it retrieves the notes to play (represented by the bit mask stored earlier) at the given time unit. The function uses the bit mask to set the GPIO pins, increment the counter for the time unit, and delay the function by the duration of time for the time interval. If the RPi receives a pause signal in between iterations of the loop, it will simply exit the loop and wait for the next command.

With this process, the scheduling algorithm only needs to run once to save the bit masks for the notes at each time unit. To change the speed of the playback, the RPi simply needs to recompute the duration of the delays with a new tempo value using the formula mentioned earlier.

E. Physical Components

The physical circuitry consists of multiple components, such as GPIO inputs, an array of MOSFETs, solenoids, diodes, and a power supply. Each component provides an essential role in making the system move and play the piano correctly.

The power source provides the voltage and current needed to actuate the solenoids. The power source has a voltage rating of 30V and a current rating of 5A. With each

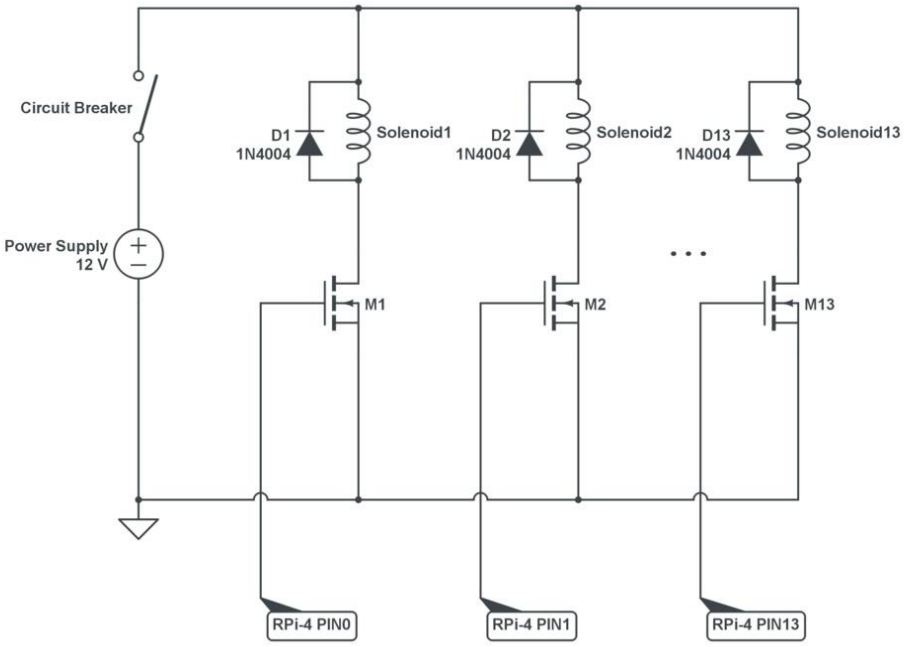

Fig. 7. Schematic of the physical circuitry

solenoid drawing at most 1A at 12V, this should be all that we need to power the system since no more than 5 keys will be pressed at a single time. In order to enforce this current restraint, we are looking into circuit breakers to stop the current flow if it gets too high, but this might not be necessary since our power source already maxes out at 5A.

The power source will be supplying the current for the 25N Adafruit Large Push-Pull solenoids when the MOSFETs are no longer in cut-off. The solenoids are large inductors that have a metal rod fixed inside of the coil. Once current is flowing through the inductor, it induces a magnetic field that causes the metal rod to move. The solenoid retracts once the current is cut from the inductor, hence the push-pull name. These solenoids will be pushing on the piano keys depending on the signals it receives from the microprocessor after the notes have been scheduled.

The output GPIO pins from the RPi 4 are connected to the gates of our MOSFETs. The gates will receive a high or low signal depending on the output of the scheduling algorithm. When high, this signal will force the MOSFET out of cut-off and allow current to flow from the source to the drain and through the solenoid since they are in series. This will power the solenoid and actuate the piano keys when properly scheduled.

The last major component of the physical circuitry is the 1N4004 diodes. The diodes are placed in parallel with the solenoids so that when the MOSFETs are turned off, the voltage spike caused by the inductor in the solenoid does not damage our power supply or microprocessor.

Lastly, we will be constructing a chassis to support the aforementioned circuitry components so that they can hover over the piano keys and press them from a stationary location. Although our chassis design may be subject to change, our preliminary implementation will involve a sturdy beam that will clamp or stand along the edge of the piano paired with a sliding box containing all of the solenoids. Fig. 8 displays a preliminary design. This box is adjustable to allow the user to select their octave range of pressable keys. Once the octave is selected, the user can proceed to upload their desired piece and the accompanyBot will perform.

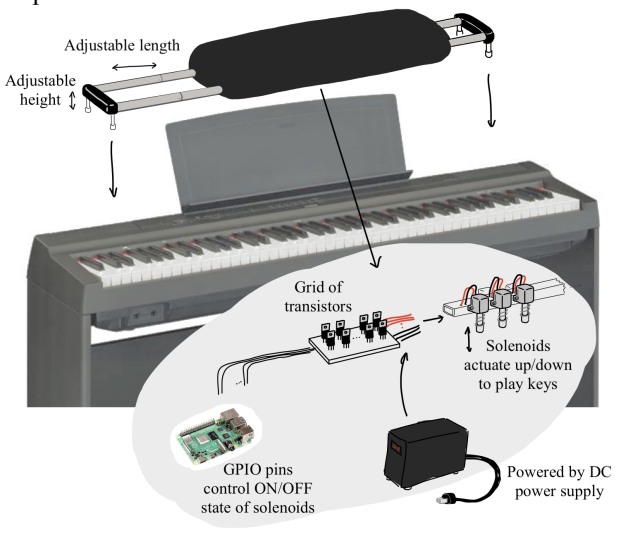

Fig. 8. Potential implementation design

VII. Test, Verification and Validation

A. Test for OMR Parser Accuracy

After the OMR Paser has completed and produced the XML file associated with the sheet music passed in, we will use the music21 library to convert the XML file back to a pdf of the sheet music and compare the original sheet music to the newly generated sheet music. Counting the number of correctly parsed notes and features and dividing by the total number of notes and features will show how much of the music matches the original, determining the OMR parser accuracy.

B. Test for Note Scheduling Accuracy

In order to test and measure the note scheduling accuracy, we will observe and record two things: the timing of the key presses and the notes that are pressed. Once the XML file has been sent to the RPi 4 and our note scheduling algorithm has been run, we will observe and record all the notes that the accompanyBot presses and compare that to the notes parsed by the OMR. Additionally, we will take the output of the tempo accuracy test to ensure enough delay was put in between notes such that it is correct according to a metronome.

C. Test for Key Press Frequency

In order to test the key press frequency, we will use a metronome set to 240 BPM, or four beats per second. If the solenoid can press the piano key at the same rate as the metronome, it will meet our use-case requirement of four key presses per second.

D. Test for Power Consumption

Once the circuitry is constructed, we will turn on five of the MOSFETs such that current is being drawn through the solenoid. We will then measure the voltage across each of the five solenoids, as well as the current through each, and do some calculations by hand to determine the total power consumed by the solenoids when they are all on at once.

E. Test for Latency

The latency will be measured as the time taken from when the longest signal is sent from the hub application to the microcontroller plus the time until the end of the longest response is received back by the hub application from the RPi microcontroller. As we build out the communication segment of the accompanyBot, we will figure out which signal(s) will have the longest byte sequences. Then, to determine latency we can write a simple helper function to run from the app as described in Fig. 9.

```
def latency():
    startTime = time.time_ns()
    SendLongestSignal() # asynchronous execution
    ReceiveLongestResponse() # returns after receiving
    endTime = time.time_ns()
    print(endTime - startTime)
```

Fig. 9. Latency Measurement Pseudocode

F. Test for Tempo Accuracy

In order to test for tempo accuracy, we will use a metronome application and set it to the BPM of either the piece or what the user has determined is an appropriate tempo. We will then audibly determine whether or not the accompanyBot is striking the piano keys at the correct time according to the metronome.

G. Test for Note Playing Accuracy

A reasonable way to test for the accuracy of notes played would be to compare sound output quality to a human or digital source. The music21 library is capable of playing back an XML converted score through the computer speakers. We will contrast this digital output with the audio playing from the solenoids to qualitatively rate the note playing accuracy. This mechanism of testing better isolates small bugs that may occur earlier on through the OMR parsing, as both the digital audio and notes played by the accompanyBot must rely on the transcribed XML, whereas a human demonstration would introduce a variable music recognition process that may skew the pure note playing test results.

VIII. Project Management

A. Schedule

The Gantt chart in Fig. 10 and Fig. 11 below lays out different deliverables from each of us every week of the semester. The last two weeks of slack are reserved for issues that may pop up from the integration of the three subsystems. We also have assignment due dates on the last few rows of the Gantt chart.

B. Team Member Responsibilities

Our team responsibilities are divided into the three major components of our project: the OMR integration and application development, the microprocessor interface and note scheduling algorithm, and the physical circuit implementation. Rahul is responsible for OMR integration and application development. Nora is responsible for the microprocessor and note scheduling algorithm. Aden is responsible for the physical circuit implementation.

Naturally, these three sections of our system rely upon each other for different input and output information, so we have all collectively agreed to work together on problems where our sections intersect. For example, we all rely upon the note scheduling and microprocessor section of the project, so we will all make contributions to ensure that there is a seamless transition from the user interface all the way down to the circuitry, which is impossible without the proper implementation of the microprocessor. Additionally, Aden's and Nora's areas of expertise intersect greatly, so they have both made contributions to one another's sections of the project in order to ease the workload and guarantee a working implementation. Rahul has also helped Nora find Python libraries that aid in parsing XML files and storing information in different OOP structures. As our project heads to completion, some of these responsibilities may change, grow, or intersect more since the project could change, but as it stands this is the current division of labor.

C. Bill of Materials and Budget

So far, we have spent $289.43 of our total budget. A detailed breakdown of the bill of materials for the final product can be found in Table 1 at the end of this document. The grand total in the table does not include money spent on the test solenoids and spare parts we purchased in addition to what is strictly necessary.

Although we have not yet purchased the material for the physical piano mount, we are currently planning on using 3D-printed material for the casing that covers the electronics and 80/20 aluminum T-slots for the adjustable framing that will support the hardware above the piano keys. We estimate that altogether these should not exceed $200 total, so we will still be within the $600 budget.

D. Risk Mitigation Plans

One main risk involved in our project is the high degree of accuracy needed in terms of meeting timing requirements. Since some music may be rhythmically complex and very fast-paced, it may be difficult to capture those melodies with our system. To mitigate this risk, we are limiting the scope of possible user inputs to a library of sheet music that does not put strain on the accompanyBot. During our demo, we will still allow users to select different songs to upload, but they will all be chosen ahead of time by us.

Physical parts are fragile and often break when constructing something new. Therefore, we have purchased additional solenoids and MOSFETs to replace anything that may break when building and integrating the whole system. We also have additional room in our budget in case some of our parts break and we need to purchase more.

In general, all of us have stayed on schedule and have been able to make consistent progress. At the moment we have a working music reader, file handler, digital audio output mechanism, and solenoid motion from the RPi. In the worst case of the application being incomplete, the notes scheduler left undeveloped, and the chassis build being unsuccessful, we can still put together an effective proof of concept demonstration of solenoids pressing down on a paper representation of a keyboard with digitally interpreted audio from piano scores we choose beforehand.

IX. Related Work

We were curious about related projects that make use of FPGAs. For MIT's digital design course, undergraduate students developed their own accompanyBot that processes audio from .wav files and plays matching notes on a keyboard from the output of an FPGA [8]. Similar to us, they made use of Python to gather notes into organized data structures but instead designed an FSM on an FPGA to control motors connected to beams that would rotate and strike the keys, whereas we opted for a Raspberry Pi that would induce solenoids to strike the keys via a linear motion.

Our inspiration to use solenoids controlled by a Raspberry Pi originates from a working keyboard player published in The MagPi, the official Raspberry Pi magazine

[9]. Their project successfully played monotonic melody lines on a keyboard. We of course seek to improve upon this project with the power of music21 and our in-built notes scheduler to be capable of playing polyphonic melodies or harmonies.

X. Summary

In summation, our design consists of three main subsystems: the local application, the Raspberry Pi 4, and the physical interface. The local application consists of an OMR parser that will take the pdf/png from the GUI and output an XML file containing information about the sheet music. This XML file will then be forwarded to the RPi 4 via USB for more parsing. The RPi 4 will then use the music21 library to extract all the vital information about note pitch, duration, and time stamp of occurrence. Then the parsed notes will be scheduled and await for the user to begin playing the piece from the GUI. Once the user decides to begin the piece, high and low signals will be sent to the MOSFETs that will allow the solenoids to draw current from the power source and press down or pull up according to the output of the note scheduling from the RPi.

As we progress individually on our subsystems, an upcoming challenge will be integrating all of these subsystems into one cohesive product. This is a challenge with just about any project that has so many moving parts like ours. Some other challenges might arise out of the physical limitations of our parts since they are bound by natural phenomena. Our solenoids, MOSFETs, and diodes are not ideal, so dealing with those issues might prove challenging. Additionally, our solenoids are bound by different time constants that might prevent them from moving as fast or as slow as we desire. Hopefully, testing and verification of our systems can reveal these issues early on so we have time to address them in our implementation.

The accompanyBot aims to improve the quality of isolated practice for singers and other instrumentalists. After rehearsing with our technology, musicians will realize more productive practice sessions without their ensembles. Wherever there is a piano, the accompanyBot can perform. With a compact product that is perfect for practice and performance, users of the accompanyBot will be able to experience an enhanced playing experience that furthers their love of music.

Footnotes

1. High Resolution music scores are limited to those generated via computer software or transcribed to modern print quality. The accompanyBot will accept and attempt to parse low quality or handwritten scores, however, at the cost of lowered accuracy and overall performance than the metrics outlined in the use-case requirements.

Glossary of Acronyms

BPM – Beats per minute
FTP – File Transfer Protocol
GPIO – General Purpose Input/Output
GUI – Graphical User Interface
MIDI – Musical Instrument Digital Interface
MOSFET – Metal Oxide Semiconductor Field Effect Transistor
OMR – Optical Music Recognition
OOP – Object Oriented Programming
RPi 4 – Raspberry Pi 4
SFTP – Secure File Transfer Protocol
USB – Universal Serial Bus
XML – Extensible Markup Language

References

[1] "A comparative study of visual and auditory reaction times on the basis of gender and physical activity levels of medical first year students" *National Library of Medicine*, Accessed on Feb 27, 2023, [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC44568

[2] "Do Electric Pianos Use a Lot of Electricity? [Explained]." Piano Sounds, Accessed on Feb 27, 2023, [Online]. Available: https://pianosounds.com/do-electric-pianos-use-a-lot-of-electricity

[3] Ebonko, Israel, "Play Sheet Music with Python, Opencv, and an Optical Music Recognition Model." *Medium*, Heartbeat, 5, Oct. 2021, https://heartbeat.comet.ml/play-sheet-music-with-python-opencv-and-an-optical-music-recognition-model-a55a3bea8fe

[4] *Github*, BreezeWhite, https://github.com/BreezeWhite/oemer

[5] Bitteur, Hervé. "Command Line Interface." *Audiveris Pages*, Hervé Bitteur, https://audiveris.github.io/audiveris/_pages/advanced/cli/

[6] "Exploring GPIO Port and Pins of STM32F4xx Discovery Board." *FastBit EBA*, 18 Nov. 2022, https://fastbitlab.com/exploring-gpio-port-pins/

[7] "Raspberry Pi and General-Purpose Input/Output." *FutureLearn*, https://www.futurelearn.com/info/courses/robotics-with-raspberry-pi/0/steps/75878

[8] Ashworth, Brendan, et al. *Piano Man FPGA Piano-Playing Robot - Massachusetts Institute of Technology*. 11 Dec. 2019, https://web.mit.edu/6.111/volume2/www/f2019/projects/brendana_Project_Final_Report.pdf

[9] King, Phil. "Piano-Playing Robot." *The MagPi Magazine*, https://magpi.raspberrypi.com/articles/piano-playing-robot

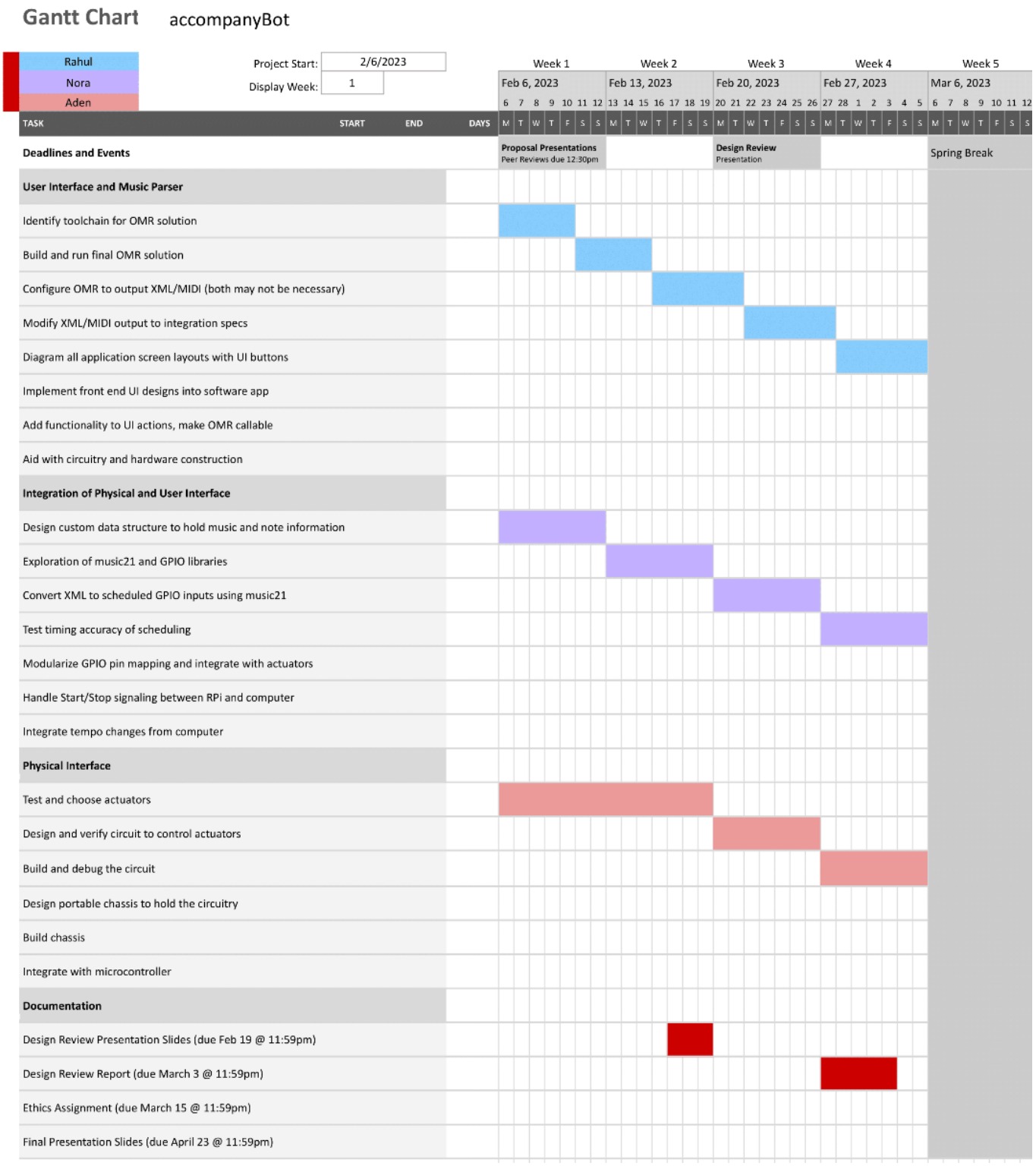

Fig. 10 Gantt Chart continued

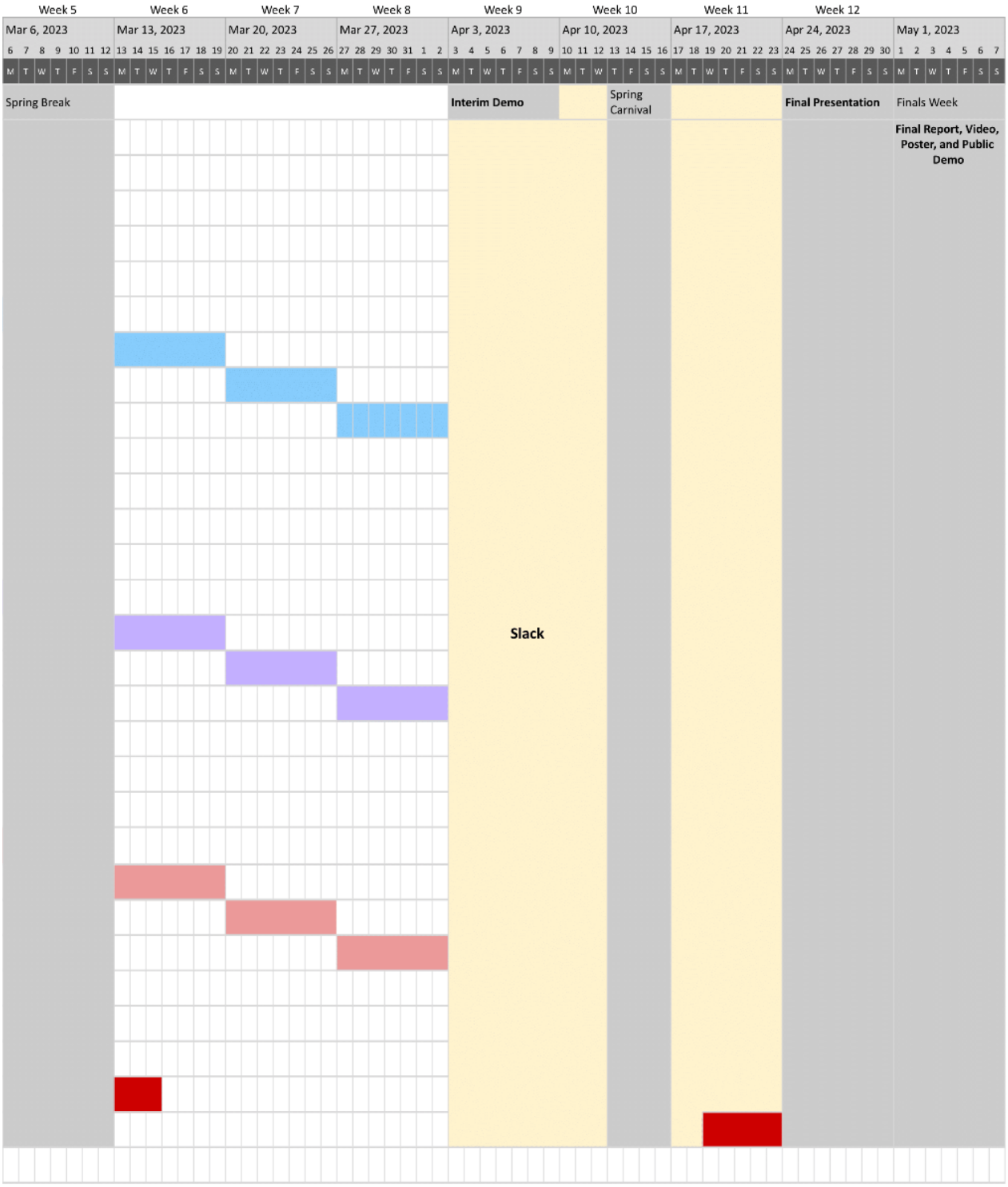

Fig. 11 Gantt Chart continued

TABLE I. BILL OF MATERIALS

Description	Manufacturer	Quantity	Cost Per Unit	Total Cost
N-Channel Power MOSFET - 30V/60A	Adafruit	13	$2.03	$26.39
Large Push-Pull Solenoid	Adafruit	13	$13.46	$174.98
1N4004 Diodes	-	13	$0	$0
Protoboards	Adafruit	2	$0	$0
Raspberry Pi 4 - 8GB	-	1	$0	$0
Electronic Keyboard	-	1	$0	$0
Power Supply 30V/5A	-	1	$0	$0
			Grand Total	**$201.37**