

**Title of Project:** Downbeat

**Team members:** Damon Anderson, Michael Halloran, Catherine Kenzie, Prakash Kumar, and Bryan Rabotnick

## I. Executive Summary

The Downbeat project is a real-time drumming transcription system. The system processes a single audio track of a live performance on a drum kit and generates sheet music corresponding to what is being played. This process consists of two key algorithmic steps: classifying different pieces of the kit which were hit, and transcribing onset times to standardized music notation. The transcribed performance is displayed on a GUI which is updated in real time. The system offers additional visual feedback by flashing LEDs mounted on a 3D drum kit model corresponding to the pieces of the drum kit detected in classification. Pressing a playback button after the performance will prompt auditory feedback of the system's transcription. The system generates and plays an audio file using audio samples of the individual pieces of the drum kit.

Almost all of the primary goals for the project have been successfully met. These achievements include designing a stand-alone working prototype consisting of a Raspberry Pi 4 and an STM Nucleo F747ZI; implementing a setup procedure to calibrates the system with the spatial setup of the drum kit; generating a steady click track at a user-selected tempo within an appropriate range; supporting the classification of five distinct drum kit pieces and the transcription of sixteenth-note rhythmic subdivisions; providing two forms of real-time visual feedback via the GUI and the LEDs; and synthesizing the transcribed performance into an audio file for playback. The only primary goal which has not been achieved is classifying and transcribing drum rudiments, such as rolls and flams. The project has surpassed original expectations for layout and packaging. The processing components of the system are contained in a compact box which also serves as a platform for the 3D model. The box includes two buttons and a knob for user input as well as access to pertinent input and output ports.

Performance of the classification algorithm is dependent on the calibration process and the tuning of multiple parameters. The algorithm can reliably classify each piece of the kit in isolation, although its accuracy in identifying the hi-hat and crash cymbal decreases when either piece is played simultaneously with another piece. Performance of the transcription algorithm is dependent on one parameter which can be set to provide an optimal balance between precision and quantization error.

## *Table of Contents*

<b>Executive Summary</b>	<b>1</b>
<b>I. Introduction</b>	<b>4</b>
<b>II. System Architecture</b>	<b>5</b>
Hardware Functionality	5
Layout and Packaging	6
<b>III. Classification</b>	<b>8</b>
The Algorithm	8
Constraints and Solutions	12
Testing and Results	13
<b>IV. Transcription</b>	<b>15</b>
The Algorithm	15
Pre-Runtime	16
Runtime Computation	18
Constraints and Solutions	19
Testing and Results	20
<b>V. GUI</b>	<b>22</b>
The Algorithm	22
Constraints and Solutions	23
Long Inputs	23
Playback Synthesis	23
Complicated Inputs	24
<b>VI. Audio Playback</b>	<b>25</b>

---

The Algorithm	25
Constraints and Solutions	26
<b>VII. 3D Printed Drum Kit</b>	<b>27</b>
Constraints and Solutions	27
<b>VIII. Summary of Constraints</b>	<b>28</b>
<b>IX. Milestones</b>	<b>29</b>
Milestone 1	29
Milestone 2	29
<b>X. Demo</b>	<b>30</b>
<b>XI. List of Parts</b>	<b>31</b>
<b>XII. Contributions of each member of team</b>	<b>33</b>
<b>XIII. References and citations</b>	<b>34</b>
<b>XIV. Appendices</b>	<b>35</b>
Appendix A: Classification Plots	35
Appendix B: Up-to-Date Classification Results	38
Appendix C: Previous Classification Results	40
Appendix D: Loading Screen Implementation	43
Appendix E: Guido Parsing and Interpretation	45
Appendix F: Alternative Transcription Methods	47
Appendix G: Hardware Specifics	48

## I. Introduction

The Downbeat project is a real-time drumming transcription system. The system uses a single microphone to capture the audio of a live performance on a drum kit consisting of a kick drum, a snare drum, a rack tom, a hi-hat, and a crash cymbal.

The system processes the audio in real time and updates a GUI display with standardized sheet music notation corresponding to the performance. The system additionally lights up LEDs mounted on a 3D drum kit model which correspond to the pieces of the drum kit that were played. Furthermore, users can press a playback button to listen to the transcribed performance using audio samples of the individual pieces of the drum kit.

The system can be broken down into two general processing steps: classification and transcription. The classification step analyzes the audio signal captured by the microphone to determine which pieces of the drum kit were played and at what times these onsets occurred. Taking this information as input, the transcription step evaluates how these onsets times correspond to standard notation of musical rhythm.

## II. System Architecture

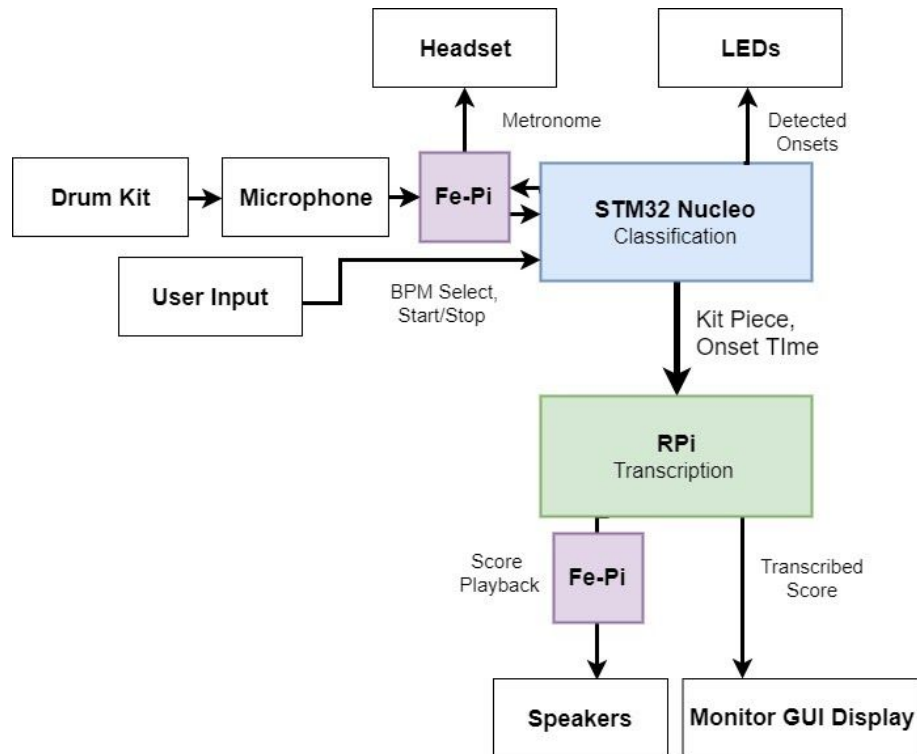


Figure 1: Block diagram of the Downbeat system.

### Hardware Functionality

The Downbeat system's algorithms are implemented on a Raspberry Pi 4 (RPi) and STM32 Nucleo F747ZI. The Nucleo serves two primary purposes: to interface with both the user and the audio input, and to perform the complex signals processing steps needed in classification. A Fe-Pi Audio Z V2 codec is used to send the drum kit input from the microphone to the Nucleo. The Fe-Pi converts the incoming audio signal from the microphone to a digital signal that can be further analyzed. Also, this Fe-Pi is used to play a click track back to the user, functioning as a metronome to keep time. The click track is generated through the use of a hardware timer interrupt. The timer is configured to interrupt the program at a specific time interval based on the potentiometer value which corresponds to the desired tempo set by the user before drumming. When the interrupt is triggered, a pre-generated click sound is sent to the Fe-Pi DAC and played back to the user through headphones. The Nucleo also interfaces with two buttons

that are configured to hardware interrupts. A ‘start/stop’ button toggles the start of a processing session in which audio is accepted and the full processing chain is run. The first press of the ‘playback’ button prompts the synthesis and playback of an audio track based on the transcribed sheet music and made from recorded drum samples. Additional presses of this button prompts playback of the generated audio file so that the user can further compare the transcription to the original performance.

The Nucleo connects and controls multiple LEDs which are configured to light up as hits of the kit are detected by the classification algorithm. These are implemented through simple GPIO functionality. Finally, the classification algorithm is flashed onto the Nucleo and constantly run on incoming audio signals. The results of the classification algorithm are sent to the RPi via a UART connection.

Once the onsets arrive on the RPi, two major steps are run to generate the sheet music. First, the transcription algorithm is run to quantize the incoming hits. Once these results are available, the GUI rendering software is run to update the screen with the newly detected hits. When the program has not been started, updated BPM values are also received from the Nucleo and displayed on the screen allowing the user to choose the correct tempo. In the stopped state, the playback interrupts are also sent to the RPi to initiate the GUI synthesizing screen. An additional Fe-Pi is mounted on the RPi in order to perform recording and playback functionality. The recording functionality is necessary to record training samples for the classification basis matrix prior to runtime. The playback is sent via the Fe-Pi to speakers so that the user can hear the playback of the generated sheet music. Altogether, the RPi receives the results of classification by UART, converts these inputs to sheet music, and displays the results on the GUI.

## Layout and Packaging

To improve user experience and increase physical robustness, most parts of the system including the RPi, Nucleo, Fe-Pis, breadboards, and wiring for hardware were encapsulated in a 200x120x75mm plastic box (see Figure 2). Drilled holes on the side of the box allows for connecting power to the processors, necessary inputs and outputs on the Fe-Pis, wiring to the external 3D model drum kit LEDs, and accessing the Nucleo’s system reset button.

With the box, users cannot accidentally disconnect or otherwise disturb the setup and they can interact with the system in a more comprehensible way. The drilled holes on the lid of the box closely fit two buttons and a potentiometer that are labeled with the parameter that they control. In the stopped state, the user turns the “BPM” potentiometer to select the tempo of their



Figure 2: Placement of hardware.

performance. The 3D model drum kit is secured to the lid of the box behind the buttons, providing visual feedback during the live drumming.

## III. Classification

### The Algorithm

The algorithm implemented in the classification step is modeled after the system proposed in [1]. This algorithm uses a separation-based approach to determine the onset times of drum kit hits. Separation-based approaches, also known as source-separation methods, separate a single-channel input signal into the distinct components that make up the signal. In this application, the input signal is the audio of the drum performance and the distinct components are the five pieces of the drum kit.

The input signal is captured by the microphone and sent through a Fe-Pi ADC at a bit depth of 16 bits and a sampling rate of 48 kHz. The digital signal is then stored in a FIFO structure on the Nucleo which continuously runs the classification algorithm. A Hamming window is applied to overlapping frames of the signal of a length of 1024 samples and a hop-size of 512 samples. The overlapping frames are converted to the frequency domain using the Fast Fourier Transform.

In order to reduce computational complexity, the signal's data is compressed by taking the summation of its spectral powers within a set of 25 frequency bands that coarsely partition the frequency spectrum. The frequency partitioning scheme initially used for classification was the Bark frequency scale. The Bark frequency scale separates the frequency range into 25 critical bands based off of human perception of auditory loudness. The Bark scale frequency band edges are [0, 100, 200, 300, 400, 510, 630, 770, 920, 1080, 1270, 1480, 1720, 2000, 2320, 2700, 3150, 3700, 4400, 5300, 6400, 7700, 9500, 12000, 15000]. This scheme was eventually adjusted to maximize the algorithm's resolution in the frequency range below 400 Hz. The adjusted frequency partitioning scheme band edges are [0, 44, 88, 132, 176, 220, 264, 308, 352, 396, 440, 510, 630, 770, 920, 1080, 1380, 1740, 2580, 4250, 6400, 7700, 9500, 12000, 15500]. A visual representation of this adjustment is shown in Figures 3 and 4. This adjustment was made because the kick drum, snare drum, and rack tom each have the majority of their acoustic energy concentrated within that range. A marginal improvement in the algorithm's ability to correctly distinguish between these pieces was observed upon adopting this altered partitioning scheme.

After performing this frequency-band summation, the algorithm uses Non-Negative Matrix Factorization (NNMF) to perform the source separation. NNMF uses an iterative heuristic to decompose a non-negative matrix  $X$  into two approximate non-negative matrix factors, the basis matrix  $B$  and the gain matrix  $G$ . In this application,  $X$  represents the time-varying spectral



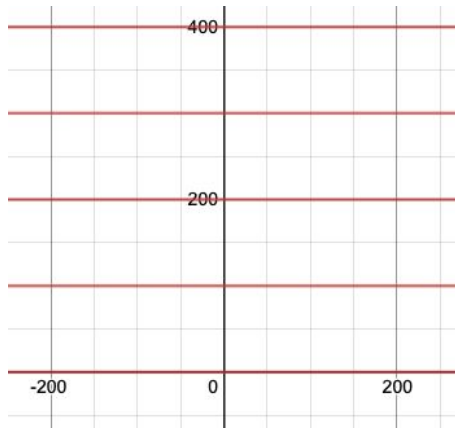


Figure 3: Bark frequency bands under 400Hz.

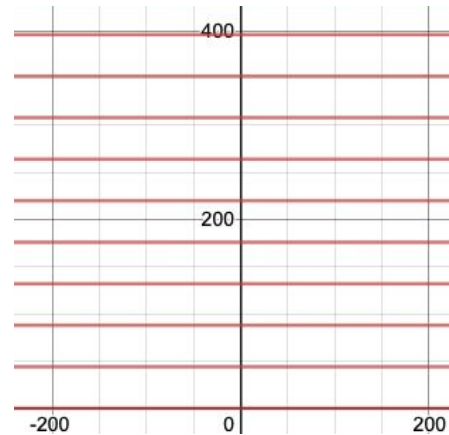


Figure 4: Adjusted frequency bands under 400Hz.

content of the drum performance,  $B$  represents “how the constituent components sound” and  $G$  represents the “time-varying amplitude envelopes” of each piece of the kit [2].

Instead of performing a blind factorization, this algorithm performs the NMF using a fixed  $B$  matrix. This provides the algorithm with prior knowledge of the spectral content of each of the five pieces of the drum kit. In order to generate the fixed  $B$  matrix, recordings of each piece of the drum kit played once in isolation are required. These recordings are manually taken every time the drum kit and microphone are set up in order to minimize deviations in spectral content due to the acoustic environment as well as any physical changes in the relative placement of the kit pieces and the microphone.

Each of these recordings, or training samples, is processed before runtime in a manner similar to the drum kit performance. The primary difference is that the basis matrix factors of the training samples are calculated by taking the average over time of each of the 25 frequency bands instead of using NMF. The decision to not use NMF for these calculations was informed by comparisons between the two approaches during the prototyping phase as well as the method implemented in [2]. The resulting basis matrix factors of each training sample are single-column matrices that are stored as the columns of the fixed  $B$  matrix used for factoring the drum kit performance.

Using the fixed  $B$  matrix, the NNMF yields a gain matrix  $G$  consisting of values representing the acoustic intensity of each kit piece at that time frame. Figure 5 plots the resulting  $G$  values for the test drum beat pattern represented by Figure 6.

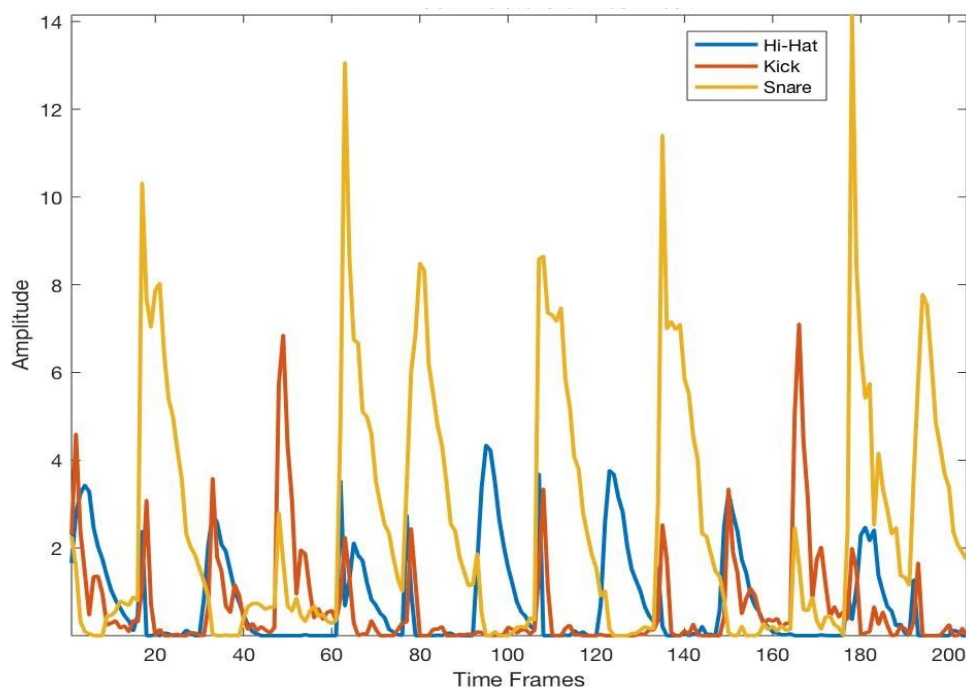


Figure 5: NNMF gain factors for test drum beat pattern.



Figure 6: Test drum beat pattern.

NNMF is an iterative algorithm which typically continues to update the matrix factors until they reach a convergence as determined by a cost function. As noted in [1], however, “ignoring the cost function’s value and forcing always ten iterations to occur, [has] no impact on the

performance” of the classification algorithm. Although observations during the prototyping phase indicated that ten iterations would not be enough to appropriately determine the G values, it was decided that 100 iterations would more than suffice. As such, the classification algorithm does not use a cost function to evaluate the convergence of the NNMF.

After running 100 iterations of the NNMF, a thresholding process is applied to the G values to determine whether or not each piece was hit. Since drum hits are characterized by a fast rise in acoustic energy, the G value for each piece must surpass a corresponding baseline threshold to be considered as a potential onset. Hitting one piece of the drum kit may also cause a sharp increase in the G value of another piece of the kit which can be erroneously mistaken for a real onset. In order to distinguish these false positive transients from the true positives, the classification algorithm evaluates the behavior of the G values after they reach relative peaks.

Multiple approaches were evaluated during the system testing phase to interpret these G values. Methods that were used but ultimately abandoned include evaluating the logarithm of the G values, the rate of their decay relative to their peak value, and the skew of the derivative of their decay. Eventually it was determined that using both the decay level and the sum of the half-wave rectified derivative of the logarithm of the G values would yield the best performance results. After 4 time frames have passed from a peak, if the G value of a piece exceeds its corresponding decay level threshold and if the sum of the half-wave rectified derivative of the logarithm of the G value over those frames is less than another corresponding threshold, then the classification algorithm determines that a true positive hit occurred for that piece. The three threshold values for each of the five pieces of the kit were tuned through a manually iterative process based upon a subjective evaluation of the classification algorithm’s accuracy. The threshold values were adjusted in an attempt to balance the algorithm’s ability to correctly identify all true positive onsets with its ability to correctly ignore all false positive onsets. Plots of all of the considered methods as applied to the hi-hat G values for the test beat in Figure 6 can be found in Appendix A.

The classification algorithm supplies the transcription algorithm with information regarding which pieces are hit and the onset times as the hits occur. While testing the system, it was observed that the onset times drifted away from a steady tempo at a consistently increasing rate. Adjustments were made to the calculation of the onset times output by the classification algorithm in order to correct for this deviation.

## Constraints and Solutions

The decision to use a single microphone to capture the drum kit performance imposed a significant constraint on the choice of a classification algorithms. As discussed in [2], the three general categories of algorithms that can be applied to this single-channel problem are source separation, template matching, and segment-and-classify. The choice of a separation-based NMF algorithm was made due to its limited complexity and the existence of multiple academic reports detailing the advantages that such an algorithm offers in solving this problem. If multiple microphones were used, other potentially-simpler algorithms could be explored since each microphone signal could be responsible for classifying an individual kit piece or a subset of the kit pieces. That being said, the introduction of multiple microphones could also introduce additional complexity as a result of synchronization and computation requirements.

During the system testing phase, the average of three single-column matrices for a single kit piece generated by three different training samples was briefly used for the fixed B matrix in an attempt to account for variations in spectral content resulting from the intensity with which a piece is hit. There was no noticeable improvement in the system's ability to classify the drum kit performance, and so for the sake of simplicity only one training sample was used for each kit piece thereafter. It should be noted, however, that the quality of the system's classification of the crash cymbal is more dependent on the training sample used to generate its column of the fixed B matrix than the other four kit pieces are for their training samples. This is likely due to the fact that the crash's acoustic properties vary significantly compared to the other kit pieces depending on how it is struck. In particular, the system yielded its best results when the crash was struck lightly on the bell of the cymbal for the training sample.

Plans were initially made to develop and implement a noise filtering step in the classification algorithm due to concerns that environmental noise could negatively affect its performance. Ultimately, these concerns were alleviated and the noise filtering step was left out. The rationale for this decision is that having the microphone placed in close proximity to the drum kit greatly reduces the impact that even loud environmental noises might have on performance. Furthermore, only noises of a percussive nature would have a likelihood of affecting the algorithm since the algorithm's thresholding process is designed to only select sounds with percussive characteristics and because persistent ambient sounds would be accounted for in the training samples.

## Testing and Results

The classification algorithm was tested by playing each piece of the kit individually and in various combinations 100 times at a constant tempo. The total percentage of correctly classified hits or pairs of hits are summarized in Figure 7.

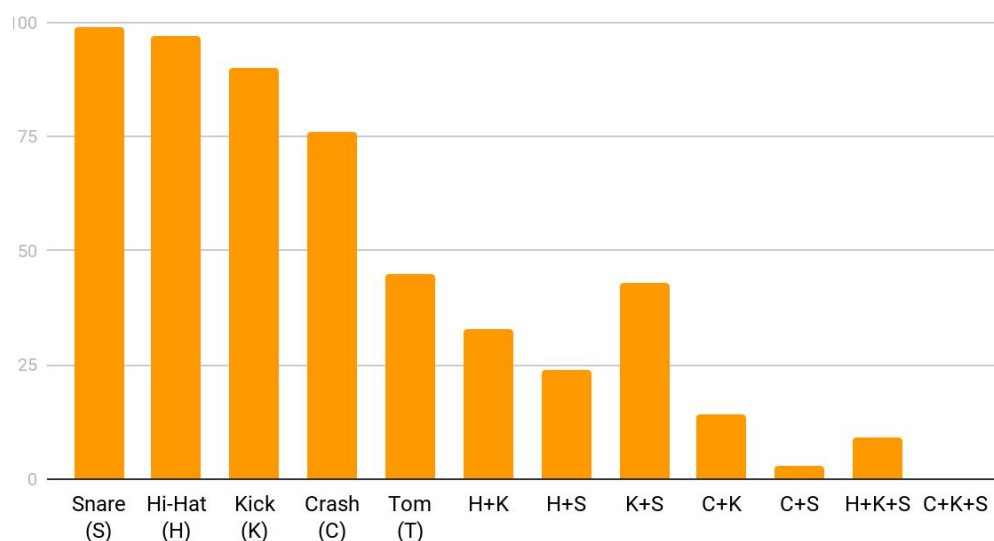


Figure 7: Total percentage of correctly classified quarter note hits at 120 BPM.

As these results indicate, the classification algorithm performs best when pieces of the kit are played individually. The performance for the crash and tom are less accurate than for the snare, hi-hat, and kick. This may possibly be due to the longer decay characteristics of the former pieces in comparison to the sharp transient nature of the latter pieces. Performance significantly deteriorates when multiple pieces of the kit are played simultaneously. This is primarily due to the algorithm's difficulty recognizing the hi-hat and crash hits, whereas it is able to fairly accurately classify the snare and the kick. The full results of this test can be found in Appendix B.

It should be noted that the algorithm's performance, and thus any test results, are highly dependent on the tuning of the G value thresholds. The test results presented in Figure 7 were recorded following a long process of tuning these thresholds to the point where they were qualitatively deemed to yield sufficiently accurate classification results. Additionally, the

algorithm's performance depends a great deal on the placement on the microphone relative to the drum kit. The results above were recorded after placing the microphone low to the ground, near the kick drum, and pointed up towards the snare. This position was settled upon because it was qualitatively observed to produce the best overall results for each of the five kit pieces. For comparison, testing results recorded using a different microphone placement are included in Appendix C.

These testing results illustrate both the current abilities and limitations of the algorithm as well as those inherent in the problem domain. While the algorithm does demonstrate positive potential, further study and testing are required in order to clearly distinguish algorithmic challenges from external issues and to refine the algorithm's performance. Drawing upon an intuitive understanding of how the system may be used could offer insight into smaller opportunities for improvement. As an example, some combinations of pieces, such as the hi-hat and crash, are unlikely to occur simultaneously. Knowing this, the algorithm could be modified so that the onset of only one of these pieces is classified at any particular time. Although this change would impose an artificial limit on what a user may perform, it has the potential to greatly improve the algorithm's performance.

## IV. Transcription

### The Algorithm

The transcription algorithm takes in hit onsets determined by classification and maps those inputs to standard drum kit sheet music. First, a terminology will be defined to help describe this process:

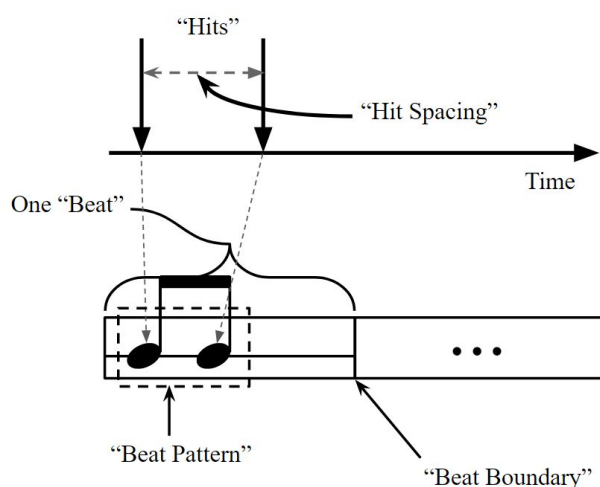


Figure 8: Example beat diagram.

- *Hit*: An onset of any instrument in the drumkit.
- *BPM*: The number of beats per minute.
- *Beat*: A unit of time that corresponds precisely to one quarter note at a given BPM.
- *Beat Pattern*: One possible combination of hits that occurs in a specified amount of time.
  - *Beat Pattern Length*: The “specified amount of time” noted above.
- *Beat Boundary*: The time of the start of a beat.
- *Resolution Limit*: The smallest recognizable partial that is considered by the transcription algorithm (e.g. 16th note, 32nd note, etc.).

- *Prior*: The probability that a given beat pattern occurs.
- *Hit Spacing*: The amount of time between two hits. If the hit is the first to occur following a beat boundary, the hit spacing is the amount of time between the hit and the beat boundary.

Before runtime, a set of possible beat patterns is decided and the prior associated with each beat pattern in that set is determined. During runtime, incoming hits are compared to the predetermined beat patterns, and the best pattern is calculated. The resulting beat patterns are sent to the visual feedback system that is described in detail in Section V.

## *Pre-Runtime*

Before runtime, all possible beat patterns are listed and the likelihood of each is calculated. One challenge was determining what scale of beat patterns would be used. To illustrate this tradeoff, consider a beat pattern length of one beat. This set consists of patterns such as a single quarter note hit, two eighth note hits, an eighth note rest followed by an eighth note hit, etc. Now, assume that the limiting resolution of the system is sixteenth notes. Then, this set of beat patterns can be estimated by noting that each sixteenth note can be either ‘on’ (a hit) or ‘off’ (a rest) which means  $2^4 = 16$  possible combinations are allowed. To fully calculate the set of beat patterns, the triplet pattern below the resolution limit (e.g. eighth note triplets in this case) must also be considered. Here, this would add  $2^3$  additional patterns, as well as a handful of patterns consisting of both eighth note triplets and sixteenth notes. Excluding the patterns with both triplets and sixteenths, a total of  $2^4 + 2^3 = 24$  possible beat patterns are included. Now, as an alternate case, consider a beat pattern length of two beats. In this case, there are 8 possible sixteenth notes and 6 possible triplets. Then, the total number of beat patterns for a beat pattern length of two beats is approximately  $2^8 + 2^6 = 320$ , again excluding the patterns with both sixteenths and triplets. Although the beat patterns with both sixteenth and triplets are excluded, the important result here is that the total number of possible beat patterns will increase exponentially as a function of the beat pattern length. Since the algorithm is limited by real time computation constraints it is favorable to consider smaller beat length patterns. However, the system accuracy will decrease as the beat pattern length decreases because less information is available. This intuitive result will be justified once the method for *prior* determination is discussed. Then, there is a tradeoff between computational complexity and system accuracy associated with the predetermined beat pattern length.

For this project, a beat pattern length of exactly one beat was determined to be the most logical and realizable model. As shown above, the number of possible beat patterns for a beat pattern



length of one beat is on the order of 24 patterns. This is certainly manageable in real-time computation as 24 comparisons is negligible compared to the time scale of drumming, even at speeds such as 240 BPM. Also, it was theorized that most drumming rhythms would be distinguished over the scale of a single beat. One main consideration is whether triplets and sixteenth notes can be distinguished accurately, and since both of these partials occupy less time than one beat, both should be transcribed accurately. On the other hand, a set of quarter note triplets will occupy more than one beat length which means it will be difficult to transcribe these properly without a longer beat pattern length. Despite this, it was decided that a longer beat pattern length would result in significant computational delay and would not be worth the slight improvement in accuracy. Therefore, a beat pattern length of one beat was used for the transcription algorithm.

The next major challenge in beat pattern considerations was determining the resolution limit. In a similar calculation as above, it can be shown that the computation complexity increases exponentially as the resolution limit becomes finer. It was initially thought that 32nd notes would be needed to fully capture the complexity of typical rhythms because 32nd notes are often found in drumming. However, due to limitations in the classification algorithm's time resolution, it was later decided that 16th notes would be sufficient. At this stage, the set of allowed beat patterns is specified as those occurring over a beat pattern length of one beat with a resolution limit of sixteenth notes.

Next, the priors were determined for each possible beat pattern for use as a weighting factor. [3] proposes a method for tempo tracking in which the probability of note occurrence is proportional to  $e^{-d(c_k)}$  where  $d(c_k)$  is the number of bits required to represent the beat pattern  $c_k$ . In this project, the beat probability was calculated as the average of the base 2 logarithm of the denominators of each hit in the beat pattern. Consider a beat pattern consisting of two sixteenth notes followed by an eighth note, then the prior is calculated as shown:

$$\begin{aligned} \text{Beat Pattern Representation: } & \left[ \frac{1}{16}, \frac{1}{16}, \frac{1}{8} \right] \\ \text{Average } \log_2: & \frac{1}{3}(\log_2(16) + \log_2(16) + \log_2(8)) = \frac{1}{3}(4 + 4 + 3) \approx 3.67 \\ \text{Probability: } & \exp\left[-\left(\frac{1}{p_{const}}\right) * 3.67\right] \end{aligned}$$

The  $p_{\text{const}}$  term is a constant that can be used to tune the prior of each beat during the pre-runtime phase. This method of prior assignment gave a realistic representation of what may be expected from a drummer; that is, quarter notes are very probable whereas sixteenth note patterns are less likely.

It should be noted here that for a larger set of considered beat patterns, the exponential distribution helps to discriminate against unlikely (more complex) beat patterns. The most complex beats will consist of several partials at the resolution limit. However, when the beat pattern length becomes smaller and approaches the resolution limit, the set of beat patterns will tend towards only containing the resolution limit partial. In this case, the prior will tend towards one for that limiting partial which means that no information can be gained by the pre-calculated prior. This is a concrete justification for the loss of accuracy associated with a decreasing beat pattern length.

## *Runtime Computation*

After all of the pre-runtime considerations are computed, the transcription algorithm analyzes incoming hits by comparing the hit spacings to those in the possible beat patterns of length one beat. The hit spacings for each beat pattern in the set of possible patterns is known before runtime and this information is made available. During runtime, each time a hit is detected, the hit spacing is calculated. Whenever a new beat boundary is observed, the hit spacings from the last beat are compared to the hit spacings of every beat pattern in the set of available beat patterns. Since the number of hits is known, then beat patterns with just the same number of hits are considered. The differences between the known hit spacings and the calculated hit spacings are computed to determine how similar the incoming beats are to each possible beat pattern. Finally, given this information, the most likely beat pattern is determined by inverting the distance and multiplying by all note probabilities. The distances are inverted so that a smaller distance implies a large likelihood. Figure 9 graphically summarizes this process.

One additional feature of the algorithm is a “rounding” of beats to handle any errors around beat boundaries. Since beats are considered in discrete chunks, unexpected behavior can occur near the end or start of beat. For example, timing uncertainty associated with the system non-idealities will cause beats to be distributed in time. Then, for a hit expected to occur exactly at the start of a beat boundary, there is a chance that it will occur before the beat boundary thus appearing in the previous beat. To minimize transcription errors, a check is implemented to determine if a beat is very close to the start of the next beat. The distance of the hit onset from the next beat is compared to the distance to the actual quantized hit, and if the hit is closer to the

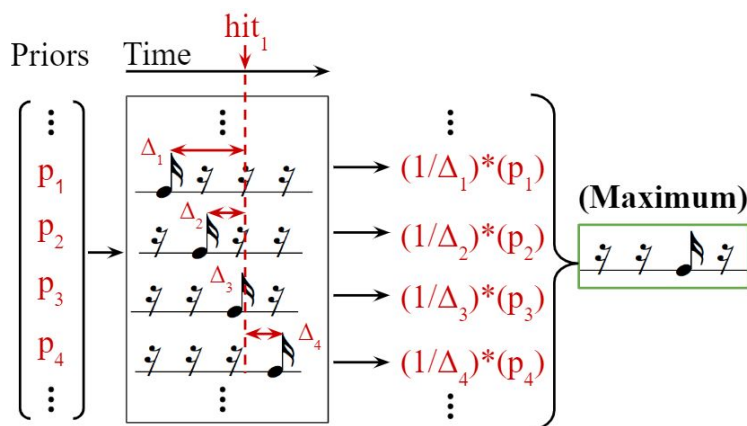


Figure 9: An overview of the transcription algorithm steps.

next beat then it is placed at the start of the next beat. This illustrates the challenge associated with analyzing discrete beat chunks as opposed to full inputs.

## Constraints and Solutions

Due to the runtime and memory constraints imposed on the entire processing chain, some analytical precision had to be sacrificed in the transcription algorithm. In particular, most transcription algorithms found in literature process an entire set of hits and analyze the most likely quantizations of the pattern by generating a number of possible outputs. In real time processing, this depth of analysis cannot be achieved as only subsets of hits can be analyzed at a given time. As a result, lower accuracy levels should be expected as only subsets of the entire input can be analyzed as it is detected.

Additional constraints were placed on the tempo and meter assumed by transcription. Since quarter notes at 120 BPM are equivalent to eighth notes at 60 BPM, various accurate transcription outputs are possible if the tempo is not fixed. If a tempo is not specified then the transcription algorithm would be responsible for tracking and estimating the tempo in real-time. This task was deemed out of scope due to its theoretical complexity. Similarly, a variable meter implies that the transcription algorithm would have to estimate the most likely meter given the inputs. Again, this estimation process was ruled out and a fixed meter of common time, or four quarter notes per measure, was always assumed. Therefore, a fixed tempo and meter were always decided before runtime and the beat patterns, priors, and spacings were calculated based on these parameters.



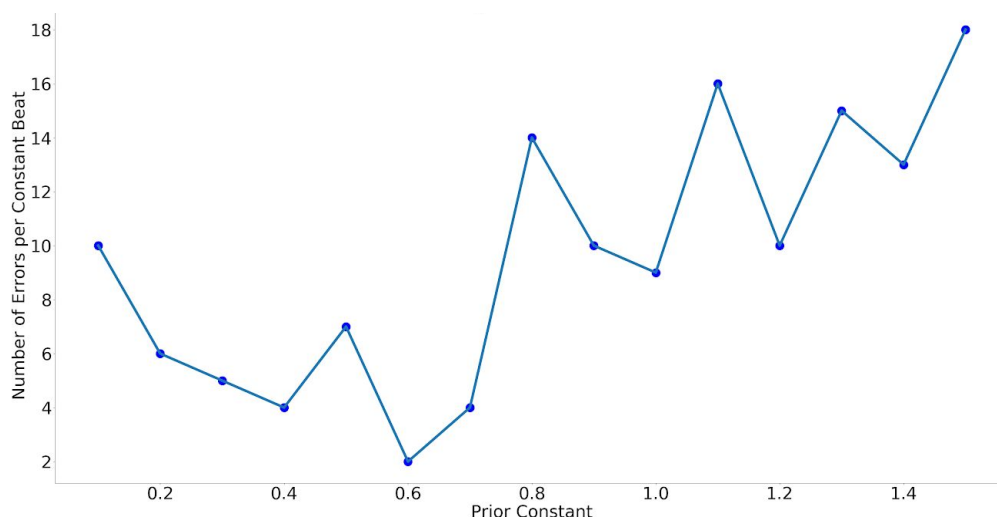




Figure 11: Transcription algorithm sensitivity to  $p_{\text{const}}$ .

uncertainty which may have induced larger errors than expected in the prototyping phase. Nonetheless, it was seen that, as the values of  $p_{\text{const}}$  were tuned lower, an optimum value occurred as the majority of rhythms were detected correctly. In the best case, two errors occurred and

typically these were issues in distinguishing patterns such as  and . This is unsurprising as these two beat patterns sound very similar, and slight variations in user playing may cause error in the algorithm. This underscores a fundamental limit of the algorithm which is that even to the human ear some rhythms sound very similar in certain contexts and therefore if the player cannot properly distinguish these rhythms then it will become very difficult for the algorithm to distinguish the rhythms since the algorithm can only reflect what the user plays. This testing showed the limits of the transcription algorithm and also revealed the potential accuracy of the algorithm if the  $p_{\text{const}}$  is tuned properly.

## V. GUI

### The Algorithm

The GUI is implemented using C and the Simple Directmedia Layer (SDL) [4]. In order to render sheet music notation, the Guido Music Notation library is used, as it is a powerful framework that supports the rendering of all music rhythms needed for the project, including triplets and sixteenth notes. Figure 12 offers an example of what the GUI looks like.



Figure 12: Downbeat GUI output when receiving real-time input.

The Guido library takes in a proprietary .gmn file that describes the notes and lengths from the output of transcription, and then creates an SVG markup file displaying the notes when opened in a browser. Guido renders this by using a special “Guido Font” library and placing notes, articulations, bars, and staff markings at specific x and y locations in the SVG. As SDL only has a simple image rendering library (SDL\_image), it does not support rendering of SVG images and only supports more basic formats like .jpg and .png. In order to mitigate this issue, conversion from SVG to PNG is needed. Instead of re-developing an SVG parser, a Python implementation of SVG to PNG conversion is used called PyCairo. After converting to a PNG, the resultant image is then rendered on the screen. A diagram of this pipeline is shown in Figure 13.

The process of re-generating .gmn files, converting to SVG, converting to PNG, and rendering is necessary every time a new input is received. All the steps required in the GUI pipeline are also sequential, causing the system to slow down. To help prevent too much slowdown, a simple change to the system was made to only re-run the render pipeline when a new beat is received. Furthermore, all rendering is pushed to a separate thread from the transcription system, allowing

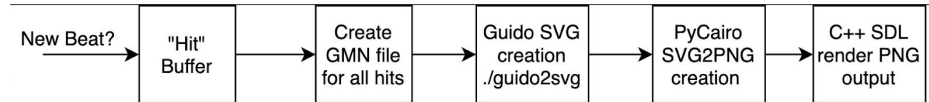


Figure 13: Guido rendering pipeline.

the UART communications from the Pi and the Nucleo to be separate from the necessary rendering calls.

## Constraints and Solutions

### *Long Inputs*

As the system is run for increasingly longer inputs, each new beat makes the rendered .gmh file larger, which in-turn makes the SVG take longer to render and the PNG take longer to convert. While initial results from small inputs showed that it would only take around 60ms to run the entire GUI pipeline, as the inputs got larger the lag became very noticeable. Because of the way .gmh files are made, there is no way to simply append to the file when a new beat is discovered; the .gmh files has to be re-generated for every new beat received, which contributes to the slowdown.

Furthermore, there is only so much real-estate on the screen the Raspberry Pi could output. As inputs got too large, the score displayed would run off the screen and be unreadable. Because of these two issues, it was decided that input sizes would be limited to 4 lines of Guido output, as any more lines would not be rendered properly. This reduces the need to develop any sort of system for the GUI to handle multiple pages of scores or scrolling, and also ensures that users will not have to wait for long periods of time after their last hit for the render to actually display. Lastly, this helps reduce the amount of time in the audio playback step, as smaller inputs mean a smaller final audio waveform to synthesize.

### *Playback Synthesis*

Since the audio synthesis is written in Python using resource heavy packages like Numpy and Pandas, the playback synthesis step is considerably slower than other steps. Therefore, it is imperative for a visual feedback system to exist that shows the user that the system is in-fact working but just taking some time to complete the synthesis step. To do this, a loading bar was

added to the synthesis step (see Figure 14). Since the synthesis step is actually a Python script call, there is no straightforward way to identify how far along the progress of the script is to accurately display a loading time. Therefore, loading times are extrapolated by fitting a line to the delay given the input number of beats. Appendix D shows the considerations used to predict how far along the synthesis is.



Figure 14: Loading Bar during audio playback synthesis.

## *Complicated Inputs*

As it is common for drum scores to consist of components with rhythms at any given beat, the possibility for some components to have various types of rests while other components are “hit” is very high. This results in the left image in Figure 15, where the Guido output is very cluttered and difficult to read. A considerable amount of processing is necessary to “hide” notes that are rests when there are other notes being rendered in the same beat. After this processing, the Guido output is much more parseable, as demonstrated in the right image in Figure 15. More information on how this parsing is done is shown in Appendix E.

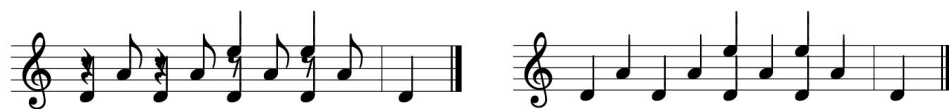


Figure 15: Difficult to interpret Guido (left) and the processed solution (right).



## VI. Audio Playback

### The Algorithm

The audio playback algorithm provides a means to playback the transcription results from a set of audio samples. The set of samples to generate the audio were originally used for training the classification algorithm. A Python script is used to handle the audio playback functionality. The playback algorithm works by first importing the Guido output in the form of a CSV containing onsets. This is imported initially as a dataframe from the Pandas software library. For  $N$  instruments, a mapping is then created between a set of integers from 0 to  $N - 1$ . From this mapping, it is then possible to extract the instruments classified at each onset into the form of a Numpy array, as well as doing likewise for the quantized onset times.

Once these critical data values have been extracted, it is then necessary to manipulate the data into a form which is amenable to the process of creating a new audio playback file. A new Numpy array representing the playback data is formed and initialized as a zeros matrix which has a length spanning the last onset's time plus the maximal time elapsed in any given recorded sample. These time lengths are measured in terms of samples, which is necessary to be consistent with the input and to produce an output at the same sample rate.

The playback audio is then created using the newly formed arrays for onset times, hit classifications, and playback. A loop is specified of range equaling the number of onsets. In this loop, a set of  $N$  conditional statements for each instrument are evaluated. If a given instrument was played, this triggers the conditional statement, which in turn triggers a for loop of range equaling the length in samples of said instruments recording. In this loop, the playback array is aligned at the onset time with the sample's first value, and start adding a coefficient augmented multiple of the sample to each space in the playback array within the length of the sample, corresponding to the onset start plus the current position in the loop.

Once this has been done for all onsets, the Numpy array representation of the transcribed beat has been successfully created, representing audio data in which each onset time corresponds to a triggering of the given instruments sample. One of the big draws for using Python in this algorithm is that once this array had been created, the SciPy library provides a function call which allows the array to be encoded into the .wav audio format.

## Constraints and Solutions

Once the base playback algorithm has been established, challenges included mixing the samples when creating the playback file. Before determining the appropriate coefficients the sound quality of the playback was suboptimal. It is necessary to apply a coefficient to all instruments which scales the samples to a small fraction of the original data values, as too much overlap distorts the file. Then it is necessary to optimize the coefficients with respect to one another such that the mix is pleasing to the ears. To put this into perspective, the kick's coefficient is five times that of the snare, which in turn is twice that of the hi-hat. It is also necessary to add the function call to the C codebase to trigger this playback, and it is pertinent to ensure that the output .csv is available when this call is made.

## VII. 3D Printed Drum Kit

To create a compelling display for presentations, a 3D model drum kit was printed with color coded LEDs mounted on the pieces of the kit. Numerous kit designs were considered, some requiring greater preprocessing, which could be accomplished with a program like Meshmixer. Ultimately, there were many issues to consider when sending the design to print, utilizing the Cura software and the Ultimaker printer in tandem. The desired 3D model is rather intricate and has a multitude of pieces aside from just the drums and cymbals, such as mounting components, pedal hardware, and stands. The relevant hardware components were printed and then anything that required mounting was manually attached.

### Constraints and Solutions

The biggest constraint in regards to 3D printing was only being able to reserve one printer for one day. This is particularly challenging because 3D printing is a time intensive process, and if satisfying results were not obtained from the first print, an alternative option would likely have had to have been considered for the presentation display. This necessitated scaling down all of the drum kit pieces intended for use, transforming orientation such that it was possible to fit all of our kit pieces onto the 8x8x8" print bed, while providing adequate support while printing, which is vital to the integrity of any given piece.

It was also necessary to ensure that there was no spillover of resin from any one piece to another, requiring careful placement in the testbed model within the Cura software. Finally, tradeoffs were made in regards to print quality versus print time. Fortunately, Cura provides a multitude of functionalities for previewing prints and made it possible to get a good idea of the final result while printing.

The 3D print was ultimately successful, and the next step of attaching the LEDs to the constituents of the kit was in order. LEDs for one instrument were soldered in strings to a positive and negative wire. Then, for each instrument these LED strings were hot glued to the 3D print. The outgoing wires were all routed together into the box to be connected to their respective GPIO pins.

## VIII. Summary of Constraints

User error and variability in drumming speed and intensity needed to be considered when implementing the system's algorithms. The three types of classification thresholds for onset detection were selected and tuned in order to optimize the algorithm's performance for what was considered to be the most likely level of variation in user performance. Similarly, the transcription  $p_{\text{const}}$  value was set to minimize error that may arise from the algorithm and not from the user. Although both of these decisions reduces the system's ability to properly handle complex user input, this limitation was deemed appropriate given the improvements that they yielded for the average use case.

Additional design compromises that were made include limiting the tempo step size to three of four beats per minute and synthesizing the audio for playback after the performance only if the user selects to do so. Each of these compromises has a practical justification. The limitation of the tempo step size was due to noise inherent in the potentiometer used to set the tempo which could produce errors in the transcription algorithm if this restriction was not imposed. The playback file is rendered using a Python script which is fairly slow. It makes sense that this script is called only when prompted by the user after the performance since it would otherwise add significant latency to the process of updating the transcription display.

## IX. Milestones

### Milestone 1

Goals for the project which were set for and achieved by Milestone #1 include prototyping the classification and transcription algorithms in Matlab; taking separate recordings of the five drum kit pieces and appropriate environmental noise and using combinations of the kick, snare, hi-hat and noise recordings for testing the classification algorithm; producing a mock-up of the GUI visual feedback display; and determining the optimal combination of components for implementing the system architecture.

While working in preparation for Milestone #1, it was decided that the noise filtering step of the classification algorithm would not sufficiently improve the algorithm's performance in order to justify its inclusion. It also became clear that both algorithms' accuracy, memory usage, and inherent limitations could only be properly assessed once they were implemented, tuned, and tested on the system's hardware. Although mock-ups of the transcription playback and the 3D model visual feedback were not completed by Milestone #1, they were both finished within the two weeks following Milestone #1.

### Milestone 2

All goals set for Milestone #2 were achieved in time, which included completing a fully functional end-to-end system and designing a draft of the poster required for Design Expo. In between Milestone #1 and Milestone #2, it became apparent that the system's memory usage did not produce any tradeoff with the accuracy of the system's algorithms. It was also decided that rudiments could not be reliably classified using the current approach. Instead of pursuing this goal by implementing a new classification algorithm, this goal was abandoned in favor of supporting the classification of distinct drum onsets performed at tempos of up to 240 beats per minute.

## X. Demo

Prior to demonstration at the Design Expo and at the final presentation, the components of the system were connected and the drum kit and microphone were arranged in a standardized manner on a 4x6' rug. The rug was used both to reduce any undesired acoustic reflections from hard floor surfaces and to set exact markers for the placement of the drum kit and the microphone stand. Once set up, new training samples were recorded to generate a fixed B matrix for the classification algorithm and the various classification thresholds were adjusted to optimize its performance.

Demonstration of the project involved a member of the team showing observers how to set the system's tempo, start and stop the system's transcription process, and play back the transcription results. The team member would play the kit to demonstrate the system's ability to recognize each of the pieces of the kit, as verified by the flashing LEDs mounted on the 3D model and the notation generated on the GUI monitor. Observers were encouraged to take a turn playing the drum kit so that they could independently evaluate the system's performance.



Figure 16: Photo of Downbeat setup at Design Expo.

## XI. List of Parts

- Parts from the lab
  - STM Nucleo F747ZI
  - Raspberry Pi 4
  - Fe-Pi Audio Z V2 codec (2)
  - Power strip
  - Jumper wires
  - Mini breadboard (3)
  - 3.5mm male to dual RCA male cable
  - Tape
  - Wires
  - RPi cables/converter for HDMI
  - Monitor
- Parts purchased for the project
  - 200x120x75mm plastic box
    - [https://www.amazon.com/LeMotech-Plastic-Electrical-Project-Junction/dp/B07D23BF7Y/ref=sr\\_1\\_11?keywords=black+plastic+box&qid=1574352593&sr=8-11](https://www.amazon.com/LeMotech-Plastic-Electrical-Project-Junction/dp/B07D23BF7Y/ref=sr_1_11?keywords=black+plastic+box&qid=1574352593&sr=8-11)
  - 30mm button (5)
    - <https://www.adafruit.com/product/473>
  - Panel Mount Right Angle 10K Potentiometers
    - <https://www.adafruit.com/product/2046>
  - Adafruit LED Sequins (5)
    - <https://www.adafruit.com/product/3377>
  - 3D-printed drum kit model
    - the reservation fee for using a 3D printer in the Duderstadt Center's Fabrication Studio was covered by the team budget.  
<https://www.thingiverse.com/thing:1229824>
- Parts provided by the team
  - Drum kit consisting of kick drum, snare drum, rack tom, hi-hat, and crash cymbal
  - Drum sticks (2)
  - 4x6' rug
  - Shure SM58S cardioid dynamic microphone
  - Samson MK10 boom microphone stand

- 20' XLR cable
- Samson MDR1248 12-channel analog audio mixer
- Technics RP-DJ1200 headphones
- Edifier R1280T powered speaker



## XII. Contributions of each member of team

Team member	Contribution	Effort
Damon Anderson:	Transcription, Hardware/Integration	20%
Michael Halloran:	Classification, LEDs, Audio Equipment	20%
Catherine Kenzie:	Classification, Hardware/Integration	20%
Prakash Kumar:	Transcription, GUI, Hardware	20%
Bryan Rabotnick:	Playback, 3D Printing, Classification	20%

## XIII. References and citations

- [1] G. Papanikas, “Real-time Automatic Transcription of Drums Music Tracks on an FPGA Platform,” MSE dissertation, Dept. Informatics Mathematical Modelling, Univ. Denmark, Lyngby, Denmark, 2012.
- [2] C. Dittmar and D. Gärtner, “Real-time Transcription and Separation of Drum Recordings Based on NMF Decomposition,” in *Proceedings of the 17th International Conference on Digital Audio Effects (DAFx)*, 2014.
- [3] Cemgil, A. T., and B. Kappen. “Monte Carlo Methods for Tempo Tracking and Rhythm Quantization.” *Journal of Artificial Intelligence Research* 18 (2003): 45–81. Crossref. Web.
- [4] “About SDL.” *Simple DirectMedia Layer - Homepage*, <https://www.libsdl.org/>.

## XIV. Appendices

### Appendix A: Classification Plots

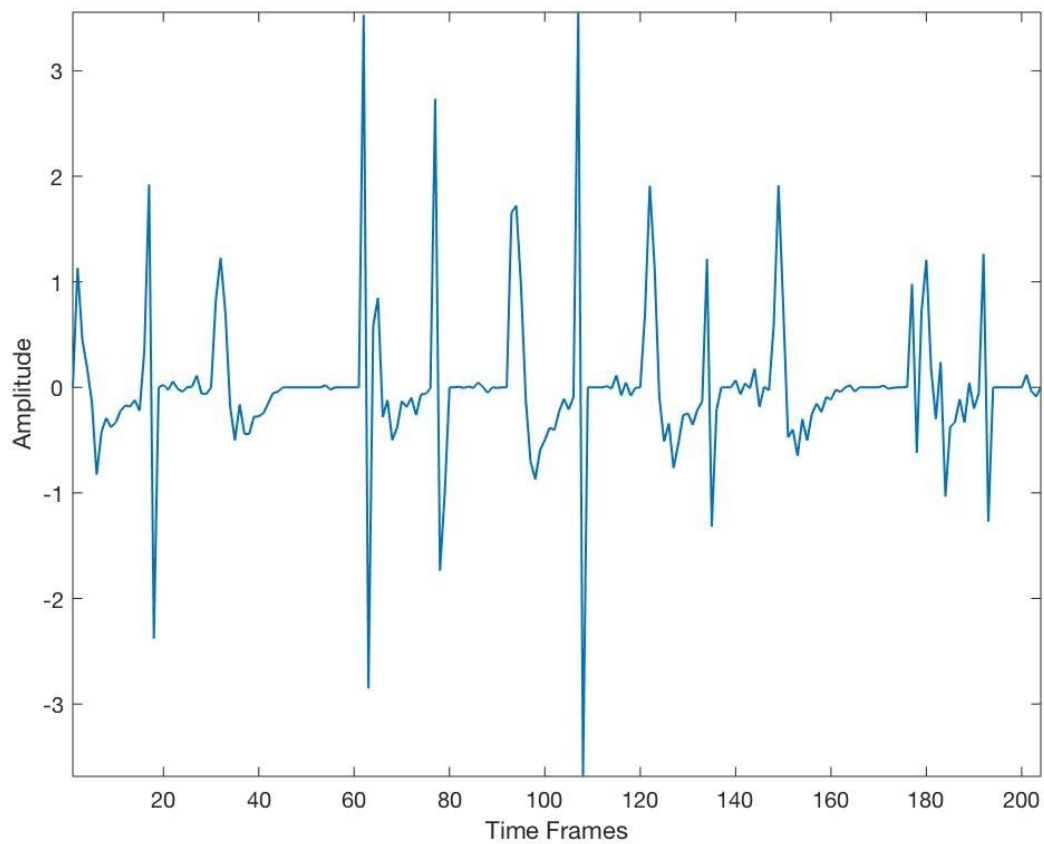


Figure A.1: Derivative of the hi-hat G values for the test drum beat pattern.

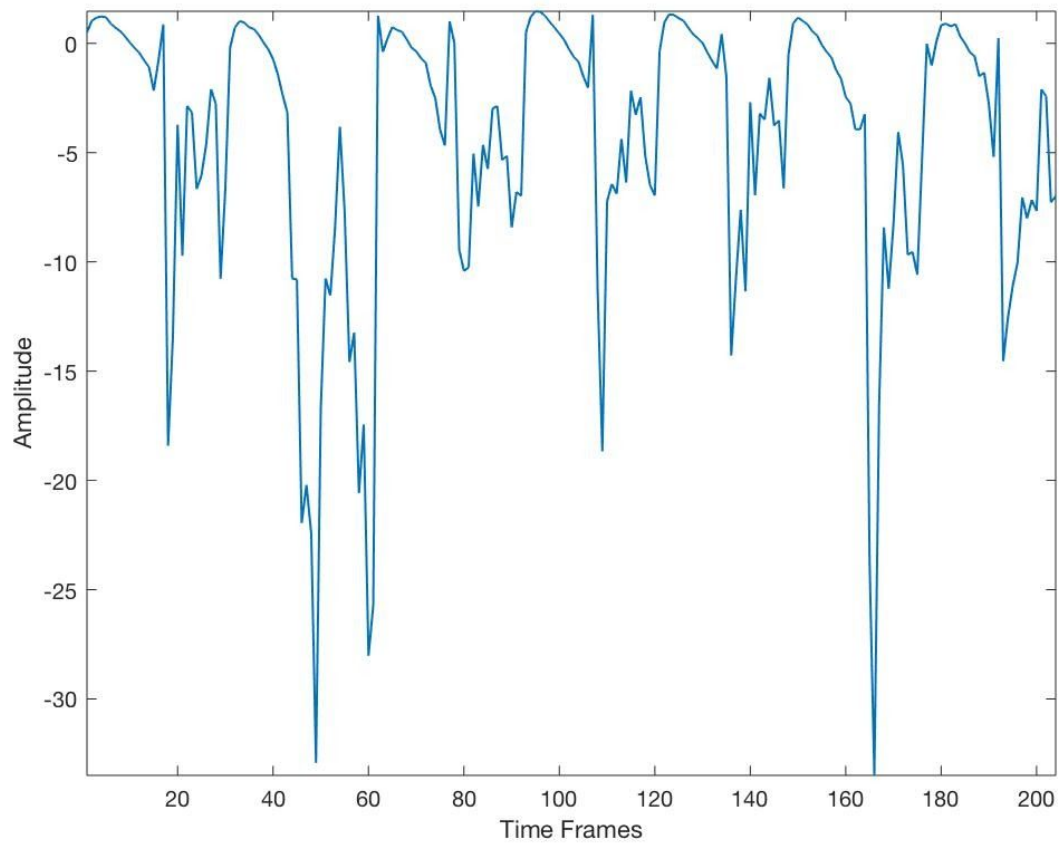


Figure A.2: Logarithm of the hi-hat G values for the test drum beat pattern.

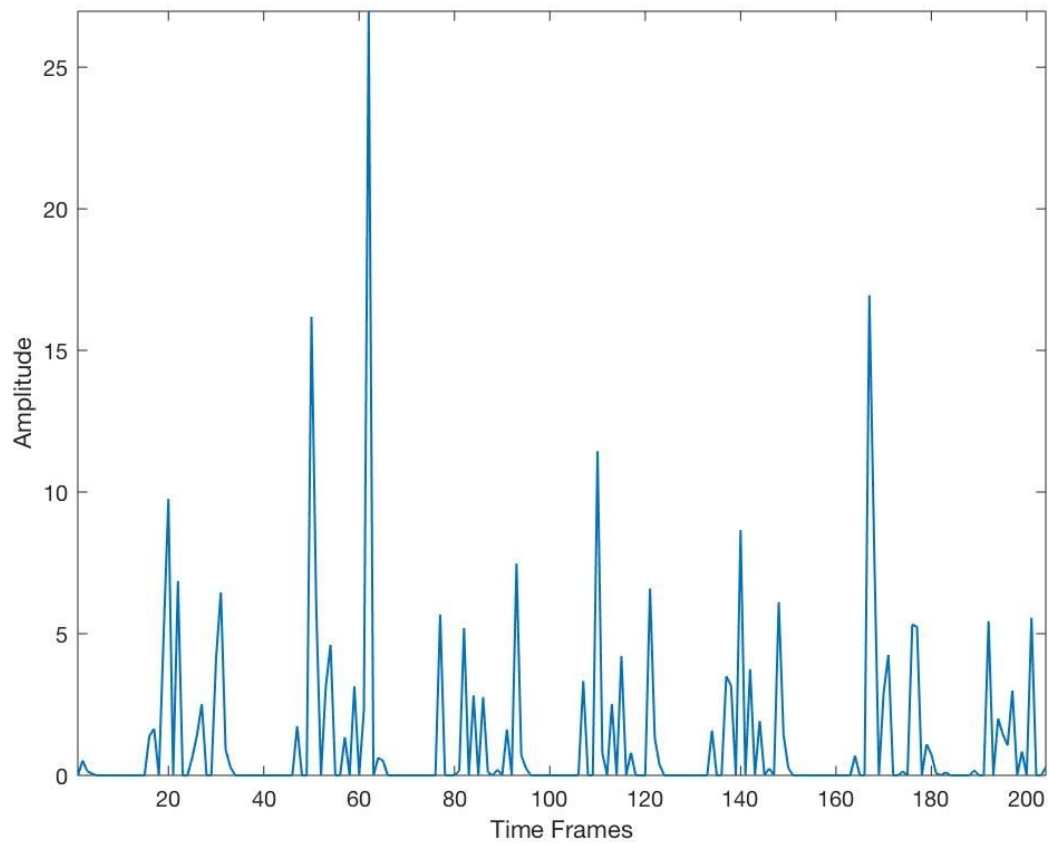


Figure A.3: Half-wave rectified derivative of the logarithm of the hi-hat G values for the test drum beat pattern.

## Appendix B: Up-to-Date Classification Results

Table B.1 contains the testing results of the classification algorithm using the most up-to-date microphone placement and thresholding tuning.

Piece(s)	Piece #1	Piece #2	Piece #3	False H	False T	False S	Total
Hi-hat (H)	97	N/A	N/A	0	0	0	97
Snare (S)	100	N/A	N/A	1	0	0	99
Kick (K)	91	N/A	N/A	0	1	0	90
Tom (T)	54	N/A	N/A	0	0	10	45
Crash (C)	87	N/A	N/A	10	0	3	76
H+K	35	94	N/A	1	0	2	33
H+S	29	94	N/A	0	0	0	24
K+S	49	100	N/A	6	3	0	43
C+K	19	94	N/A	11	1	4	14
C+S	6	96	N/A	11	0	1	3
H+K+S	14	71	96	0	0	0	9
C+K+S	0	61	86	8	3	0	0

Table B.1: Up-to-date classification test results.

Each piece or combination of pieces were played 100 times in a row at a tempo of 120 beats per minute. The table details how many times the classification algorithm correctly detected an onset by the piece(s) as well as how many times a false onset was detected, including the false detection of two onsets for one true onset. The number of times that the classification algorithm simultaneously identified all of the pieces that were played without error is recorded in the Total column. Only combinations of pieces that were deemed likely to be played by drummers were tested.

## Appendix C: Previous Classification Results

Tables C.1, C.2, and C.3 contain the testing results of the classification algorithm at three different tempos using a different microphone placement and thresholding tuning than those used in Appendix B.

Pieces(s)	Piece #1	Piece #2	Piece #3	False H	False K	False S	Total
Hi-hat (H)	100	N/A	N/A	0	0	0	100
Snare (S)	100	N/A	N/A	0	1	0	99
Kick (K)	56	N/A	N/A	0	8	1	48
H+K	96	55	N/A	0	6	1	46
H+S	28	96	N/A	0	0	0	24
K+S	26	93	N/A	0	0	0	23
H+K+S	56	29	89	1	1	0	13

Table C.1: Previous classification test results at 60 BPM.



# Downbeat Final Report

EECS 452  
Fall 2019

Piece(s)	Piece #1	Piece #2	Piece #3	False K	False S	Total
Hi-hat (H)	100	N/A	N/A	0	0	100
Snare (S)	100	N/A	N/A	1	0	99
Kick (K)	68	N/A	N/A	10	0	58
H+K	93	81	N/A	18	3	58
H+S	29	100	N/A	0	0	29
K+S	31	96	N/A	1	0	29
H+K+S	7	18	99	0	0	3

Table C.2: Previous classification test results at 120 BPM.

# Downbeat Final Report

EECS 452  
Fall 2019

Piece(s)	Piece #1	Piece #2	Piece #3	False K	False S	Total
Hi-hat (H)	99	N/A	N/A	0	0	99
Snare (S)	99	N/A	N/A	2	0	98
Kick (K)	67	N/A	N/A	0	0	67
H+K	32	57	N/A	0	5	21
H+S	13	100	N/A	0	0	13
K+S	55	86	N/A	0	0	51
H+K+S	18	41	90	0	0	7

Table C.3: Previous classification test results at 235 BPM.

These tests were performed before submitting the poster for Design Expo. At that point, only the kick, snare, and hi-hat were adequately tuned and tested. Each piece or combination of pieces were played 100 times in a row. The table details how many times the classification algorithm correctly detected an onset by the piece(s) as well as how many times a false onsets was detected, including the false detection of two onsets for one true onset. The number of times that the classification algorithm simultaneously identified all of the pieces that were played without error is recorded in the Total column.

## Appendix D: Loading Screen Implementation

When it came to implementing the loading screen, it was first thought to use the index of the loop / length of the loop of the synthesis as a percentage for how far in progress the script was. This removes the amount of time needed to import packages and write the output file itself in the estimation, but takes into account the linear relationship between size of input and length of run time. Therefore, this seemed like an appropriate estimate to use, as when inputs became very large, the time for other processes becomes negligible in comparison.

While obtaining the index of the script while it was running was relatively easy in the Python script itself, getting the C code to know the status of the Python script became difficult. Because both operations were running on the same thread, there had to be some two-way communication method between both scripts. Therefore, reading/writing to a file was out of consideration, as the Python script could not simultaneously be writing to the file while the C code was reading from it.

One idea that was brought up was to “create” files during the Python implementation as progress passed set thresholds. The idea was to essentially create an arbitrary number of set files (say 10) that, when created, meant that some percentage of the synthesis was complete. If there were 10 files, then after the first file was created, the program was at 10% completion, the second file meant 20%, etc. While this method would create an accurate (although considerably discrete) calculation rather than an estimate, the method would require a considerable amount of development. Since the team was on a tight timeline, this method was dropped by utilizing the much simpler method of extrapolating load times based on a line of fit for input beat size.


In order to estimate loading times, beats of three different sizes were taken and timed using the Python timer function. After this, a line of best fit was made with the three different beat sizes and then used to extrapolate all future load times. The reason for taking three data points was to remove any outliers from only selecting two data points, without having to construct larger experiments which would have taken more time. It was found that the three points did form a fairly linear fit, and worked pretty well for overall system use. In the average case, the system took longer on busier beats with many components hit and shorter for times when only one component (such as the snare) was hit, but this estimation did not take into account this. Instead, a multiplier of a guess of how many components may be hit was added to the slope of the linear fit, and tested until it “felt right.”

While this estimate was actually fairly off of the actual loading time, simply having a loading bar that moved somewhat close to where the program was actually helped users feel like the program

was working with them rather than against them, and that nothing broke during synthesis. While many more accurate methods could have been used to estimate loading progress, the one used was certainly good enough for the purposes of the project.

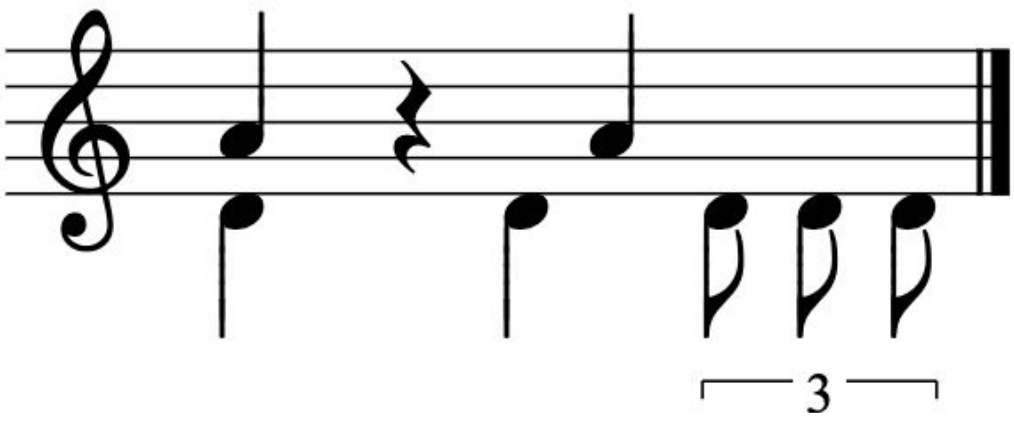
## Appendix E: Guido Parsing and Interpretation

An example of pre-processed and post-processed Guido is shown below, to show the types of processing done to make the Guido output look acceptable.



`{ [ \staff<1> d/4 _/8 d/8 d/12 d/12 d/12], [ \staff<1> a/4 _/4 a/8 _/8 ] }`

Figure E.1: Unprocessed Guido that is hard to understand.



```
{ [ \staff<1> \stemsDown d/4 \noteFormat<size=0> d/8 \noteFormat<size=1> d/4 d/12 d/12  
d/12], [\staff<1> \stemsUp a/4 \noteFormat<size=0> _/4 \noteFormat<size=1> a/8  
\noteFormat<size=0> c/8 \noteFormat<size=1>] }
```

Figure E.1: Processed Guido that looks visually better.

The GMN generated that for the “unprocessed Guido” is easy to follow, but very difficult to interpret. By removing some of the unnecessary rests, and fixing the direction of the stems, it is far easier to read and understand. The only way to hide these rests is to actually render them as “notes” of arbitrary pitch, and then use `\noteFormat<size=0>` to set their size to 0, which doesn’t render them.

## Appendix F: Alternative Transcription Methods

Before fully implementing the absolute spacing method for transcription, one other method was attempted in which a similar comparison computation was performed in the “time” domain. The possible beat patterns were mapped onto a buffer that contained the number of samples per beat, and the similarity between beat patterns and incoming hits were calculated in much the same way. In this case, the computation consisted of a much larger matrix multiplication where the basis matrix contains all possible beat vectors and the input vector contained 0s where no input was classified and 1s where an input was classified. It was found that to make this work accurately, the inputs could not be entered into the input vector simply as a 1 or 0, rather some sort of decay function was necessary to capture the possibility of variation in the input note. Furthermore, it was clear that rests must be given a value of -1 so that where rests lined up a positive value ( $-1 * -1$ ) was calculated and where hits lined up a positive value ( $1*1$ ) was detected. Similar to the method detailed above, the same priors were used to modify the amplitude of the hits thus applying the same probability weighting method. This method has two major drawbacks: the computational complexity is significantly increased, and it was far less robust in the prototyping phase. One important criteria for success in the prototyping phase was that a perfect input should imply a perfect output, and this method could never achieve this. This was most likely due to the fact that small changes were amplified as a result of the negative weighting of rests. This method also used far more constant relating to the width of the hit function, the amplitude of the hit, and the same parameters for the basis matrix hits. Finally, this method never provided any advantage over the simple absolute distance comparisons and thus the latter method was chosen as a more robust and simple algorithm.

## Appendix G: Hardware Specifics

Specific details regarding the addition of hardware interrupts, timers, and potentiometer readings are given here. These details are primarily added to document difficulties encountered with using the Nucleo documentation. This section should serve as an example of how to implement certain timer interrupt and hardware interrupt functionality should it be relevant in future offerings of the course.

```
15
16 static TIM_HandleTypeDef g_TimHandle3 = {
17     .Instance = TIM5
18 };
19
20 static TIM_HandleTypeDef g_TimHandle = {
21     .Instance = TIM2
22 };
23
24 void TIM2_IRQHandler()
25 {
26     __HAL_TIM_CLEAR_FLAG(&g_TimHandle, TIM_FLAG_UPDATE);
27     HAL_TIM_IRQHandler(&g_TimHandle);
28     HAL_TIM_PeriodElapsedCallback(&g_TimHandle);
29 }
30
```

Figure G.1: Configuration of TIM2 and TIM5 hardware timers (1).



```
7=void ConfigureTIM()
8 {
9     __TIM2_CLK_ENABLE();
10    g_TimHandle.Init.Prescaler = 0;
11    g_TimHandle.Init.CounterMode = TIM_COUNTERMODE_UP;
12    g_TimHandle.Init.Period = 108000000;
13    g_TimHandle.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
14    g_TimHandle.Init.RepetitionCounter = 0;
15
16    TIM2->EGR = TIM_EGR_UG;
17    TIM2->DIER = TIM_DIER_UIE;
18    TIM2->CR1 = TIM_CR1_CEN;
19
20    HAL_NVIC_SetPriority(TIM2_IRQn, 15,0);
21    HAL_NVIC_EnableIRQ(TIM2_IRQn);
22
23    HAL_TIM_Base_Init(&g_TimHandle);
24    HAL_TIM_Base_Start(&g_TimHandle);
25
26    __TIM5_CLK_ENABLE();
27    g_TimHandle3.Init.Prescaler = 0;
28    g_TimHandle3.Init.CounterMode = TIM_COUNTERMODE_UP;
29
30    g_TimHandle3.Init.Period = 4294967295;
31    g_TimHandle3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
32    g_TimHandle3.Init.RepetitionCounter = 0;
33
34    TIM5->EGR = TIM_EGR_UG;
35    TIM5->DIER = TIM_DIER_UIE;
36    TIM5->CR1 = TIM_CR1_CEN;
37
38    HAL_TIM_Base_Init(&g_TimHandle3);
39    HAL_TIM_Base_Start(&g_TimHandle3);
40 }
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2
```

```
void ConfigureADC()
{
    GPIO_InitTypeDef GPIOA_InitStructure;

    __GPIOA_CLK_ENABLE();
    __ADC1_CLK_ENABLE();

    GPIOA_InitStructure.Pin = GPIO_PIN_3;
    GPIOA_InitStructure.Mode = GPIO_MODE_ANALOG;
    GPIOA_InitStructure.Pull = GPIO_NOPULL;
    HAL_GPIO_Init(GPIOA, &GPIOA_InitStructure);

    HAL_NVIC_SetPriority(ADC_IRQn, 0, 0);
    HAL_NVIC_EnableIRQ(ADC_IRQn);

    ADC_ChannelConfTypeDef adcChannel;

    g_AdcHandle.Instance = ADC1;

    g_AdcHandle.Init.ClockPrescaler = ADC_CLOCKPRESCALER_PCLK_DIV2;
    g_AdcHandle.Init.Resolution = ADC_RESOLUTION_12B;
    g_AdcHandle.Init.ScanConvMode = DISABLE;
    g_AdcHandle.Init.ContinuousConvMode = ENABLE;
    g_AdcHandle.Init.DiscontinuousConvMode = DISABLE;
    g_AdcHandle.Init.NbrOfDiscConversion = 0;
    g_AdcHandle.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
    g_AdcHandle.Init.ExternalTrigConv = ADC_EXTERNALTRIGCONV_T1_CC1;
    g_AdcHandle.Init.DataAlign = ADC_DATAALIGN_RIGHT;
    g_AdcHandle.Init.NbrOfConversion = 1;
    g_AdcHandle.Init.DMAContinuousRequests = ENABLE;
    g_AdcHandle.Init.EOCSelection = ADC_EOC_SEQ_CONV;

    HAL_ADC_Init(&g_AdcHandle);

    adcChannel.Channel = ADC_CHANNEL_3;
    adcChannel.Rank = 1;
    adcChannel.SamplingTime = ADC_SAMPLETIME_480CYCLES;
    adcChannel.Offset = 0;

    if (HAL_ADC_ConfigChannel(&g_AdcHandle, &adcChannel) != HAL_OK)
    {
        asm("bkpt 255");
    }
}

if (HAL_ADC_PollForConversion(&g_AdcHandle, 1000000) == HAL_OK)
{
    adc_val = HAL_ADC_GetValue(&g_AdcHandle);

    if (abs(adc_val - prev_adc_val) > 75)
    {
        BPM = 60 + (adc_val * (180.0 / 4095));
        cycles_per_period = 10800000.0 * (60.0 / BPM);
        len = sprintf(packet_buffer, "BPM, %i \n", BPM);
        SendPacket(SID_TO_WHATEVER, (uint8_t *)packet_buffer, len);
        TIM2->ARR = cycles_per_period;
        prev_adc_val = adc_val;
    }
}
```

Figure G.3: ADC configuration for use with potentiometer.

Figure G.3 shows an example of configuration settings for the Nucleo ADC to be used with an analog GPIO pin. The top graphic shows the necessary configuration call that should be done before the main body of code is run, and the bottom graphic shows the method for reading the value from the ADC during runtime.

```
void EXTI9_5_IRQHandler(void)
{
    time_diff = TIM5->CNT;
    TIM5->CNT = 0;

    if((__HAL_GPIO_EXTI_GET_IT(GPIO_PIN_9) != RESET) && !run_main)
    {
        __HAL_GPIO_EXTI_CLEAR_IT(GPIO_PIN_9);
        if (!run_main && time_diff > 30000000){
            len = sprintf(packet_buffer, "%s\n", "Playback");
            SendPacket(SID_TO_WHATEVER, (uint8_t *)packet_buffer, len);
        }
    }
}

__GPIOC_CLK_ENABLE();
GPIOC_InitStructure.Pin = GPIO_PIN_12 | GPIO_PIN_9;
GPIOC_InitStructure.Mode = GPIO_MODE_IT_RISING;
GPIOC_InitStructure.Speed = GPIO_SPEED_HIGH;
GPIOC_InitStructure.Pull = GPIO_PULLDOWN;
HAL_GPIO_Init(GPIOC, &GPIOC_InitStructure);
```

Figure G.4: GPIO interrupt configuration.

Figure G.4 shows the settings to configure a hardware interrupt pin. Only the interrupt call for GPIO\_PIN\_9 is shown as EXTI9\_5\_IRQHandler is used only for pins 5 to 9. An additional call was made to EXTI15\_10\_IRQHandler for GPIO\_PIN\_12.

```
static void SystemClock_Config(void)
{
    RCC_ClkInitTypeDef RCC_ClkInitStruct;
    RCC_OscInitTypeDef RCC_OscInitStruct;

    __HAL_RCC_PWR_CLK_ENABLE();
    __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE2);

    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
    RCC_OscInitStruct.HSEState = RCC_HSE_ON;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
    RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
    RCC_OscInitStruct.PLL.PLLM = 8;
    RCC_OscInitStruct.PLL.PLLN = 288;
    RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
    RCC_OscInitStruct.PLL.PLLQ = 6;
    HAL_RCC_OscConfig(&RCC_OscInitStruct);

    RCC_ClkInitStruct.ClockType = (RCC_CLOCKTYPE_SYSCLK | RCC_CLOCKTYPE_HCLK | RCC_CLOCKTYPE_PCLK1 | RCC_CLOCKTYPE_PCLK2);
    RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
    RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
    RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;
    HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4);
    SystemCoreClockUpdate();

    if (HAL_GetREVID() == 0x1001)
        __HAL_FLASH_PREFETCH_BUFFER_ENABLE();
}
```

Figure G.5: System clock settings.

Figure G.5 shows the clock settings as they were configured for use with the ADC and TIM2/5 hardware interrupts.