Final Design Report

A conductible music player device

Comp 3601

Adam Smallhorn, z3332654

Team Orange

Adam Smallhorn
Geoff Choy
Scott Smith

Table of contents

Executive summary

This report presents the designing of an interactive music player built on the Digilent Nexys board with a Spartan 3 FPGA. This design makes use of an IR sensor to detect a user specified beat and play music back at that speed. An application to write and download songs was written to use with the device. In this design, the spec design objectives and extension objectives were met.

Introduction

The task of creating a music player seems trivial at first. However, under the surface there are many vast and complicated technologies at play. The project specification called for the creation of a conductible music player based on the Digilent Nexys board with Spartan 3 FPGA.

Whilst music devices are relatively common, the use of the Nexys board meant that many things that are taken for granted (such as pulse width modulation, digital to analog convertors, firmware) did not exist, and had to be developed and implemented from scratch using VHDL. Furthermore the spec called for a complicated control mechanism with a baton and interaction between the device and custom PC software.

Fortunately, many of these functions are replicable in hardware such as the Spartan 3 FPGA, and some tools are available to aid device to PC communication.

Feature Set

Feature Overview

The core features of our design were based off the requirements of the project specifications – namely : playback of music downloaded from a PC, sensing the motion of a baton to control the music tempo, and the development of custom PC software to create music files. Part way into the development of the device, we realised it would be trivial to implement some extra features including: fast forward, rewind, play, pause, stop and volume control.

The PC software was designed to allow users to intuitively compose songs within the app and to easily download the song to the device within seconds. To achieve this, we built a graphical user interface (GUI) including a full scale C3-C6 keyboard with functional keys. For music composing, the GUI allows the user to set music settings like note type (normal, slurred or staccato) with a radio button, beats per minute with a slider or text box, and note length with a selector. (See appendix 1 – Application screenshot). The user is able to input music by clicking the piano notes in the GUI, by typing them in directly in a text box, or by loading from a text file. At any time, the user is able to preview the song on their PC by using the play, pause and stop buttons. To download the song to the board, we include simple buttons within the GUI. The ***Encode*** button will translate the music file in a byte format suitable for the board, and the ***Program to board*** button will transfer the byte file to the board.

User Interface and usability was also a factor, so we include simple features like a clear button to remove the previously generated notes, a success dialog box when the file had successfully downloaded to the board, and something as simple as remembering the previous directory used for file browsing so you don't have to navigate there repeatedly.

On the device itself, there are several features for users to change music playback. The simplest are ***play***, ***pause*** and ***stop*** buttons (buttons 0, 1 and 2 respectively). ***Pause*** will stop the music at a note and not resume until the ***play*** button is pushed again, and ***stop*** will stop the music and return to be beginning of the song when ***play*** is pushed again. Another feature is tempo control. The user can choose between playing at the tempo specified in the file and a variable tempo by setting switch sw7 to on or off respectively. If set to variable tempo, the user can conduct with the baton over the sensor, and after 4 swings the playback speed will change and the display will update with the new current BPM. If the user would prefer to use a button over a baton, they can activate sw4 and use button 3 to control the tempo. Fast-forward and rewind were some extension features that were able to easily implement. The user can activate the FF/RW mode by setting sw6. The user can choose between the fast-forward or rewind operation with switch 3, when button 3 is pushed the song will proceed at maximum speed forwards or backwards through the song. Additionally, we saw the value of adding volume control as an extension feature. The user can change the playback volume using sw2 and sw1 to select between 4 volume settings as a binary value.

Deviation from initial plans

Our initial feature set did not include volume or playback control (play, pause, stop, fast-forward or rewind features). Volume control was included because after some initial testing of notes on the board, it became clear that volume control was a very convenient feature, especially whilst in the lab with other people, and this would likely be a useful feature for users as well. Play back control was added as during development we saw that it could be added quite easily, and presented a convenient set of additional features that are consistent with other music devices on the market.

However, our initial design report did include some features that did not make it in the final prototype: playback statistics in the PC software, a filter circuit to clean up the speaker quality and an extended range of notes. Playback statistics in the PC software represented an ambitious view of what might be possible, but was not seriously pursued when development was started due to the large scope of work. A filtering circuit for the speaker was not included because when we began testing, the speaker sound output was of sufficient quality that additional filtering was not necessary. Likewise, whilst preparations were made to file formats to include a wider range of tones, as we moved closer to the deadline, this feature was deemed non-critical, so was removed to ensure more essential features could be implemented reliably.

Limitations of final design

The only significant limitation of our final prototype was the maximum song length that was able to be played. Our original plan was to support play back lengths of several hundred notes, however due to a reliability issue with writing data to the board, we had to resort to checking that each byte of data was successfully written, and rewriting it if there was an error. This meant having to supply an 8-bit address value with every write. As our encoding format uses 2 bytes per note, this resulted in a maximum of 128 notes per song. In practice, for an average song at 80 BPM this meant nearly 1.5 minutes of playback. Although not ideal, this still allowed for the majority of songs to be played.

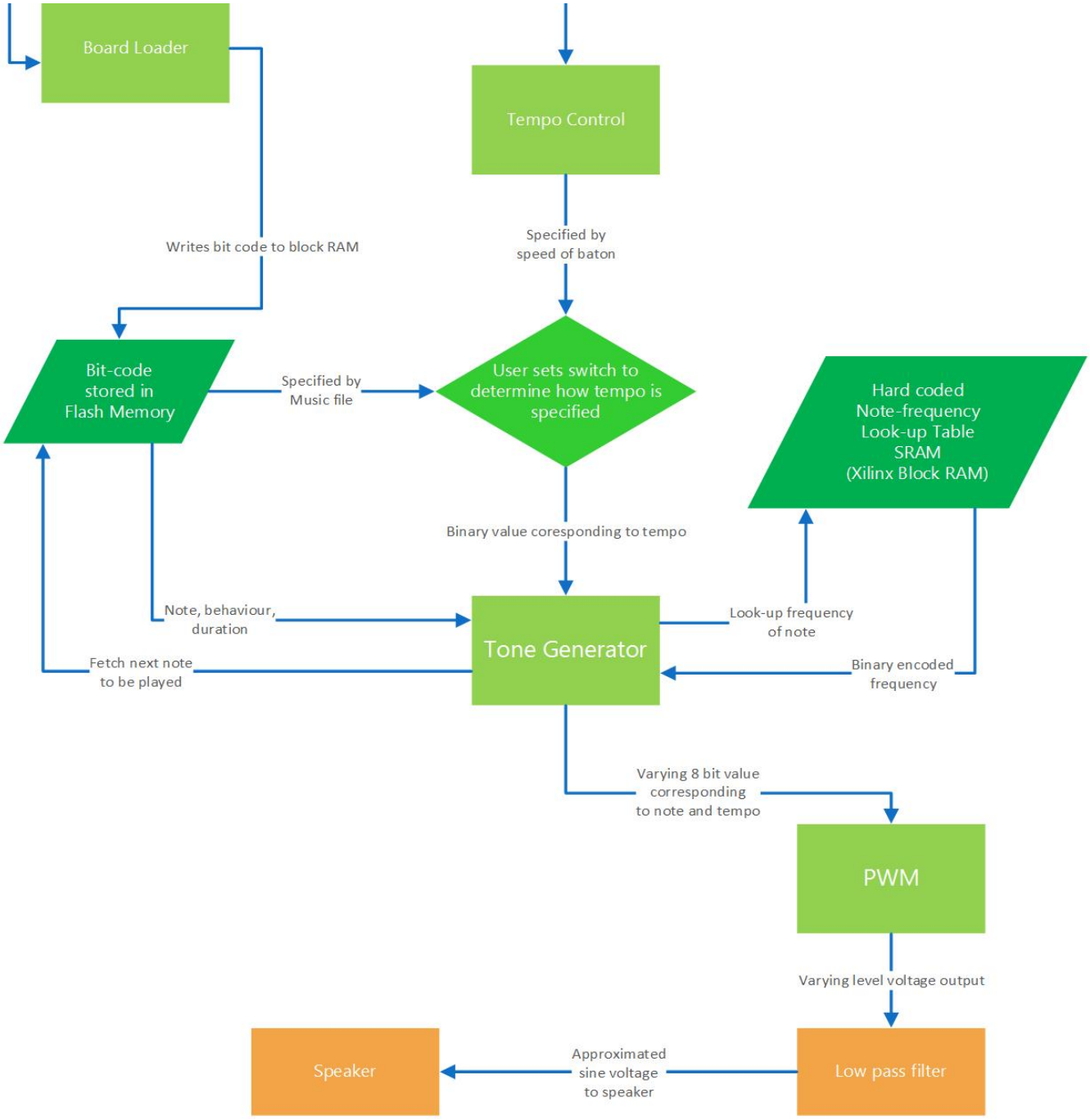
Task Breakdown

In approaching this project, we sought to strike a balance between being able to work independently on separate tasks and being easily able to integrate these components together. The key to this was dividing the components so they had minimal and well defined interactions with other components.

We divided our project into the following areas:

- **PC Software (Software) –** This component was responsible for the creation of music files, reading files, encoding songs in the byte file format and allowing the user to download to the board. This component was only required to interact with the downloader, calling it in the background via command line and passing in a txt file as a parameter.
- **Downloader App (Software), Downloader VHDL (VHDL), Block memory (VHDL) –** The downloader had two parts, the C++ Windows executable (which took in a file from the command line, and wrote each byte to the board) and the VHDL that receives the stream of bytes and writes them to block ram. The Downloader app interacts with the PC software, and the block ram is accessed by the timing module.
- **Detection circuit (Electronics) –** The detection was performed by an IR sensor and comparator which supplied a digital signal to the BPM calculation module.
- **Detection module (VHDL) –** This component takes the digital signal from the sensor and creates two values – beats per minute (BPM), and milliseconds per beat (MPB), which is how long to play a crotchet in milliseconds. BPM is sent to the 7-segement display and MPB is sent to the playing circuit.
- **Timing module (VHDL) –** The timing module is responsible for fetching the notes and data from block ram, interpreting the data and sending the note data to PWM unit.
- **PWM unit (VHDL) -** PWM converts the note data into an analog signal for output in the speaker.

Music Player – Process Diagram


Board Loader
Tempo Control
Writes bit code to block RAM
Specified by
speed of baton
Bit-code
stored in
Flash Memory
Specified by
Music file
User sets switch to
determine how tempo is
specified
Hard coded
Note-frequency
Look-up Table
SRAM
(Xilinx Block RAM)
Binary value coresponding to tempo
Note, behaviour,
duration
Tone Generator
Look-up frequency
of note
Fetch next note
to be played
Binary encoded
frequency
Varying 8 bit value
corresponding
to note and tempo
PWM
Varying level voltage output
Speaker
Approximated
sine voltage
to speaker
Low pass filter

Hardware Design

The hardware component of this project was for baton detection. After much consternation, we decided upon using an Infrared sensor, as it displayed favourable characteristics over the alternative sensors we investigated - ultrasonic sensors were prone to error near low ceilings and a magnetometer required specialised modifications to the baton. We used the Sharp GP2Y0A21K0F analog distance sensor with a range of 10-80cm. When the baton was present above the sensor the voltage would climb from a resting voltage of approximately 1.1V to near 4 V. We used this output with a Texas Instruments LM331 digital comparator with a reference voltage set at 1.5V. The comparator was wired into the FPGA as a digital signal representing whether a baton was detected.

The sensor had some strange side effects, including a periodic spike of 1.9 V every few milliseconds. We suspected this was leakage from the internal clock the sensor must use to calculate the distance of objects. We were able to overcome this by adding de-bouncing for the signal as it entered the FPGA, thus negating the temporary periodic spikes.

We did not require a specialised baton, so any thin cylindrical object, such as a pen, is compatible with the sensor.

FPGA Design

Task breakdown

Most of the logic of the device exists on the FPGA as different modules. Those modules can be roughly divided into the following parts: ***downloader***, ***baton detection & BPM display***, ***playing module***, and ***PWM & tone generation***.

Downloader

The VHDL downloader is the component that interfaces with the PC downloader software via EPP (Enhanced Parallel Port). To do this it requires a shared data bus, and several control signals shared with the PC. The VHDL downloader itself has two registers (one for storing addresses, and one for storing incoming data) and a block ram module. When the PC downloader wants to write to an address in memory, it must first send the address first, as the data bus is only 8 bits wide. When the PC downloader sends an address, the VHDL will cache this value in a register. When the PC sends a piece of data, the VHDL stores the data in a temporary data register, then uses the address register to provide the destination address for the block ram module.

The code is based off an example file ‘DpimRef’ provided by Digilent and uses a finite state machine to interact with the PC via EPP.

Baton detection & BPM display

The baton detection VHDL module receives a digital input from the electronic sensor that represents whether a baton has been detected or not. The task of the baton detection module is to interpret the swing detections to calculate the new tempo.

The particular algorithm that we implement to do this requires 4 swings through the sensor’s beam, of which, two are used to calculate a new tempo. The first swing though the beam is detected, and sets of a timer. The timer is not stopped until a second and third swing is detected, thus the timer value

represents two whole swings. (see appendix 2). This value is shifted to represent a value called milliseconds per beat (MPB) and is supplied to the playing module.

The MPB value is also converted into BPM with by using repeated subtraction to implement division, and then is converted again into BCD for display on the 7segment display.

Playing module

The playing module is responsible for fetching the notes and data from block ram, interpreting the data and sending the note data to PWM unit. The playing module uses a finite state machine to control whether the device is in playing mode, stop mode, pause mode, fast-forward mode or rewind mode. When in playing, fast-forward or rewind mode, the module will fetch 2 bytes of data from memory. The finite state machine will then enter a process data state, which parses the note type (normal, staccato etc) and length from the data. Separate sub-modules calculated the total duration of a beat and the the duration of the note based on the note length, note type and the current MSB (milliseconds per beat) value from the baton detection unit. A separate unit within the timing module controls when the PWM module should receive a value, and when it should receive nothing so that a pause between notes is heard.

PWM & Tone generation

The tone generator is made up of many sub components including lookup tables, PWM unit and registers. Its role is to take a note received from the timing module, and output an analog signal that will produce a note of a particular frequency in a speaker.

In order to play a note, the tone generator receives an 8-bit identifier which is used to index a lookup table. The lookup table also takes in volume level, and based on these two input values will output the *current duty cycle value* and a *hold-duration* to the PWM unit. The current duty cycle value is one of 64 'slices of a sine wave'; that is to say; one of 64 values of range 0-255 that emulate the amplitude of a sine wave. If the volume is not 100%, then the value amplitude value is instead read from a separate lookup table which has values that represent a wave at the desired volume (this could have been replaced with shifting, but we did not have time to make this change). The hold-duration value from the look up table is used to determine the frequency of the note. If this value is larger, the PWM unit will stay on the same duty cycle for a longer period of time, resulting in a lower frequency sine wave.

Prevention of glitching on note switching was possible by holding the PWM value during a rest note or a note gap and continuing from that value when the next duty cycle value was fetched from the lookup table.

Nexys board features

Our final prototype makes use of the following Nexys board components:

- **Buttons** – Buttons 0, 1 and 2 are reserved for play, pause and stop. Button 3 is reserved for special functions depending on the switch settings, but is used to simulate a baton swing, and fast-forward/rewind. The buttons were chosen because they represent the most convenient input method for simple commands such as play and stop.
- **Switches** – Switches are used to control some of the modes of the board. For example, two switches are dedicated to volume, one switches between baton detection and BPM from the

file. Switches were chosen for these functions because there are fewer buttons, and some thing such as settings and modes are easier to see what state they are in by looking at the switch.

- **7 Segment display** – The current BPM detected on the sensor is displayed on the 7-seg display.
- **Block ram** – The downloader uses block ram to implement the board memory. Block ram was used over SDRAM as it was easier to first implement, and didn't present any performance or storage limitations.
- **100MHz clock** – We used the 100MHz clock on the Nexys board for no other reason than we first started prototyping with 100MHz and never down converted the code to a slower speed. This didn't provide any limitations to our design, but a slower clock could have also been used.

FPGA hardware – software interface

The FPGA is able to interface between hardware and software because of a dedicated Windows downloader program, the Digilent DPCUTIL DLL, and a special finite state machine on the board designed to handle incoming data.

The downloader program, which was completely software, was custom built to take in a file from command line, and write each byte to the board using the API provided in the DPCUTIL library. Most of the hard work of interfacing between software and hardware is done by the API in the DPCUTIL on our behalf. The DPCUTIL.dll library was supplied by Digilent, and uses EPP over USB to interface with the Spartan FPGA. The EPP uses a shared 8-bit data bus, and 4 main control signals (Address Strobe, Data Strobe, Write, and Wait) to control the flow of data over the bus. During EPP the computer acts as master, and the device as slave. On the FPGA, we developed a finite state machine based off a Digilent reference design that loads addresses and data from EPP and transfers the data into block ram for use in the rest of the design.

Waveform Generation

Our final design uses 8-bit pulse width modulation to create analog signals to drive the speaker. A lookup table stores 64 samples of a sine wave and a counter is used to iterate through all 64 values. These values are supplied to the PWM unit as its duty cycle. In order to produce notes of different frequency, the counter does not increment on every clock tick, but instead only increments after the time has elapsed that is $1/64^{th}$ of the desired wave's period. This means that all 64 samples of the wave will have been sent to the speaker in the same amount of time as the period of the wave, effectively creating a wave of the desired frequency in the speaker. The logic for enabling the counter to increment was implemented as an internal timer in the lookup table module. Different frequencies were obtained finding a new value for the internal counter in a dedicated lookup table. This value would change the internal counter in the lookup table, which would cause the counter iterating through each sample of the wave to slow or increase, resulting in a different frequency wave.

Software Design

The PC software was created using Java, Swing for the graphical user interface, and netbeans to auto-generate large parts of the underlying code. For the graphical user interface, most buttons used action listeners and updated or inserted the necessary information based on the user's input. File input was built using netbeans' existing templates, and the input was then used to display in a JTextArea so users could edit the music file. Some of the user interface aspects such as note type and duration were build using JSlider and radio buttons.

One of the extension features of the PC software was the ability for the user to preview their song by playing it in the app. This was built using javax.midi. Sequencer methods such as Sequencer.stop(), Sequencer.pause(), Sequencer.resume() allowed easy playback control without the need for multithreading.

Downloading to the board required the generation of the binary file in the same directory as the java app and calling our downloader application to run, with the binary file specified as a command line input. The downloader.exe would output a success or failure message to stdio which would be parsed by the java app, and used to show a success or failure dialog box in the GUI.

The downloader.exe program was written using the Digilent DCPUTIL API and dynamic linked library. This provided simple functions that could place a byte in a register on the board, and handled the lower level driver/USB interaction with the board. The downloader.exe read a file from the command line, opened a connection to the board, and looped through the input file writing each byte to the board. We encountered reliability issues with writing to the board, so in order to ensure accurate downloading, the downloader.exe would read back the last byte written, and if necessary write it again. This gave us vastly superior accuracy at the expense of limiting our file size to 256 bytes, as data could no longer be streamed into the registers. At the end of the transaction with the board, the downloader.exe closes the connection with the board, and returns a success message to the java app.

Final Budget

This project came in significantly under budget with a total cost of approximately $17.

Component	**Description**	**Price**
Sharp GP2Y0A21K0F	Sharp Analog distance sensor, range 10-80 cm	$15.00
lm331	Digital comparator	$2.00
Resistors		Priceless.

Retrospect

Compared to our initial project plan there have been a few modifications along the way. Our timeline planning was not very accurate – a lot of tasks took longer than expected although this didn't impact our ability to finish in time. Also, many of the Gantt chart tasks were very generic tasks so it was difficult to be precise about what had been achieved and what had not. Our budget however, was very accurate. Our initial extension features were not overly ambitious, but we did not realise that some extension features would be quite trivial to implement, such as fast forward and volume control. Nor did we anticipate having as much difficulty as we did with our sensors.

At the risk of sounding trite, if we were able to start all over again we would do many things differently. We would establish clearer interfaces between the different VHDL components. We encountered a few issues when integrating the pulse width modulation and the timing module like bit-widths of vectors and active high vs active low signals. We would of course, start sooner in the semester, so that we could get the fundamentals completed earlier. There were too many delays with our sensor circuit and the ordering of new sensors. Our downloader wasn't complete until the last moment, which caused added stress and complications. Our PC software should have focused on playback first, before adding novelty features.

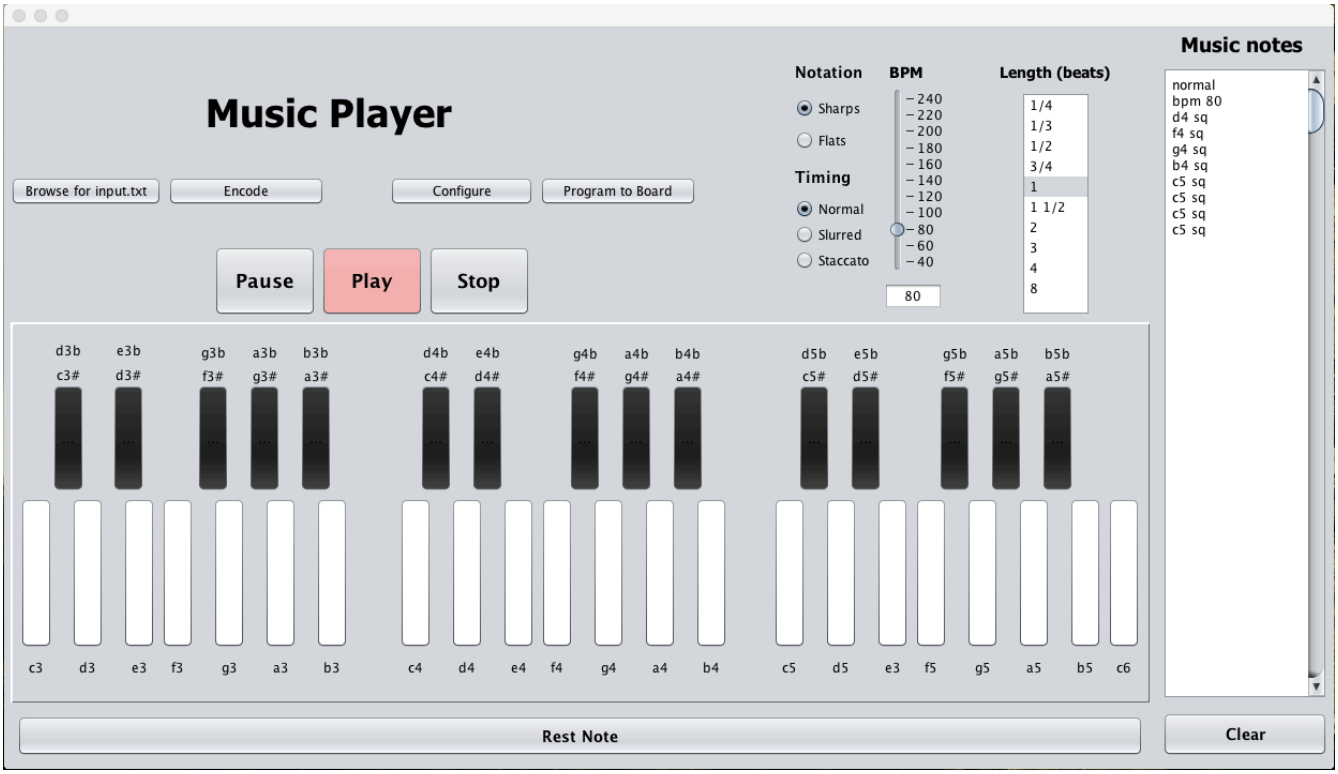
In terms of our PC software, using the netbean's system to auto generate code created a lot of headaches. While it was easier than designing the GUI manually, it resulted in massive amounts of code which was hard to re-structure and debug. Javax.midi was used instead of JFugue, but this resulted in a poorer sound quality. JFugue's note playback, in contrast, was more consistent and sounds closer to real piano notes. There were also some UI tweaks we would have made : default notes should have been crotchets, and creating an interface to allow for tempo changes midway through a song.

For the VHDL components, more rigorous testing of the different components was necessary. We encountered several bugs with our playback module which was only picked up when components were integrated. If we had more time, tone generation could have been optimised by using bit-shifting to change the volume dynamically instead of consulting separate lookup tables for hardcoded amplitudes. We didn't have enough time to fully debug our downloader VHDL, which we worked around by checking each byte that was written, but this had the side effect of our file size being limited to 256 bytes.

All in all, we were quite happy with having achieved all we set out to achieve and thoroughly enjoyed the project and the challenges it presented.

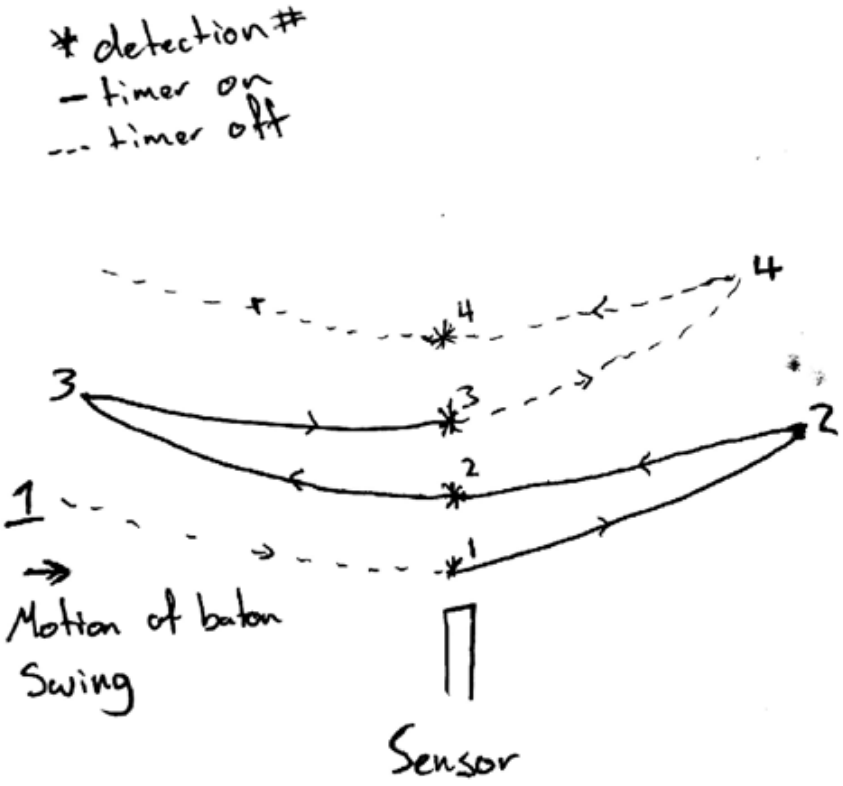
Appendices

Appendix 1



Appendix 1. The Music Player PC Application

Appendix 2



Appendix 2: Diagram of the motion and periods when the timer is active.