Eight String Laser Harp

University of British Columbia

Ahren Spadinger-Fengler

ABSTRACT

For my PHYS 319 project I built an 8 string laser harp, with audio mixing capabilities. The harp uses 8 pairs of lasers and photodiodes to detect input, which is processed by an MSP430 microcontroller. Sound mixing is performed on the MSP430 before a wave representing the currently playing notes is outputted to a speaker. The laser harp performed quite well, with many basic songs such as Take on Me, Twinkle Twinkle Little Star and the Tetris Theme being able to be played on it. A variety of different waveforms such as sine, saw and square waves could be used to simulate different sounding instruments. The harp was well tuned, but its audio output was very quiet due to a lack of voltage amplification of the signal wave, making it not ideal for actual performances.

1.0 INTRODUCTION

I’ve played various musical instruments for a majority of my life, so for my PHYS 319 project I decided to build a laser harp. This is an electronic instrument that replaces the strings of a regular harp with lasers. In order to play a specific note, instead of plucking a string, you instead block the laser beam with your hand. This is then detected by a photodiode, sending a signal to a microcontroller, which then plays that note over a speaker.

The first laser harp was invented by Geoffrey Rose in 1976, and consisted of 10 laser beams in an octagonal frame[1]. Since then many other laser harp designs have been developed including both framed designs, which have each laser shine on their own photodiode; and frameless designs, which shoot laser beams out in quick succession and use a single photodiode to detect the light reflecting off of the performers hand[2]. For this project I built a framed design using 8 lasers and 8 photodiodes, which was processed by an MSP430F5529 microcontroller. This microcontroller detected which notes were being played, mixed the notes together, and outputted the appropriate sound wave to a speaker. The laser harp was able to play various waveforms including sine, saw and square waves, and had a way to adjust which set of notes the harp was currently being used, so that different scales could be played.

2.0 THEORY

In order to have built a laser harp, there were many components that needed to be used and understood. These included the photodiodes used for detecting the lasers, the Digital to Analog converter used for creating the output wave, the low pass filter used for reducing noise in the output wave, and the speaker used for playing the output wave.

2.1 How a Photodiode Works

A photodiode is an electrical component which converts light energy into current. It does this by having 3 regions: a positive region (connected to anode) and negative region (connected to cathode) separated by a depletion region[3]. Inside the depletion region is an electric field pointing towards the positive region[3]. Light can hit electrons in the valence band of atoms in the depletion region, causing the electrons to be freed[3]. The electrons now in the presence of the electric field in the depletion region are moved to the negative region causing a current to form[3]. This current is proportional to the intensity of the light hitting the photodiode[3].

2.2 How a Ladder DAC (Digital to Analog Converter) Works

A Digital to Analog Converter (DAC) is used for converting a digital input (a series of high and low voltages) to an Analog output (a singular voltage). The output voltage should be proportional to the binary number represented by the inputs[4]. For example, if a DAC has 8 input pins, you could input binary numbers ranging from 0 to 255. If this binary number is D, then the output voltage is determined by[7]:

$$V_{out} = V_{in}\left(\frac{D}{256}\right) \qquad \text{(Eq 2.0)}$$

A Ladder DAC works by having a series of resistors with latches connected between the resistors, where each latch represents one digital input[4]. Depending on the latch position, the current flowing through the latch will need to travel through more or fewer resistors[4]. This causes the voltage from each latch to be divided by a different amount, with the output voltage being the sum of these divided voltages[4]. Using this method we can convert a digital input to an analog voltage.

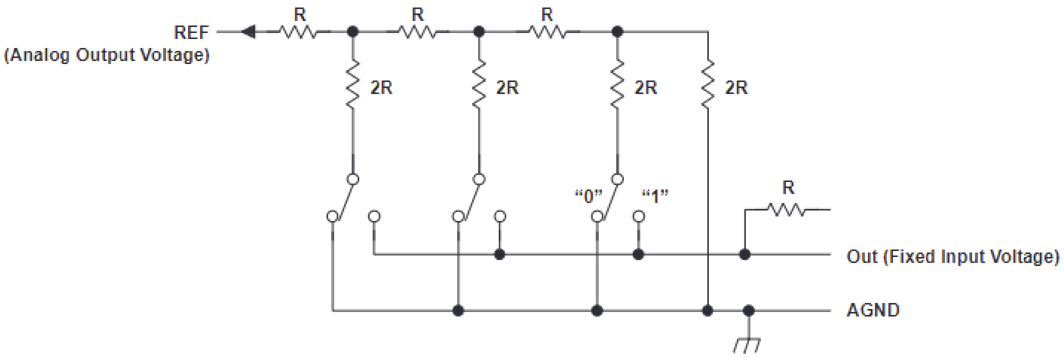


Figure 2.1: The basic structure of a ladder DAC, taken from the TLC7528CN data sheet[7], with 3 latches (3 bit digital input).

2.3 How a Low Pass Filter Works

A basic low pass filter can be built using an RC circuit where the output voltage V_{out} exponentially decays towards the input voltage V_{in} at a rate of $\omega = \frac{1}{RC}$. If we have an input wave with a frequency of $f_c = \frac{1}{2\pi RC}$, the amplitude of the wave is reduced to 70.7% of its original value[5]. This is known as the cutoff frequency[5]. Beyond this point the gain of the wave decays logarithmically with frequency[5] as seen in Figure 2.2.

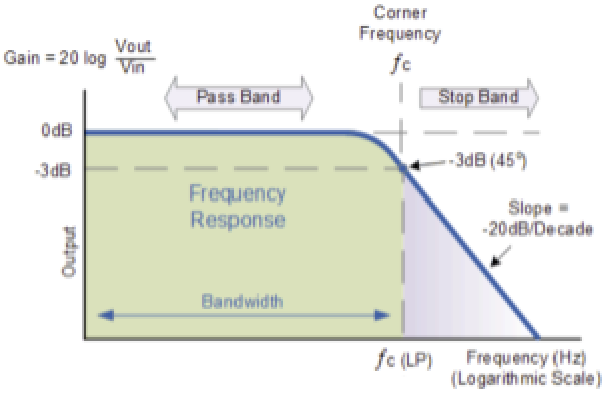


Figure 2.2: Wave amplitude as a function of frequency once passed through a low pass filter. This figure was taken directly from reference [5]. Here the cutoff frequency is labelled f_c.

2.4 How a Speaker Works

A speaker converts an alternating current to a sound wave by passing the current through a copper inductor surrounded by a fixed magnet[6]. By altering the current through the inductor, different magnetic fields can be formed, inducing motion in the inductor[6]. The copper inductor is attached to the cone of the speaker[6]. When the inductor causes the cone of the speaker to move, this creates a pressure wave that we perceive as sound[6].

3.0 APPARATUS

The apparatus consisted of three main components, the input device (harp), the control unit (MSP430) and the output device (speaker system). In addition to the MSP430, 3.3V and 5V power supplies were used to power the apparatus. The program loaded onto the MSP430 can be found in Appendix L, with individual parts of the code highlighted in Appendices A-K.

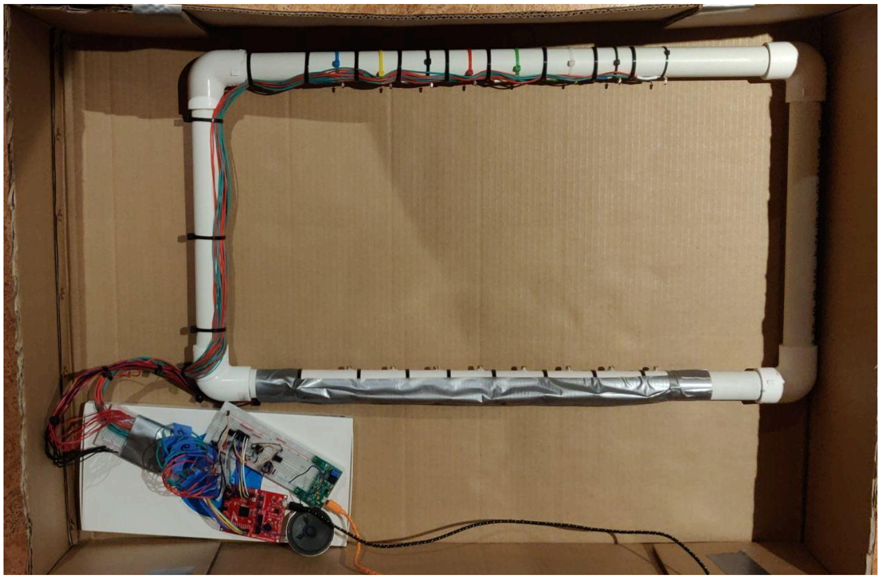


Figure 3.0: The completed apparatus sitting in a cardboard box used for transporting the device. The apparatus is powered off.

3.1 Input Device (Harp)

The actual harp consisted of 8 pairs of photodiodes and lasers lined up in parallel. These components were connected to a frame built out of 1 inch diameter PVC piping, formed into a rectangular shape 28” by 16” large. The lasers were spread evenly across one of the long edges of the frame, and each shot its laser beam into a photodiode on the opposite edge (Figure 3.1).

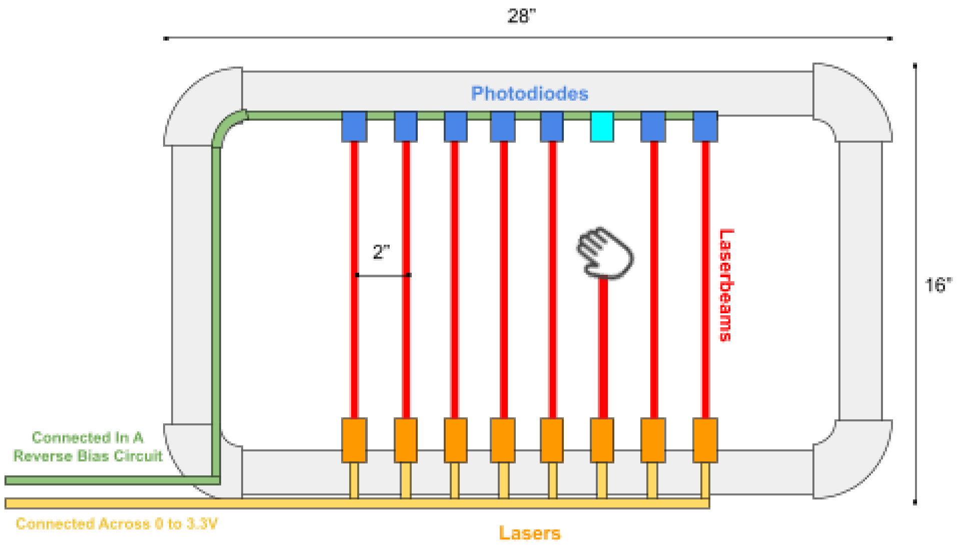


Figure 3.1: Basic Structure of the Laser Harp. A rectangular frame holds in place 8 pairs of lasers and photodiodes. Notes can be played by blocking different lasers with your hand.

The lasers were 3V-5V 6mm red dot lasers [amazon link], that were each connected across a 3.3V power supply on the bread board. These lasers were inserted into one of the long ends of the frame, each spaced 2” apart, and each pointing at a photodiode on the opposite edge. Super glueing the lasers in place prevented them from shifting around. The photodiodes were 3mm clear round headed receiver diodes [amazon link], held in place on the frame using zip ties. Each one was connected in a reverse bias mode across a 10 kOhm resistor. By connecting MSP430 input pins between the photodiode and resistor, we were able to detect high voltage readings when the laser shone on the diode, and low voltage readings when the laser was blocked (Figure 3.2).

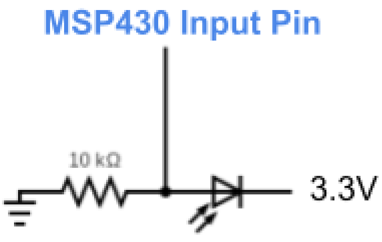


Figure 3.2: Photodiode set up in reverse bias mode. This circuit was constructed for each of the 8 photodiodes, using an integrated resistor block to save space.

3.2 Processing Unit (MSP 430)

As seen in Figure 3.3, the MSP430 used:

- 8 pins for input; one for each laser on the harp
- 8 pins for audio signal output; allowing for a 256 bit output
- 2 pins for signal output control; to interface with the DAC
- 1 GND pin connected to the breadboard ground

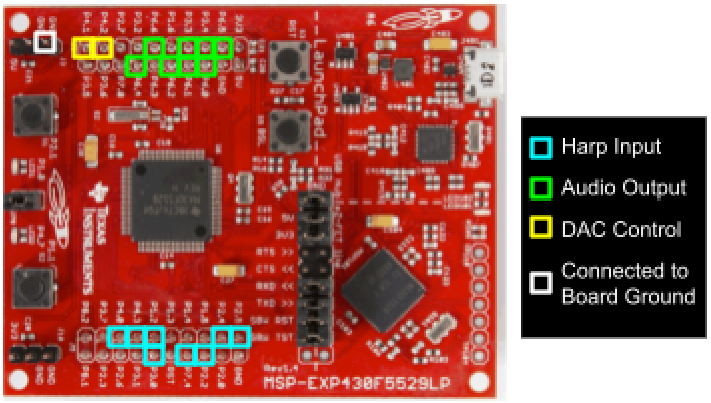


Figure 3.3: Outline of pin usage on the MSP430. The photo of the microcontroller was taken from the first page of the Development Kit[12].

3.2.1 Processing Input

In order to read signals from the harp, each pin had its PxOUT bit set to 0, its PxDIR bit set to 0 and PxREN bit set to 1 (Appendix I). This enabled input for the pins using a pull down resistor.

The read() function (Appendix E) was used to read in the harp inputs as an 8 bit number, with each bit representing whether a specific laser was hitting its diode (1) or not (0). BIT0 represented the lowest note and BIT7 represented the highest note.

This function was run repeatedly in a while loop (Appendix J). Its output was then used to update an unsigned int isPlaying[8] array. Each element in the array held a masking with 2 possible states: a 16 bit number of all 1's (NOTE_ON), or a 16 bit number of all 0's (NOTE_OFF). This was later applied as a mask to each note's signal during sound mixing (Appendix K). I decided to use a mask instead of a boolean to keep track of each note, as this made the sound mixing code more efficient as no branching would need to be used.

Also in the while loop were controls for using the buttons P1.1 and P2.1 on the MSP430 as inputs to increment the type of waveform (instrument) and scale being used by the laser harp (Appendix J). In total the harp had 3 different waveforms (Sine, Square, Saw) and 4 different scales (C4 Major, A4 Minor, A4 Dorian, G4 Mixolydian). The P1.1 and P2.1 buttons were initialized as input pins using a pull up resistor (Appendix I).

3.2.2 Sound Mixing

Sound samples were mixed and sent out as part of an interrupt routine that ran at a rate of once every 512 clock cycles. The code to set up this interrupt can be found in Appendix G. The goal for the sound mixing was to be able to mix the waveforms of up to 8 notes at once, hence allowing the harp to play any combination of notes in its currently loaded scale.

This sound mixing code was the most challenging piece of code to write for the project. This is because, in order to achieve a quality sound we would want to be sending out wave samples from the microcontroller tens of thousands of times a second; the common sampling rate for digital audio is $44.1\ kHz$. Since by default the MSP430 operates at a $1\ MHz$ clock speed, this would only allow roughly 22 clock cycles to process each sample, which is not enough to perform proper sound mixing. By increasing the clock speed we could get more clock cycles per sample, but our routine for processing each sample would still need to be very efficient.

3.2.3 Writing An Efficient Interrupt

To write an efficient sound mixing interrupt I had to apply some clever programming tricks to avoid using multiplication, division and floating point numbers, since the MSP430 did not have dedicated hardware for these operations, meaning they were very slow to execute. The waveforms for the different instruments were stored in 256 element arrays of integers, with each integer ranging from -128 to 127 (Appendix C). The three main waveforms I implemented can be seen in Figure 3.4.

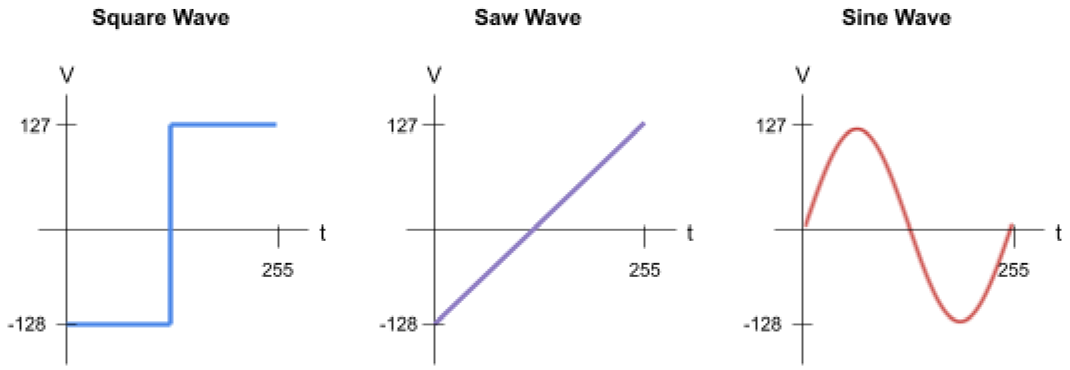


Figure 3.4: The square, saw and sine waveforms: t represents the sample index ranging from 0 to 255, while V represents the sample value, ranging from -128 to 127.

In order to play these waveforms at different frequencies, we needed to adjust how we read in samples for different notes. In order to achieve this we gave each of the 8 notes a counter, stored in the unsigned int counter[8] array. This counter represented where each note was along its given waveform at any given time and was treated as a fixed point number, with the most significant 8 bits representing the array index, and the least significant 8 bits representing a decimal for further precision (Figure 3.5). Representing the wave position of each note in this way had 2 main advantages:

(1) It allowed us to be more precise in the rate at which we decided to increment the counter for each note, as we had an additional 8 decimal bits of precision.

(2) Once the counter reached the end of the sample it would automatically circle back to the beginning due to integer overflow; no additional operations were needed for this to work.

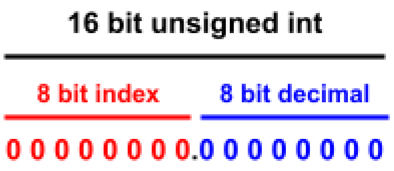


Figure 3.5: How an unsigned int is used as a fixed point number with 8 decimal bits.

In order to progress the wave, we only needed to increment the counter during each call of the interrupt and use the most significant 8 bits of the counter as an index into our waveform arrays to find the new sample for that note. Meanwhile we could adjust the frequency of each note by adjusting how much we ticked up its counter during each run of the interrupt. We will call this increment a given note's $DELTA$.

$$DELTA = \frac{NOTE\_FREQUENCY \times INT\_COUNT \times SAMPLE\_RATE}{CLOCK\_SPEED} \qquad \text{(Eq 3.0)}$$

Where:

- $NOTE\_FREQUENCY$ is the frequency in Hz.
- INT_COUNT is the number of possible integers ($2^{16} = 65536$)
- $SAMPLE\_RATE$ is the number of clock cycles between samples (512)
- $CLOCK\_SPEED$ is the clock speed of the MSP430 in Hz; by default roughly 1048000, although this is increased in the next section.

The $DELTA$'s for notes in the seventh octave were calculated (Appendix A). These values could then be bit shifted to the right to get note deltas in lower octaves (Appendix B). For each note, we added its $DELTA$ to its counter each time the interrupt runs. This $DELTA$ was determined by the currently active scale and specific note in said scale. We then added together the waveform samples of each note, using our isPlaying[8] array to mask out notes which were not playing. After this the sample ranged from -128×8 to 127×8, so we added 1024 to have the sample range from 0 to 255×8. By doing a right bit shift by 3 we could divide this value by 8 to get a sample between 0 to 255 which we could then output as an 8 bit number. Using all these optimizations we could get an interrupt routine that runs in only 512 clock cycles (Appendix K).

3.2.4 Increasing the Clock Speed

The other key component to maximizing the sample rate was increasing the clock speed. By default the MSP430 operates at a roughly $1MHz$ clock speed. In order to increase the clock speed I used the configureClock() function in Appendix F to increase this to around $12\ MHz$. The code for this function was taken from sample code developed by Texas Instruments[11]. This code adjusts the frequency of the DCO (Digitally Controlled Oscillator) used as the main clock on the MSP430. This was the only part of my program that I did not write myself. Since setting the DCO in this way actually allows it have a range of clock frequencies around $12\ MHz$, to find the exact clock frequency, I worked backwards from equation 3.0. Once the harp was built I used a tuner to adjust the clock speed in equation 3.0, until the frequency of the notes was correct. It was found that the actual clock speed was approximately $12.3\ Mhz$. This allowed for a sample rate of $24.1\ Khz$, which was not quite the ideal $44.1\ Khz$, but quite close.

3.2.5 Outputting to the DAC

Similar to the read function for input, we also had a write function (Appendix D) for outputting our 8 bit sample to the DAC. For each output pin (both audio and DAC control) I initialized their PxDIR to 1 and PxOUT to 0, except for P4.2 which was used for enabling writing on the DAC. This pin had its PxOUT initialized to 1 (writing disabled) (Appendix H). When writing a value I first updated the audio output pins, and then grounded the Write Enable pin on the DAC to write the value, before raising it again.

3.3 Output Device (Speaker)

The output system consisted of many small components. First the 8 bit audio output from the MSP was fed through a DAC (Digital to Analog Converter) to translate the output to a voltage between $0V$ and $5V$. Due to this voltage not actually being able to power a load, it was then fed through a buffer op amp. After the buffer op amp, the signal was passed through a low pass filter to smooth out the output wave, and finally through an additional op amp to amplify the current. A DC offset was then applied using a $200\mu F$ capacitor, before the signal was fed through a speaker.

3.3.1 Digital to Audio Converter

The TLC7528CN Dual 8 Bit Multiplying Digital to Analog Converter was used as the DAC in this circuit, with its configuration shown in Figure 3.6.

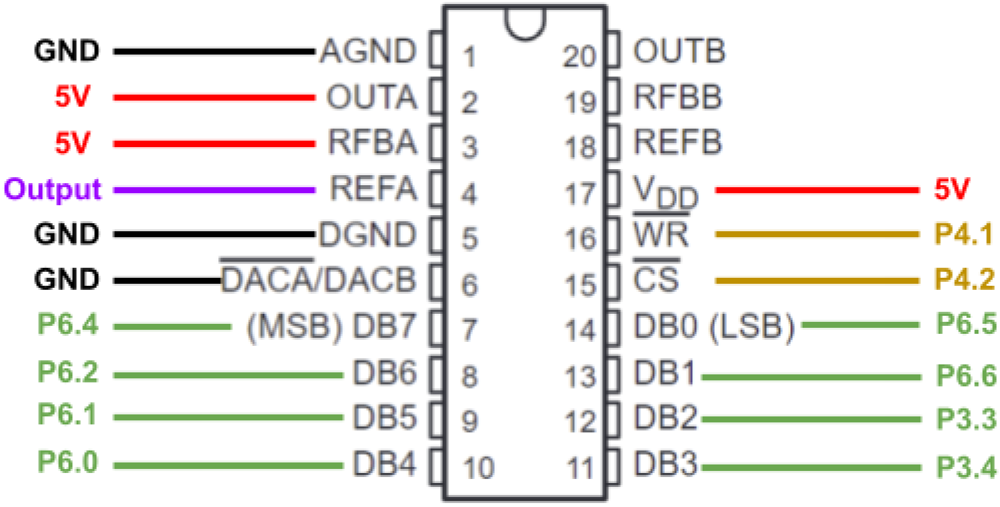


Figure 3.6: The Pin Configuration Diagram for the TLC7528CN (from the manual's 1st page)[7] with its connections to the MSP430 and bread board. The $OUTB$, $RFBB$ and $REFB$ pins were left unconnected as only input A was being used.

This chip had 2 different DAC input/outputs, the A and B DAC, but only the A input/output was used[7]. The control pin for switching between the 2 DACs was $\overline{DACA}/DACB$[7]. When held low input A was used and when held high input B was used[7], hence this input was grounded so input A was always used. The chip also had 2 pins for input control: the $\overline{CS}$ pin (Chip Select) and $\overline{WR}$ pin (Write Enable)[7]. These were connected to the MSP430, with both needing to be held low for the DAC to update its output. Since the Chip Select pin was used to control input to the entire chip[7], it was just held low the entire time, as we were only using the A input. The Write Enable pin would be momentarily held low to update the DAC in the write function on the MSP430 (Appendix D).

The input pins $DB0$ to $DB7$ were used for inputting our 8 bit audio sample into the DAC, and were connected to the audio output pins on the MSP430, with the most significant bit ($DB7$) connected to P6.4, and the least significant bit ($DB0$) connected to P6.5. The DAC was used in Voltage-Mode Operation (Figure 11 from the manual)[7]. In this configuration, the $REFA$ pin acted as the Analog output, while the $RFBA$ and $OUTA$ pins were held at a fixed input voltage, in this case $5V$ [7]. Finally the power and ground pins VDD, $AGND$ and $DGND$ were connected to power and ground respectively.

3.3.2 Wave Processing

After the DAC, the wave it outputted was then fed through a wave processing unit (Figure 3.7) based roughly on this guide by Amanda Ghassaei[10]; before being played through a speaker.

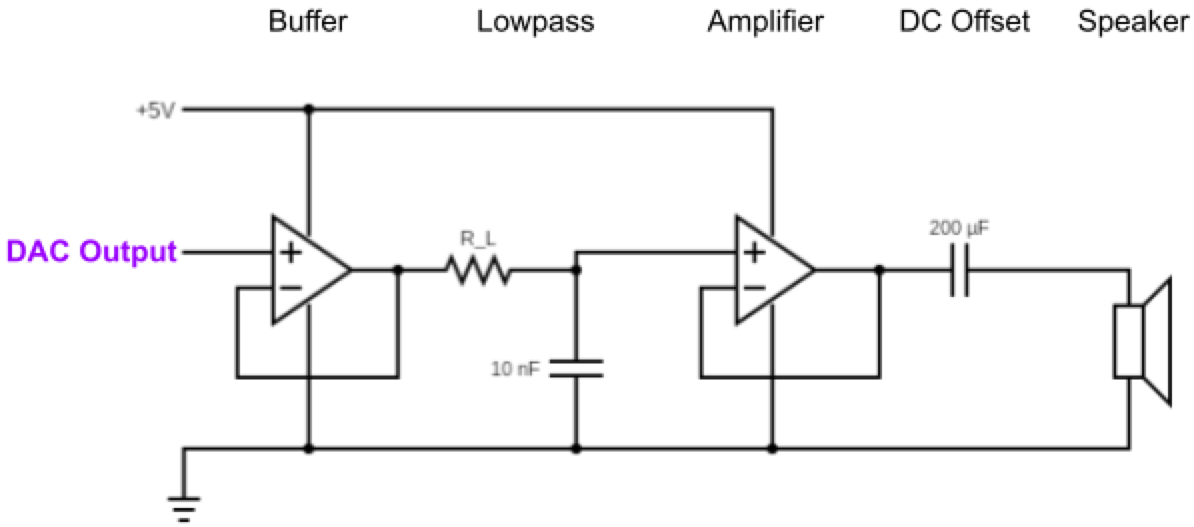


Figure 3.7: Output wave processing unit, transforming the voltage output from the DAC into a wave that can be played through a speaker.

Since the DAC output voltage was quite unstable when applied to a load, we had to first pass it through a buffer op amp to protect the output voltage. This op amp was set up as a voltage follower, meaning that the output voltage would be equal to the input voltage. An $LM358$ was used as the op amp for its low output current of $20mA$[9] minimizing energy loss in the low pass filter.

After the buffer, we then fed the signal through a low pass filter. Since the DAC could only output 256 possible voltage values, it led to small voltage steps appearing in its output wave. This low pass filter was used to smooth those out, with a cutoff frequency determined by:

$$f_{cutoff} = \frac{1}{2\pi R_L C_L} \qquad \text{(Eq 3.1)}$$

We would want a cutoff frequency no higher than the sample rate of our audio $24kHz$. Using a $C_L = 10nF$ capacitor, this would leave $R_L \geq 665\ \Omega$. Due to noise in the DAC output, a cutoff frequency of approximately $408Hz$ was used, or an $R_L = 39k\Omega$. This is discussed further in the Results section. After the low pass filter, the current of the wave then needed to be amplified to be able to play the speaker. For this another op amp - the $TCA0372DP1$ - was used for its high output current of 1 amp[8]. The current amplified wave was then fed through a $200\mu F$ capacitor acting as a DC offset, so that our wave would oscillate around $0V$, instead of $2.5V$. The wave was then fed through an 8Ω $0.4W$ speaker.

4.0 RESULTS:

4.1 Input accuracy

The laser harp itself was quite accurate in detecting notes. When the laser shone on a diode the MSP430 input pin would read at a voltage of approximately $3.079V$, and when the laser was blocked it would read at $0.386V$. The major issue was that if the laser harp was in sunlight it could not register any notes. Even sunlight on a cloudy day shining through a skylight was enough to prevent the diode readings to drop below $2.971V$, which was not enough to be detected as low by the MSP430.

4.2 Wave Quality and Amplitude

While initially a lower resistance of $R_L = 3.3\ k\Omega$ was used in the low pass filter, this led to a very noisy output as seen in Figure 4.0. In order to reduce noise I tried to maximize R_L while still allowing a signal to get through. An $R_L = 39k\Omega$ was found to work quite well, as it drastically reduced noise while still allowing for the notes to pass through. This dampened some of the higher notes which were above the cutoff frequency of $408Hz$, but this actually helped to counteract the fact that higher pitched notes naturally sound louder at similar wave amplitudes.

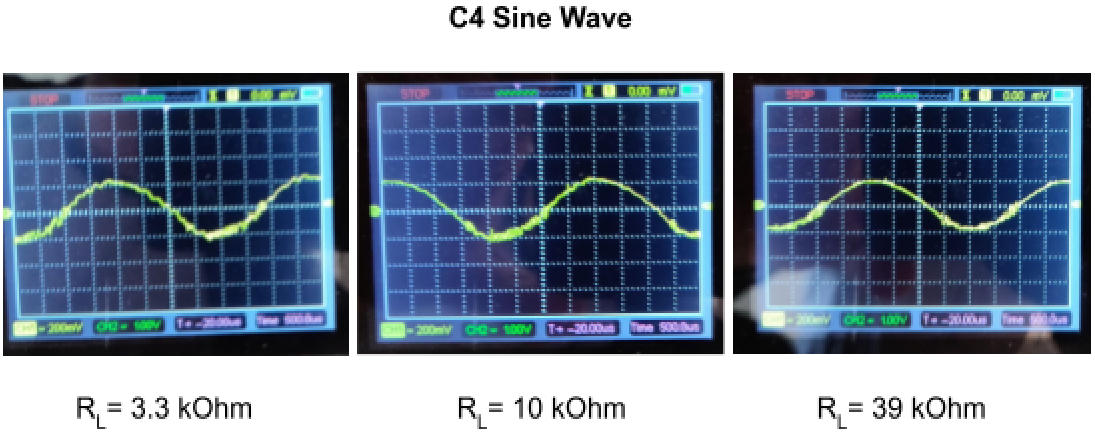


Figure 4.0: C4 sine wave (261.6 Hz), output readings on an oscilloscope at different R_L values used in the low pass filter. The higher the resistance, the less noisy the wave.

As seen in Figure 4.1 however, this stronger low pass filter did distort the square and saw waves. Regardless, they still sounded fine when played through the speaker.

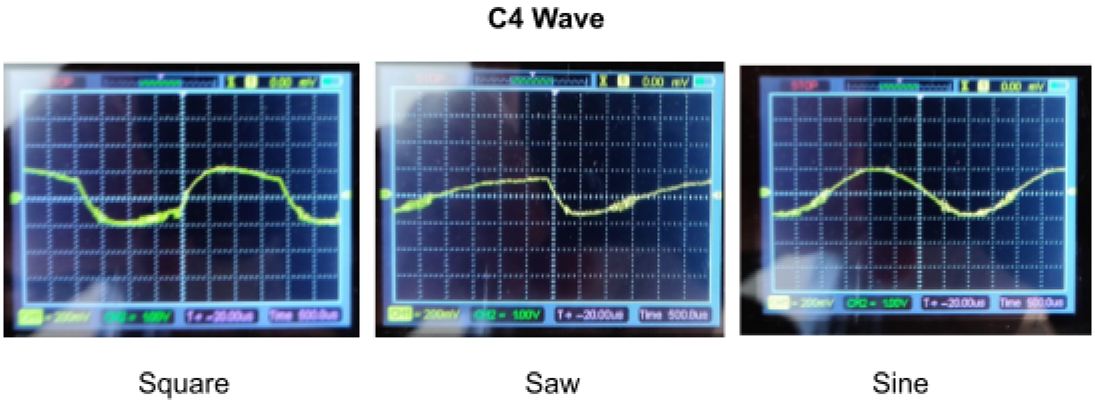


Figure 4.1: C4 wave (261.6 Hz), output readings on an oscilloscope, using different waveforms at $R_L = 39k\Omega$.

Also of note is that the waves had quite a low amplitude, with the peak to peak amplitude sitting at around $400mV$ as seen in Figure 4.1. The expected amplitude for a single note should have been roughly $5V\ /\ 8\ =\ 625mV$, to prevent peaking when all 8 notes were being played, so having only a $400mV$ amplitude is unexpected, but may have been caused by power dissipation in the circuit. This wave amplitude was very quiet when played through the speaker, but even if it was $625mV$, this would not have made a significant difference.

4.3 Frequency Accuracy

Each note in the C4 major scale was also tested to see how its frequency compared to the expected frequency for that note. For each note a maximum and minimum frequency reading was taken based on fluctuations in the readings during data collection. The percent error between these maximum and minimum frequencies and the expected frequencies was then found and plotted in Figure 4.2.

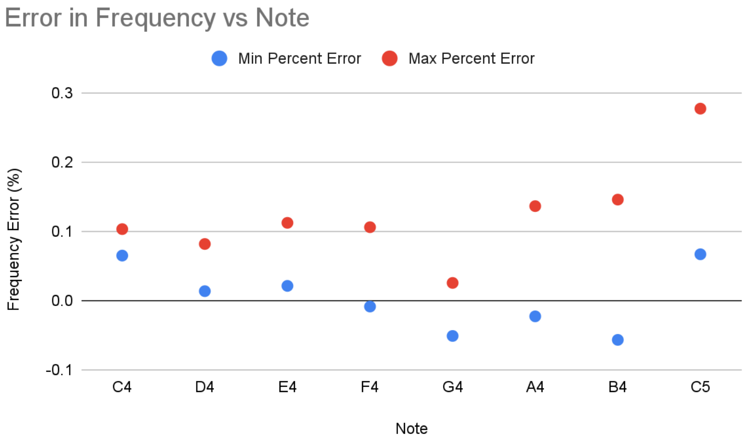


Figure 4.2: The error in frequency vs note for the output of the harp for notes in the C4 major scale. All notes had very low percent error, but most tended to be pitched slightly too high. Frequencies were recorded on a Oneplus 6 mobile phone using https://muted.io/chromatic-tuner/.

Overall the percent error in the frequencies was very small, with the largest absolute percent error sitting below 0.3%. Most notes tended to be pitched slightly too high however.

4.4 Demo

Here are some videos of me demoing the harp. The awkward camera angle for most of them was due to the speaker being very quiet, hence forcing me to place my phone microphone right next to the speaker to pick up the sound from the harp.

Demo	**Waveform Used**	**Scale Used**
C Major Scale	Square	C4 Major
Crab Rave	Sine	G4 Mixolydian
Take On Me	Square	A4 Dorian
Tetris Theme	Saw	A4 Minor
Mary had a Little Lamb	Sine	C4 Major
Twinkle Twinkle	Sine	C4 Major
Sound Mixing Demo	Sine	C4 Major

5.0 DISCUSSION:

Overall the laser harp worked quite well, but had a few quirks. Things that worked well included the sound mixing functionality, and the accuracy of the note frequencies. Things could be improved include:

- Wave Amplitude: the wave amplitude was very small, leading to the speaker being very quiet. This could be fixed by adding a voltage amplifier to the end of the wave processing unit. For example a Non Inverting Amplifier circuit using an op amp and resistors could increase the amplitude of the wave.
- Wave Quality: the waves were quite noisy requiring a very strong low pass filter to be

used. Overall wave quality could be improved by having cleaner wiring to prevent possible interference effects.

- Pin Usage: the pins used on the MSP430 were not especially efficient for reading and writing to, due to them being quite spaced out across different registers, requiring multiple lines of code to read or write a single value. This was mainly due to many pins on my MSP430 being broken, hence getting a better working microcontroller could help reduce write and read times, and might therefore lead to a higher sample rate.

The harp could also benefit from the following features:

- A display interface, for showing the current scale and waveform selected.
- A stand, so that the harp could stand vertically, allowing the instrument to more easily be played; having the instrument lie flat on a table led to me constantly banging my hand into the table while playing notes.
- An exponential decay on note amplitudes when a note is released, to more closely mimic the sound of real instruments. Right now the instant cutoff sounds a bit jarring.

6.0 CONCLUSION:

Overall the harp worked quite well, but due to it being very quiet it is not ideal for actual performances. This project was a lot of work to build, but I learned many things, especially about converting a digital signal to an audio output. The sound mixing code also challenged my programming skills, but was my favourite part of the project due to my background in computer science. I had to learn a lot about the computational limitations of the MSP430 in order to write efficient code. I would recommend this project to anyone who has an interest in music or audio engineering.

References:

1. “Build your own laser harp using Arduino” by Ibrar Ayyub

   https://duino4projects.com/build-your-own-laser-harp-using-arduino/#google_vignette

2. “Laser Harp: How does it work?” by Theremin Hero

   https://www.youtube.com/watch?v=nuM2Jw78u8Y

3. “What is Photodiode?” by Automatedo

   https://www.youtube.com/watch?v=rNoHLOumplk

4. “How Do DACs Work?” by Element14 Presents

   https://www.youtube.com/watch?v=YAxrmoVtEtE

5. “Passive Low Pass Filter” by Electronics Tutorials

   https://www.electronics-tutorials.ws/filter/filter_2.html

6. “How Speakers Make Sound” by Animagraffs

   https://www.youtube.com/watch?v=RxdFP31QYAg

7. TLC7528 Datasheet by Texas Instruments

   https://www.ti.com/lit/ds/symlink/tlc7528.pdf?ts=1744316167015&ref_url=https%253A%252F%252Fwww.google.com%252F

8. TCA0372 Datasheet by Mototrala

   https://www.alldatasheet.com/datasheet-pdf/view/5752/MOTOROLA/TCA0372.html

9. LM358 Datasheet by Texas Instruments

   https://www.ti.com/lit/ds/symlink/lm358.pdf?ts=1744289053951&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FLM358

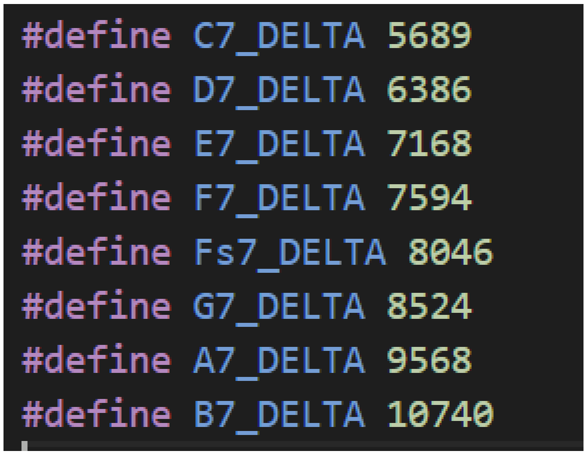
10. “Arduino Audio Output” by Amanada Ghassaei

    https://www.instructables.com/Arduino-Audio-Output/

11. “MSP430F552x Demo - Software Toggle P1.1 with 12MHz DCO” by Bhargavi Nisarga (Texas Instruments) https://dev.ti.com/tirex/explore/node?node=A__AFkcScYdp-l.HKcp8rxS9w__msp430ware__IOGqZri__LATEST
12. “MSP430F5529 LaunchPad™ Development Kit” by Texas Instruments https://www.ti.com/lit/ug/slau533d/slau533d.pdf?ts=1744311113652

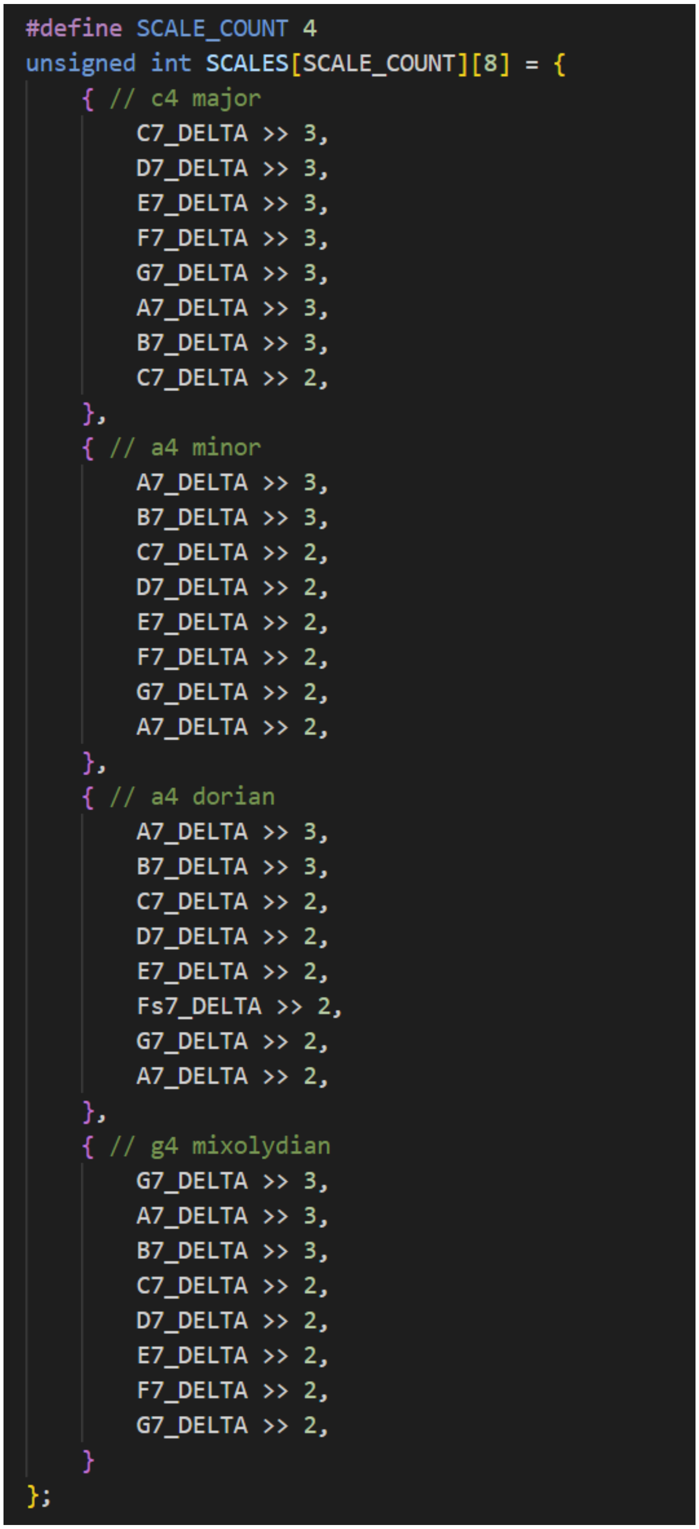
Appendix:

Appendix A: The note deltas for notes in the C7 major scale. These values can be bit shifted to the right to get note deltas in lower octaves, like seen in Appendix B.



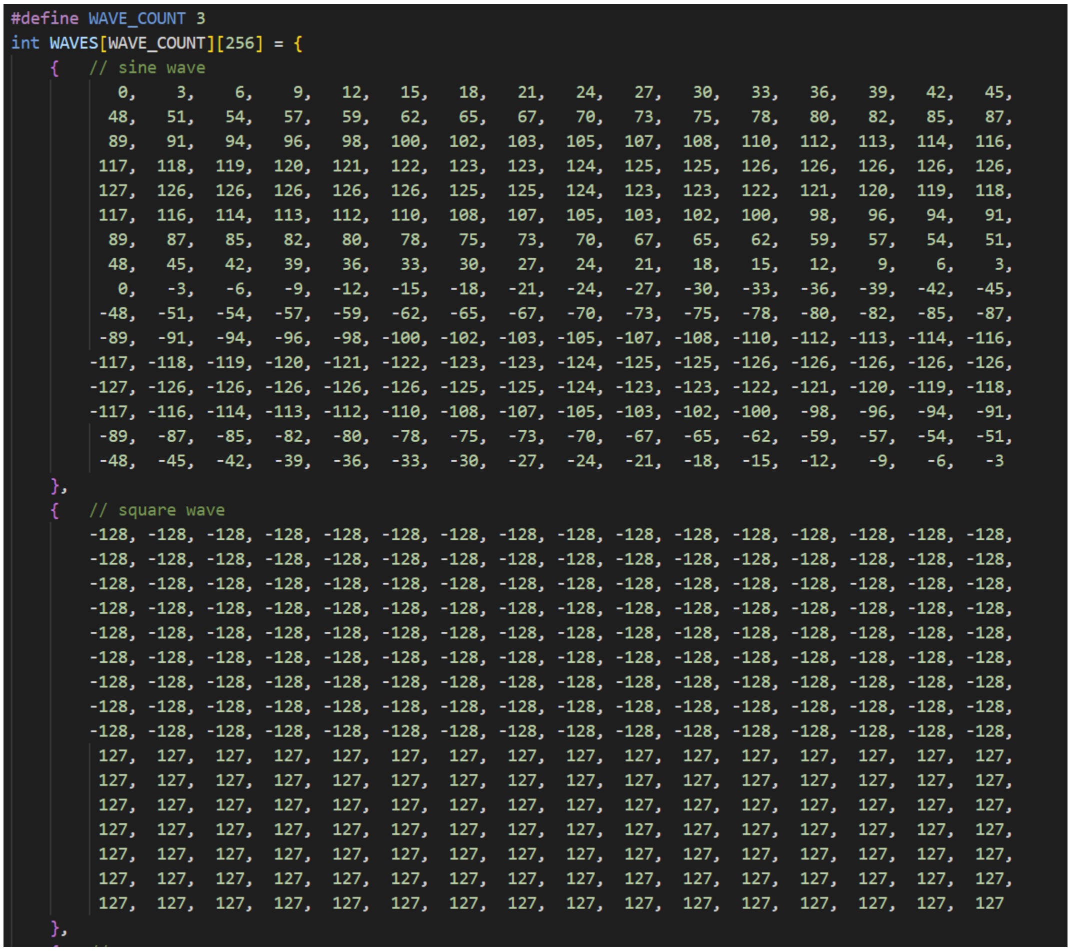
```
#define C7_DELTA 5689
#define D7_DELTA 6386
#define E7_DELTA 7168
#define F7_DELTA 7594
#define Fs7_DELTA 8046
#define G7_DELTA 8524
#define A7_DELTA 9568
#define B7_DELTA 10740
```

Appendix B: The scales available on the harp.

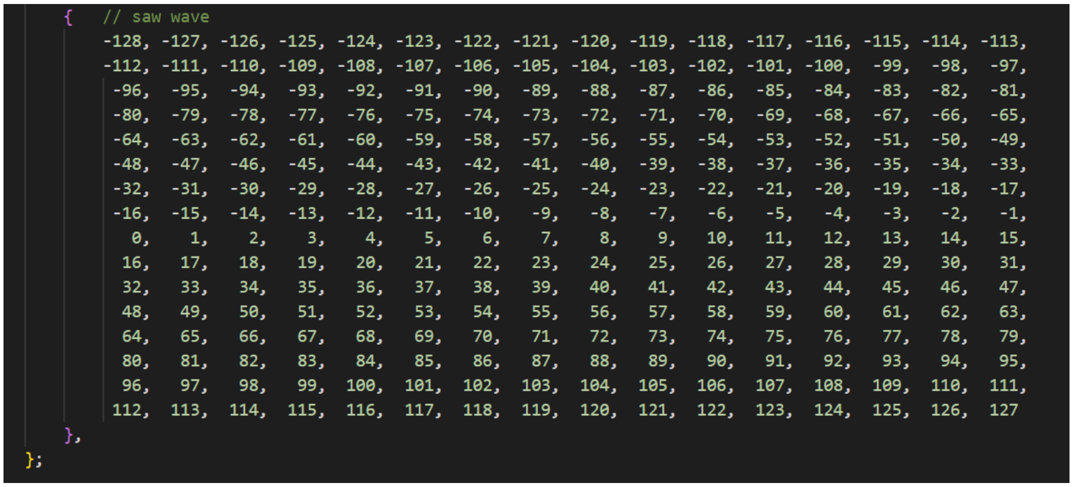


```c
#define SCALE_COUNT 4
unsigned int SCALES[SCALE_COUNT][8] = {
    { // c4 major
        C7_DELTA >> 3,
        D7_DELTA >> 3,
        E7_DELTA >> 3,
        F7_DELTA >> 3,
        G7_DELTA >> 3,
        A7_DELTA >> 3,
        B7_DELTA >> 3,
        C7_DELTA >> 2,
    },
    { // a4 minor
        A7_DELTA >> 3,
        B7_DELTA >> 3,
        C7_DELTA >> 2,
        D7_DELTA >> 2,
        E7_DELTA >> 2,
        F7_DELTA >> 2,
        G7_DELTA >> 2,
        A7_DELTA >> 2,
    },
    { // a4 dorian
        A7_DELTA >> 3,
        B7_DELTA >> 3,
        C7_DELTA >> 2,
        D7_DELTA >> 2,
        E7_DELTA >> 2,
        Fs7_DELTA >> 2,
        G7_DELTA >> 2,
        A7_DELTA >> 2,
    },
    { // g4 mixolydian
        G7_DELTA >> 3,
        A7_DELTA >> 3,
        B7_DELTA >> 3,
        C7_DELTA >> 2,
        D7_DELTA >> 2,
        E7_DELTA >> 2,
        F7_DELTA >> 2,
        G7_DELTA >> 2,
    }
};
```

Appendix C: Waveform samples available on the harp

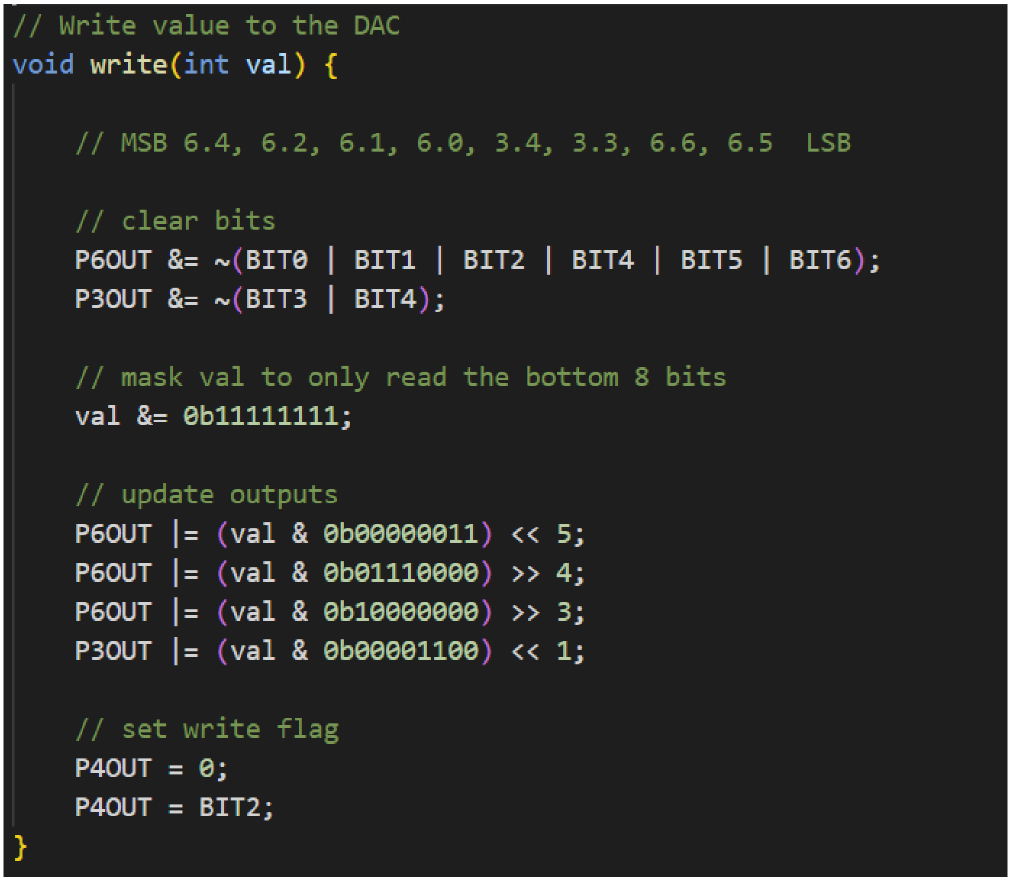


```
#define WAVE_COUNT 3
int WAVES[WAVE_COUNT][256] = {
    {   // sine wave
           0,    3,    6,    9,   12,   15,   18,   21,   24,   27,   30,   33,   36,   39,   42,   45,
          48,   51,   54,   57,   59,   62,   65,   67,   70,   73,   75,   78,   80,   82,   85,   87,
          89,   91,   94,   96,   98,  100,  102,  103,  105,  107,  108,  110,  112,  113,  114,  116,
         117,  118,  119,  120,  121,  122,  123,  123,  124,  125,  125,  126,  126,  126,  126,  126,
         127,  126,  126,  126,  126,  126,  125,  125,  124,  123,  123,  122,  121,  120,  119,  118,
         117,  116,  114,  113,  112,  110,  108,  107,  105,  103,  102,  100,   98,   96,   94,   91,
          89,   87,   85,   82,   80,   78,   75,   73,   70,   67,   65,   62,   59,   57,   54,   51,
          48,   45,   42,   39,   36,   33,   30,   27,   24,   21,   18,   15,   12,    9,    6,    3,
           0,   -3,   -6,   -9,  -12,  -15,  -18,  -21,  -24,  -27,  -30,  -33,  -36,  -39,  -42,  -45,
         -48,  -51,  -54,  -57,  -59,  -62,  -65,  -67,  -70,  -73,  -75,  -78,  -80,  -82,  -85,  -87,
         -89,  -91,  -94,  -96,  -98, -100, -102, -103, -105, -107, -108, -110, -112, -113, -114, -116,
        -117, -118, -119, -120, -121, -122, -123, -123, -124, -125, -125, -126, -126, -126, -126, -126,
        -127, -126, -126, -126, -126, -126, -125, -125, -124, -123, -123, -122, -121, -120, -119, -118,
        -117, -116, -114, -113, -112, -110, -108, -107, -105, -103, -102, -100,  -98,  -96,  -94,  -91,
         -89,  -87,  -85,  -82,  -80,  -78,  -75,  -73,  -70,  -67,  -65,  -62,  -59,  -57,  -54,  -51,
         -48,  -45,  -42,  -39,  -36,  -33,  -30,  -27,  -24,  -21,  -18,  -15,  -12,   -9,   -6,   -3
    },
    {   // square wave
        -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128,
        -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128,
        -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128,
        -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128,
        -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128,
        -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128,
        -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128,
        -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128,
        -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128,
         127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,
         127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,
         127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,
         127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,
         127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,
         127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,
         127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127
    },
```



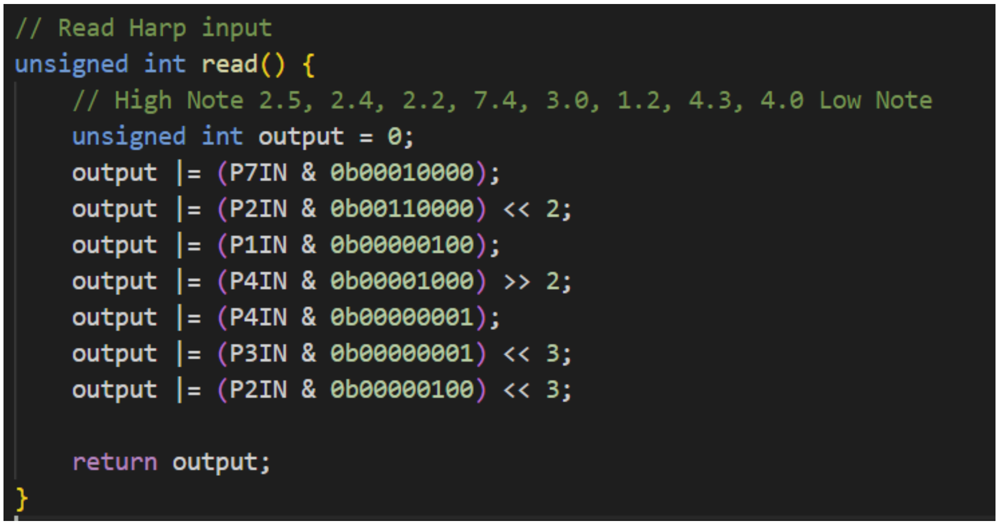
```
    {   // saw wave
        -128, -127, -126, -125, -124, -123, -122, -121, -120, -119, -118, -117, -116, -115, -114, -113,
        -112, -111, -110, -109, -108, -107, -106, -105, -104, -103, -102, -101, -100,  -99,  -98,  -97,
         -96,  -95,  -94,  -93,  -92,  -91,  -90,  -89,  -88,  -87,  -86,  -85,  -84,  -83,  -82,  -81,
         -80,  -79,  -78,  -77,  -76,  -75,  -74,  -73,  -72,  -71,  -70,  -69,  -68,  -67,  -66,  -65,
         -64,  -63,  -62,  -61,  -60,  -59,  -58,  -57,  -56,  -55,  -54,  -53,  -52,  -51,  -50,  -49,
         -48,  -47,  -46,  -45,  -44,  -43,  -42,  -41,  -40,  -39,  -38,  -37,  -36,  -35,  -34,  -33,
         -32,  -31,  -30,  -29,  -28,  -27,  -26,  -25,  -24,  -23,  -22,  -21,  -20,  -19,  -18,  -17,
         -16,  -15,  -14,  -13,  -12,  -11,  -10,   -9,   -8,   -7,   -6,   -5,   -4,   -3,   -2,   -1,
           0,    1,    2,    3,    4,    5,    6,    7,    8,    9,   10,   11,   12,   13,   14,   15,
          16,   17,   18,   19,   20,   21,   22,   23,   24,   25,   26,   27,   28,   29,   30,   31,
          32,   33,   34,   35,   36,   37,   38,   39,   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
    },
};
```

Appendix D: The write function, used for outputting audio samples to the DAC



```c
// Write value to the DAC
void write(int val) {

    // MSB 6.4, 6.2, 6.1, 6.0, 3.4, 3.3, 6.6, 6.5  LSB

    // clear bits
    P6OUT &= ~(BIT0 | BIT1 | BIT2 | BIT4 | BIT5 | BIT6);
    P3OUT &= ~(BIT3 | BIT4);

    // mask val to only read the bottom 8 bits
    val &= 0b11111111;

    // update outputs
    P6OUT |= (val & 0b00000011) << 5;
    P6OUT |= (val & 0b01110000) >> 4;
    P6OUT |= (val & 0b10000000) >> 3;
    P3OUT |= (val & 0b00001100) << 1;

    // set write flag
    P4OUT = 0;
    P4OUT = BIT2;
}
```

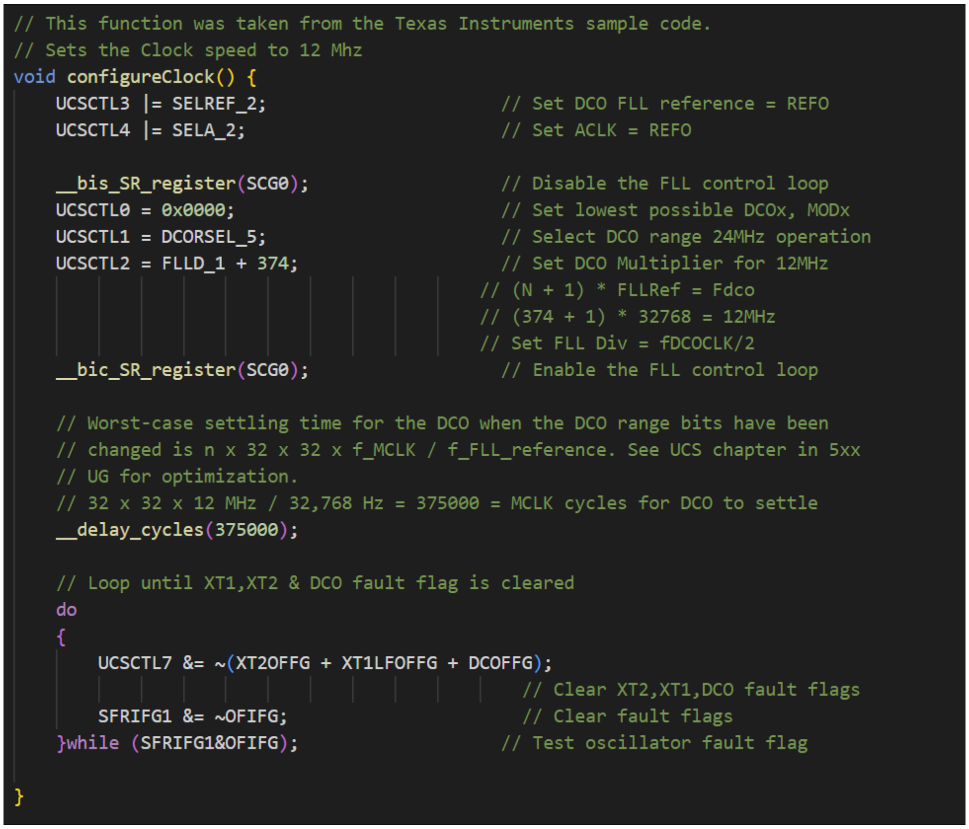
Appendix E: The read function, used for reading which notes are being played on the harp.



```
// Read Harp input
unsigned int read() {
    // High Note 2.5, 2.4, 2.2, 7.4, 3.0, 1.2, 4.3, 4.0 Low Note
    unsigned int output = 0;
    output |= (P7IN & 0b00010000);
    output |= (P2IN & 0b00110000) << 2;
    output |= (P1IN & 0b00000100);
    output |= (P4IN & 0b00001000) >> 2;
    output |= (P4IN & 0b00000001);
    output |= (P3IN & 0b00000001) << 3;
    output |= (P2IN & 0b00000100) << 3;

    return output;
}
```

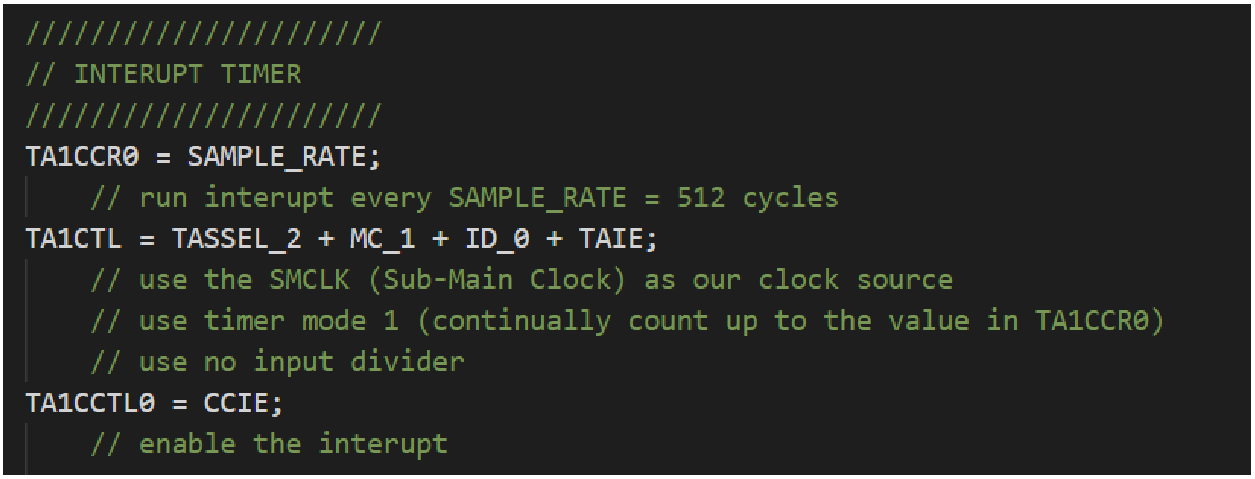
Appendix F: The configureClock function, used for configuring the MSP430 DCO to 12 MHz.

This code, including comments, was taken from the Texas Instruments code samples[11].



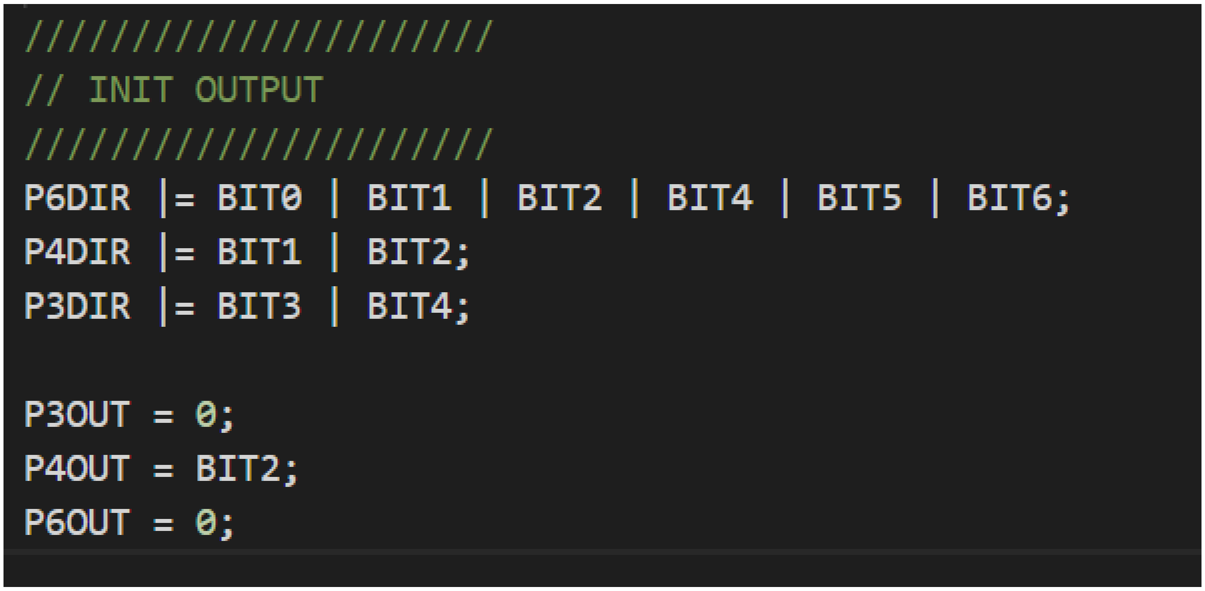
```
// This function was taken from the Texas Instruments sample code.
// Sets the Clock speed to 12 Mhz
void configureClock() {
    UCSCTL3 |= SELREF_2;                      // Set DCO FLL reference = REFO
    UCSCTL4 |= SELA_2;                        // Set ACLK = REFO

    __bis_SR_register(SCG0);                  // Disable the FLL control loop
    UCSCTL0 = 0x0000;                         // Set lowest possible DCOx, MODx
    UCSCTL1 = DCORSEL_5;                      // Select DCO range 24MHz operation
    UCSCTL2 = FLLD_1 + 374;                   // Set DCO Multiplier for 12MHz
                                            // (N + 1) * FLLRef = Fdco
                                            // (374 + 1) * 32768 = 12MHz
                                            // Set FLL Div = fDCOCLK/2
    __bic_SR_register(SCG0);                  // Enable the FLL control loop

    // Worst-case settling time for the DCO when the DCO range bits have been
    // changed is n x 32 x 32 x f_MCLK / f_FLL_reference. See UCS chapter in 5xx
    // UG for optimization.
    // 32 x 32 x 12 MHz / 32,768 Hz = 375000 = MCLK cycles for DCO to settle
    __delay_cycles(375000);

    // Loop until XT1,XT2 & DCO fault flag is cleared
    do
    {
        UCSCTL7 &= ~(XT2OFFG + XT1LFOFFG + DCOFFG);
                                                // Clear XT2,XT1,DCO fault flags
        SFRIFG1 &= ~OFIFG;                      // Clear fault flags
    }while (SFRIFG1&OFIFG);                   // Test oscillator fault flag

}
```

Appendix G: Code for initializing the interrupt to run at a rate of once every SAMPLE_RATE = 512 clock cycles.



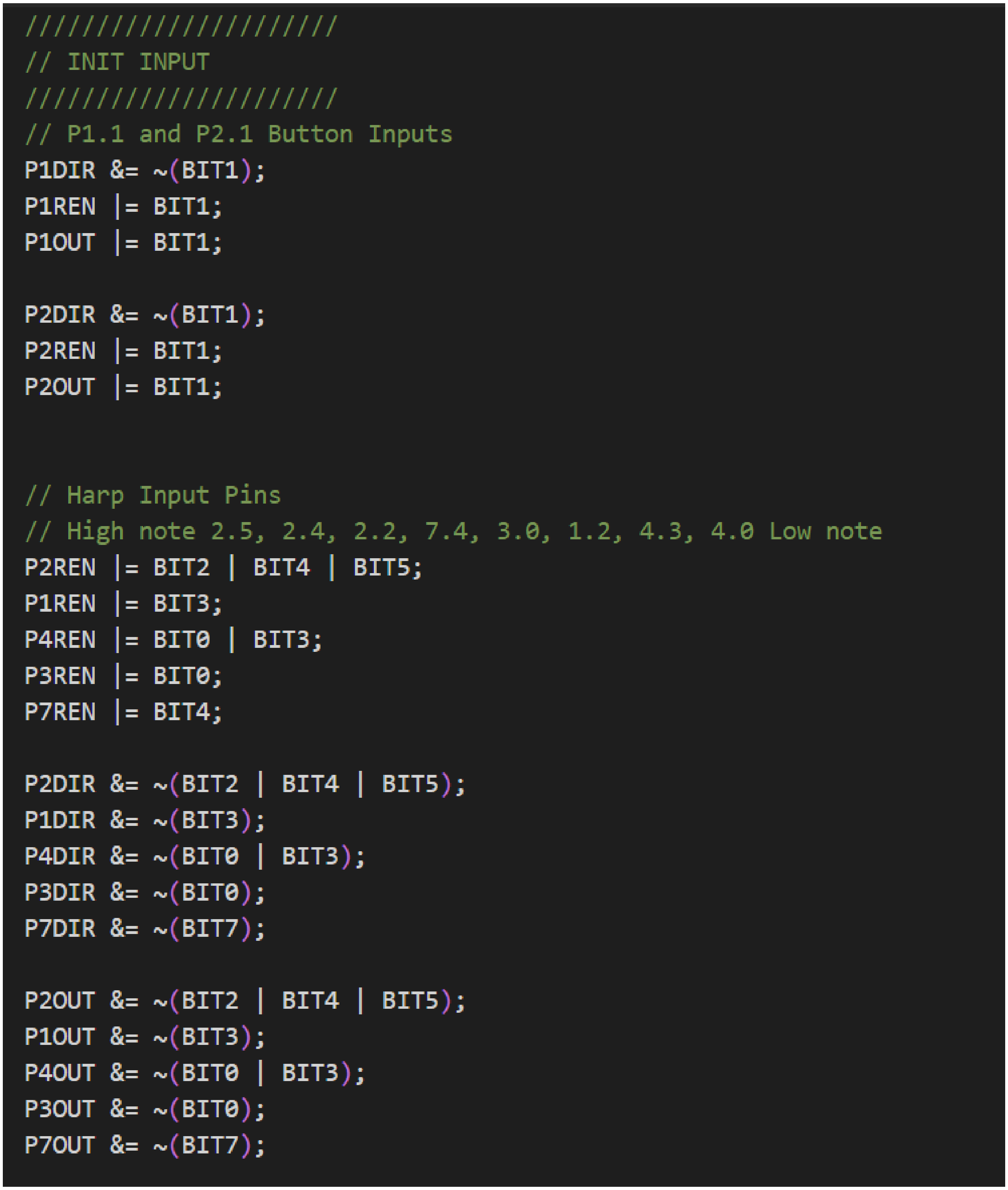
```
//////////////////////
// INTERUPT TIMER
//////////////////////
TA1CCR0 = SAMPLE_RATE;
    // run interupt every SAMPLE_RATE = 512 cycles
TA1CTL = TASSEL_2 + MC_1 + ID_0 + TAIE;
    // use the SMCLK (Sub-Main Clock) as our clock source
    // use timer mode 1 (continually count up to the value in TA1CCR0)
    // use no input divider
TA1CCTL0 = CCIE;
    // enable the interupt
```

Appendix H: Code for initializing the output pins



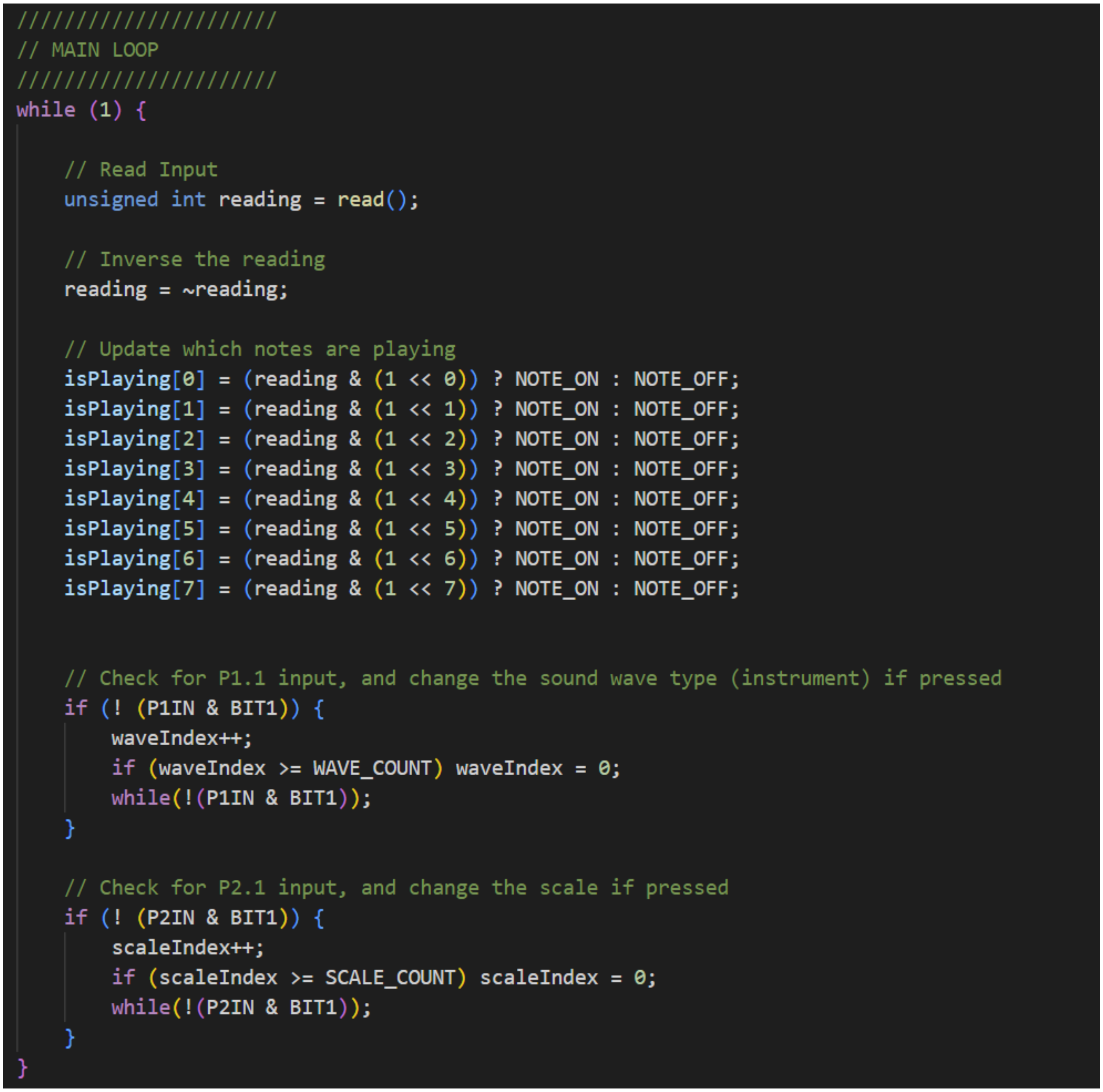
```
//////////////////////
// INIT OUTPUT
//////////////////////
P6DIR |= BIT0 | BIT1 | BIT2 | BIT4 | BIT5 | BIT6;
P4DIR |= BIT1 | BIT2;
P3DIR |= BIT3 | BIT4;

P3OUT = 0;
P4OUT = BIT2;
P6OUT = 0;
```

Appendix I: Code for initializing the input pins and buttons P1.1 and P2.1



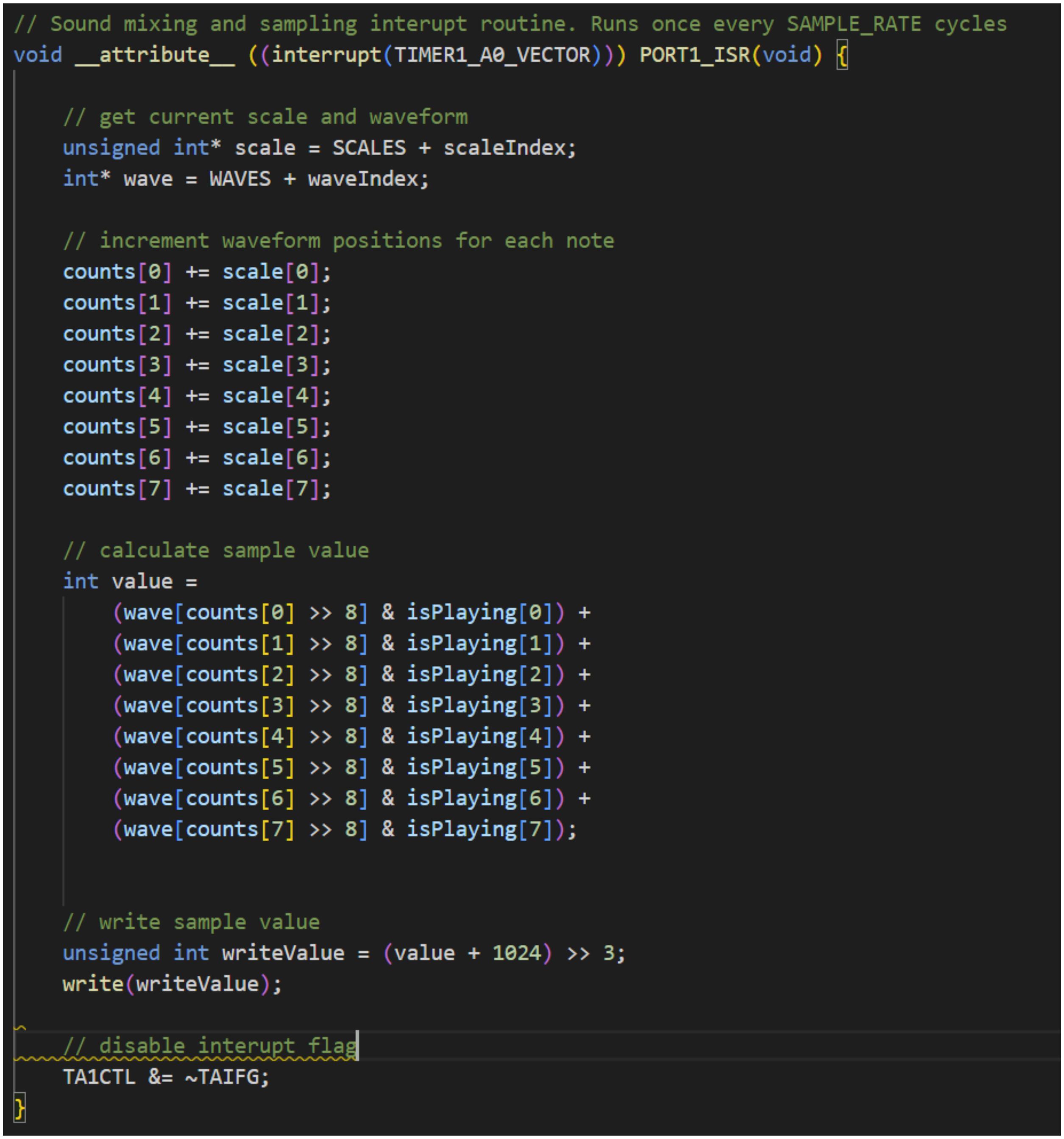
```
//////////////////////
// INIT INPUT
//////////////////////
// P1.1 and P2.1 Button Inputs
P1DIR &= ~(BIT1);
P1REN |= BIT1;
P1OUT |= BIT1;

P2DIR &= ~(BIT1);
P2REN |= BIT1;
P2OUT |= BIT1;


// Harp Input Pins
// High note 2.5, 2.4, 2.2, 7.4, 3.0, 1.2, 4.3, 4.0 Low note
P2REN |= BIT2 | BIT4 | BIT5;
P1REN |= BIT3;
P4REN |= BIT0 | BIT3;
P3REN |= BIT0;
P7REN |= BIT4;

P2DIR &= ~(BIT2 | BIT4 | BIT5);
P1DIR &= ~(BIT3);
P4DIR &= ~(BIT0 | BIT3);
P3DIR &= ~(BIT0);
P7DIR &= ~(BIT7);

P2OUT &= ~(BIT2 | BIT4 | BIT5);
P1OUT &= ~(BIT3);
P4OUT &= ~(BIT0 | BIT3);
P3OUT &= ~(BIT0);
P7OUT &= ~(BIT7);
```

Appendix J: The main loop



```c
//////////////////////
// MAIN LOOP
//////////////////////
while (1) {

    // Read Input
    unsigned int reading = read();

    // Inverse the reading
    reading = ~reading;

    // Update which notes are playing
    isPlaying[0] = (reading & (1 << 0)) ? NOTE_ON : NOTE_OFF;
    isPlaying[1] = (reading & (1 << 1)) ? NOTE_ON : NOTE_OFF;
    isPlaying[2] = (reading & (1 << 2)) ? NOTE_ON : NOTE_OFF;
    isPlaying[3] = (reading & (1 << 3)) ? NOTE_ON : NOTE_OFF;
    isPlaying[4] = (reading & (1 << 4)) ? NOTE_ON : NOTE_OFF;
    isPlaying[5] = (reading & (1 << 5)) ? NOTE_ON : NOTE_OFF;
    isPlaying[6] = (reading & (1 << 6)) ? NOTE_ON : NOTE_OFF;
    isPlaying[7] = (reading & (1 << 7)) ? NOTE_ON : NOTE_OFF;


    // Check for P1.1 input, and change the sound wave type (instrument) if pressed
    if (! (P1IN & BIT1)) {
        waveIndex++;
        if (waveIndex >= WAVE_COUNT) waveIndex = 0;
        while(!(P1IN & BIT1));
    }

    // Check for P2.1 input, and change the scale if pressed
    if (! (P2IN & BIT1)) {
        scaleIndex++;
        if (scaleIndex >= SCALE_COUNT) scaleIndex = 0;
        while(!(P2IN & BIT1));
    }
}
```

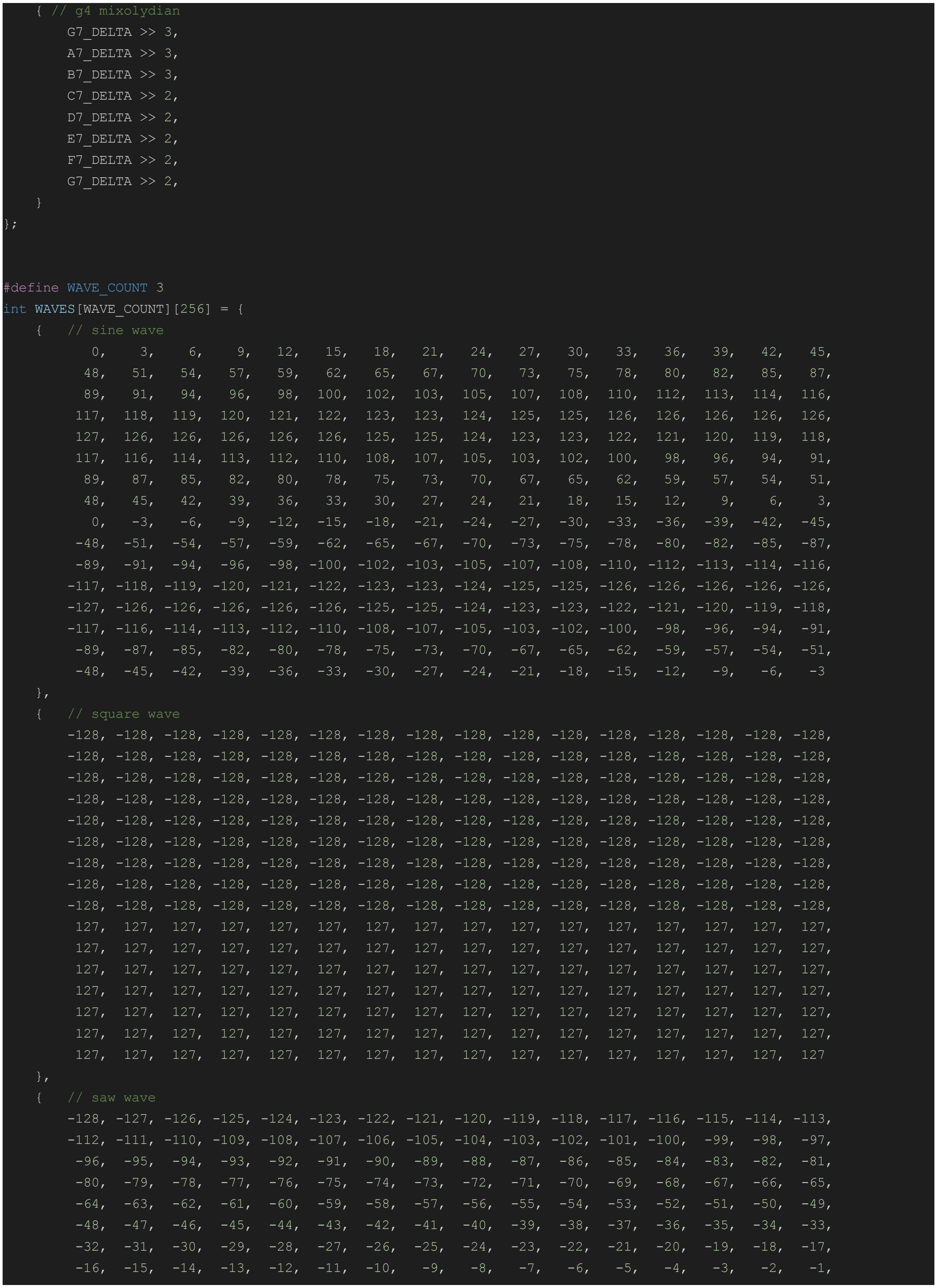
Appendix K: The interrupt routine



```c
// Sound mixing and sampling interupt routine. Runs once every SAMPLE_RATE cycles
void __attribute__ ((interrupt(TIMER1_A0_VECTOR))) PORT1_ISR(void) {

    // get current scale and waveform
    unsigned int* scale = SCALES + scaleIndex;
    int* wave = WAVES + waveIndex;

    // increment waveform positions for each note
    counts[0] += scale[0];
    counts[1] += scale[1];
    counts[2] += scale[2];
    counts[3] += scale[3];
    counts[4] += scale[4];
    counts[5] += scale[5];
    counts[6] += scale[6];
    counts[7] += scale[7];

    // calculate sample value
    int value =
        (wave[counts[0] >> 8] & isPlaying[0]) +
        (wave[counts[1] >> 8] & isPlaying[1]) +
        (wave[counts[2] >> 8] & isPlaying[2]) +
        (wave[counts[3] >> 8] & isPlaying[3]) +
        (wave[counts[4] >> 8] & isPlaying[4]) +
        (wave[counts[5] >> 8] & isPlaying[5]) +
        (wave[counts[6] >> 8] & isPlaying[6]) +
        (wave[counts[7] >> 8] & isPlaying[7]);


    // write sample value
    unsigned int writeValue = (value + 1024) >> 3;
    write(writeValue);

    // disable interupt flag
    TA1CTL &= ~TAIFG;
}
```

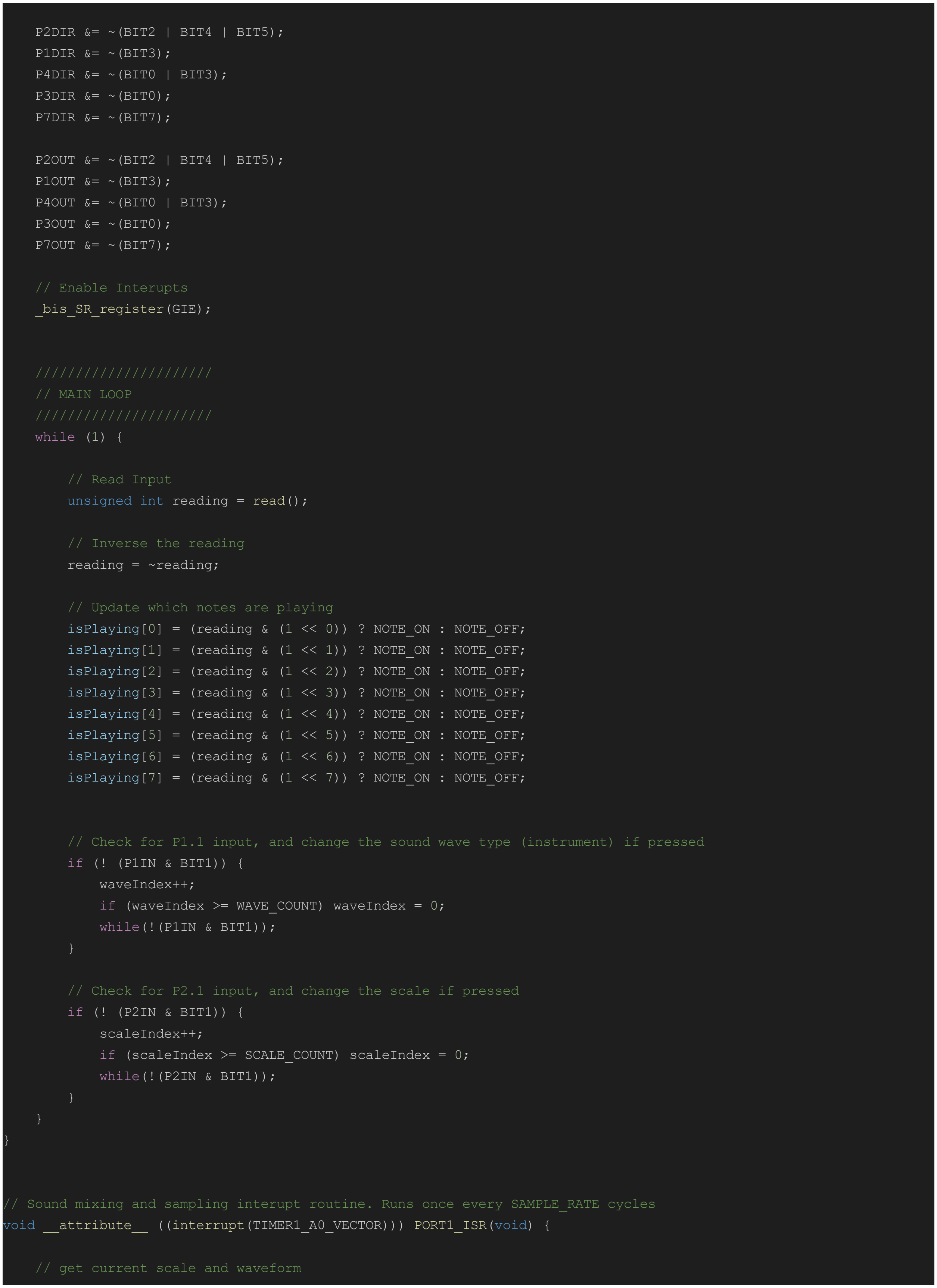
Appendix L: The entire program running on the MSP430.

```
#include "driverlib.h"
#include "intrinsics.h"
#include <msp430.h>


#define NOTE_ON 0xffff
#define NOTE_OFF 0x0000
#define SAMPLE_RATE 512



#define C7_DELTA 5689
#define D7_DELTA 6386
#define E7_DELTA 7168
#define F7_DELTA 7594
#define Fs7_DELTA 8046
#define G7_DELTA 8524
#define A7_DELTA 9568
#define B7_DELTA 10740


unsigned int counts[8];
unsigned int isPlaying[8];
unsigned int scaleIndex = 0;
unsigned int waveIndex = 0;

#define SCALE_COUNT 4
unsigned int SCALES[SCALE_COUNT][8] = {
    { // c4 major
        C7_DELTA >> 3,
        D7_DELTA >> 3,
        E7_DELTA >> 3,
        F7_DELTA >> 3,
        G7_DELTA >> 3,
        A7_DELTA >> 3,
        B7_DELTA >> 3,
        C7_DELTA >> 2,
    },
    { // a4 minor
        A7_DELTA >> 3,
        B7_DELTA >> 3,
        C7_DELTA >> 2,
        D7_DELTA >> 2,
        E7_DELTA >> 2,
        F7_DELTA >> 2,
        G7_DELTA >> 2,
        A7_DELTA >> 2,
    },
    { // a4 dorian
        A7_DELTA >> 3,
        B7_DELTA >> 3,
        C7_DELTA >> 2,
        D7_DELTA >> 2,
        E7_DELTA >> 2,
        Fs7_DELTA >> 2,
        G7_DELTA >> 2,
        A7_DELTA >> 2,
    },
```

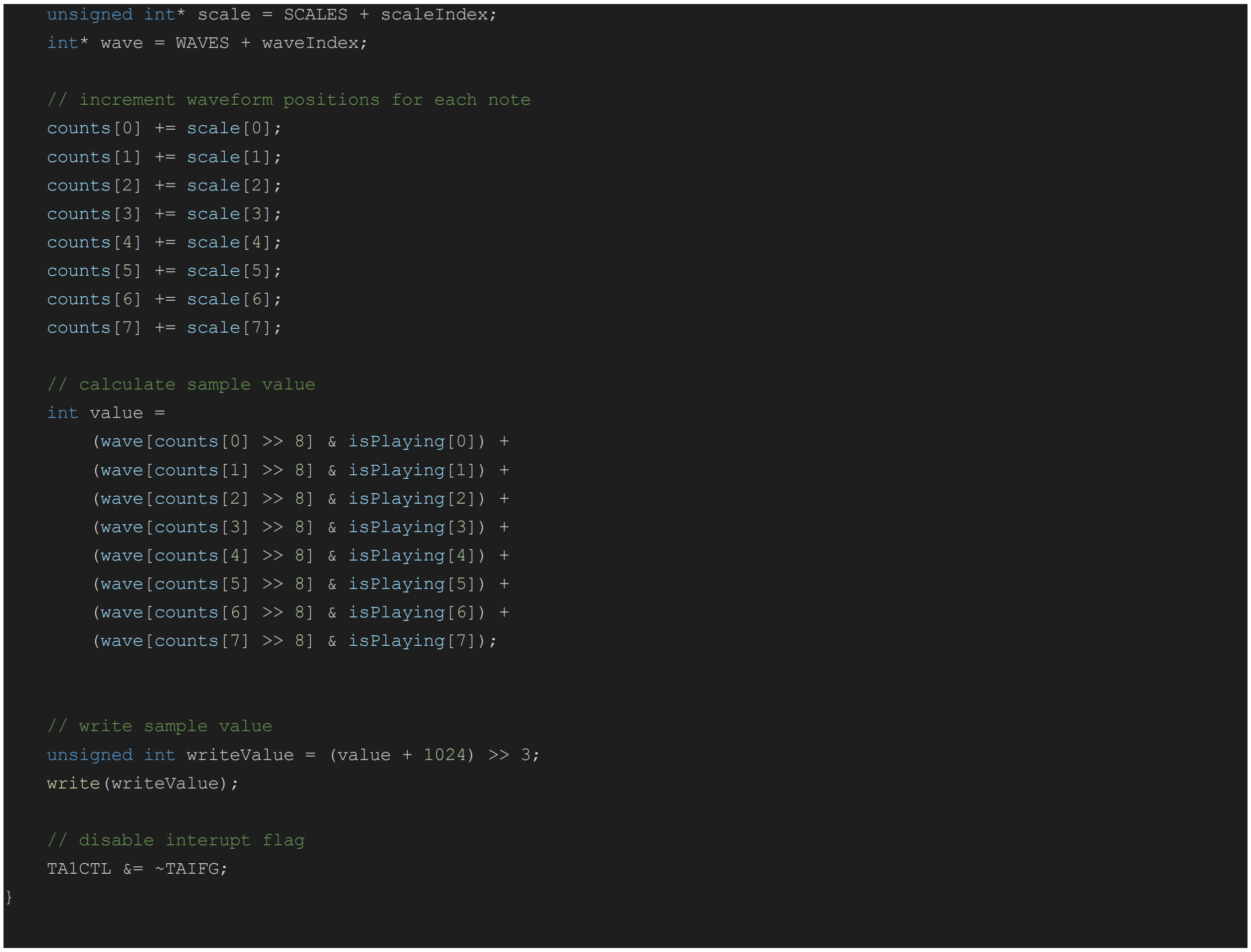


```
    { // g4 mixolydian
        G7_DELTA >> 3,
        A7_DELTA >> 3,
        B7_DELTA >> 3,
        C7_DELTA >> 2,
        D7_DELTA >> 2,
        E7_DELTA >> 2,
        F7_DELTA >> 2,
        G7_DELTA >> 2,
    }
};


#define WAVE_COUNT 3
int WAVES[WAVE_COUNT][256] = {
    {   // sine wave
           0,    3,    6,    9,   12,   15,   18,   21,   24,   27,   30,   33,   36,   39,   42,   45,
          48,   51,   54,   57,   59,   62,   65,   67,   70,   73,   75,   78,   80,   82,   85,   87,
          89,   91,   94,   96,   98,  100,  102,  103,  105,  107,  108,  110,  112,  113,  114,  116,
         117,  118,  119,  120,  121,  122,  123,  123,  124,  125,  125,  126,  126,  126,  126,  126,
         127,  126,  126,  126,  126,  126,  125,  125,  124,  123,  123,  122,  121,  120,  119,  118,
         117,  116,  114,  113,  112,  110,  108,  107,  105,  103,  102,  100,   98,   96,   94,   91,
          89,   87,   85,   82,   80,   78,   75,   73,   70,   67,   65,   62,   59,   57,   54,   51,
          48,   45,   42,   39,   36,   33,   30,   27,   24,   21,   18,   15,   12,    9,    6,    3,
           0,   -3,   -6,   -9,  -12,  -15,  -18,  -21,  -24,  -27,  -30,  -33,  -36,  -39,  -42,  -45,
         -48,  -51,  -54,  -57,  -59,  -62,  -65,  -67,  -70,  -73,  -75,  -78,  -80,  -82,  -85,  -87,
         -89,  -91,  -94,  -96,  -98, -100, -102, -103, -105, -107, -108, -110, -112, -113, -114, -116,
        -117, -118, -119, -120, -121, -122, -123, -123, -124, -125, -125, -126, -126, -126, -126, -126,
        -127, -126, -126, -126, -126, -126, -125, -125, -124, -123, -123, -122, -121, -120, -119, -118,
        -117, -116, -114, -113, -112, -110, -108, -107, -105, -103, -102, -100,  -98,  -96,  -94,  -91,
         -89,  -87,  -85,  -82,  -80,  -78,  -75,  -73,  -70,  -67,  -65,  -62,  -59,  -57,  -54,  -51,
         -48,  -45,  -42,  -39,  -36,  -33,  -30,  -27,  -24,  -21,  -18,  -15,  -12,   -9,   -6,   -3
    },
    {   // square wave
        -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128,
        -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128,
        -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128,
        -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128,
        -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128,
        -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128,
        -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128,
        -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128,
        -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128, -128,
         127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,
         127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,
         127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,
         127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,
         127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,
         127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,
         127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127,  127
    },
    {   // saw wave
        -128, -127, -126, -125, -124, -123, -122, -121, -120, -119, -118, -117, -116, -115, -114, -113,
        -112, -111, -110, -109, -108, -107, -106, -105, -104, -103, -102, -101, -100,  -99,  -98,  -97,
         -96,  -95,  -94,  -93,  -92,  -91,  -90,  -89,  -88,  -87,  -86,  -85,  -84,  -83,  -82,  -81,
         -80,  -79,  -78,  -77,  -76,  -75,  -74,  -73,  -72,  -71,  -70,  -69,  -68,  -67,  -66,  -65,
         -64,  -63,  -62,  -61,  -60,  -59,  -58,  -57,  -56,  -55,  -54,  -53,  -52,  -51,  -50,  -49,
         -48,  -47,  -46,  -45,  -44,  -43,  -42,  -41,  -40,  -39,  -38,  -37,  -36,  -35,  -34,  -33,
         -32,  -31,  -30,  -29,  -28,  -27,  -26,  -25,  -24,  -23,  -22,  -21,  -20,  -19,  -18,  -17,
         -16,  -15,  -14,  -13,  -12,  -11,  -10,   -9,   -8,   -7,   -6,   -5,   -4,   -3,   -2,   -1,
           0,     1,     2,     3,     4,     5,     6,     7,     8,     9,    10,    11,    12,    13,    14,    15,
          16,    17,    18,    19,    20,    21,    22,    23,    24,    25,    26,    27,    28,    29,    30,    31,
          32,    33,    34,    35,    36,    37,    38,    39,    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
    },
};

// Write value to the DAC
void write(int val) {

    // MSB 6.4, 6.2, 6.1, 6.0, 3.4, 3.3, 6.6, 6.5  LSB

    // clear bits
    P6OUT &= ~(BIT0 | BIT1 | BIT2 | BIT4 | BIT5 | BIT6);
    P3OUT &= ~(BIT3 | BIT4);

    // mask val to only read the bottom 8 bits
    val &= 0b11111111;

    // update outputs
    P6OUT |= (val & 0b00000011) << 5;
    P6OUT |= (val & 0b01110000) >> 4;
    P6OUT |= (val & 0b10000000) >> 3;
    P3OUT |= (val & 0b00001100) << 1;

    // set write flag
    P4OUT = 0;
    P4OUT = BIT2;
}

// Read Harp input
unsigned int read() {
    // High Note 2.5, 2.4, 2.2, 7.4, 3.0, 1.2, 4.3, 4.0 Low Note
    unsigned int output = 0;
    output |= (P7IN & 0b00010000);
    output |= (P2IN & 0b00110000) << 2;
    output |= (P1IN & 0b00000100);
    output |= (P4IN & 0b00001000) >> 2;
    output |= (P4IN & 0b00000001);
    output |= (P3IN & 0b00000001) << 3;
    output |= (P2IN & 0b00000100) << 3;

    return output;
}



// This function was taken from the Texas Instruments sample code. See the copyright notice inside.
// Sets the Clock speed to 12 Mhz
void configureClock() {

/* --COPYRIGHT--,BSD_EX
 * Copyright (c) 2012, Texas Instruments Incorporated
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * *  Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * *  Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * *  Neither the name of Texas Instruments Incorporated nor the names of
 *    its contributors may be used to endorse or promote products derived
 *    from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
 * OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
 * OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
 * EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 *******************************************************************************
 *
 *                       MSP430 CODE EXAMPLE DISCLAIMER
 *
 * MSP430 code examples are self-contained low-level programs that typically
 * demonstrate a single peripheral function or device feature in a highly
 * concise manner. For this the code may rely on the device's power-on default
 * register values and settings such as the clock configuration and care must
 * be taken when combining code from several examples to avoid potential side
 * effects. Also see www.ti.com/grace for a GUI- and www.ti.com/msp430ware
 * for an API functional library-approach to peripheral configuration.
 *
 * --/COPYRIGHT--*/


    UCSCTL3 |= SELREF_2;                      // Set DCO FLL reference = REFO
    UCSCTL4 |= SELA_2;                        // Set ACLK = REFO

    __bis_SR_register(SCG0);                  // Disable the FLL control loop
    UCSCTL0 = 0x0000;                         // Set lowest possible DCOx, MODx
    UCSCTL1 = DCORSEL_5;                      // Select DCO range 24MHz operation
    UCSCTL2 = FLLD_1 + 374;                   // Set DCO Multiplier for 12MHz
                                            // (N + 1) * FLLRef = Fdco
                                            // (374 + 1) * 32768 = 12MHz
                                            // Set FLL Div = fDCOCLK/2
    __bic_SR_register(SCG0);                  // Enable the FLL control loop

    // Worst-case settling time for the DCO when the DCO range bits have been
    // changed is n x 32 x 32 x f_MCLK / f_FLL_reference. See UCS chapter in 5xx
    // UG for optimization.
    // 32 x 32 x 12 MHz / 32,768 Hz = 375000 = MCLK cycles for DCO to settle
    __delay_cycles(375000);

    // Loop until XT1,XT2 & DCO fault flag is cleared
```

```
    do
    {
        UCSCTL7 &= ~(XT2OFFG + XT1LFOFFG + DCOFFG);
                                                // Clear XT2,XT1,DCO fault flags
        SFRIFG1 &= ~OFIFG;                      // Clear fault flags
    }while (SFRIFG1&OFIFG);                   // Test oscillator fault flag

}


void main(void){
    WDTCTL = WDTPW|WDTHOLD; // Stop WDT

    // Set clock to 12 Mhz
    configureClock();

    //////////////////////
    // INTERUPT TIMER
    //////////////////////
    TA1CCR0 = SAMPLE_RATE;
        // run interupt every SAMPLE_RATE = 512 cycles
    TA1CTL = TASSEL_2 + MC_1 + ID_0 + TAIE;
        // use the SMCLK (Sub-Main Clock) as our clock source
        // use timer mode 1 (continually count up to the value in TA1CCR0)
        // use no input divider
    TA1CCTL0 = CCIE;
        // enable the interupt


    //////////////////////
    // INIT OUTPUT
    //////////////////////
    P6DIR |= BIT0 | BIT1 | BIT2 | BIT4 | BIT5 | BIT6;
    P4DIR |= BIT1 | BIT2;
    P3DIR |= BIT3 | BIT4;

    P3OUT = 0;
    P4OUT = BIT2;
    P6OUT = 0;

    //////////////////////
    // INIT INPUT
    //////////////////////
    // P1.1 and P2.1 Button Inputs
    P1DIR &= ~(BIT1);
    P1REN |= BIT1;
    P1OUT |= BIT1;

    P2DIR &= ~(BIT1);
    P2REN |= BIT1;
    P2OUT |= BIT1;


    // Harp Input Pins
    // High note 2.5, 2.4, 2.2, 7.4, 3.0, 1.2, 4.3, 4.0 Low note
    P2REN |= BIT2 | BIT4 | BIT5;
    P1REN |= BIT3;
    P4REN |= BIT0 | BIT3;
    P3REN |= BIT0;
    P7REN |= BIT4;
```



```
    P2DIR &= ~(BIT2 | BIT4 | BIT5);
    P1DIR &= ~(BIT3);
    P4DIR &= ~(BIT0 | BIT3);
    P3DIR &= ~(BIT0);
    P7DIR &= ~(BIT7);

    P2OUT &= ~(BIT2 | BIT4 | BIT5);
    P1OUT &= ~(BIT3);
    P4OUT &= ~(BIT0 | BIT3);
    P3OUT &= ~(BIT0);
    P7OUT &= ~(BIT7);

    // Enable Interupts
    _bis_SR_register(GIE);


    //////////////////////
    // MAIN LOOP
    //////////////////////
    while (1) {

        // Read Input
        unsigned int reading = read();

        // Inverse the reading
        reading = ~reading;

        // Update which notes are playing
        isPlaying[0] = (reading & (1 << 0)) ? NOTE_ON : NOTE_OFF;
        isPlaying[1] = (reading & (1 << 1)) ? NOTE_ON : NOTE_OFF;
        isPlaying[2] = (reading & (1 << 2)) ? NOTE_ON : NOTE_OFF;
        isPlaying[3] = (reading & (1 << 3)) ? NOTE_ON : NOTE_OFF;
        isPlaying[4] = (reading & (1 << 4)) ? NOTE_ON : NOTE_OFF;
        isPlaying[5] = (reading & (1 << 5)) ? NOTE_ON : NOTE_OFF;
        isPlaying[6] = (reading & (1 << 6)) ? NOTE_ON : NOTE_OFF;
        isPlaying[7] = (reading & (1 << 7)) ? NOTE_ON : NOTE_OFF;


        // Check for P1.1 input, and change the sound wave type (instrument) if pressed
        if (! (P1IN & BIT1)) {
            waveIndex++;
            if (waveIndex >= WAVE_COUNT) waveIndex = 0;
            while(!(P1IN & BIT1));
        }

        // Check for P2.1 input, and change the scale if pressed
        if (! (P2IN & BIT1)) {
            scaleIndex++;
            if (scaleIndex >= SCALE_COUNT) scaleIndex = 0;
            while(!(P2IN & BIT1));
        }
    }
}


// Sound mixing and sampling interupt routine. Runs once every SAMPLE_RATE cycles
void __attribute__ ((interrupt(TIMER1_A0_VECTOR))) PORT1_ISR(void) {

    // get current scale and waveform
```



```
    unsigned int* scale = SCALES + scaleIndex;
    int* wave = WAVES + waveIndex;

    // increment waveform positions for each note
    counts[0] += scale[0];
    counts[1] += scale[1];
    counts[2] += scale[2];
    counts[3] += scale[3];
    counts[4] += scale[4];
    counts[5] += scale[5];
    counts[6] += scale[6];
    counts[7] += scale[7];

    // calculate sample value
    int value =
        (wave[counts[0] >> 8] & isPlaying[0]) +
        (wave[counts[1] >> 8] & isPlaying[1]) +
        (wave[counts[2] >> 8] & isPlaying[2]) +
        (wave[counts[3] >> 8] & isPlaying[3]) +
        (wave[counts[4] >> 8] & isPlaying[4]) +
        (wave[counts[5] >> 8] & isPlaying[5]) +
        (wave[counts[6] >> 8] & isPlaying[6]) +
        (wave[counts[7] >> 8] & isPlaying[7]);


    // write sample value
    unsigned int writeValue = (value + 1024) >> 3;
    write(writeValue);

    // disable interupt flag
    TA1CTL &= ~TAIFG;
}
```