MP3 Compression: How it Works and Why it Matters

Morgan Zolob - V00823098

CSC 461

July 5, 2018

Table of Contents

Table of Figures

1.0 Introduction

This report will cover MP3 audio compression. MP3 is a lossy audio compression technology developed in the early 1990s by the Moving Picture Experts Group (MPEG). It was designed to provide a high compression ratio without creating noticeable quality loss. Due to it's great compression performance and lack of competitors, MP3 usage spread quickly in the early 90s, especially through the internet. This eventually led to it's near-universal support in audio players and devices today. The goal of this report is to discuss the history of MP3 compression, discuss how and why it works, and finally talk about my attempt to implement a software MP3 decoder on an Arduino Nano.

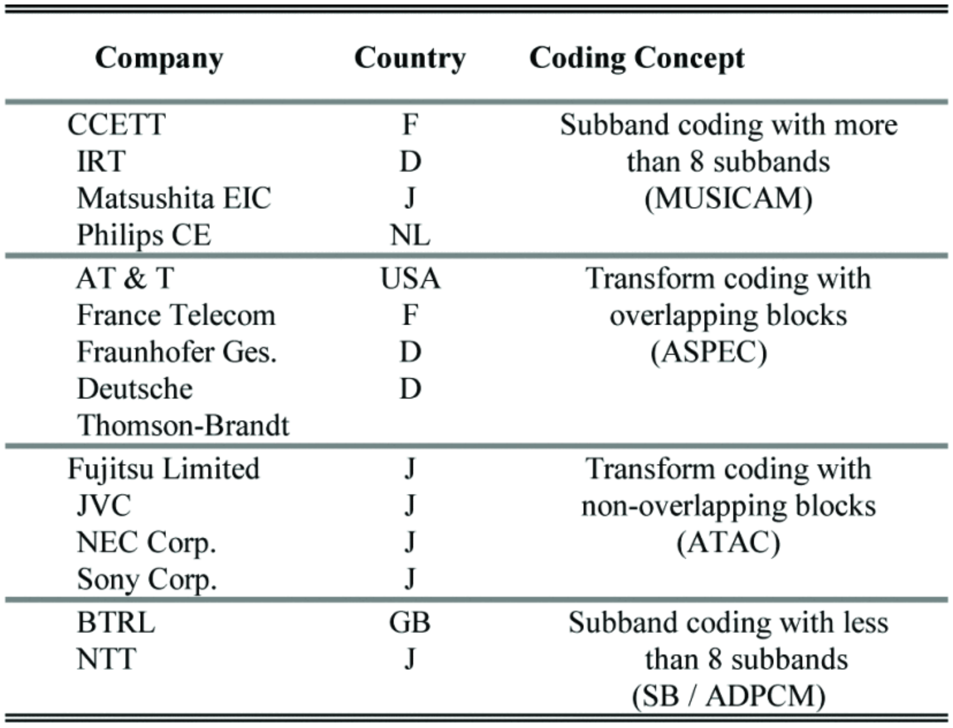
2.0 History of MP3 Compression

Before delving too deep into the details of MP3 compression, and why it is important, it is important to understand the history of MP3 compression. This section will cover how the MP3 format came to be, and discuss the impact it had on online music distribution.

2.1 The Creation of MP3

The idea of being able to remove information from audio without having any perceived change has been around far longer than computers or the need for compressible digital audio. In 1894, Alfred M. Mayer, an American physicist, made the discovery of auditory masking. He reported that a tone could be rendered inaudible by another tone of a similar, but lower, frequency [1]. This concept was further investigated by the likes of Richard Ehmer, Ernst Terhard, and others [2, 3].

By 1988, the year MPEG was established, extensive research had been done on the topics of psychoacoustic masking (psychoacoustics is the study of sound perception) and several audio compression algorithms using these ideas had been proposed or were in use. Despite this, there was no universal standard, which is one of the reasons MPEG was established. In December of 1988, the group called for proposals for a standard audio compression scheme [4]. In response, fourteen audio coding algorithms were submitted. Several of these algorithms were very similar, so they were placed into four groups, and the members in each group collaborated on merging their ideas into a single algorithm.



Company	Country	Coding Concept
CCETT	F	Subband coding with more than 8 subbands (MUSICAM)
IRT	D	
Matsushita EIC	J	
Philips CE	NL	
AT & T	USA	Transform coding with overlapping blocks (ASPEC)
France Telecom	F	
Fraunhofer Ges.	D	
Deutsche Thomson-Brandt	D	
Fujitsu Limited	J	Transform coding with non-overlapping blocks (ATAC)
JVC	J	
NEC Corp.	J	
Sony Corp.	J	
BTRL	GB	Subband coding with less than 8 subbands (SB / ADPCM)
NTT	J	

Figure 1 - The fourteen contributors, and the resulting algorithm groups [4]

The four grouped algorithm proposals were submitted back to MPEG in October of 1989. These proposals were then tested, but two failed due to hardware issues, so only two of the proposals, MUSICAM and ASPEC, could be fully tested. These both proved to have their own pros and cons, with MUSICAM offering less complexity and shorter decoding delay, and ASPEC offering better audio quality at the same bitrate. Because of this, a new group was formed with the goal of combining the two algorithms into a final standard, which was finally released in August of 1993 [5].

This ISO standard defined three different layers, with each layer building upon the previous one. The base layer, Layer I (commonly referred to as MP1) was the simplest and easiest to encode/decode, but offered the worst compression results. Layer II (MP2) was more complex but offered better compression, and Layer III (MP3) offered the best compression results, at the price of having the most complexity out of the three layers.

The different layers were intended to be used for different purposes. MP1 is considered mostly obsolete nowadays, however MP2 is still commonly used in the broadcast industry, and MP3 is one of the most well-known and widely used audio compression formats in the world today.

2.2 MP3 and The Internet

MP3 came about around the same time as the birth of the modern internet, and as such, it was influential in the distribution of audio over the internet. Early online music archives, such as the Internet Underground Music Archive, initially experimented with distributing uncompressed audio [6], but with the low internet speeds of the era, this would have been extremely slow and inefficient. The advent of MP3 allowed these sites to distribute near-lossless quality audio with much lower file sizes, which quickly grew the file format's popularity. By the late 90's, software players such as Winamp, and portable hardware MP3 players like the Rio player rapidly increased the popularity of online MP3 distribution, despite legal fightbacks from the RIAA due to the rampant copyright infringement arising from the ease of distribution and creation of MP3 files [6].

Although MP3 distribution on the internet had a rough start legally, and still continues to be used in unauthorized music sharing today through mediums such as peer-to-peer filesharing, it's help in increasing the popularity of online music distribution has led to the rise of the popular legal music streaming services in the world today, such as Spotify and Google Play Music. In fact, in 2016, online music streaming became the most popular way people consume music, outranking both physical and digital sales [7].

3.0 Why MP3 Compression is Important

One may initially wonder why there is a need to compress audio at all. And indeed, storing uncompressed audio does have it's merits. For archival purposes, uncompressed, or at least lossless audio formats are definitely the best choice, as any other storage option, such as MP3, will result in information loss. But many people don't need archival quality music. For the general consumer, they just need a way to listen to music on their smartphone, or car stereo, and don't want to consume large amounts of their valuable storage space. This is where the MP3 compression format shines.

3.1 File Size Savings

The main goal of any compression scheme is to reduce the amount of information one must store. Lossless compression schemes do this while retaining the ability to perfectly reproduce the original information, while lossy compression schemes such as MP3 give up perfect reproducibility in order to achieve much higher compression results. This can be seen by looking at the compression ratios of MP3 compared to a lossless compression scheme such as FLAC. At the common bitrate of 128 kbit/s, MP3 offers a compression ratio of 11:1 over uncompressed CD quality audio [8]. Compare this to FLAC, which at best achieves a compression ratio of around 2:1 [9], and the benefits of MP3 can clearly be seen.

In order to see these compression ratios for myself, I decided to devise my own test. I took an album of uncompressed music, specifically Fall Out Boy's "Save Rock and Roll", and compared the file sizes using various forms of compression. The results can be seen in the table below.

"Save Rock and Roll" by Fall Out Boy
(41 minutes, 37 seconds long)

File Format	**File Size**
Uncompressed WAV	420.8 MB
Lossless Compression with FLAC	295.6 MB
MP3 at 320 kbit/s	97.4 MB
MP3 at 128 kbit/s	40.2 MB

Figure 2 - Album file size with various audio coding schemes

As you can see, these results line up well with the compression ratios described above. My 128 kbit/s MP3 achieved a compression ratio of about 10.5:1, and my FLAC results achieved a compression ratio of 1.4:1 (Note that this is fairly far from the 2:1 above, but that was a best-case result. Results in the range of 1.5:1 or worse are common for FLAC). My higher quality 320 kbit/s MP3 also achieved quite good results, with a 4.3:1 compression ratio. Of course, such high compression ratios beg the question, how noticeable is the loss in information?

3.2 Little Noticeable Quality Loss

When you are storing just one bit for every ten in the original audio file, one would assume that there must be a fairly significant loss in the quality of the audio. This is not necessarily true, however. In an online test conducted in 2012 which surveyed over 3500 people, it was concluded that "people can't hear a difference at bitrates above 128kbps" [10]. The same experiment also mentions that the LAME MP3 encoder (one of the more popular free MP3 encoders) defaults to a 192 kbit/s variable bitrate (variable bitrates allow the encoder to use more/less bits where necessary, averaging to the stated bitrate). This default seems to be well-chosen, as it provides a bit of a safety above the 128 kbit/s threshold, and should, according to the results of the experiment, result in no discernable quality loss over the original, CD-quality audio.

Again, I wanted to test these findings for myself, so I set up a comparison test between the original, uncompressed CD audio, a 320 kbit/s MP3, and a 128 kbit/s MP3 for the song “Miss Missing You” from the previously mentioned Fall Out Boy album. I did this using the free Audio editor “Audacity” and lining up the three different audio tracks, then switching between them by pressing the “Solo” button on the track I wanted to hear.

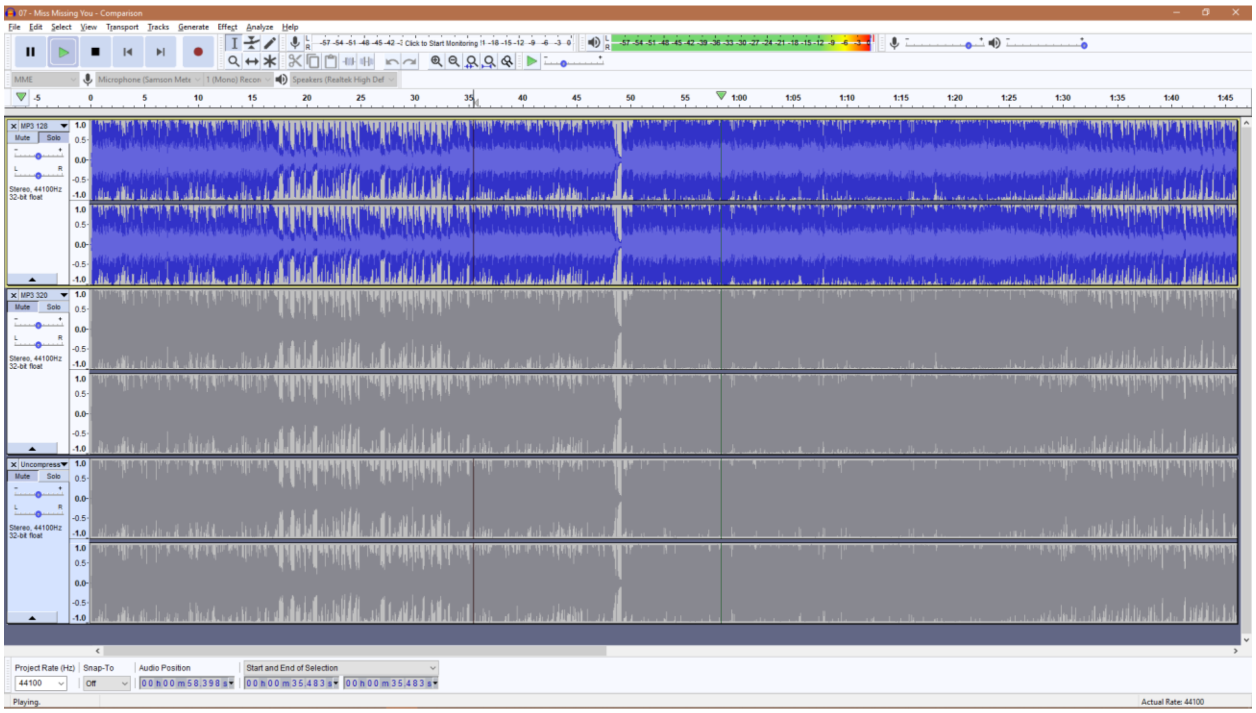

07 - Miss Missing You - Comparison
File Edit Select View Transport Tracks Generate Effect Analyze Help
MP3 128
MP3 320
Uncompress
Mute Solo
Stereo, 44100Hz 32-bit float
Project Rate (Hz) 44100
Snap-To Off
Audio Position 00 h 00 m 58.398 s
Start and End of Selection 00 h 00 m 35.483 s 00 h 00 m 35.483 s
Playing.
Actual Rate: 44100

Figure 3 - Audio quality comparison using Audacity

I found that I couldn’t hear any difference between any of the files. I did these tests using a decent pair of headphones (AKG K182’s) and the built in sound card in my computer. Perhaps with better audio equipment or a different song I might be able to tell the difference, but from what I can tell, 128 kbit/s is definitely approaching the limits of having no discernable audio differences. Testing with lower bitrate audio, however, such as 64 kbit/s, or even 94 kbit/s, did produce very noticeable quality degradation, so 128 kbit/s does indeed seem to be around the lower limit, as the experiment discussed earlier indicated.

4.0 How MP3 Compression Works

This report has discussed the history of MP3 and why it is important, as well as some examples of the compression results it can achieve. But how exactly is MP3 compression able to achieve such high compressibility without audible quality loss? It does so through several tricks, mostly revolving around how people hear and process sounds. There are many scenarios where an audio file will contain sounds that a human will normally never be able to hear, due to either the sound's frequency, or auditory effects such as masking. By removing these sounds, MP3 is able to reduce file size significantly without noticeable loss in audio quality. This section will cover how this works in detail.

4.1 The Human Frequency Range

Humans do not hear all frequencies the same. Our ears are most sensitive between frequencies of 2 kHz and 5 kHz, and the absolute limits of human hearing are around 20 Hz to 20 kHz. At higher frequencies, however, our ears are much less sensitive, and in fact most adults will be unable to hear any sounds above around 16 kHz. You can see this demonstrated in the graph below, which shows the equal loudness curves defined in the ISO 226-2003 standard [11]. A sound at any frequency along one of these curves will be perceived by a human as having the same loudness. From this graph, it is clear that humans are much less sensitive to very low and very high sounds, as those sounds must be much louder in reality for humans to hear them as equal to sounds where we are most sensitive (around 2-5 kHz).

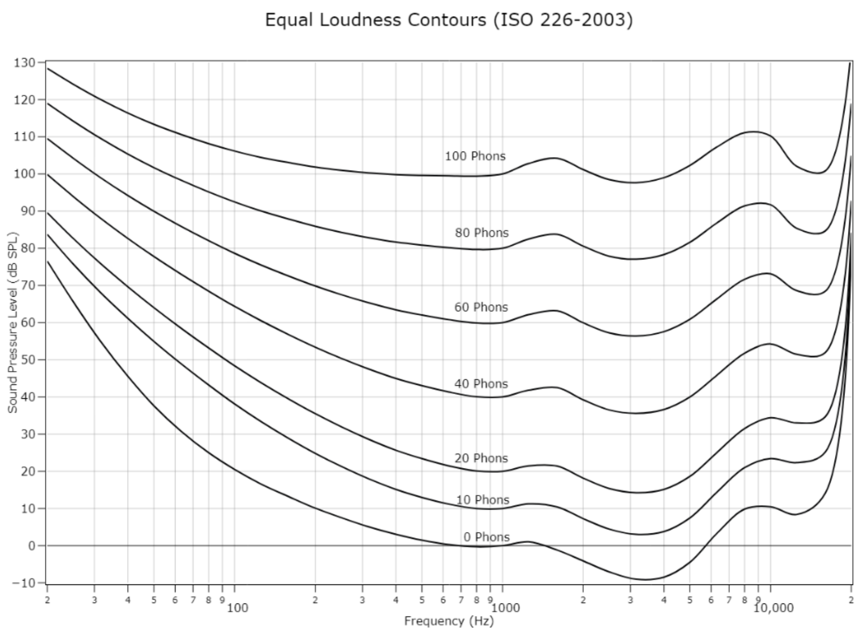


Figure 4 - ISO 226-2003 Equal Loudness Contours

MP3 compression takes advantage of this. Sounds that are outside of the human hearing range can be completely discarded, and quiet sounds in the frequency ranges that we are less sensitive too can be discarded as well, as we wouldn't be able to hear them. For example, a 20 Hz tone can only be heard if its louder than about 60 dB, however sounds in the 2-5 kHz range can easily be heard at much lower levels than this [12]. By removing, or using less data to store these inaudible sounds, MP3 can save a significant amount of data.

4.2 Auditory Masking Effects

Another quirk of human hearing that MP3 can take advantage of is audio masking. There are different types of masking effects, but they all follow the same concept; some sounds will "mask" other sounds, that is, make them inaudible to humans.

One example of this is frequency masking. If two sounds are close to each other in frequency (for example, 100 Hz and 110 Hz), but differ in loudness, then the louder sound will mask the quieter one if they are both played simultaneously, even though if played separately both sounds would be perfectly audible [13]. This means that MP3 compression can discard the quieter sound entirely, saving space. Furthermore, due to the masking effect of the loud sound, MP3 can heavily compress that section of audio, as the compression "noise" will also be masked [12].

Another example of masking is temporal masking. As the name implies, this effect applies to sounds nearby in time, rather than in frequency. A loud, short sound (known as a transient sound), will mask weaker sounds for a very short period of time around it (generally tens of milliseconds) [13, 14]. This applies to both preceding and succeeding sounds, that is, it applies to sounds both before and after the masking sound. MP3 compression can thus remove the quieter sounds around a loud transient sound, which will again save more space.

4.3 Traditional Compression – Huffman Coding

The final way MP3 compression can save space is by utilizing a traditional, lossless compression scheme. MP3 does this using Huffman coding, which is the same compression scheme used by archive formats such as GZIP. This report won't go in depth into how Huffman coding works – but a simple example would be encoding the text string "HELLO". Without utilizing compression, storing this string using UTF-8 would take up 40 bits (one byte per character). Using Huffman coding, a tree can be constructed (see the figure below) and then the string can be encoded using far fewer bits (for this example, "HELLO" would become 1011100110, which uses only 10 bits). The string can then be fully decoded back to the original string by using that same tree. Note that internally this tree will be stored as a table, it is just displayed as a tree here to aid in understanding.

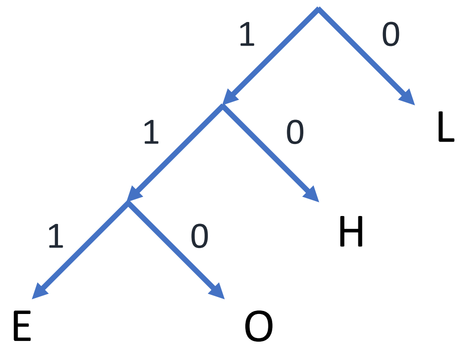


Figure 5 - Huffman tree for encoding/decoding the string "HELLO"

Using Huffman coding, MP3 can maximize its compression ratio without any additional data loss. Storing the Huffman tables used in this step, however, would take a significant amount of space. To combat this, the MP3 standard defines several pre-made Huffman tables which have been designed to provide optimal compression for "typical" audio samples [12]. The MP3 encoder uses these tables to apply the Huffman coding, so the tables themselves never have to actually be stored with the MP3 file. Instead, only a reference to which table was used needs to be stored. The MP3 decoder will know all the tables internally, and will just use those references to figure out which tables to use.

4.4 Not All MP3's Are Created Equal

It is important to note that while the MP3 standard acknowledges these compression methods, it does not define any specific parameters for how they should be used. This decision is left up to the encoder [12]. This means that while one MP3 encoder may decide that a certain sound will be inaudible and remove it, a different encoder may decide that the sound is important and keep it. For this reason, two MP3 files, created at the same bitrate from the same source material, may sound quite different depending on what encoder was used to create each file.

5.0 Developing an Arduino MP3 Decoder

As part of this project, I wanted to investigate using an Arduino to decode MP3 files. I hoped that this would help me in further understanding the technical details of MP3 compression, and imagined that it would be a fun project to work on. This section will detail the steps I took in creating the decoder, and my findings along the way.

5.1 Research

To begin, I started with some research into Arduinos and MP3 decoders. The Arduino I chose to work with was an Arduino Nano, which uses an ATmega328 microcontroller clocked at 16 MHz. It has 2 KB of ram and 32 KB of program storage.

My initial research indicated that real-time MP3 decoding could be implemented using as low as 24 MIPS (millions of instructions per second) [15]. While this is a little outside the capabilities of the Arduino I had chosen, I hoped that by using only mono, low-bitrate, low-sample-rate MP3 files as input, I could get the decoding requirements low enough to achieve real-time decoding on the Arduino.

My next step was reading the fantastic article “Let’s build an MP3-decoder!” by Björn Edström [12]. This article discussed the technical details of the MP3 file format, and went over the construction of an MP3 decoder. After reading this article, I began to realize that MP3 decoding is not a simple task, so while I had originally planned to write my decoder from scratch, I instead decided to attempt to port an existing MP3 decoder.

From there, I began researching different open-source MP3 decoder implementations. There was one included in Edström’s article, but it was written in Haskell, which I have never worked with before, so porting it would be difficult for me. The next decoder I looked at was minimp3 by KeyJ [16]. This decoder looked promising, as it was written in C and used fixed-point math, which would be good for the Arduino, which lacks a floating-point unit. However, the code turned out to be very difficult to read and understand, so I decided to move on. Next, I looked at minimp3 by lieff [17]. This decoder was a sort of spiritual successor to KeyJ’s, but I found that it was much easier to read and modify. It did, however, use floating-point math, which would have to be emulated on the Arduino. I decided that for me at least, this performance hit was worth the trade-off of being much easier to modify.

5.2 Audio Playback on Arduino

Another goal I had for this project was to be able to play back the decoded MP3 using the Arduino. This meant that the Arduino had to be able to play uncompressed, PCM audio (which the MP3 would decode into). Initially this seemed like it might be problematic, as the official Audio library for the Arduino, which allows for WAV file playback, only works on the Arduino Due, because it contains an onboard DAC (digital to analog converter) [18]. However, after finding a different audio library, TMRpcm [19], which did support my Arduino, I was able to get WAV files to playback from an SD card to a standard speaker through an AUX cable. These files had to have a fairly low sample-rate (22050 Hz), but this was fine, as I was planning to use low sample-rate MP3s anyways.

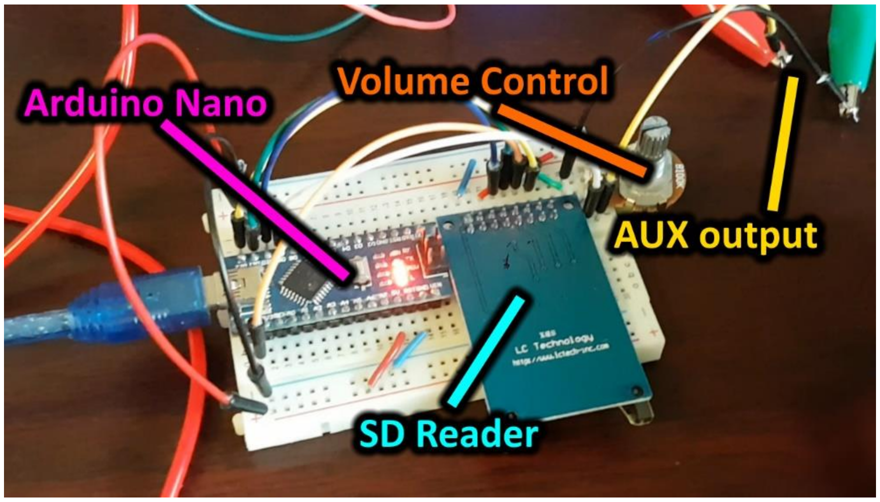
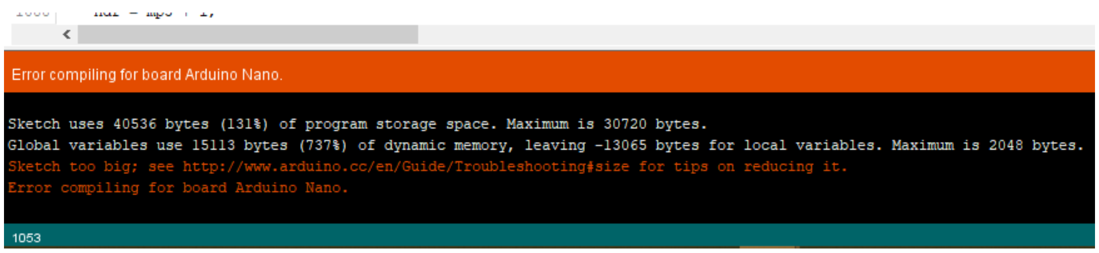


Figure 6 – Picture of my setup for audio playback from Arduino

5.3 Porting an MP3 Decoder to Arduino

Once I confirmed that the Arduino would be able to playback audio. I began working on porting the MP3 decoder mentioned above (minimp3) to the Arduino. This is where I began to run into issues. MP3 decoding requires a significant amount of memory. MP3 files are split into “frames”, which can be a few hundred bytes long before being decompressed. In order to efficiently decode MP3’s, you need to be able to store the whole uncompressed frame in memory, along with several other intermediate buffers. Once decompressed, a frame will have 1152 audio samples, which will take up around 2.3 KB of memory. The Arduino Nano only has 2 KB of memory to work with, and the SD card library already uses up around 500 bytes of that. Clearly much more memory was needed. In my initial attempt to port the minimp3 library to the Arduino, I found that I was using over 700% of the available memory with global variables alone.



Error compiling for board Arduino Nano.

```
Sketch uses 40536 bytes (131%) of program storage space. Maximum is 30720 bytes.
Global variables use 15113 bytes (737%) of dynamic memory, leaving -13065 bytes for local variables. Maximum is 2048 bytes.
Sketch too big; see http://www.arduino.cc/en/Guide/Troubleshooting#size for tips on reducing it.
Error compiling for board Arduino Nano.
```

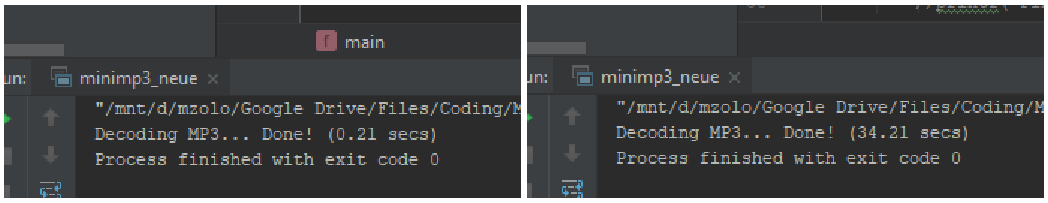
Figure 7 – Screenshot of the Arduino IDE error showing too much memory was being used

The first thing I looked into in order to optimize this was moving the Huffman tables into program memory, rather than storing them in RAM. I quickly realized that this wouldn't solve anything, though, as the sketch was also already using more than the available program memory (at around 131%).

It was at this point that I started to realize this might be a harder task then I initially thought. Still, I didn't want to give up, so I started looking at how I could save memory. To start with, I removed all the code relating to MP3 features I wasn't planning on using, such as stereo sound and high bitrates. This allowed me to reduce the size of several buffers by a significant amount, though still nowhere near the amount that I needed.

The next thing I attempted was using the SD card as a form of additional memory. By refactoring the code to read/write to a temporary file on the SD card, rather than using the limited RAM, I thought I could probably decrease the RAM requirements enough to work on the Arduino.

While this strategy may work, it proved to be very tedious. It took me several hours just to refactor one variable (a buffer for MP3 granules) out of memory and on to file storage. Additionally, the performance of the decoder was reduced drastically. On my desktop, it went from being able to decode a three-minute song in under one second when only using RAM, to taking over 34 seconds with that one variable being moved to file storage.



```
"/mnt/d/mzolo/Google Drive/Files/Coding/M
Decoding MP3... Done! (0.21 secs)
Process finished with exit code 0
```

```
"/mnt/d/mzolo/Google Drive/Files/Coding/M
Decoding MP3... Done! (34.21 secs)
Process finished with exit code 0
```

Figure 8 – Decoder output before (left) and after (right) moving a variable to file storage

It was at this point that I decided to call this experiment a failure. Even if I could move enough variables out of RAM and into file storage, the amount of time it would take for me to refactor that much code would be tremendous, and the performance hit on the decoder would be astronomical. Furthermore, I still hadn't solved the problem of the decoder's code using more than the Arduino's available program storage. To solve that I would have to remove some of the code, but I had already stripped out everything that I could whilst still having the decoder work. Ultimately, MP3 decoding simply requires more resources than the Arduino has available, so the goal I set out to achieve simply wasn't possible with this hardware.

5.4 A Success, Even in Failure

Although my goal of building an Arduino MP3 decoder was a failure, I still got to work and become very familiar with the source code of the minimp3 MP3 decoder. Since the whole point of building an MP3 decoder for the Arduino was to learn more about the technical details of MP3 decoding, I believe that this project was still a success. I now know and understand much more of how MP3 files work and how they are decoded, and believe that this was a worthwhile learning experience despite being unable to produce a working final product.

6.0 Conclusions

MP3 is an extremely useful audio compression technology that allows for compression ratios of up to 11:1 without essentially no quality loss. It helped greatly in popularizing audio distribution over the internet, paving the way for popular streaming services today such as Spotify and Google Play Music. While using lossy audio compression such as MP3 does have drawbacks, it is good enough for day to day use, with the majority of people being unable to distinguish a 128 kbit/s MP3 from an original, uncompressed CD audio file. Using MP3 allows people to store tens of thousands of songs on devices such as smartphones and MP3 players, rather than just a few hundred if they were to use a lossless format.

The MP3 format is also surprisingly complex for an audio format defined nearly 30 years ago. Even modern low-power, general purpose processors, such as those found in the Arduino, will struggle to decode MP3 files due to its relatively high memory requirements.

All in all, MP3 compression is a fascinating and complex technology that has helped shape the way we listen to and distribute audio in the world today.

7.0 References

[1] A. M. Mayer, "XXIII. Researches in Acoustics.—No. IX," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science,* vol. 37, pp. 259-288, 3 1894.

[2] E. Terhardt, G. Stoll and M. Seewann, "Algorithm for extraction of pitch and pitch salience from complex tonal signals," *The Journal of the Acoustical Society of America,* vol. 71, pp. 679-688, 3 1982.

[3] R. H. Ehmer, "Masking by Tones vs Noise Bands," *The Journal of the Acoustical Society of America,* vol. 31, pp. 1253-1256, 9 1959.

[4] H. G. Musmann, "Genesis of the MP3 audio coding standard," *IEEE Transactions on Consumer Electronics,* vol. 52, pp. 1043-1049, 8 2006.

[5] ISO, *ISO/IEC 11172-3:1993 — Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s — Part 3: Audio,* 1993.

[6] D. Morton, Sound Recording: The Life Story of a Technology, Greenwood Press, 2004.

[7] E. Crawford, "20I6 Year-end Music Report," 2017.

[8] W. S. Gan and S. M. Kuo, Embedded Signal Processing with the Micro Signal Architecture, Wiley, 2007.

[9] N. Zachary, "FLAC compression level comparison," June 2014. [Online]. Available: http://z-issue.com/wp/flac-compression-level-comparison/. [Accessed 6 July 2018].

[10] J. Atwood, "Concluding the Great MP3 Bitrate Experiment," June 2012. [Online]. Available: https://blog.codinghorror.com/concluding-the-great-mp3-bitrate-experiment/. [Accessed 6 July 2018].

[11] ISO, *ISO 226:2003 — Normal equal-loudness-level contours,* 2003.

[12] B. Edström, "Let's build an MP3-decoder!," October 2008. [Online]. Available: http://blog.bjrn.se/2008/10/lets-build-mp3-decoder.html. [Accessed 6 July 2018].

[13] T. Wilburn, "The AudioFile: Understanding MP3 compression," October 2007. [Online]. Available: https://arstechnica.com/features/2007/10/the-audiofile-understanding-mp3-compression/. [Accessed 6 July 2018].

[14] B. Lincoln, "Temporal Masking," March 1998. [Online]. Available: https://ccrma.stanford.edu/bosse/proj/node21.html. [Accessed 6 July 2018].

[15] B. Samuel and A. Jhunjhunwala, "Real time implementation and optmization of MP3 decoder on DSP," in *2008 Canadian Conference on Electrical and Computer Engineering*, 2008.

[16] M. Fiedler, "Honey, I shrunk the MP3 decoder," January 2007. [Online]. Available: https://keyj.emphy.de/minimp3/. [Accessed 6 July 2018].

[17] lieff, "minimp3," January 2018. [Online]. Available: https://github.com/lieff/minimp3. [Accessed 6 July 2018].

[18] Arduino, "Simple Audio Player," [Online]. Available: https://www.arduino.cc/en/Tutorial/SimpleAudioPlayer. [Accessed 8 July 2018].

[19] TMRh20s, "Arduino WAV Playback Direct from SD Card," June 2012. [Online]. Available: http://tmrh20.blogspot.com/2012/06/arduino-wav-playback-from-sd-card-new.html. [Accessed 8 July 2018].