Project 3.3 Long ISP: 8-Bit R/2R Audio Player

Purpose

The purpose of this ISP, the 8-Bit Audio Player is to play rotary-encoder-selectable-by-name 8-bit, 16 KHz, mono wav files using an R/2R digital to analog converter (see Project 2.1) in place of a traditional I2C or SPI DAC.

Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI4M/2425/ISPs.html#logs

Project proposal: http://darcy.rsgc.on.ca/ACES/TEI4M/2425/images/RJAudio.png

Project GitHub: https://github.com/rohan-development/3.3-Long-ISP-R2R-Audio-Player

PCB schematic: https://oshwlab.com/rjamal/ipod-thing

Theory

There are 2 main classes of audio files: compressed audio, and uncompressed. The dominant audio file format in modern devices is MP3, which is a type of compressed audio. In order to play most standard audio files (regardless of compression), samples of the amplitude at a known sample rate must be obtained. In an uncompressed audio file, the raw sample values are already available in the file; in a compressed audio file, the samples must be obtained by applying an algorithm to the compressed data. These sample rates are then converted to an analog voltage, using a DAC, and amplified to be perceived as sound (see Figure 1).

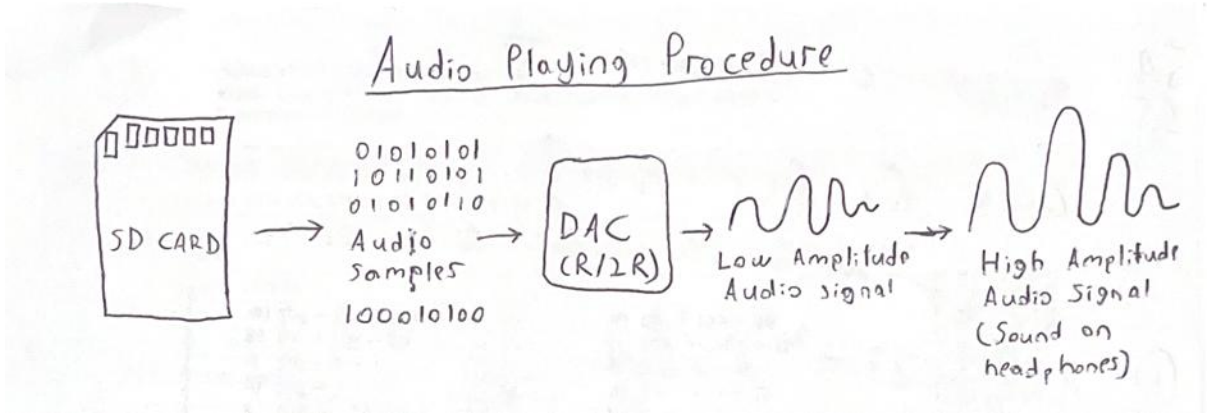


Figure 1. Audio Playing Sequence Using a DAC

When referencing the Atmega328P, most compression algorithms require significantly more computing power than available to playback in real time. For this reason, the R/2R audio player utilizes Waveform Audio File Format (WAV) files to store audio data, which simply store the raw discrete samples.

The basic concept of 8-Bit R/2R Audio Player comes from Project 2.1, the R/2R DAC. An R/2R ladder, or DAC, is a simple DAC that converts an 8-bit, discrete value, to an analog voltage between 0 and the supply voltage using the principle of voltage division. It included 8 switches that could be turned on or off, for a total of 256 steps between 0 and 5V. By connecting the 8 switch inputs to a microcontroller, a stored 8-bit digital value can be converted to an analog voltage. In this configuration, the R/2R DAC enables the microcontroller to emulate PWM with true analog voltage, instead of a duty cycle-based approximation.

Procedure

On a very basic level, to play audio, the 8-Bit Audio Player makes use of an R/2R ladder and 8-bit, 16 KHz WAV files (see Theory section). In this configuration, an ATmega328P reads audio samples from an SD card, and sequentially outputs the samples onto the R/2R ladder at a rate of 16 KHz. To advance through the samples reliably at 16 KHz, a timer 1 interrupt is used. The raw signal from the output of the R/2R DAC is filtered through a capacitor, and then amplified through an LM386 audio amplifier, before finally being sent to headphones through a 3.5 mm audio jack.

Parts Table	
Quantity	**Description**
13	Assorted Capacitors
1	3.5 mm Audio Jack
25	±0.1% 10 KΩ Resistor
38	Assorted Standard Resistor
2	Atmega328P SMD MCU
1	USB-C Connector
1	LM386 Audio Amplifier
1	MCP4551T Digipot
2	16 MHz Crystal Oscillator

The ATmega328P has only 2 KB of memory, while an average WAV file can be in excess of 5000 KB. To get around this limitation, a buffer is used. An 8-bit unsigned array with 256 locations (256 bytes) is created. In this configuration, the first 256 bytes are read from the SD card and placed in the buffer array. Then, a timer1-driven interrupt with a frequency of 16 KHz is initialized Each time the interrupt is triggered, an index is incremented. In the `loop()` function, the R/2R DAC is constantly updated with the value of the current index of the buffer array. To do this, the `PORTD` register is directly manipulated. By wiring the LSB of the DAC to pin 0, the second LSB to pin 1, and so on up to the MSB wired to pin 7, a WAV sample value can be directly put on PORTD, which automatically generates the correct analog voltage. When the index reaches the end of the array, it is reloaded. As a result, the buffer needs to be large enough that the MCU does not access the SD card too often to keep up, but also small enough that it can reload the entire buffer reliably between interrupts. Using this process, the original audio signal can be approximated (see Figure 1).

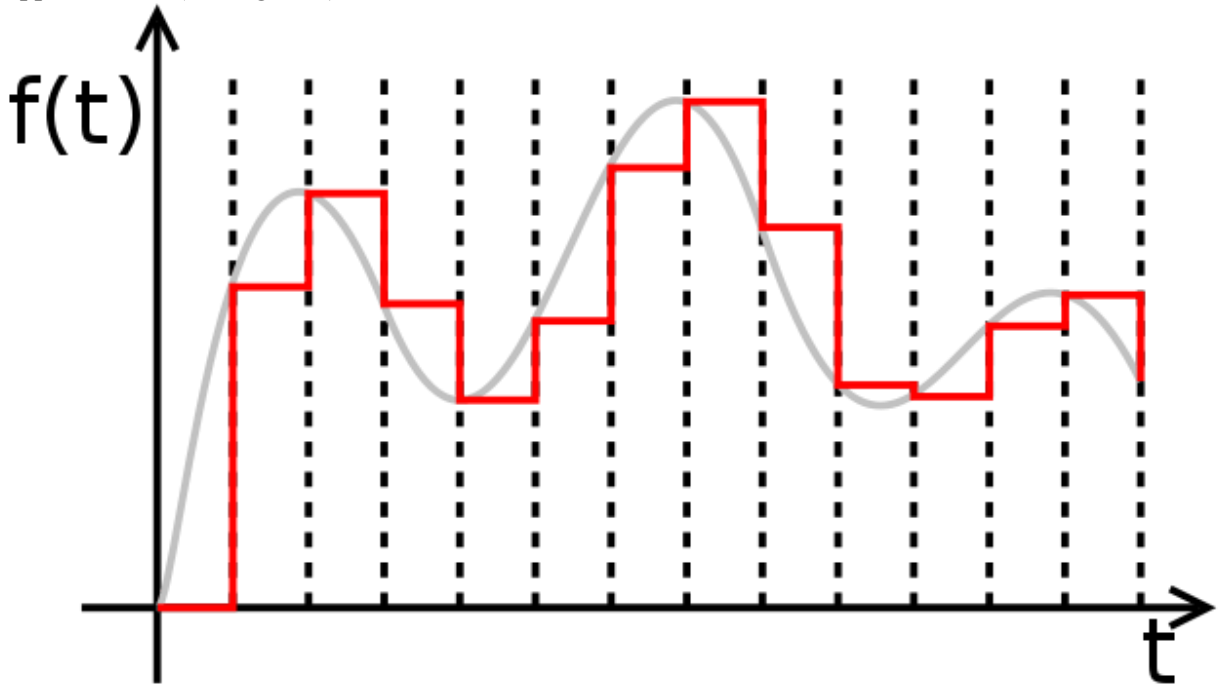


Figure 1. Audio Signal Approximation using Discrete Signaling

The 8-Bit Audio Player has 2 MCUs used: one "slave" to read from the SD card and update the DAC, and a master to coordinate which song to play, and by responding to user input through a rotary encoder and to control a TFT screen with the song progress and a list of songs. To communicate between MCUs, I2C is utilized. When the master wants to receive any information from the slave, it must send a request. Due to the way that I2C works (see Project 2.6.1), when a request is sent, the same *request routine* is run. A request routine is a function run when an I2C request is received. This means that the same data point is sent for every request, in a standard configuration. The request routine is simply a function. It is not, strictly speaking, an ISR. This means that many blocks of code can be efficiently run within the request routine, and blocking calls are considered acceptable.

To request multiple data points, a command code system is implemented. In this configuration, the master sends a code corresponding to a data point, which is placed into a buffer. Then, the master sends a request. Within the request routine, a subroutine is included for each point (see Figure 2).

Figure 2. Code Data Correspondences

Code	Data Point
0	Song length
1	N/A
2	Song names
3	Number of songs

The master MCU only has direct access to the display, the rotary encoder, and the I2C bus. Since it does not have access to the SD card, it cannot directly read the song names to put on the display. To read the song names, it requests the data from the player MCU. To do this, it first sends a value of 2, to place in the code buffer (see Figure 2). Then, it requests 32 bytes from the player, which reads the SD card, parses it into a character array buffer, and sends 1 character (byte) at a time to the master which reconstructs the data and parses it into a 2D array of characters. Since the master can only hold a finite number of song names in its 2 KB of memory, the player maintains a global index based on the last song sent. This means that on the next song name request, the next song is automatically sent.

In addition to a request routine, the player MCU has a *receive routine*. Unlike the request routine, the receive routine is an ISR, and it is called when data is written to the player. To aid the command code system, the first byte received is always the code. Some codes warrant a second byte (see Figure 3).

Figure 3. Code Action Correspondences

Code	Action	Extra bytes
0	Changes song	1
1	Toggles Playback	0
2	N/A	0
3	N/A	0

The code system is global, and not specific to the data direction. As a result, most codes are only valid in either Figure 2 (MISO), or Figure 3 (MOSI). The exception to this is the code 0, which, strictly speaking, corresponds to a song change. When a 0 is received, the `songChanged` flag is set, and a second byte is read, which is stored into the song number buffer. On the next iteration of the loop, the `songChanged` flag is picked up, and the song is set to the number in the song number buffer. When the song is changed, the master requires the length of the song each time, which is stored on the slave. As a result, the master sends a data request. Since the code 0 corresponds to a song change, an additional request code is not required; the data point requested immediately after a song change is always song length.

To select the desired song, a simple for loop is utilized. In the loop, an index is set such that the loop runs *song* times. In the loop are two important parts: a next file line, and a hidden file skipping line. In this configuration, since the loop is run a certain number of times, the next file is chosen the same number of times, skipping to the desired file. The index number is built in the same fashion which maps an index value to the filename, so a number can be easily linked with a filename from the display. The hidden file skipping line simply decrements the loop index if the file starts with "`._`".

When a song is selected, after the master updates the slave, it goes to the "now playing" screen (see Figure 4). This screen shows a music icon, which is stored as an array in program memory, to save RAM. It also shows the current song progress in seconds, the total song length, a progress bar, and the name of the track (see Figure 4).

To display this screen, when the song is selected, the music icon is drawn from program memory (only once, which saves resources). The song track name is also drawn once. Then, a timer1-driven interrupt which triggers once per second is started, updating the song progress in real time. To display the progress bar, the map function is used. It simply maps the song progress from a value between 0 and `songLength`, to a value between 0 and 100. The progress bar is 100 pixels wide, so a green rectangle with the result of the map is drawn overtop of the white background bar.

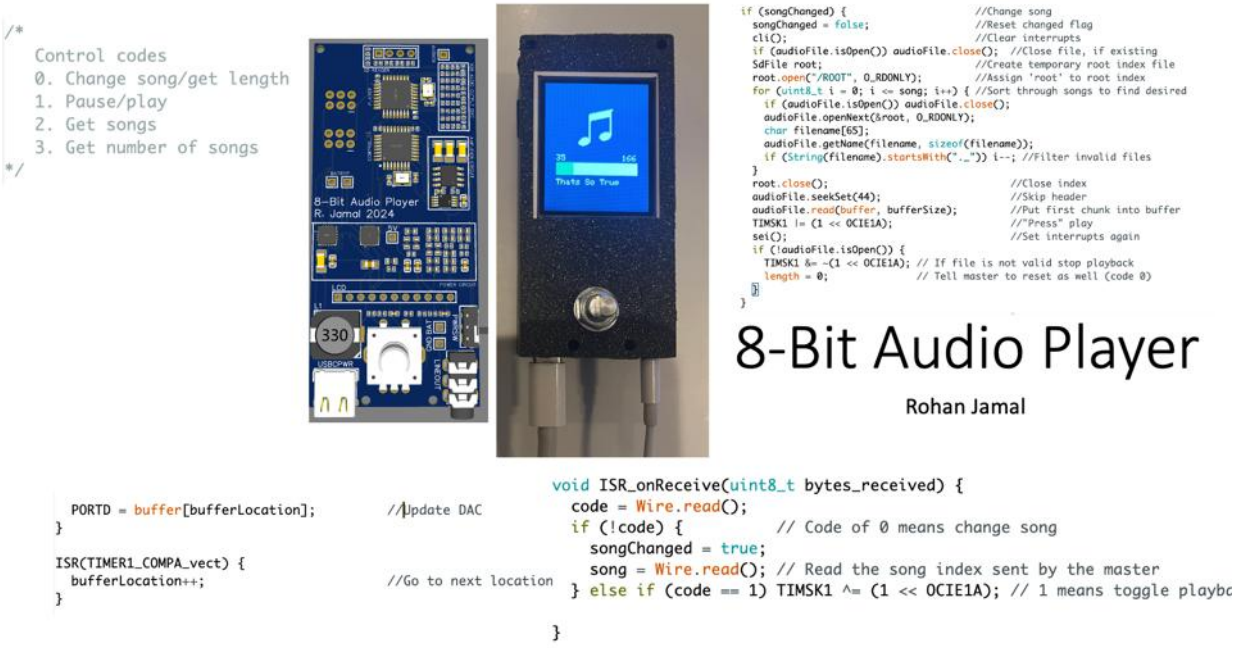
Figure 4. Now Playing Screen

On the now playing screen, when the rotary encoder is pressed, the master sends a code of 1 to the player, which toggles the OCIE1A bit of the TIMSK register. This bit enables or disables the execution of the ISR on the overflow of timer1. This effectively toggles playback. Since the MCU playing the music is independent from the one connected to the screen, the progress bar is updated independently as well; when playback is started, the master downloads the song length and starts a timer. It does not send a request with the song progress each second.

The PCB for the 8-Bit Audio Player includes a boost converter, as well as a battery charging IC. This allows the 5 V system to be used effectively with a 3.7 V lithium-ion battery, with charging over USB-C. The boost converter steps the voltage up from 3.7 V to 5 V. The battery charging IC regulates the 5 V from the USB-C connector to effectively charge the lithium battery.

Media

Project video: https://youtu.be/TbR3IQ8DSpg



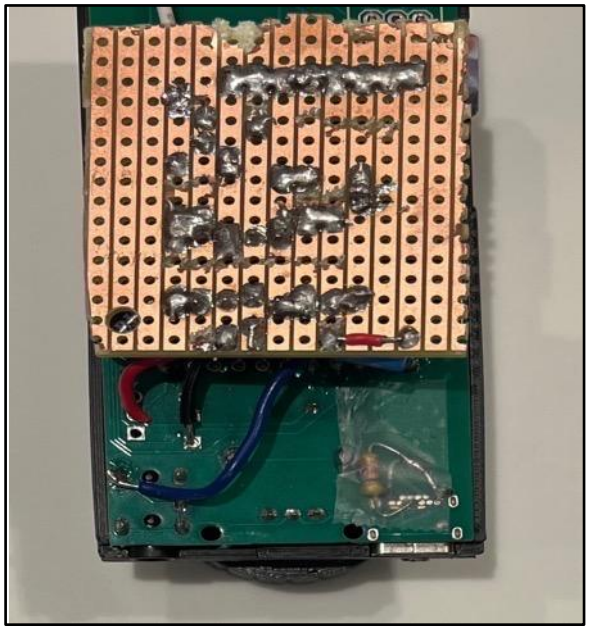
ISP Presentation Background

Left to Right: PCB, Assembled Device without Cover, Assembled PCB

New Year's Day Selected

New Years Day Playing (32/257 Seconds)

Device Side Profile (SD Slot)



PCB Rear (External Amplifier)

Rotary Encoder Handle

Device Ports (Bottom)

Reflection

This has definitely been my favourite ISP thus far. Out of all my ISPs, I think this is my most functional one. My bike speedometer worked in the end, but it was never practical. This audio player is completely practical for sitting in one place whilst doing homework and listening to music. I learned a ton about how music is played in general, I learned a ton of strategies in code (it's just practice, I suppose), and I really feel that I pushed my PCB design during this project. Overall, I am super happy with the result as I feel that I accomplished just about everything that I said I would (more in some areas). The only "copout" that I ended up going with was 8 bits instead of 12, as all information I found pointed towards inaccuracies of the R/2R ladder beyond 8-10 bits. It also would not have made much sense from a port perspective.

I think that this is really not a testament to any skill but time management. This is something I normally struggle with, but I had my PCB done early enough that I was not scrambling at the last minute, and I had much time to debug and perfect my code and 3D design.

I know that I could have used an ESP32, or some other microcontroller to easily play the audio without an external DAC, but for me, the real satisfaction was building my own DAC from scratch, and working with the limited resources of the 328P

That being said, there are some things that I hope to fix: I think that if I get a chance, I will reorder the PCB, without the power circuit and use larger resistors and capacitors so I can surface mount them myself in order to address a few issues (such as lack of volume control and the absence of a functional onboard amplifier). Overall, this was a great project, and while thinking about my next ISP, I definitely have audio in mind. (My code and EasyEDA files are included in the GitHub and Oshwlab links in Reference).