STM32F4 MP3 Player

Project Overview

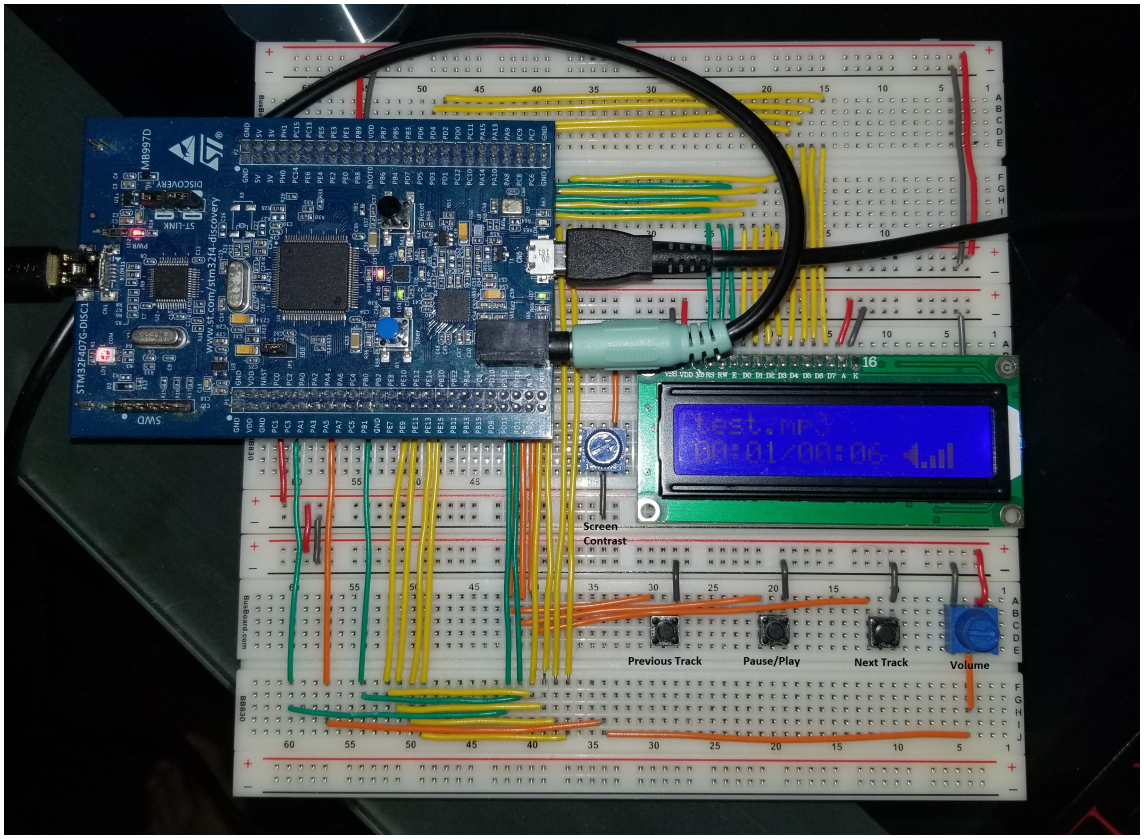

Figure 1: MP3 Player Prototype

The main goals of this project was to expand my knowledge about the STM32 toolchain, programming with arm cortex, and familiarizing myself with FreeRTOS. The MP3 player decodes mp3 files and uses DMA transfers to send the PCM data to the audio DAC. One of the basic features that this MP3 player has is dynamic volume adjustment located in figure 1 by the blue potentiometer in the bottom right corner of the breadboard. Next, the mp3 player can pause and resume playback with the button in the middle and switch songs while playback is paused or resumed using two buttons for previous and next track. Finally, there is the capability to adjust screen contrast using the blue potentiometer to the left of the LCD screen.

Methodology

Mp3 Decoding:

- First the mp3 data is read from the USB into an input buffer for decoding.
- Next, 1 mp3 frame is decoded using the minimp3 library and the PCM data is stored in an audio buffer.
- Finally, a memmove operation is used to correctly position input buffer for the next frame decoding.

Audio Playback:

- The (ping pong) audio buffer is used in conjunction with DMA callbacks.
- DMA transfers are utilized to move audio PCM data from memory to cs43L22 audio DAC for playback.
- Callback functions are used for half transfers and complete transfers of PCM data, these call backs set flags such that the decoding process can continue on the next half of the ping pong buffer.
- In the case of a full transfer callback, it also changes DMA transfers to transmit data from the beginning of the audio buffer.

Track Linked List:

When the program first starts up it creates a doubly linked list of all the music tracks. This linked list holds the track name, a pointer to next track and a pointer to the previous track. After the linked list is created, the head and tail of the linked list are made to point to each other such that the LL makes a ring. This track information is used to traverse between music tracks forwards and backwards using the button on the breadboard.

LCD:

A driver file was written to facilitate communication with the LCD (LCM1602a). The LCM1602a is capable of 8 bit and 4 bit transmission. 8 bit transmission was used as the MCU has ample GPIO ports and there is a disadvantage with the delay between sending nibbles that 4 bit transmission requires. Custom volume characters were created for this project by writing 5x8 character patterns to the LCD controller's CGRAM at the program startup. The mp3 name displayed on the first line of the LCD screen continuously wraps around the display which speed depends on how often the update LCD function is called and a adjustable delay variable.

FreeRTOS:

FreeRTOS was used to schedule the different tasks that necessary for the program to function properly. These tasks include, mp3 decoding and playback, USB communication, updating the LCD, volume ADC polling, and button polling. In a previous version of this project no Real Time Operating System was used and this lead the code to being very non intuitive to follow and understand as the project size and scope increased.

Project Challenges

Robotic/Slowed Down Audio:

Name	Type	Stack Information	Activations	Total Blocked Time	Total Run Time	Time Interrupted	CPU Load
SysTick	#15		58 380		1.042 001 625 s	0.000 000 ms	0.20 %
Scheduler			95 816		4.188 029 250 s	8.927 063 ms	0.82 %
USBH_Thread	@3	436 @ 0x2000B540	91 117	7.611 836 688 s	11.272 623 750 s	68.123 438 ms	2.19 %
Update LCD	@2	924 @ 0x2000ABB0	246	0.006 126 625 s	0.941 062 063 s	2.353 188 ms	0.18 %
Read ADC	@2	436 @ 0x2000B000	117	0.005 112 188 s	0.007 723 563 s	0.000 000 ms	0.00 %
Read Buttons	@2	436 @ 0x2000B268	1 172	0.078 900 813 s	0.312 758 188 s	0.000 000 ms	0.06 %
Mp3 Play-back	@2	17924 @ 0x200064F8	85 748	0.000 000 000 s	492.257 052 375 s	957.068 875 ms	96.07 %
Task 0x314	@0	0 @ 0x00000000	1	0.000 000 000 s	0.000 000 000 s	0.000 000 ms	0.00 %
Idle			0		0.000 000 000 s	0.000 000 ms	0.00 %

Figure 2: CPU Task Information

Orignally, the audio playback did not work properly. The sound was audibly slower and distorted than normal mp3 playback. My first thought was that the sample rate of mp3 file was incorrectly lowered somewhere in the code. I tried increasing it however the audio distortion remained. Next, I thought the audio buffer datatype I was using to store the audio PCM data was of the incorrect size so I tried uint16_t and uint32_t, this just exacerbated audio distortion. Finally, I changed the code optimization from debug to O2 and the audio playback was excellent.

Pausing White Noise:

The BSP (Board Support Package) audio pause and resume functions did not work consistently. Half the time when the track is paused there is very audible white noise. After many attempts I was not able to establish causality, sometimes there was noise and sometimes it was fine. Therefore it was simpler to right my own pause and resume functions instead. To pause audio playback I the program shuts down playback entirely and to resume the program simply reinitialized playback. There is one lost mp3 frame with this method however since most mp3s only have a sample rate of 44100hz only 26 ms of audio is lost while resuming playback which is not noticeable.

Track Switch Stuttering:

When switching audio tracks when either the song ends or using the Previous/Next buttons on the breadboard, there was an audio stuttering noise for the half second it takes to start playing the next song. I suspected that the PCM data in the audio buffer was just being indiscriminately repeated from the DMA Transfers. The simple solution I found to solve this big was to clear the buffer while the program switches music tracks.

Memory Management:

- The primary difficulty of this project was managing memory as the STM32f407 only has 128 kB of ram. In this respect, the first obstacle was trying to find the track length of each mp3 file so that it could be displayed on the LCD screen. Trying to count mp3 frames was not useful considering the Cortex M4's low performace when compared typical desktop computers. This operation which would take 0.01 seconds on a desktop computer takes over 6 seconds on th STM32f407. This was not particularly useful when trying to display total time of a song on the LCD. There was another way to find the total samples of a track, if it is an Variable Bitrate mp3 than there would be a VBR TAG in the mp3 file with the sample information. If this tag could be found, then total duration of the mp3 could be calculated. Luckily the minimp3 library provides this functionality but the library's default behavior will use all 128kB of ram to perform the search. After consulting with the Github author it became possible to reduce this to 16kB, small enough to be utilized in this project.

- The next instance of memory management problems was trying to incorporate FreeRTOS into the project. The necessary memory required in the FreeRTOSconfig.h file was very large. After playing around a bit, all the HardFault errors were stopped however no audio was being played. After a day of trying to isolate the problem, it became apparent that the minimp3 decoding function was not returning any samples because of a memory limitation problem. Trying to increase the memory allocated for the decoding task was difficult because the program was already near the limit for the total memory utilization. Finally, after reducing the audio buffers the problem was able to be solved.

- The final difficulty regarding the memory was trying to introduce Seggar SystemView to analyze and debug the RTOS behavior to make sure the program was working as intended. The program was already at the limit for memory before this point so it was crucial to cut dow buffers to their minimum size when SystemView was being utilized. Since SystemView was only for debugging purposes it was possible to make the audio input buffer smaller than the advised amount without any problems for certain songs, enough to get snapshots of how to RTOS was behaving.

Conclusion

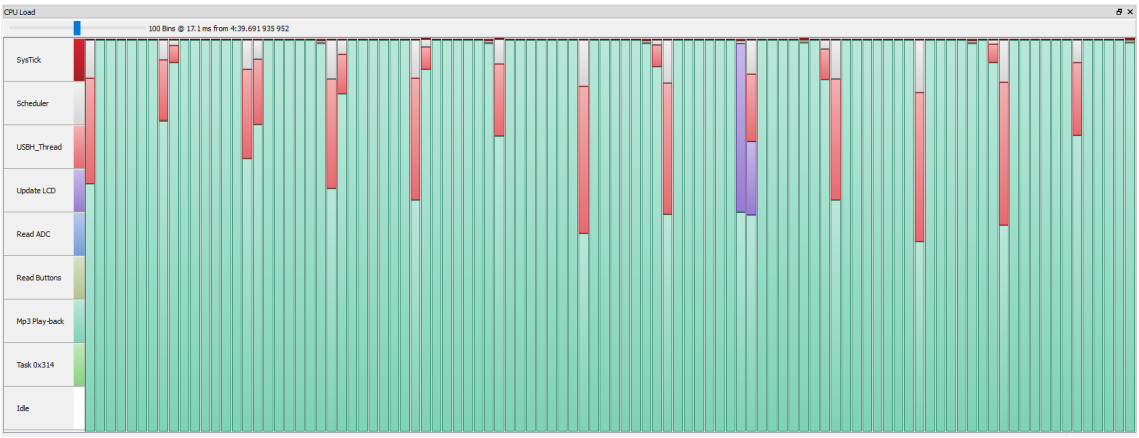

Figure 3: Typical CPU Load

I was successful in achieving the goals of this project as I am much more familiar with the STM32 tool-chain, programming with arm cortex, and FreeRTOS. I think there is still room for improvement in optimizing the decoding and playback process. In figure 3 above it is evident that the majority of the CPU load is spent on the MP3 Playback task. During the project I briefly considered using a ring buffer to decode the mp3s rather than costly memmove operations, however the minimp3 library only supports ring buffers on Linux OS. If I were to come back to this project at a later date implementing a ring buffer for decoding would be my first priority.

References

[1] "STM32F4 Series," STMicroelectronics. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32f4-series.html. [Accessed: 21-Mar-2021].

[2] Lieff, "lieff/minimp3," GitHub. [Online]. Available: https://github.com/lieff/minimp3. [Accessed: 21-Mar-2021].

[3] "Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions," FreeRTOS, 09-Mar-2021. [Online]. Available: https://www.freertos.org/. [Accessed: 21-Mar-2021].

Appendix

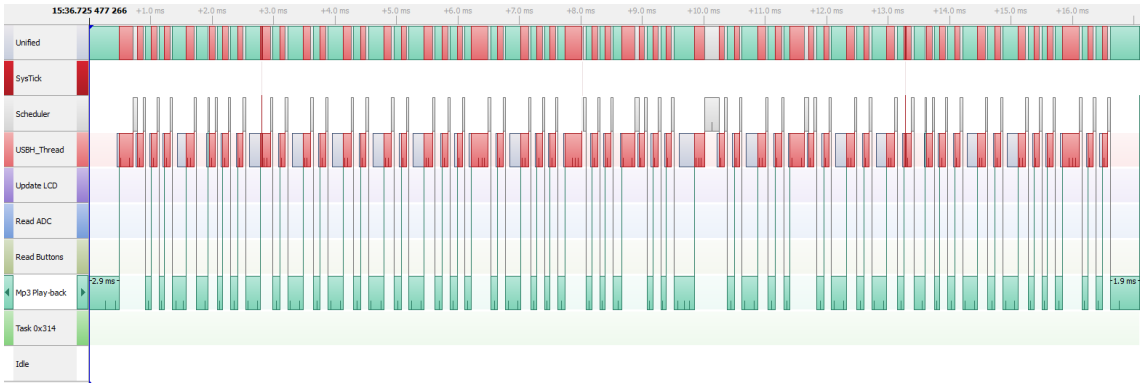

Figure 4: USBH Process

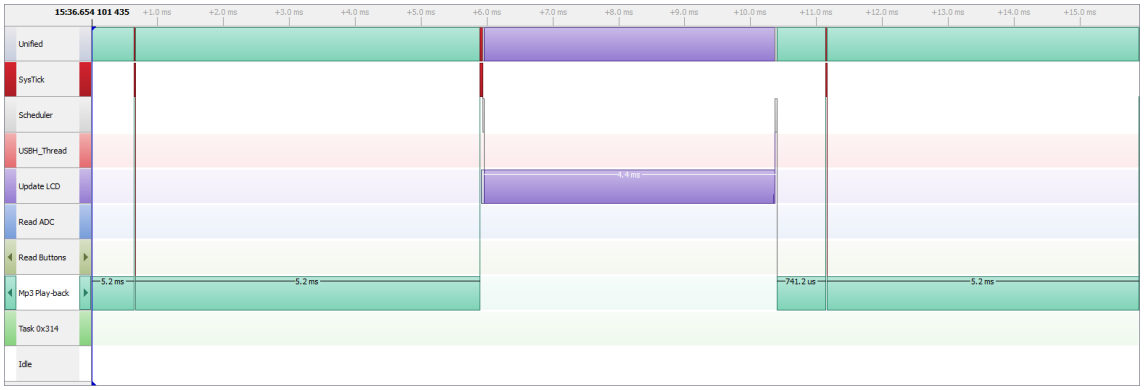

Figure 5: LCD Process

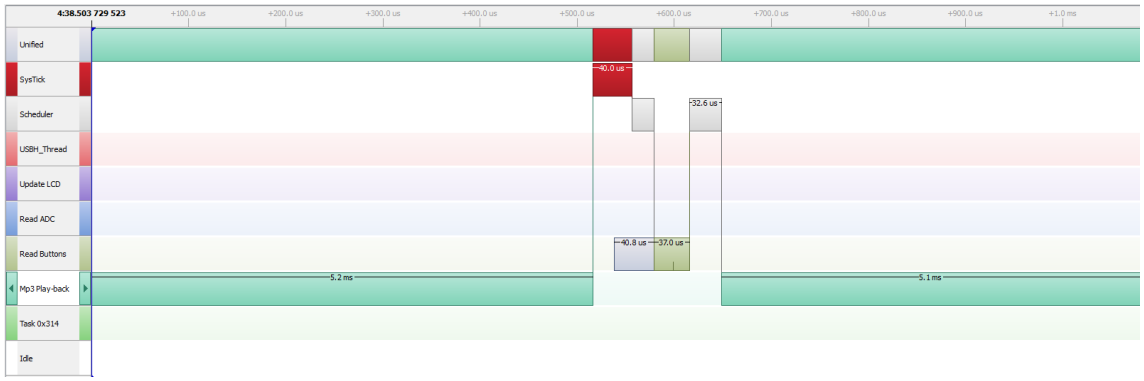

Figure 6: Poll Buttons Process

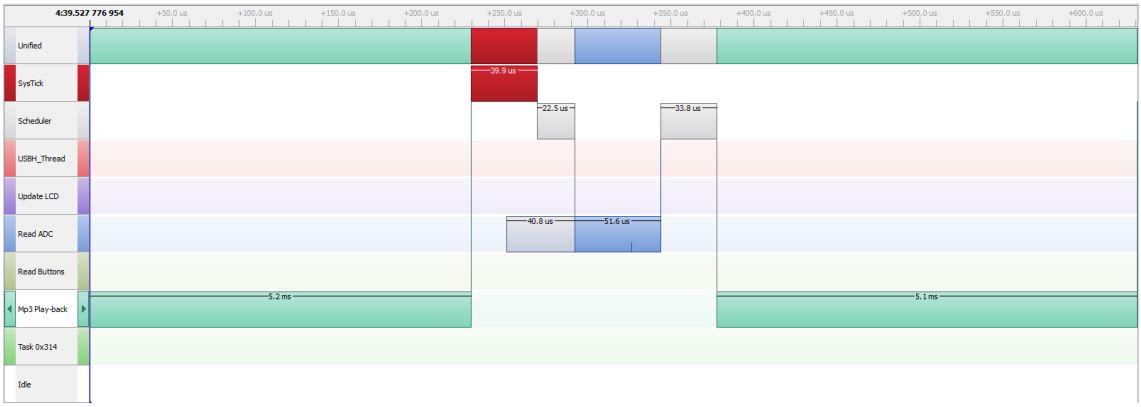

Figure 7: Poll ADC Process