Dane Walton

William Archer

Lab 03 Report

Part 1

In part 1 we were able to generate a certain frequency output using the timer compare registers and using interrupts. This served as a baseline to frequency generation.

Part 2

In part 2 we were able to create a function that allowed us to generate any frequency we wanted within the limits of the registers. This in the future would allow us to create the delays required for the unique frequency of each button.

Part 3

Part 3 was all about incorporating the keypad into the circuitry. We had to write code that iterated through the different rows and then checked each column to try and pinpoint which button was being pressed. We iterated through using an interrupt that would check every four milliseconds.

Part 4

In part 4, we had to both build a circuit to concert the wave from square to sinusoidal. This was accomplished using a combination of op amps, capacitor, resistors, and other circuitry. It allowed us to make audible signals. Also, we had to add to each button declaration a certain frequency that would be generated corresponding to the row and column of the button pushed.

Part 5

Part 5 was relatively easy as it was a basic conversion of the timer compare to CTC. In this instance it reset the counter every time it matched so it eliminated the need for two interrupts.

Code

1. #include <avr/io.h>
2. #include <avr/interrupt.h>
3. #include "util/delay.h"
4. #include <stdio.h>
5. #include <stdlib.h>
6. #include "uart.h"
8. //Define Frequencies
9. #define Freq1C 1209
10. #define Freq2C 1336
11. #define Freq3C 1477
12. #define FreqAC 1633
13. #define Freq1R 697
14. #define Freq4R 770
15. #define Freq7R 852
16. #define FreqStarR 941


20. //OC1A used as PB1
21. **int** HiOrLo;
22. **long** unsigned rowStallTime;
23. **long** unsigned columnStallTime;
24. **char** next\_char;
25. **int** iterate;
26. **char** first[8];
27. **char** second[8];
28. **int** rowHolder;
29. **char** keyPress;
31. /\*
32. Need to create a while loop for while key is depressed in each if statement
33. or branch to helper function and play out frequency on OC1A and OC1B.
34. \*/
36. **long** freqCalc(**long** inHertz) {
37. **return** 16000000/(2\*inHertz);
38. }
40. **short** **int** freqCalcZero(**long** inHertz) {
41. **return** 250000/(2\*inHertz);
42. }

45. ISR(TIMER2\_COMPA\_vect) {
46. **if** (rowHolder == 1) {
48. **if** (PIND & 0x04) {
49. keyPress = '1';
50. **int** test;
51. OCR1A = freqCalc(Freq1R);
52. OCR0A = freqCalcZero(Freq1C);
53. test = OCR0A;
54. printf("%i\n", test);

57. **return**;
58. //printf("1");
59. } **else** **if** (PIND & 0x08) {
60. keyPress = '2';

63. OCR1A = freqCalc(Freq1R);
64. OCR0A = freqCalcZero(Freq2C);
66. **return**;
67. //printf("2");
68. } **else** **if** (PIND & 0x10) {
69. keyPress = '3';

72. OCR1A = freqCalc(Freq1R);
73. OCR0A = freqCalcZero(Freq3C);
75. **return**;
76. //printf("3");
77. } **else** **if** (PIND & 0x20) {
78. keyPress = 'A';
80. OCR1A = freqCalc(Freq1R);
81. OCR0A = freqCalcZero(FreqAC);
83. **return**;
84. //printf("A");
85. }
87. rowHolder++;
89. PORTC &= ~(1 << PC2);
90. PORTC |= (1 << PC3);
91. PORTC &= ~(1 << PC4);
92. PORTC &= ~(1 << PC5);
94. } **else** **if** (rowHolder == 2) {

97. **if** (PIND & 0x04) {
98. keyPress = '4';
100. OCR1A = freqCalc(Freq4R);
101. OCR0A = freqCalcZero(Freq1C);
103. **return**;
104. //printf("4");
105. } **else** **if** (PIND & 0x08) {
106. keyPress = '5';
108. OCR1A = freqCalc(Freq4R);
109. OCR0A = freqCalcZero(Freq2C);
111. **return**;
112. //printf("5");
113. } **else** **if** (PIND & 0x10) {
114. keyPress = '6';
116. OCR1A = freqCalc(Freq4R);
117. OCR0A = freqCalcZero(Freq3C);
119. **return**;
120. //printf("6");
121. } **else** **if** (PIND & 0x20) {
122. keyPress = 'B';
124. OCR1A = freqCalc(Freq4R);
125. OCR0A = freqCalcZero(FreqAC);
127. **return**;
128. //printf("B");
129. }
131. rowHolder++;
133. PORTC |= (1 << PC4);
134. PORTC &= ~(1 << PC2);
135. PORTC &= ~(1 << PC3);
136. PORTC &= ~(1 << PC5);
138. } **else** **if** (rowHolder == 3) {


142. **if** (PIND & 0x04) {
143. keyPress = '7';
145. OCR1A = freqCalc(Freq7R);
146. OCR0A = freqCalcZero(Freq1C);
148. **return**;
149. //printf("7");
150. } **else** **if** (PIND & 0x08) {
151. keyPress = '8';
153. OCR1A = freqCalc(Freq7R);
154. OCR0A = freqCalcZero(Freq2C);
156. **return**;
157. //printf("8");
158. } **else** **if** (PIND & 0x10) {
159. keyPress = '9';
161. OCR1A = freqCalc(Freq7R);
162. OCR0A = freqCalcZero(Freq3C);
164. **return**;
165. //printf("9");
166. } **else** **if** (PIND & 0x20) {
167. keyPress = 'C';
169. OCR1A = freqCalc(Freq7R);
170. OCR0A = freqCalcZero(FreqAC);
172. **return**;
173. //printf("C");
174. }
176. rowHolder++;
178. PORTC &= ~(1 << PC2);
179. PORTC &= ~(1 << PC3);
180. PORTC &= ~(1 << PC4);
181. PORTC |= (1 << PC5);

184. } **else** **if** (rowHolder == 4) {
186. **if** (PIND & 0x04) {
187. keyPress = '\*';
189. OCR1A = freqCalc(FreqStarR);
190. OCR0A = freqCalcZero(Freq1C);
192. **return**;
193. //printf("\*");
194. } **else** **if** (PIND & 0x08) {
195. keyPress = '0';
197. OCR1A = freqCalc(FreqStarR);
198. OCR0A = freqCalcZero(Freq2C);
200. **return**;
201. //printf("0");
202. } **else** **if** (PIND & 0x10) {
203. keyPress = '#';
205. OCR1A = freqCalc(FreqStarR);
206. OCR0A = freqCalcZero(Freq3C);
208. **return**;
209. //printf("#");
210. } **else** **if** (PIND & 0x20) {
211. keyPress = 'D';
213. OCR1A = freqCalc(FreqStarR);
214. OCR0A = freqCalcZero(FreqAC);
216. **return**;
217. //printf("D");
218. }
220. rowHolder = 1;
222. PORTC |= (1 << PC2);
223. PORTC &= ~(1 << PC3);
224. PORTC &= ~(1 << PC4);
225. PORTC &= ~(1 << PC5);
226. }
228. OCR0A = 0;
229. OCR1A = 0;
231. }

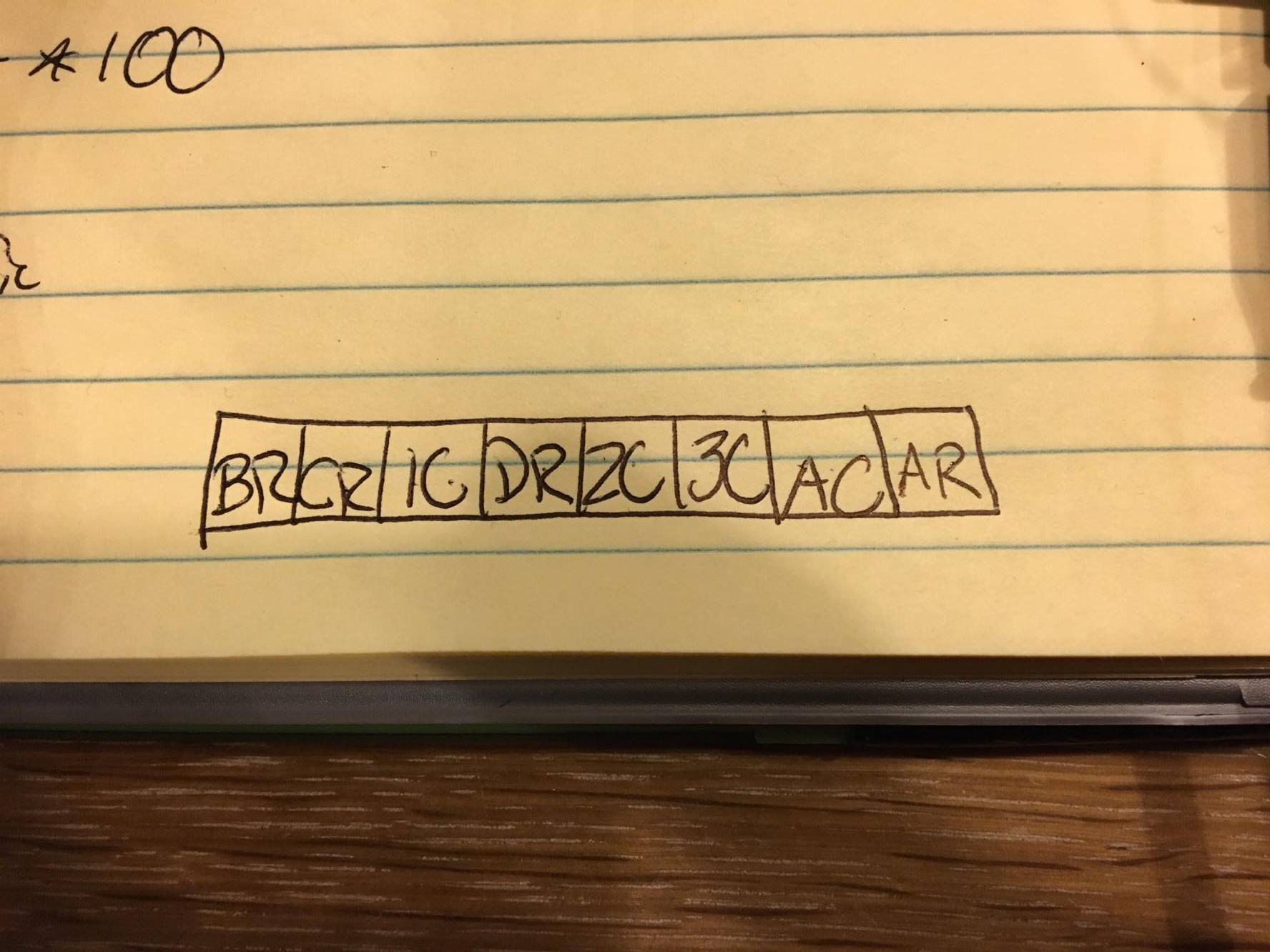
234. **int** main(**void**) {
236. DDRB |= 0xFF;
237. DDRC |= 0xFF;
238. DDRD |= 0x40;
239. PORTC = 0x00;
240. rowHolder = 1;
241. rowStallTime = 0;
242. columnStallTime = 0;
244. //Set rows early to prevent timing error
245. PORTC |= (1 << PC2);
246. PORTC &= ~(1 << PC3);
247. PORTC &= ~(1 << PC4);
248. PORTC &= ~(1 << PC5);
250. uart\_init();

253. //Timer 0 Config
254. TCCR0A = 0x42;
255. TCCR0B = 0x03;
256. TIMSK0 = 0x00;
257. OCR0A = 0;

260. //Timer 1 Config
261. TCCR1A = 0x40;
262. TCCR1B = 0x09;
263. TIMSK1 = 0x00;
265. OCR1A = 0;

268. //Timer 2 Config
270. TCCR2A |= 0x02;
271. TCCR2B |= 0x05;
272. TIMSK2 |= 0x02;
273. OCR2A = 0xFF;
275. /\*
277. //Get the first number, break on space, and get the second number
278. scanf("%s", &first);
279. scanf("%s", &second);

282. printf("First: %s\n", first);
283. printf("Second: %s\n", second);
285. //Convert the strings collected to longs
286. long unsigned freqInput = strtol(&first, NULL, 10);
287. long unsigned durationInput = strtol(&second, NULL, 10);
289. //Test print statements for the input freq and duration
290. printf("Freq: %i\n", freqInput);
291. printf("Dur: %i", durationInput);
293. \*/

296. sei();
298. **while**(1) {
299. }
301. }
302. The TCNT1 register is used to generate the low frequencies of the DTMF generator (the columns). The OCR1A/B register is used as an output/compare for the timer to know when to start and stop the pulses. TIMSK1 is used to configure the interrupts for Timer 1. TCCR1A/B is used to enable the CTC mode.
303. Minimum and maximum frequency respectively of Timer 1 are 122Hz and 8,000,000Hz. For Timer 0, it is 122Hz and 31,372Hz.
304. 1 clock tick = 6.25x10^(-5)ms. You can use this as a conversion to go back and forth between units.
305. Minimum and maximum frequency respectively of Timer 1 are 122Hz and 8,000,000Hz. For Timer 0, it is 122Hz and 31,372Hz.
306. 

The first character in this diagram is the key that’s being considered, and then the second character is either an R for row, or a C for column of the first character. These pins were connected correspondingly to the pins outlined on the directions.

1. We used the Timer2 Compare Output register to set “stallTimes” and had that updated every time the interrupt was made. To scan for keypresses, we wired up our keys to be high on columns, and then rows would be ground. We would then alternate columns by making one low and the rest high. Using pull-up resistors, we ensured the voltage would go to ground when the switch became active. Then using the PortD GPIO pins, we set them to input and would check to see if one was going low.
2. We would use pull-up resistors to ensure current was flowing. The keypad acted as a switch for the column and row so we could determine which key was being pressed. This is a pulldown switch. If it was a pullup switch, we would only need to change the output to low on PortB registers and then alternate one column each clock tick as high to scan for key presses.
3. We didn’t use a prescaler, but it would be useful in order to be able to set clock ticks more easily and achieve lower frequencies. We would be able to achieve lower frequencies since a scalar would let us lower our dividing number and circumvent the limitation we have on the size of the timer counters.
4. CTC mode is easier to deal with, but you have to consistently change the frequencies. Using interrupts, its easy to change the frequencies you need to output, but deals with overflows and other interrupts which may have more bugs.