The Last Music Trainer
Embedded System Design, Lab 4

Ben Lorenzetti

October 8, 2015

Contents

1 Objectives and Problem Descriptions **2**
1.1 Beginner Music Trainer 2

2 Procedure **2**
2.1 Circuit Design 2
2.2 Data Design 4
2.3 Data Generation from RTTTL Files 5
2.4 Implementation Flowchart 5

3 Expected Results **5**
3.1 Laboratory Demonstration 5

4 Experiment and Design Revisions **5**
4.1 Changes During Debugging 5

5 Observations **7**
5.1 Sound Intensity and Memory Mapping 7

6 Discussion **7**
6.1 Discussion 7

7 Exercises **7**

8 Implementation Code **9**
8.1 The-Last-Music-Trainer.bs2 9
8.2 generate-rtttl-song-data.c 13

1 Objectives and Problem Descriptions

1.1 Beginner Music Trainer

The objective of this lab is to develop microcontroller-based, beginner music trainer with the following features:

1. Be capable of playing simple songs using PBasic's `FREQOUT Pin, Duraction, Freq1 {, Freq2}` command. This including the ability to
   a. play any piano tone in the 4^{th}–7^{th} octave {C, C#, D, D#, E, F, F#, G, G#, A, A#, B};
   b. change the base tempo's whole note duration;
   c. play each note for {1, 1/2, 1/4, 1/8, 1/16, 1/32} of the base tempo;
   d. play each note for $1\frac{1}{2} * \{1, 1/2, 1/4, 1/8, 1/16, 1/32\}$ (dotten notes) of the base tempo; and
2. allow user to select from a menu of 5 Ring Tone Text Transfer Language (RTTTL) songs, using a pushbutton switch–each push causes an advance to the next song.
3. allow user to increase the base tempo by a factor of 1–4 using a potentiometer rotary knob.
4. display the current note being played on a 7-segment display, using the decimal point to indicate sharp notes. Furthermore, display the note's octave with individual LEDs.
5. play each song in an infinite loop if user does not advance to the next song.

2 Procedure

2.1 Circuit Design

Objectives 1-4 from the problem specifications (section 1.1) each require some hardware for their functionality. Playing simple songs requires a piezo element, user song selection requires a pushbutton, adjusting the base tempo requires a rotary potentiometer knob, and displaying the current note requires a 7-segment display.

The circuit diagram for these 4 hardware sections are shown in figure 1.

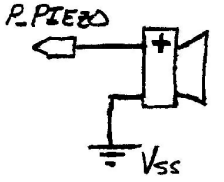
For the pushbutton circuit, a capacitor is added in parallel to the switch; if the capacitor is charged immediately after reading the pin, this capacitive switch will have memory in case the switch is contacted while the controller is busy.

For the rotary potentiometer, the RC time constant was found in a prior lab to be

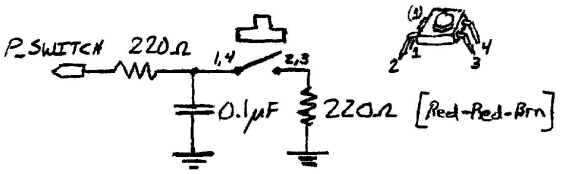
$$\tau = \left[\frac{0.06365 R_{pot}}{\Omega} + 636.5\right] * [2\mu s] \tag{1}$$

The time constant is given in units of 2 μs because that is the measurement unit of the PBasic Stamp controller. The potentiometer resistance has a domain of 0 Ω-10 kΩ, resulting in an expected range of 636.5-1273. The and the range needs to be mapped linearly to a speed factor of 1–4 and, from experience in the last lab, there should be built in tolerance in case the time measurement is outside the expected range. This can be done with the mapping

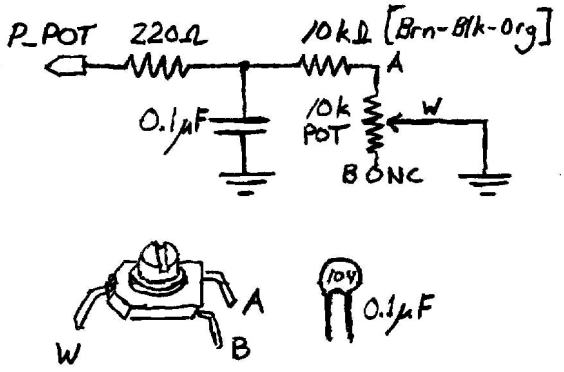
$$y = \frac{1}{256}\left(\frac{\tau}{[2\mu s]} - 443\right) + 1 \tag{2}$$



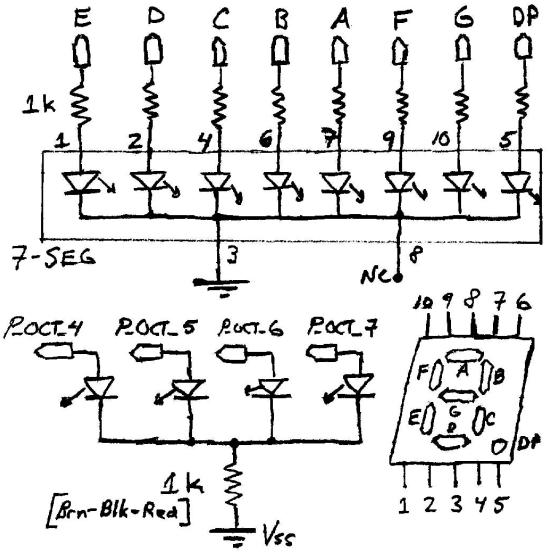
(a) Piezo Circuit



(b) Pushbutton Switch Circuit



(c) Potentiometer RC Circiut



(d) LED Display Circuit

Figure 1: Hardware for The Last Music Trainer

Table 1: 10-Bit Note Encoding Truth Table

Length	∗	Bit Code	Dec.	Whole Note Divisor	Letter	#	Bit Code	Dec.	7^{th} Octave Freq
𝅝	0	000	0	(1 << 0)	A♮	0	000	0	3520.0
𝅝.	1	000	8		A♯	1	000	8	3729.3
𝅗𝅥	0	001	1	(1 << 1)	B♮	0	001	1	3951.1
𝅗𝅥.	1	001	9		C♮	0	010	2	2093.0
♩	0	010	2	(1 << 2)	C♯	1	010	10	2217.5
♩.	1	010	10		D♮	0	011	3	2349.3
♪	0	011	3	(1 << 3)	D♯	1	011	11	2489.0
♪.	1	011	11		E♮	0	100	4	2637.0
𝅘𝅥𝅯	0	100	4	(1 << 4)	F♮	0	101	5	2793.8
𝅘𝅥𝅯.	1	100	12		F♯	1	101	13	2960.0
𝅘𝅥𝅰	0	101	5	(1 << 5)	G♮	0	110	6	3136.0
𝅘𝅥𝅰.	1	101	13		G♯	1	110	14	3322.4
					P	x	111	7,15	-
					Octave		Bit Code	Dec.	7^{th} Freq. Divisor
					4^{th}		00	0	$1 << (3-0)$
					5^{th}		01	1	$1 << (3-1)$
					6^{th}		10	2	$1 << (3-2)$
					7^{th}		11	3	$1 << (3-3)$

2.2 Data Design

For this project, many songs will have to be stored in the Stamp's nonvolatile EEPROM memory. The smaller the format of the song data, the better, because more-and longer-songs will be able to be played. Based on the full set of possible notes from the specification in section 1.1 and on the RTTTL data format, the full information for a single note can be encoded in 10 bits.

A single note is encoded according to the following scheme:

```
Bits[9:8] = Note Octave
Bit [7]   = Sharp Flag
Bits[6:4] = Note Letter
Bit [3]   = Dotted Flag
Bits[2:0] = Note Type (whole, half, etc.)
```

For example, listed most-significant bit first, the note `00|1|001|0|010 = 146` would be octave=00, sharp=1, letter=1, dotted=0, and note_type=010. The mapping of the bit codes to physical, musical quantities are shown in table 1.

For a full song, the data is encoded according to the following byte scheme:

```
[Song X Base Tempo Byte] [Song Length Byte] [Data Byte 0] [Data Byte 1] ...
... [Data Byte N] [Song X+1 Base Tempo Byte]...
```

There are two bytes for the song meta data, a base tempo of 0-255 beats per minute and song length of 0-255 notes. The number of data bytes is $N =$ number of notes $* \frac{10\text{bits/note}}{8\text{ bits/byte}}$. If the size of the data does not fall on a byte boundary, N is rounded up to the nearest integer and the extra bits are padded with zeros. The next song starts at byte N+1.

The bit and byte packing orders are both least significant first. For example, take bytes 0-4 of Let it Be by the Beatles.

```
|  B0=100  |  B1=26   |  B2=68   |  B3=206   | B4=40     |
| 00100110 | 01011000 | 00100010 | 01 110011 | 0001 0100 |
|Base Tempo|-#of Notes|----Note 0----|---Note 1----|-Low 4 Bits of Note 2 |
| 00100110 | 01011000 | 00100010   01|110011   0001|0100                   |
```

For human decoding, Note 0 is rewritten with its most-significant bit first:

```
Note 0 = 1001000100; octave=10, sharp=0, letter=100, dotted=0, note_type=100
```

2.3 Data Generation from RTTTL Files

It would be very monotonous to encode RTTTL songs to my 10-bit format by hand, so I wrote a C script for automating the process. This program is included in section 8.2.

2.4 Implementation Flowchart

Unfortunately the minimum size for a note (10 bits) does not fall evenly on a byte boundary. Therefore, there is a tradeoff between data compactness and program complexity. I chose to prioritize data compactness, and the result is a programming implementation with most of its complexity involved in unpacking data.

The implementation flowchart is shown in figure 2. Most of the states and logic are dedicated to unpacking data in a bitwise manner. The only states/transitions not involved in the bitwise unpacking of data are shown as dotted lines in the flowchart.

3 Expected Results

3.1 Laboratory Demonstration

I expected my microcontroller and circuit to behave as described by the problem specifications in section 1.1. For the lab demonstration, the TA should be able to recognize at least 5 songs, switch between songs with the pushbutton, change the song speed, and see notes displayed on the 7-segment display and octave LEDs.

4 Experiment and Design Revisions

4.1 Changes During Debugging

During debugging, I spend some time fixing problems associated with the bit packing scheme. There were three other bugs I fixed:

First, I was implementing pause notes as notes of zero frequency (`FREQOUT pin, duration, 0`), but this resulting is a clicking noise that was unpleasent. I fixed this by wrapping the `FREQOUT` statement with IF (freq=0) logic.

Second, the songs were being played too fast. I fixed this by moving the speed factor from the potentionmenter (1-4) from the denominator of the duration variable to the numerator.

Third, an A note was being played at the beginning of every song. I fixed this by adding an initialization value for `note` in the Reset_Song_Parameters state.

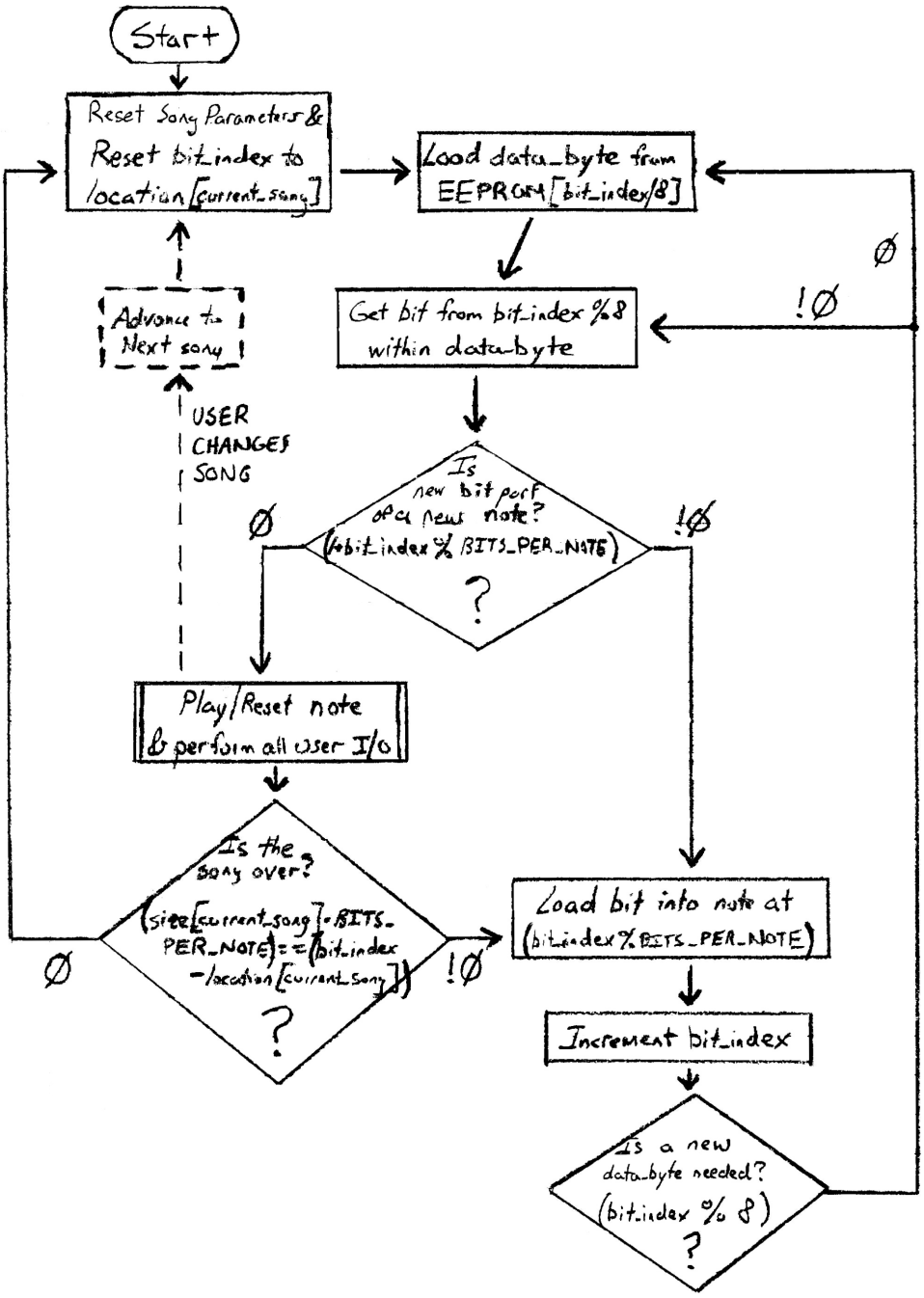


Figure 2: Music Trainer Implementation Flowchart

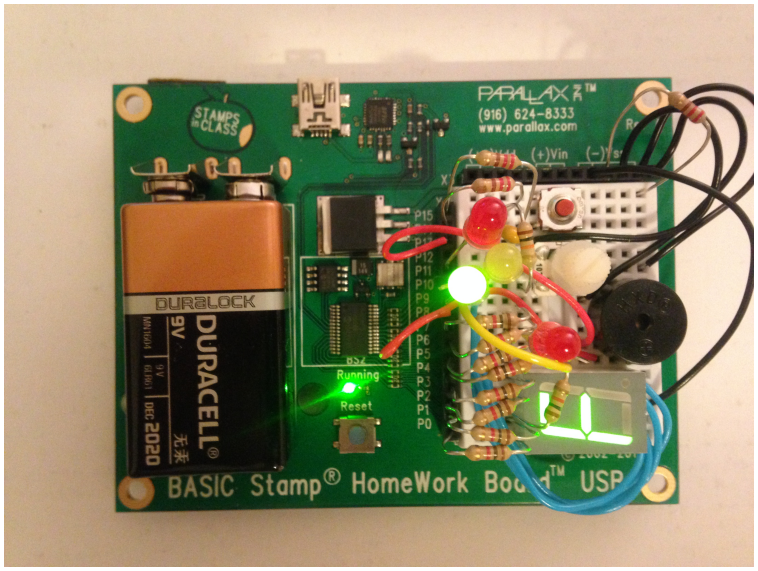


Figure 3: My Music Trainer Playing The Final Countdown

5 Observations

5.1 Sound Intensity and Memory Mapping

The last music trainer worked as expected after some initial debugging. I did notice two interesting things. First, the piezo speaker seemed to play some notes louder than others. Second, I looked at the memory map for my program and it seems my bit packing scheme worked well–there was plenty of additional room for more songs.

A photo of my music trainer and its memory map are shown in figure 3 and figure 4.

6 Discussion

6.1 Discussion

Microcontrollers can easily be used to generate simple tones with pulse width modulation on digital I/O pins. Simple songs such as the original cell phone ring tones can be played as long as more than a single tone at any moment is not required. (Actually you could play more tones at once with additional pins and a lower level language).

7 Exercises

There were no exercises for this lab.

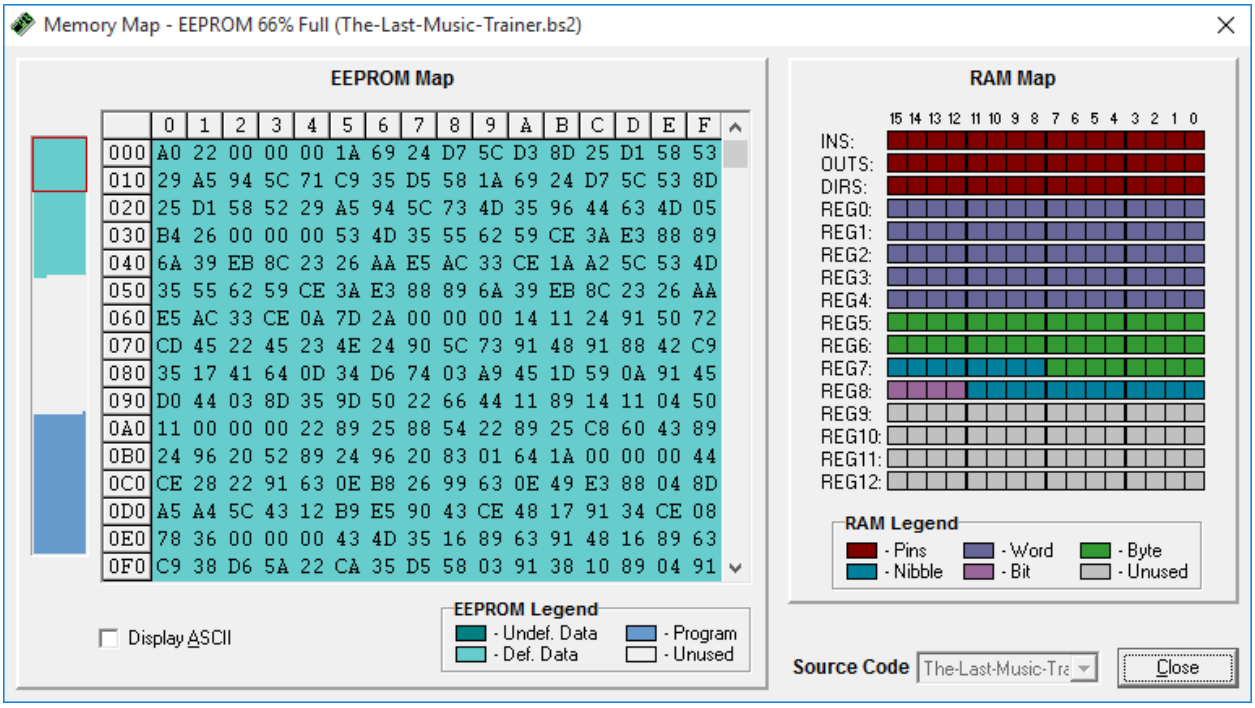


Figure 4: EEPROM Memory Mapping with Six Songs

8 Implementation Code

8.1 The-Last-Music-Trainer.bs2

```
' The-Last-Music-Trainer.bs2
' {$STAMP BS2}
' {$PBASIC 2.5}

' I/O Port Constants
P_PIEZO CON 13
P_POT CON 14
P_SWITCH CON 15
P_OCT_4 CON 8
P_SHARP CON 4
OUTS = $0000 + (1 << P_SWITCH)
DIRS = $FFFF

' Constants Related to the Bit Packing of Song Data
BITS_PER_NOTE CON 10
NOTE_MASK CON %0001110000
NOTE_SHIFT CON 4
SHARP_FLAG CON 7
TYPE_MASK CON %0000000111
TYPE_SHIFT CON 0
DOTTED_FLAG CON 3
OCTAVE_MASK CON %1100000000
OCTAVE_OFFSET CON 8

' Constants Related to the Byte Packing of Song Data
TEMPO_OFFSET CON 0
NUMBER_OF_NOTES_OFFSET CON 1
FIRST_NOTE_OFFSET CON 2

' Timing Constants
CHARGE_TIME CON 1
MILLISECONDS_PER_MINUTE CON 60000

' Store Several Songs in EEPROM
NUMBER_OF_SONGS CON 6

' 1. Let it Be
DATA 100, 26, 68, 206, 40, 34, 145, 99, 14, 184, 38, 153, 99, 14, 73
DATA 227, 136, 4, 141, 165, 164, 92, 67, 18, 185, 229, 144, 67, 206, 72, 23
DATA 145, 52, 206, 8

' 2. Final Countdown
DATA 125, 42, 20, 17, 36, 145, 80, 114, 205, 69, 34, 69, 35, 78, 36
DATA 144, 92, 115, 145, 72, 145, 136, 66, 201, 53, 23, 65, 100, 13, 52, 214
DATA 116, 3, 169, 69, 29, 89, 10, 145, 69, 208, 68, 3, 141, 53, 157, 80, 34
DATA 102, 68, 17, 137, 20, 17, 4

' 3. Star Wars
DATA 180, 38, 83, 77, 53, 85, 98, 89, 206, 58, 227, 136, 137, 106
DATA 57, 235, 140, 35, 38, 170, 229, 172, 51, 206, 26, 162, 92, 83, 77, 53
```

```
DATA 85, 98, 89, 206, 58, 227, 136, 137, 106, 57, 235, 140, 35, 38, 170
DATA 229, 172, 51, 206, 10

' 4. Walk of Life; Tempo=160, Notes=34
DATA 160, 34, 26, 105, 36, 215, 92, 211, 141, 37, 209, 88, 83, 41
DATA 165, 148, 92, 113, 201, 53, 213, 88, 26, 105, 36, 215, 92, 83, 141
DATA 37, 209, 88, 82, 41, 165, 148, 92, 115, 77, 53, 150, 68, 99, 77, 5

' 5. Tarzan
DATA 120, 54, 67, 77, 53, 22, 137, 99, 145, 72, 22, 137, 99, 201, 56
DATA 214, 90, 34, 202, 53, 213, 88, 3, 145, 56, 16, 137, 4, 145, 56, 144, 140
DATA 99, 173, 37, 162, 92, 67, 77, 53, 22, 137, 99, 145, 72, 22, 137, 99, 201
DATA 56, 214, 90, 34, 202, 53, 213, 88, 3, 145, 56, 16, 137, 4, 145, 56, 144
DATA 140, 99, 173, 5

' 6. Van Halen Eruption
DATA 120, 299, 165, 146, 86, 84, 121, 164, 21, 85, 30, 105, 69, 149
DATA 71, 90, 81, 165, 146, 86, 84, 121, 164, 21, 85, 30, 105, 69, 149, 71, 90
DATA 81, 5, 145, 86, 84, 65, 164, 21, 85, 16, 105, 69, 21, 68, 90, 81, 5, 145
DATA 86, 84, 65, 164, 21, 85, 16, 105, 69, 21, 68, 90, 109, 5, 209, 86, 93, 65
DATA 180, 85, 87, 16, 109, 213, 21, 68, 91, 117, 5, 209, 86, 93, 65, 180, 85, 87
DATA 17, 109, 213, 85, 68, 91, 117, 21, 17, 85, 94, 69, 68, 149, 87, 17, 81, 229
DATA 85, 68, 84, 121, 21, 17, 85, 94, 69, 68, 149, 87, 17, 81, 229, 85, 68, 84
DATA 121, 37, 18, 85, 86, 69, 68, 149, 85, 17, 81, 101, 85, 68, 84, 89, 21, 17
DATA 85, 86, 69, 68, 149, 85, 35, 81, 101, 213, 72, 84, 117, 53, 82, 87, 80, 141
DATA 212, 21, 84, 35, 117, 5, 213, 72, 93, 65, 53, 82, 87, 80, 141, 212, 21, 84
DATA 36, 117, 5, 21, 73, 93, 65, 69, 82, 84, 94, 145, 20, 149, 87, 36, 69, 229
DATA 21, 73, 81, 121, 69, 82, 84, 94, 145, 21, 145, 87, 100, 69, 228, 21, 89, 17
DATA 121, 69, 178, 87, 100, 121, 20, 21, 201, 81, 145, 21, 209, 88, 100, 97, 164
DATA 22, 89, 24, 169, 69, 22, 68, 98, 145, 5, 145, 88, 100, 121, 20, 21, 89, 94
DATA 69, 53, 22, 89, 17, 141, 69, 86, 68, 99, 145, 133, 145, 90, 100, 97, 164, 22
DATA 89, 16, 137, 69, 22, 68, 98, 145, 229, 81, 84, 100, 121, 20, 213, 88, 16
DATA 137, 53, 22, 68, 98, 141, 229, 81, 84, 99, 121, 20, 213, 88, 22, 97, 53, 150
DATA 69, 88, 141, 213, 21, 68, 99, 117, 5, 145, 88, 22, 97, 37, 150, 69, 88, 137
DATA 213, 17, 84, 98, 117, 4, 149, 88, 21, 121, 37, 86, 69, 94, 137, 69, 145, 85
DATA 98, 81, 100, 85, 84, 27, 117, 21, 213, 70, 93, 69, 181, 81, 87, 81, 109, 212
DATA 85, 84, 27, 117, 21, 213, 70, 93, 69, 181, 81, 87, 81, 109, 212, 85, 84, 20
DATA 89, 21, 21, 69, 22


' Declare Variables
song VAR Nib
note VAR Word
current_bit VAR Bit
bit_index VAR Word
song_start_byte VAR Word
data_byte VAR Byte
note_type VAR Nib
note_dotted VAR Bit
note_letter VAR Byte
note_sharp VAR Bit
oct7_freq VAR Word
note_octave VAR Nib
default_tempo VAR Byte
```

```
song_size VAR Byte
notes_played VAR Byte
speed_factor VAR Nib
duration VAR Word
pushbutton_switch VAR Bit


Start:
song = 0
song_start_byte = 0

Reset_Song_Parameters:
  PAUSE 1000
  READ (song_start_byte + TEMPO_OFFSET), default_tempo
  READ (song_start_byte + NUMBER_OF_NOTES_OFFSET), song_size
  bit_index = 0
  note = %0001110000
  DEBUG "song=", DEC song, ", song_size=", DEC song_size, ", tempo=", DEC default_tempo
  DEBUG ", song_start_byte=", DEC song_start_byte, CR

Load_Data_Byte_from_EEPROM:
  READ (song_start_byte + FIRST_NOTE_OFFSET + (bit_index >> 3)), data_byte

Get_Bit_from_Data_Byte:
  current_bit = 1 & (data_byte >> (bit_index // 8))

Is_New_Bit_Part_of_a_New_Note:
  IF (0 = (bit_index // BITS_PER_NOTE)) THEN Perform_All_User_IO
  'else Load current_bit into note

Load_Bit_into_Note:
  note = note | (current_bit << (bit_index // BITS_PER_NOTE))

Increment_Bit_Index:
  bit_index = 1 + bit_index

Is_a_New_Data_Byte_Needed:
  IF (0 = (bit_index // 8)) THEN Load_Data_Byte_from_EEPROM
  'Else'
  GOTO Get_Bit_from_Data_Byte

Perform_All_User_IO:
  ' Display Letter of 7-Segment Display
  note_letter = (note & NOTE_MASK) >> NOTE_SHIFT
  LOOKUP note_letter, [$AF, $E3, $C6, $E9, $C7, $87, $6F, $8F], note_letter
  OUTS = (OUTS & $FF00) | note_letter
  ' Display "Sharp" Decimal Points on 7-Segment Display
  note_sharp = (note & (1 << SHARP_FLAG)) >> SHARP_FLAG
  OUTS = (OUTS & ($FFFF - (1 << P_SHARP))) | (note_sharp << P_SHARP)
  ' Display Octave LEDs
  OUTS = (OUTS & ($FFFF - ($F << P_OCT_4))) | (1 << (P_OCT_4 + (3 & note_octave)))
  ' Read Speed from Potentiometer
  HIGH P_POT
  PAUSE CHARGE_TIME
```

```
  RCTIME P_POT, 1, duration
  speed_factor = ((duration - 443) >> 8) + 1
  duration = MILLISECONDS_PER_MINUTE / default_tempo
  duration = duration * speed_factor
  ' Read Pushbutton Switch for Advancing to Next Song
  pushbutton_switch = (INS & (1 << P_SWITCH)) >> P_SWITCH
  DIRS = DIRS | (1 << P_SWITCH)
  OUTS = OUTS | (1 << P_SWITCH)
  PAUSE CHARGE_TIME
  DIRS = DIRS & ($FFFF - (1 << P_SWITCH))
  IF (0 = pushbutton_switch) THEN Advance_to_Next_Song
  ' ELSE Is the song Over?

Play_and_Reset_Note:
  ' Decode Note
  note_type = (note & TYPE_MASK) >> TYPE_SHIFT
  duration = duration >> note_type
  note_dotted = (note & (1 << DOTTED_FLAG)) >> DOTTED_FLAG
  duration = duration + (note_dotted * (duration >> 1))
  note_letter = (note & (NOTE_MASK | SHARP_FLAG)) >> NOTE_SHIFT
  LOOKUP note_letter, [3520, 3951, 2093, 2349, 2637, 2794, 3136, 0, 3729, 0, 2218, 2489, 0, 2960, 3322,
  note_octave = (note & OCTAVE_MASK) >> OCTAVE_OFFSET
  ' Play the Note
  IF (0 = oct7_freq) THEN Skip_Conditional
    FREQOUT P_PIEZO, duration, (oct7_freq >> (3-note_octave))
  Skip_Conditional:
  ' Reset Note
  note = 0

Is_the_Song_Over:
  IF (bit_index >= (song_size * BITS_PER_NOTE)) THEN Reset_Song_Parameters
  ' ELSE
  GOTO Load_Bit_into_Note:

Advance_to_Next_Song:
  PAUSE 1000
  ' Recharge the switch again--just in case someone has sticky fingers
  DIRS = DIRS | (1 << P_SWITCH)
  OUTS = OUTS | (1 << P_SWITCH)
  PAUSE CHARGE_TIME
  DIRS = DIRS & ($FFFF - (1 << P_SWITCH))
  ' Find the song start location in EEPROM
  song = (song + 1) // NUMBER_OF_SONGS
  song_start_byte = 0
  i VAR Nib
  i = 0
  DO WHILE (i < song)
    READ (song_start_byte + NUMBER_OF_NOTES_OFFSET), data_byte
    song_start_byte = song_start_byte + FIRST_NOTE_OFFSET + (((data_byte * BITS_PER_NOTE) + 7) / 8)
    i = i + 1
  LOOP
  GOTO Reset_Song_Parameters:
```

8.2 generate-rtttl-song-data.c

```
/*  generate-rtttl-song-data.c
 *
 *  Use this program to convert RTTTL format text strings into
 *  byte-packed data for use in my PBasic music trainer.
 *
 *  The RTTTL string should be in a text file, passed in through
 *  the command line. The output will be a set of bytes in base-10
 *  digits which can be copied directly following PBasic's DATA command
    .
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUFFER_SIZE 1000
#define ASCII_ZERO_CONVERSION 48

int log_base_two (int input) {
        int temp;
        for (temp = 0; input > (1 << temp); temp++) {}
        return temp;
}

int main (int argc, char* argv[])
{
        char rtttl_buffer[BUFFER_SIZE];
        int rtttl_size;
        int ret_val;
        int default_dur = 0;
        int default_oct = 0;
        int default_beat = 0;
        int note_buffer[BUFFER_SIZE];
        int note_buffer_size = 0;
        int note = 0;
        int sharp = 0;
        int duration = 0;
        int octave = 0;
        int dotted = 0;
        int data_buffer[BUFFER_SIZE];
        int data_buffer_size = 0;

        if (argc <= 1) {
                printf ("Usage: %s rtttl_file.txt\n", argv[0]);
                exit (EXIT_FAILURE);
```

```
}

FILE* fp;
fp = fopen (argv[1], "r");
if (!fp) {
        printf ("Error: Could not find file %s\n", argv[1]);
}

for (rtttl_size = 0; rtttl_size < (BUFFER_SIZE-1); rtttl_size
   ++) {
        ret_val = fscanf (fp, "%c", (rtttl_buffer + rtttl_size)
           );
        if (ret_val == EOF)
                break;
        if (ret_val != 1) {
                printf ("Error reading RTTTL file: fscanf()
                   returns %d\n", ret_val);
                exit (EXIT_FAILURE);
        }
}
rtttl_buffer[rtttl_size] = 0;
fclose (fp);
printf ("RTTTL Format:\n%s", rtttl_buffer);

char* iter;
iter = strstr (rtttl_buffer, "d=");
if (!iter) {
        printf ("Error: default duration not found\n");
        exit(EXIT_FAILURE);
}
iter += 2;
default_dur = *iter - ASCII_ZERO_CONVERSION;
if (*(++iter) != ',') {
        default_dur *= 10;
        default_dur += *iter - ASCII_ZERO_CONVERSION;
}
printf ("Default Duration:\t%d\n", default_dur);

iter = strstr (rtttl_buffer, "o=");
if (!iter) {
        printf ("Error: default octave not found\n");
        exit(EXIT_FAILURE);
}
default_oct = *(iter + 2) - ASCII_ZERO_CONVERSION;
printf ("Default Octave:  \t%d\n", default_oct);

iter = strstr (rtttl_buffer, "b=");
if (!iter) {
```

```
        printf ("Error: default beat not found\n");
        exit(EXIT_FAILURE);
}
for (iter = iter + 2; (*iter) != ':'; iter++) {
        if (!iter) {
                printf ("Error: default beat never ends...\n");
                exit (EXIT_FAILURE);
        }
        default_beat *= 10;
        default_beat += *iter - ASCII_ZERO_CONVERSION;
}
printf ("Default Beat:     \t%d\n", default_beat);

iter++;
while (iter - rtttl_buffer < rtttl_size) {
        if (*iter < 'a') {
                duration = *(iter++) - ASCII_ZERO_CONVERSION;
                if (*iter < 'a') {
                        duration *= 10;
                        duration += *(iter++) -
                           ASCII_ZERO_CONVERSION;
                }
        }
        else
                duration = default_dur;
        duration = log_base_two (duration);
        if ((*iter < 'a') || (*iter > 'p')) {
                printf ("Error: at iter=%d, expecting letter
                   between a-p\n", (int)(iter-rtttl_buffer));
                exit (EXIT_FAILURE);
        }
        if (*iter == 'p')
                note = 'h' - 'a', iter++;
        else
                note = *(iter++) - 'a';
        if (*iter == '#')
                sharp = 1, iter++;
        else
                sharp = 0;
        if (*iter == '.')
                dotted = 1, iter++;
        else
                dotted = 0;
        if ((*iter >= '4') && (*iter <= '7'))
                octave = *(iter++) - '4';
        else
                octave = default_oct;
```

```
                note_buffer[note_buffer_size] = (15 & duration) + (
                    dotted << 3)
                note_buffer[note_buffer_size] += (note << 4) + (sharp
                    << 7) + (octave << 8);
                if (*iter == ',')
                        iter++, note_buffer_size++;
                else
                        break;
        }

        printf ("Unpacked Data Array (starting with lowest bit of
            lowest byte) (size=%d):\n", note_buffer_size);
        for (int i=0; i<(note_buffer_size * 10); i++) {
                printf ("%d", ((note_buffer[i/10] >> (i%10))&1));
                if (i%8 == 7)
                        printf ("|");
        }

        for (int bit=0; bit < (note_buffer_size * 10); bit++) {
                data_buffer[bit/8] += ((note_buffer[bit/10] >> (bit%10)
                    )&1) << (bit % 8);
                if (0 == bit % 8)
                        data_buffer_size++;
        }

        printf ("\nPacked Data Array: ([Beats per Minute][Total Notes][
            Byte 1][Byte 2]...[Byte N])\n");
        printf ("(where N = Total Notes * 10/8 rounded up)\n");
        printf ("%d, %d, 0, 0, 0", default_beat, note_buffer_size);
        for (int i=0; i<data_buffer_size; i++)
                printf (", %d", (int) data_buffer[i]);
        printf ("\n");

        return 0;
}
```