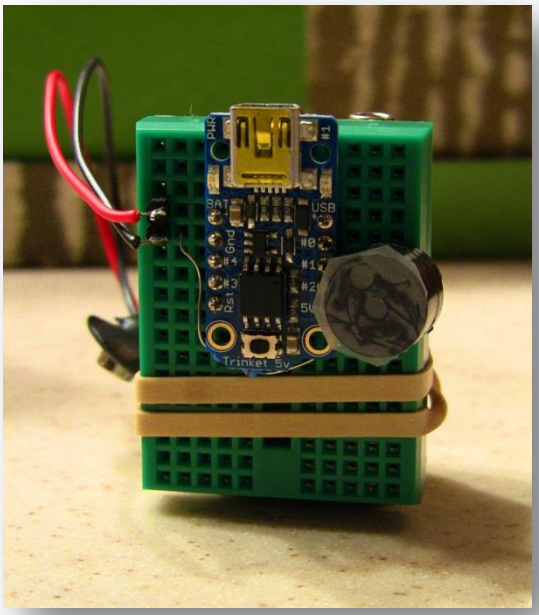


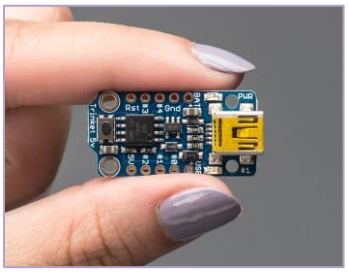
A Tone Generator for Trinket

Bruce E. Hall, W8BH

1) INTRODUCTION

The Trinket is a tiny microcontroller board by Adafruit. At $8 per board, it's an easy and inexpensive way to add microcontroller power to any project. The Arduino IDE makes development a snap. Unfortunately, not all Arduino software is compatible with this little board. One such example is the Tone Library. You can add it to your Trinket project, but it doesn't make a sound! In this tutorial I will show you how to make your Trinket sing.

2) HOW TO START



Get yourself a Trinket, if you don't have one! Everything you need is at Adafruit: the 5V trinket, a piezo buzzer, a small breadboard, and a USB-miniB cable. I also recommend a 200-300 ohm resistor to limit current through the piezo.

Next, you need to add the Arduino IDE to your computer. The easiest way is to use an IDE preconfigured for the Trinket. Go to the Adafruit learning site "Introducing Trinket" to learn more, and download the software.

The hardware couldn't be simpler: plug your trinket into a breadboard, and connect the (+) lead of your piezo to digital pin 1, labelled #1 on the trinket board. Connect the other piezo lead to ground (labelled GND on the trinket) through a 200 to 300 ohm resistor. That's it.

3) THE TONE LIBRARY

There are many, many libraries already written for the Arduino, saving you the time and effort of coding these yourself. But the libraries are written for Arduino hardware in mind, which uses AVR ATmega-series microcontrollers. Over time, chips have been added and the software has evolved, but not all AVR microcontrollers are supported. The Trinket uses an ATtiny85 microcontroller: powerful for its size, but not 100% Arduino compatible.

The Arduino Tone Library is an example of code that isn't currently Trinket compatible. The original library, written by Brett Hagman, is here. Check it out! A lot of the code is necessary to support various controllers (and multiple counters in those controllers, and multiple output pins for flexibility). We can create a much smaller library by using a single ATtiny85 counter and a single, dedicated output pin.

The tone library uses a hardware counter to create the audio output. A counter is a microcontroller component that knows how to count without any software intervention. It is controlled by reading and writing various memory registers on the microcontroller. The Trinket's microcontroller contains two 8-bit counters, called timer0 and timer1. Of these, timer1 gives us the most flexibility for making audio.

4) COUNTER BASICS

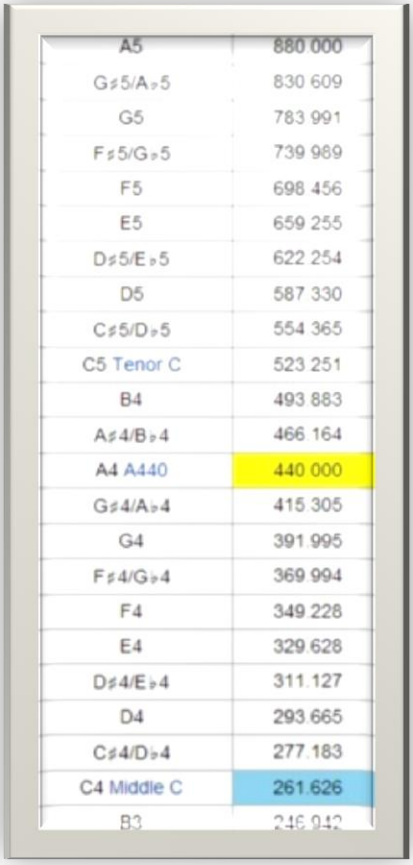
One of the counter modes is called CTC, or "Clear on Timer Compare Match". The counter increments its register with each input pulse, and then resets to zero whenever the counter value matches a user-determined level. For example, if the compare register value is 250, the counter increments from 0 to 250 and then resets itself to 0. If the input clock is 8 MHz, the counting cycle repeats at a frequency of 8/250 MHz or 32 kHz. We can program specific events to occur during each reset, including software interrupts and output pin state changes.

We can use the CTC mode to toggle an output pin. Toggling the pin at audio frequencies will cause the attached piezo element to vibrate and create the audible tones. In the example above, 32 kHz is above the audible range. We either need to count *higher* (not possible with an 8-bit counter) or count *slower*.

Enter the 'prescalar'. The prescalar sits in front of the counter, and divides the microcontroller clock by specific values. For example, we can program our prescalar to divide the clock by 64. Using our example above, the 8 MHz clock divided by 64 = 125 kHz. If we set our output compare register to 250, the counter now resets 125,000/250 = 500 times a second. An output pin toggling at this rate will create a square wave at half this frequency, or 500/2 = 250 Hz. For our 8 MHz Trinket, frequency output for any combination of prescalar (ps) and output compare register (ocr) values is: freq(Hz) = 8 MHz/ps/ocr/2.

Now let's consider two fundamental properties of a musical note: pitch and duration.

5) PITCH



A5	880.000
G♯5/A♭5	830.609
G5	783.991
F♯5/G♭5	739.989
F5	698.456
E5	659.255
D♯5/E♭5	622.254
D5	587.330
C♯5/D♭5	554.365
C5 Tenor C	523.251
B4	493.883
A♯4/B♭4	466.164
A4 A440	440.000
G♯4/A♭4	415.305
G4	391.995
F♯4/G♭4	369.994
F4	349.228
E4	329.628
D♯4/E♭4	311.127
D4	293.665
C♯4/D♭4	277.183
C4 Middle C	261.626
B3	246.942

We want more than just an audible frequency; we want to create pitches that map to the common 12-note chromatic musical scale. The 88-key piano gives us a reasonable range, and the frequency for each key is well known. The piano frequencies roughly span 30 to 4000 Hz in 7 octaves. The frequency ratio between any two adjacent keys is the twelfth root of 2 (1.05946). Doubling a frequency raises the pitch by 12 notes, or one octave.

Using frequency numbers to specify musical pitches can be cumbersome, so I use scientific pitch notation to specify each musical note. Each note is specified by its letter, followed by an octave number. For example, the lowest C on the piano keyboard is C0. Middle C is C4. Tuning A, at 440 Hz, is A4. A list of defines is all you need:

```
#define A4     440
#define AS4    466
#define B4     494
```

Now think about our prescalar/counter again. How can we set our prescalar and counter to generate these exact frequencies? We can't, but we can get close. Frequencies within a few Hz will sound OK on our low-fidelity piezo. The tone library runs through each available prescalar, and checks to see if the corresponding compare value is in our 8-bit range (255 or less).

The prescalar hardware in timer1 of the ATtiny85 controller is unique: it is more flexible than most Atmel AVR prescalars, giving us 15 different binary divisors from 1 through 16384. The flexibility of this particular prescalar module let us use a single while-loop to find a suitable value. Here is the code:

```
// scan through prescalars to find the best fit
 uint32_t ocr = F_CPU/frequency/2;
 uint8_t  prescalar = 1;
 while (ocr>255)
 {
   prescalar++;
   ocr /= 2;
 }
```

Let's see what happens for A4=440 Hz. F_CPU is the microcontroller clock frequency of 8 MHz = 8,000,000. As a first attempt, before the while loop starts, the ocr = 8 MHz/440/2 = 9090. This number is greater than 255, so the loop begins. For loop iteration, the prescalar is increased and the ocr is halved. After 6 loop iterations, the prescalar value is 7 and the ocr is 142. The loop terminates with these values since the ocr is <= 255. The prescalar divisor is 2^(n-1) = 2^6 = 64.

Now check to see if the resulting prescalar and OCR will give an acceptable output frequency. Using our frequency formula above, freq(Hz) = 8 Mhz/64/142/2 = 440.1 Hz.

Setting the counter's prescalar and output compare registers is easy: you just have to know their names:

```
TCCR1 = 0x90 | prescalar;
OCR1C = ocr-1;
```

OCR1C is the compare register for timer1 in CTC mode. The TCCR1 (timer/counter1 control register) contains additional bits that control the counter mode and result of a match. The table below shows each named bit in the register. Value 0x90 sets bits 7 & 4, which turn on CTC mode and set the output pin OCR1 to toggle on a match, respectively. The lower four clock select "CS" bits contain the prescalar value:

TCCR1 – Timer/Counter1 Control Register

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
CTC1	PWM1A	COM1A1	COM1A0	CS13	CS12	CS11	CS10
1	0	0	1	ps	ps	ps	ps

For more information about this register, see the ATtiny85 datasheet at Atmel.com. Output OCR1 corresponds to bit one of port B, PB1, which is labelled as pin #1 on the Trinket PCB.

The few lines of code above are all we need to generate musical pitch. Next, we need a method of controlling how long the note should play.

6) DURATION

Musical note duration is generally described as fractions of the "whole note". The duration of a quarter note is ¼ of a whole note, a sixteenth note is 1/16 as long as a whole note, and so on. But let's think like engineers for a moment, and consider duration in seconds instead. If I wanted my A4 = 440 Hz note to last one second, it requires 440 cycles. During this one second, the counter toggles the output line 880 times. For a 0.1 second note, the output line toggles 88 times. More generally, for any given note duration and pitch, the output line must toggle (2*pitch*duration) times. If we specify the duration in milliseconds, the toggle count = 2*pitch*duration/1000. Here is the code:

```
  // Calculate the toggle count
toggle_count = frequency * duration / 500;
```

By counting the number of times the output line toggles, we can measure and control the sound duration. So far, the counter toggles the output line, but doesn't tell us when it is toggling. To get that notification we'll configure the counter to generate a software interrupt. The timer interrupt mask register controls these interrupts:

TIMSK – Timer/Counter Interrupt Mask Register

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
-	OCIE1A	OCIE1B	OCIE0A	OCIE0B	TOIE1	TOIE0	-

To enable the interrupt, set the Timer/Counter1 Output Compare Interrupt Enable (OCIE1A) to logic 1. You can use bit-manipulation logic to enable this bit, or use the built-in macro "bitWrite" which is easier to read:

```
// enable timer1 output compare interrupt
bitWrite(TIMSK, OCIE1A, 1);
```

We need an interrupt service routine that counts the toggles, and stops the output whenever the specified time has passed. Interrupt service routines have a special format. The code looks like this:

```
ISR(TIMER1_COMPA_vect)
{
  if (toggle_count != 0)
    toggle_count--;
  else
    TCCR1 = 0x90;                  // stops the counter
}
```

This routine counts toggles by decrementing the count on each toggle. When the count gets to zero, the note duration is complete and the output should stop. There are several ways of stopping output. For example, you could redefine the output pin as an input. Or you could reconfigure the counter to not toggle. I chose to zero out the prescalar bits in TCCR1, which disables the input clock. You can also use this ISR for other purposes, such as toggling additional output pins.

7) FINAL BITS

Amazingly, once we code for musical pitch & duration, there isn't much left to do. The main routine reads notes one at a time from a large array, and then calls a routine to play them. Normally you would want to read two values at a time: the first for pitch and the second for duration. But I am going to cheat a bit by picking a piece that uses the same note duration throughout: the "Prelude in C" from the Well-Tempered Clavier, Book One, by J. S. Bach. There are 544 sixteenth notes, followed by an ending whole note:

```
void PlayBach()
{
  int len = 150;
  int dly = 190;
  for (int count=0; count<noteCount; count++)
  {
    TrinketPlay( pgm_read_word( &noteArray[count] ), len);
    delay(dly);
  }
  TonePlay(C4,2000);  // final note
}
```

I fixed the note length at 150 milliseconds, followed by a 40 millisecond pause between notes. You can adjust the note length for speed. The pause length controls the musical articulation.

If your piece is longer than a hundred notes, you may run out of SRAM memory where variables are usually stored. The Arduino IDE does not warn you, and your code will start doing strange things. To avoid the memory issue, put your note array in program memory instead of SRAM. Include pgmspace.h at the beginning of your code, and add the word PROGMEM to the start of your array declaration. The function pgm_read_word is used to read 16-bit variables from program memory.

```
#include <avr/pgmspace.h>
PROGMEM prog_uint16_t  noteArray[] =
{
  C4, E4, G4, C5, E5, G4,…
}
```

A YouTube video of my switched-on-Bach Trinket is here. Consider stripping out the music and note/duration definitions, and putting the other bits into a proper library. I put everything into a single file to keep things simple. Enjoy!

8) SOURCE CODE:

```
//
//  --------- TTone, a "Trinket" tone generator --------------
//
//  Author:   Bruce E. Hall <bhall66@gmail.com>
//  Date:     29 Dec 2013
//  Hardware: Trinket, using AVR ATtiny85
//  Software: Arduino 1.0.5
//  Size:     2352 bytes
//
//  This small application uses Timer1 on the ATtiny85
//  to implement a tone generator & play some music.
//
//  Connect a piezo buzzer between PB1 (#1) and ground.
//
//  The software is loosely based on the Arduino Tone Library,
//  but has been simplified for use in this small application.
//
//  The tune is "Prelude in C", from the Well-Tempered Clavier,
//  Book One, by J. S. Bach
//


/*************************************************
* Note Pitch Constants
*************************************************/

#define B0  31
#define C1  33
#define CS1 35
#define D1  37
#define DS1 39
#define E1  41
#define F1  44
#define FS1 46
```

```
#define G1  49
#define GS1 52
#define A1  55
#define AS1 58
#define B1  62
#define C2  65
#define CS2 69
#define D2  73
#define DS2 78
#define E2  82
#define F2  87
#define FS2 93
#define G2  98
#define GS2 104
#define A2  110
#define AS2 117
#define B2  123
#define C3  131
#define CS3 139
#define D3  147
#define DS3 156
#define E3  165
#define F3  175
#define FS3 185
#define G3  196
#define GS3 208
#define A3  220
#define AS3 233
#define B3  247
#define C4  262
#define CS4 277
#define D4  294
#define DS4 311
#define E4  330
#define F4  349
#define FS4 370
#define G4  392
#define GS4 415
#define A4  440
#define AS4 466
#define B4  494
#define C5  523
#define CS5 554
#define D5  587
#define DS5 622
#define E5  659
#define F5  698
#define FS5 740
#define G5  784
#define GS5 831
#define A5  880
#define AS5 932
#define B5  988
#define C6  1047
#define CS6 1109
#define D6  1175
#define DS6 1245
#define E6  1319
#define F6  1397
#define FS6 1480
#define G6  1568
#define GS6 1661
#define A6  1760
#define AS6 1865
#define B6  1976
#define C7  2093
#define CS7 2217
#define D7  2349
#define DS7 2489
#define E7  2637
#define F7  2794
```

```
#define FS7 2960
#define G7  3136
#define GS7 3322
#define A7  3520
#define AS7 3729
#define B7  3951
#define C8  4186
#define CS8 4435
#define D8  4699
#define DS8 4978


/*************************************************
* Note Duration Constants  (not used in this app)
*************************************************/

#define FN 1    // 64th note
#define TN 2    // 32nd note
#define DTN 3   // dotted 32nd note
#define SN 4    // 16th note
#define DSN 6   // dotted 16th note
#define EN 8    // 8th note
#define DEN 12  // dotted 8th note
#define QN 16   // quarter note
#define DQN 24  // dotted quarter note
#define HN 32   // half note
#define DHN 48  // dotted half note
#define WN 64   // whole note
#define DWN 96  // dotted whole note


/*************************************************
* Music Content is stored in the following array.
* PROGMEM is needed because of the large data size.
* "Prelude in C" Well-Tempered Clavier, J.S.Bach
*************************************************/


#define NOTECOUNT 544             // length of array

#include <avr/pgmspace.h>         // needed for PROGMEM

PROGMEM prog_uint16_t  noteArray[] =
{
  C4, E4, G4, C5, E5, G4, C5, E5, //measures 1-2, notes 1-32
  C4, E4, G4, C5, E5, G4, C5, E5,
  C4, D4, A4, D5, F5, A4, D5, F5,
  C4, D4, A4, D5, F5, A4, D5, F5,

  B3, D4, G4, D5, F5, G4, D5, F5, //measures 3-4, notes 33-64
  B3, D4, G4, D5, F5, G4, D5, F5,
  C4, E4, G4, C5, E5, G4, C5, E5,
  C4, E4, G4, C5, E5, G4, C5, E5,

  C4, E4, A4, E5, A5, A4, E5, A5, //measures 5-6, notes 65-96
  C4, E4, A4, E5, A5, A4, E5, A5,
  C4, D4, FS4,A4, D5, FS4,A4, D5,
  C4, D4, FS4,A4, D5, FS4,A4, D5,

  B3, D4, G4, D5, G5, G4, D5, G5, //measures 7-8, notes 97-128
  B3, D4, G4, D5, G5, G4, D5, G5,
  B3, C4, E4, G4, C5, E4, G4, C5,
  B3, C4, E4, G4, C5, E4, G4, C5,

  A3, C4, E4, G4, C5, E4, G4, C5, //measures 9-10, notes 129-160
  A3, C4, E4, G4, C5, E4, G4, C5,
  D3, A3, D4, FS4,C5, D4, FS4,C5,
  D3, A3, D4, FS4,C5, D4, FS4,C5,

  G3, B3, D4, G4, B4, D4, G4, B4, //measures 11-12, notes 161-192
  G3, B3, D4, G4, B4, D4, G4, B4,
```

```
  G3, AS3,E4, G4, CS5,E4, G4, CS5,
  G3, AS3,E4, G4, CS5,E4, G4, CS5,

  F3, A3, D4, A4, D5, D4, A4, D5, //measures 13-14, notes 193-224
  F3, A3, D4, A4, D5, D4, A4, D5,
  F3, GS3,D4, F4, B4, D4, F4, B4,
  F3, GS3,D4, F4, B4, D4, F4, B4,

  E3, G3, C4, G4, C5, C4, G4, C5, //measures 15-16, notes 225-256
  E3, G3, C4, G4, C5, C4, G4, C5,
  E3, F3, A3, C4, F4, A3, C4, F4,
  E3, F3, A3, C4, F4, A3, C4, F4,

  D3, F3, A3, C4, F4, A3, C4, F4, //measures 17-18, notes 257-288
  D3, F3, A3, C4, F4, A3, C4, F4,
  G2, D3, G3, B3, F4, G3, B3, F4,
  G2, D3, G3, B3, F4, G3, B3, F4,

  C3, E3, G3, C4, E4, G3, C4, E4, //measures 19-20, notes 289-320
  C3, E3, G3, C4, E4, G3, C4, E4,
  C3, G3, AS3,C4, E4, AS3,C4, E4,
  C3, G3, AS3,C4, E4, AS3,C4, E4,

  F2, F3, A3, C4, E4, A3, C4, E4, //measures 21-22, notes 321-352
  F2, F3, A3, C4, E4, A3, C4, E4,
  FS2,C3, A3, C4, DS4,A3, C4, DS4,
  FS2,C3, A3, C4, DS4,A3, C4, DS4,

  GS2,F3, B3, C4, D4, B3, C4, D4, //measures 23-24, notes 353-384
  GS2,F3, B3, C4, D4, B3, C4, D4,
  G2, F3, G3, B3, D4, G3, B3, D4,
  G2, F3, G3, B3, D4, G3, B3, D4,

  G2, E3, G3, C4, E4, G3, C4, E4, //meaures 25-26, notes 385-416
  G2, E3, G3, C4, E4, G3, C4, E4,
  G2, D3, G3, C4, F4, G3, C4, F4,
  G2, D3, G3, C4, F4, G3, C4, F4,

  G2, D3, G3, B3, F4, G3, B3, F4, //measures 27-28, notes 417-448
  G2, D3, G3, B3, F4, G3, B3, F4,
  G2, DS3,A3, C4, FS4,A3, C4, FS4,
  G2, DS3,A3, C4, FS4,A3, C4, FS4,

  G2, E3, G3, C4, G4, G3, C4, G4, //measures 29-30, notes 449-480
  G2, E3, G3, C4, G4, G3, C4, G4,
  G2, F3, G3, C4, F4, G3, C4, F4,
  G2, F3, G3, C4, F4, G3, C4, F4,

  G2, F3, G3, B3, F4, G3, B3, F4, //measures 31-32, notes 481-512
  G2, F3, G3, B3, F4, G3, B3, F4,
  C2, C3, G3, AS3,E4, G3, AS3,E4,
  C2, C3, G3, AS3,E4, G3, AS3,E4,

  C2, C3, F3, A3, C4, F4, C4, A3, //measures 33-34, notes 513-544
  C4, A3, F3, A3, F3, D3, F3, D3,
  C2, B2, G4, B4, D5, F5, D5, B4,
  D5, B4, G4, B4, D4, F4, E4, D4,
};


volatile uint32_t toggle_count;


// TrinketTone:
// Generate a square wave on a given frequency & duration
// Call with frequency (in hertz) and duration (in milliseconds).
// Uses Timer1 in CTC mode.  Assumes PB1 already in OUPUT mode.
// Generated tone is non-blocking, so routine immediately
// returns while tone is playing.

void TrinketTone(uint16_t frequency, uint32_t duration)
```

```
{
    // scan through prescalars to find the best fit
     uint32_t ocr = F_CPU/frequency/2;
     uint8_t  prescalarBits = 1;
     while (ocr>255)
     {
       prescalarBits++;
       ocr /= 2;
     }

    // CTC mode; toggle OC1A pin; set prescalar
    TCCR1 = 0x90 | prescalarBits;

    // Calculate note duration in terms of toggle count
    // Duration will be tracked by timer1 ISR
    toggle_count = frequency * duration / 500;

    OCR1C = ocr-1;                  // Set the OCR
    bitWrite(TIMSK, OCIE1A, 1);     // enable interrupt
}


// Timer1 Interrupt Service Routine:
// Keeps track of note duration via toggle counter
// When correct time has elapsed, counter is disabled

ISR(TIMER1_COMPA_vect)
{
  if (toggle_count != 0)            // done yet?
    toggle_count--;                 // no, keep counting
  else                              // yes,
    TCCR1 = 0x90;                   // stop the counter
}


// PlayBach:
// Plays "Prelude in C", which is held in noteArray
// Uses PROGMEM to store array, due to large size

void PlayBach()
{
  int len = 150;                    // modify for speed
  int dly = 190;                    // modify for articulation
  for (int count=0; count<NOTECOUNT; count++)
  {
    TrinketTone( pgm_read_word( &noteArray[count] ), len);
    delay(dly);
    if ((count>512) and (count<540)) // slow down (rit.) at end.
    {
      len += 3;
      dly += 5;
    }
  }
  TrinketTone(C4,1500);             // final note
  delay(1500);
}

void setup()
{
  pinMode(1, OUTPUT);               // enable OUTPUT (PB1, #1)
  PlayBach();                       // Music!
  pinMode(1, INPUT);                // disable OUTPUT
}


void loop()
{
    // once is enough.  Really.
}
```