Sound & Light Machine
Firmware Documentation
For Sound & Light Machine article in MAKE Magazine #10

By Mitch Altman

Firmware is the computer program that controls a microcontroller. [*Why not call it software? Firmware is somewhere between software and hardware – it is a software that directly controls hardware*]. In this project we converted the MiniPOV into a Sound & Light Machine (SLM) simply by changing the firmware, along with some minor hardware changes.

This document is intended to teach you enough about the SLM firmware to be able to create your own brainwave sequences. The next section ("Overview of the SLM firmware") is all you really need to read. But if you are interested in learning more about how the firmware works, I have also given plenty of extra detail.

Firmware is actually a bunch of lines of text in a file, written with a text editor. Collectively the lines of text tell the microcontroller exactly what to do. To program the microcontroller with the firmware it is convenient to use a special script file called a "makefile." (For this project, the makefile runs a C compiler, an assembler, a linker, an object converter, and programming software -- but you don't really need to understand any of this since the makefile does everything for you.)

Overview of the SLM firmware

The microcontroller for the SLM comprises most of the *hard*ware for the SLM, and is used primarily as a timing device to output square-waves that pulse LEDs in front of each eye and speakers for each ear. The SLM *firm*ware tells the microcontroller exactly how to create a sequence of these square-waves for blinking LEDs and pulsing speakers at brainwave frequencies. The sequence of square-waves mimics the sequence of a brain's frequencies as it goes into desired states of consciousness. The SLM firmware that I wrote generates a sequence that starts from a state of being awake, goes into deep meditation, hangs out there awhile, and then into being awake again (feeling wonderful).

The SLM's brainwave sequence is determined by a table called the "Brainwave Table" at the beginning of the SLM firmware. The Brainwave Table contains several "Brainwave Elements". Each Brainwave Element consists of a "Brainwave Type" (Beta, Alpha, Theta, or Delta) and a "Brainwave Duration" (which is how long to play this Brainwave Type, before doing the next Brainwave Element in the table). To change the SLM's brainwave sequence, simply change the Brainwave Table in a text editor, save the file, and run the makefile (as described in the article).

The Brainwave Table on page 8 is an easy-to-read representation of the actual table in the firmware that I created for the meditation sequence. The actual table in the firmware is shown on pages 12 and 13, called "brainwaveTab". You can see how it correlates to the more readable one on page 8. 'b' is for Beta, 'a' is for Alpha, 't' is for Theta, and 'd' is for delta. The last element must have '0' for its brainwave Type (to tell the firmware that it reached the END of the table). The times to play each brainwave (the Durations) are given in units of 1/10 milliseconds – divide the desired durations by 10,000 to get these values in brainwaveTab.

CAUTION: Please do not use the SLM if you are, or think you may be prone to seizures. Also, please be aware that certain conditions, such as ADHD, can worsen by entraining to lower brainwave frequencies. If you experience any problems, please discontinue using this SLM.

The above is really all you need to know to create your own brainwave sequences.

The rest of this document gives more in-depth explanation of how the SLM firmware does its thing.

Some words about the microcontroller

The microcontroller used in this project is the ATTINY2313, made by Atmel. It is part of their AVR family of microcontrollers. The datasheet for this chip can be downloaded from Atmel's website: http://www.atmel.com/dyn/resources/prod_documents/doc2543.pdf

Although many different microcontrollers are well suited for performing the tasks necessary for a Sound & Light Machine, the Atmel AVR family of chips has many advantages. One of them is that there is no need to buy a programmer for these chips, since it has a built in programmer – all you need is a computer with a serial port (or a USB connector and an inexpensive USB-to-serial converter) and some software. Another advantage is that all of the software necessary for programming the chip is free and open source. A free C-language compiler is available, so no one needs to learn the AVR assembly language [*Assembly language is the low-level language that microcontrollers understand and that all firmware used to be written in. Each microcontroller family has its own firmware language. When you compile a C program a compiler creates assembly language that the micro runs. Well, that's not quite accurate – actually, assembly language is for (very geeky) humans, whereas the micro runs on machine language, which is the binary numbers stored in program memory that the assembly language represents. There is another piece of software called an assembler that converts assembly language to machine language*]. Another really wonderful advantage is that there is a user forum called AVRfreaks.org that, 24/7, is full of geeks that love answering questions you may have about using the AVR family of micros. Being new to AVR chips, I made use of the user forum when I came up against a problem I needed help with. At 11pm I posted my question, and by 11:30pm someone pointed me to info where I could solve my problem. Sweet!

Most software for designing hardware and creating firmware is made for PCs running Windows. Another nice aspect of using AVR chips is that all of the software needed is available for Windows, Mac OS, and Linux.

Brainwave Frequencies the SLM firmware will generate

Brains generate brainwaves across a frequency spectrum. The spectrum is divided into four bands: Beta, Alpha, Theta, Delta. [*More recently a fifth brainwave band has been added: Gamma, but we are sticking with the original four for this project, since they are more well understood, and we can accomplish a lot with just these four.*] For the SLM I have chosen one frequency for each Brainwave Type. See the table on the next page:

Type	**Band**	**Frequency used in SLM**	**Associated States**
Delta waves	½Hz to 4Hz	2.2Hz	deep unconscious, intuition and insight
Theta waves	4Hz to 8Hz	6.0Hz	subconscious, creativity, deep relaxation
Alpha waves	8Hz to 13Hz	11.1Hz	spacey and dreamy, receptive and passive
Beta waves	13Hz to 30Hz	14.4Hz	conscious thought, external focus

Generating Square-Waves with the microcontroller

LED square-wave timing
The square-waves for pulsing the LEDs for each eye are generated by a timing loop in the firmware. Within this timing loop the firmware manually toggles microcontroller output pins. [*"Toggling" means: if the output pin was High, clear it to Low; and if the pin was Low, set it to High.*] Each LED is connected to an output pin on the microcontroller. The firmware toggles these output pins together, so that the LEDs connected to these output pins will pulse exactly the same.

As an example of how the LED timing works, let's say that we wish to blink the LEDs for the eyes at the Beta frequency of 14.4Hz. This means that the LEDs at each eye should blink High and Low 14.4 times per second. This means that the LED should toggle High and Low every 0.0347 seconds (which is 34.7 milliseconds, or 34.7ms). Here's the math:

The Duration of the LED blinking High and then Low (one cycle, or the "period"):
1 / 14.4Hz = 0.0694 sec = 69.4ms
The duration for ½ the period (this is the duration of the High and the duration of the Low):
69.4ms / 2 = 34.7 ms

Since the microcontroller's datasheet tells us how long it takes for the microcontroller to perform each instruction, we can write a loop of code that does nothing but delay for this length of time. Then, to toggle the LED at the Beta frequency, we turn the LED on (by setting the output pin of the microcontroller to High), delay for 34.7ms, and then turn the LED off (by clearing the output pin to Low), delay for 34.7ms, and repeat this process.

Speaker square-wave timing
We could do a similar thing for the audio, but the timing would get complex, since the brainwave audio should be binaural beats (as explained in the SLM article), rather than simply pulsing at brainwave frequencies. Fortunately, the microcontroller has some built-in functionality that greatly simplifies our task of creating the binaural beat frequencies at the speakers.

The microcontroller used in the SLM project has two hardware timers built into it. The SLM firmware makes use of these two timers to create the binaural beats. One timer, called Timer 0 (or T0), is set up to generate a Base Frequency of about 400Hz that is fed into one ear's speaker. The other timer, called Timer 1 (or T1), is used to generate Offset Frequencies that are fed into the other ear's speaker. When the Offset Frequency in one ear is heard together with the 400Hz in the other ear, the listener hears a beat frequency of the desired brainwave frequency. To get a beat frequency of 14.4Hz, T1 is configured to output about 414.4Hz.

Once the firmware starts T0 and T1, they will generate square-waves on their own, without firmware intervention, until the firmware tells the timers to stop.

The Microcontroller's Hardware Timers (details)

Here are the nitty-gritty details of how the firmware sets up the microcontroller's two hardware timers to output binaural beats.

Both of the timers in the microcontroller can be set up in various modes, according to the specifications in the microcontroller's datasheet. To create square-waves for the sound that feeds into the headphones, the firmware sets up the timers in a mode that toggles each timer's output pin for every time the timer times out (explained more, below). The datasheet calls this CTC mode (which they say stands for Clear Timer on Compare Match). In this mode, there is a Compare Register where the timing value is stored. The Timer counts upward from 0, one count for each clock (more on the clock, below). When the Timer reaches the value in the Compare Register (which is called "timeout", the output pin for the Timer is automatically toggled, and the timer starts counting upward from 0 again. The clock frequency is determined by the clock of the microcontroller and the Prescaler. In this project, we use the microcontroller's internal clock oscillator, which is set up to run at 8.0MHz. This frequency can be divided down by a Prescaler, which can divide by powers of 2: 1, 2, 4, 8, etc., up to 256. The formula for determining the output square-wave frequency, Fo, for the timer's output pin for this mode (as given in the datasheet) is:

$$Fo = Fclk / (2 * P * (1+C))$$

where

Fclk = 8,000,000Hz
P = Prescale value
C = Compare Register value
Fo = output square-wave frequency

Each timer (T0 and T1) can be set up independently. Each has its own output pin.

As an example, let's see how to use the two timers to generate Beta binaural beats. To generate the Base Frequency of about 400Hz, T0 is used with T0 Prescale = 256 and T0 Compare Register = 38, resulting in Fo = 400.641Hz (this is the closest we can get to 400Hz). The Offset Frequency for binaural beats is generated in T1, with T1 Prescale = 1, and T1 Compare Register = the appropriate value to generate the Brainwave Type. To get a binaural beat of about 14.4Hz (the Beta frequency), we need to set T1 to output 14.4Hz higher than the frequency of T0, which comes to 415.041Hz. Using the above formula to get as close as we can to 415.041Hz, we set the T1 Compare Register to 9637, giving an output frequency of 415.024Hz. The perceived beat frequency will be:

$$415.024Hz - 400.641Hz = 14.383Hz$$

which is pretty close to 14.4Hz.

Given a Base Frequency of 400.641Hz (generate on T0), below is a table which shows the value of the T1 Compare Register for creating an Offset Frequency as close as we can get for each of the four Brainwave Types ("OCR1A" is how the datasheet abbreviates the T1 Compare Register). See the table on the next page:

Type	**OCR1A**	**Offset Frequency Fo**	**Difference from 400.641Hz Base Frequency**
Delta waves	9928	402.860Hz	2.219Hz
Theta waves	9836	406.628Hz	5.987Hz
Alpha waves	9714	411.734Hz	11.093Hz
Beta waves	9637	415.024Hz	14.383Hz

Timing the LEDs and Speakers together

To play a given Brainwave Type for a certain Duration, the firmware starts T0 at the Base Frequency of 400.641Hz (which goes to one speaker), then starts T1 at the Offset Frequency for the given Brainwave Type (which goes to the other speaker), and the user hears the binaural beats for the given Brainwave Type. The firmware then keeps T0 and T1 going for the given Duration (the timing for the Duration is accomplished by a delay loop, which is described in the next paragraph). This takes care of the sound. What about the blinking lights?

Recall earlier that we would use a delay loop for creating the square-waves for blinking the LEDs. We can use this same delay loop for waiting the specified Duration for the given Brainwave Element. Cool, yes? While the hardware timers are playing the binaural beats, we just keep blinking the LEDs with the timing loop for the given Duration. Then we've accomplished creating binaural beats for a given Brainwave Type, along with blinking lights at the same Brainwave Type, played for a given Duration.

An example should make this clear. If we want to play Beta frequencies for 60 seconds, the firmware does the following:

```
Start T0 to toggle one ear at the Base Frequency of 400.641Hz
Start T1 to toggle the other ear at the Offset Frequency of 415.024Hz
Manually toggle the LEDs at each eye at 14.4Hz using the delay loop
Keep manually toggling the LEDs until all of the delays in the delay loop
    add up to 60 seconds (how to do this will be described later)
```

After the Duration of 60 seconds is over, the firmware could then stop manually toggling the LEDs to turn off the lights, and then turn off T0 and T1 to stop the sound at the speakers. Or, the firmware could start a different brainwave frequency playing at the LEDs and speakers for another Duration.

Generating A Brainwave Sequence with SLM firmware

The sequence of Brainwave Elements in the Brainwave Table is what determines the brainwave sequence generated by the SLM.

The firmware starts by playing the first Brainwave Element from the Brainwave Table. It does this (as described above) by starting T0 at the Base Frequency, starting T1 at the Offset Frequency for the Brainwave Type (given in the Brainwave Element), and allows the timers to continue for the Duration (given in the Brainwave Element) while blinking the LEDs at a rate for the given

Brainwave Type. Then (leaving T0 playing the Base Frequency) it grabs the next Brainwave Element from the Table, sets up T1 for the new specified Brainwave Type and plays it, along with blinking the LEDs at the new Brainwave Type's frequency, and plays it for the new specified Duration. Then (leaving T0 playing the Base Frequency) it grabs the next Brainwave Element from the Table. Etc.

This process continues till there are no more Brainwave Elements in the Brainwave Table. Then the firmware turns off both T0 and T1, stops manually toggling the output connected to the LEDs, and puts the microcontroller into a low-power mode.

(An easy-to-read representation of the Brainwave Table I created for the meditation sequence is shown on page 8.)

We can state the above more concisely by being a bit more geeky about it:

```
Initialize hardware and variables

start Timer 0 with Base Frequency into speaker #1

Do
    get next Brainwave Element (consisting of Type and Duration) from brainwaveTab
    start Timer1 with binaural beat Offset Frequency for Brainwave Type into speaker #2
    blink LEDs at brainwave frequency for the Brainwave Type
    and wait for the specified Duration
Until end of brainwaveTab

stop Timer 0 and Timer 1 to stop sound at both speakers
turn off LEDs
put microcontroller to sleep (low-power mode)
```

Let's get a little bit more geeky, and state things more in the way our specific microcontroller likes to see things (according to the datasheet). First of all, here is how the hardware is set up:

1 LED for each eye:
   LED #1: PB0 (pin 12)
   LED #2: PB1 (pin 13)

1 speaker for each ear:
   speaker #1: 0C0A / PB2 (pin 14)
   speaker #2: 0C1A / PB3 (pin 15)

And here is the algorithm for the firmware:

```
Main:
Initialize
  set Port B for outputs (DDRB = 0xFF)
  setup Timer 0 for CTC mode and for toggling OC0A (which is Timer 0's output pin)
  start Base-Frequency square wave = 400.641Hz on Timer 0
      (Prescale = 256,  OCR0A = 38) to output sound to OC0A (connected to speaker #1)
  setup Timer 1 for CTC mode and for toggling OC1A (which is Timer 1's output pin)
      (Prescale = 1, OCR1A not set yet)

MainLoop:
  get next brainwaveElement  (which consists of a brainwaveType and a brainwaveDuration)
  if brainwaveType = 0, then quit the MainLoop    (when brainwaveType = 0 we are done)
  otherwise, do the following:
    start T1 square wave = binaural beat offset for the brainwaveType
        (0CR1A=Offset Frequency for brainwaveType) to output sound to OC1A (connected to
        speaker #2)

    loop to blink LEDs at PB0 and PB1 at correct timing for the brainwaveType
    keep looping for brainwaveDuration
endofMainLoop

stop Timer 0 and Timer 1 to stop sound at both speakers
turn off both LEDs
put processor to sleep (low-power mode)
```

The actual firmware is given starting on page 11. It is described in more detail starting on page 9.

How to specify a brainwave sequence for this SLM

CAUTIONS: BLINKING LIGHTS SHOULD BE AVOIDED BY ANYONE PRONE TO SEIZURES. If you are sensitive to seizures, please disconnect the LEDs in this project (your brain will effectively entrain with the binaural beat audio alone). CERTAIN CONDITIONS, SUCH AS ADHD, CAN WORSEN WITH THETA AND DELTA ENTRAINMENT. Please discontinue using this SLM if you experience any problems.

Since we can only generate one brainwave frequency at a time in this SLM project, we need to create a brainwave sequence that tries to keep your brain active at various brainwave frequencies by going back and forth between two brainwave frequencies. For example, if we want to go from fully awake to somewhat dreamy, we would generate Beta for awhile, then Alpha for awhile, and go back and forth between Beta and Alpha, but reducing the length of Beta and increasing the length of Alpha with each iteration.

Another thing to keep in mind when creating your own brainwave sequence is to start by meeting your user where they are at and then take them where they want to go. For example, if you want to create a sequence to bring people to sleep, then start with being awake and conscious (Beta), then, after awhile, start adding some spaceyness (Alpha).

The brainwave sequence for meditation that I created uses the above techniques. The Brainwave Table for it is given on the next page.

Brainwave Table
(Meditation sequence)

Element number	Brainwave Type	Brainwave Duration (sec.)
1	Beta	60
2	Alpha	10
3	Beta	20
4	Alpha	15
5	Beta	15
6	Alpha	20
7	Beta	10
8	Alpha	30
9	Beta	5
10	Alpha	60
11	Theta	10
12	Alpha	30
13	Theta	20
14	Alpha	30
15	Theta	30
16	Alpha	15
17	Theta	60
18	Alpha	15
19	Beta	1
20	Alpha	15
21	Theta	60
22	Delta	1
23	Theta	10
24	Delta	1
25	Theta	10
26	Delta	1
27	Theta	30
28	Alpha	15
29	Beta	1
30	Alpha	15
31	Theta	30
32	Alpha	15
33	Beta	1
34	Alpha	20
35	Beta	5
36	Alpha	20
37	Beta	15
38	Alpha	15
39	Beta	20
40	Alpha	10
41	Beta	25
42	Alpha	5
43	Beta	60
44	END	**Total = 856 sec** (about 14¼ min)

Description of the actual SLM firmware

I'll start the description with the Brainwave Table (somewhat near the top of the firmware). It is called "brainwaveTab" in the firmware. The brainwaveTab contains several lines, each of which is a "brainwaveElement". Each brainwaveElement consists of a bwType ('b' for Beta, 'a' for Alpha, 't' for Theta, 'd' for Delta) and a bwDuration (which is expressed in an odd time – to get seconds, divide the number by 10,000). The last brainwaveElement must have '0' for its bwType to indicate the END of the brainwaveTab.

Now let's look at the bottom of the firmware to the "main" function. The main function starts by initializing the hardware registers of the microcontroller: no interrupts [*interrupts are a way for hardware to call a function when an event happens – we aren't using interrupts in the SLM firmware*], set all pins to outputs, and make sure that all of these output pins are Low. Then start T0 to output the Base Frequency using CTC mode, toggling output pin OC0A (connected to speaker #1). Then set up T1 to output the Offset Frequency (but we won't start T1 here – we start T1 in the do_brainwave_element function, as described in its paragraph, below). Then there is a loop that grabs each brainwaveElement in brainwaveTab, sending each brainwaveElement to the do_brainwave_element function to do the work. The loop finishes if it finds a brainwaveElement with a bwType of '0'. Once we've gone through the entire brainwaveTab, the firmware stops T0 and T1 (to stop the sound), enables sleep-mode (which will put the micro into a low-power mode), waits 1 second (to let things settle – this is way longer than necessary, but that won't hurt anything), turns off all output pins, makes all pins inputs (since this uses less battery power), and then puts the micro to sleep.

OK, now look back to the top of the firmware. The "include" statements at the beginning let the C compiler know where to look for some code that is pre-written, and comes with the compiler (these are known as library functions).

The "define" statements that come next define some constants used for the microcontroller's hardware timers.

Next comes the Brainwave Table, described earlier.

Next comes the delay loop function, "delay_one_tenth_ms". All it does is go around a loop a specified number of times, performing no useful task except to delay for the correct amount of time. It delays for a given number of 0.1ms (1/10 millisecond). So, for example, to delay 60 seconds, the input to this routine should be 600,000. One thing a little odd about this routine is that I had to tell it to do *some*thing; otherwise the C compiler would see that it didn't actually perform any useful task, and it would "optimize" the code away (this is because the C compiler we are using is an "optimizing compiler"). In the delay loop I told the microcontroller to toggle an output pin in Port D (which we don't use), which forces the compiler from optimizing the loop into non-existence. The function uses a constant called "DelayCount". I chose the value of DelayCount=87 by trial and error so that the function delays very close to increments of 1/10ms.

The next function is called "blink_LEDs". It has 3 inputs, and it does what the name implies. And it does it at a given timing, and for a given Duration (all given in units of 1/10ms). Rather than create a square-wave with equal High times and Low times, the function can make a square-wave with a specified High time (called "onTime") and a different Low time (called "offTime"). The function turns on the LED, waits onTime (by calling delay_one_tenth_ms), turns off the LED, and waits offTime (by calling delay_one_tenth_ms). It does this in a loop until the specified duration is reached. The number of times to go through the loop is calculated by dividing the specified duration by onTime+offTime. Let's look at an example to see how this works: if we want the LEDs to blink at Beta Frequency of 14.4Hz for 60 seconds, then duration = 600,000, onTime = 347, and offTime = 347. Each run through the loop takes 34.7ms+34.7ms=0.0694seconds, so we want to run through the loop for 600,000/(347+347)=864 times (truncated to an integer). The total time will be: 864 * 0.0694sec = 59.96sec, which is pretty close to the specified 60 seconds duration.

The next function is "do_brainwave_element". It has one input, telling it which brainwaveElement to play. (Keep in mind that in the C language, the first element of a table is 0, rather than 1.) This function grabs the specified brainwaveElement from the brainwaveTab, and gets the brainwaveElement's bwType and bwDuration. The function contains one section for each of the four Brainwave Types. Each section is very similar. For example, let's look at the section for Alpha (bwType='a'). It sets the output compare register for T1 (which is called "OCR1A") to 9714, to start T1 with an Offset Frequency of 411.734Hz to create a binaural beat for the user at 11.1Hz (which is Alpha waves). Then it calls the blink_LEDs function with the correct onTime and offTime and total Duration to make the LEDs blink at Alpha frequencies for the given length of time. Here's the math for the LED blink-rate:

onTime = 451   (0.0451seconds)
offTime = 450   (0.0450seconds)

So the LEDs blink on and off at this rate:

1 / (0.0451 + 0.0450)sec = 11.1Hz (which is Alpha waves)

Here is a table of values for each Brainwave Type:

Type	OCR1A	Offset Frequency Fo	Binaural Beat Frequency	onTime	offTime	LED blink Frequency
Delta waves	9928	402.860Hz	2.219Hz	225.3ms	225.3ms	2.219Hz
Theta waves	9836	406.628Hz	5.987Hz	83.5ms	83.5ms	5.988Hz
Alpha waves	9714	411.734Hz	11.093Hz	45.1ms	45.0ms	11.099Hz
Beta waves	9637	415.024Hz	14.383Hz	34.7ms	34.7ms	14.409Hz

You can see from the above table that for each brainwave type, do_brainwave_element is able to blink the LEDs at a rate that is very close to the binaural beat frequency.

The final function is the main function, which was described earlier.

And that is it. Except for one gotcha. While I was creating the firmware, to make it easier to debug I used a small brainwaveTab with only 5 brainwaveElements. Eventually I got the firmware to work great with this small brainwaveTab. But then when I created the full brainwaveTab, the firmware didn't work. What the?! After trying to figure out why the firmware would not work with the full-sized brainwaveTab, I finally formulated a question and posted it to the AVR user forum mentioned earlier. I got my answer within 30 minutes. It turns out that the C compiler tells the microcontroller to transfer the whole brainwaveTab structure to RAM when the firmware starts up. And that can not work because the microcontroller's RAM is only 128 bytes – and the brainwaveTab is way bigger than 128 bytes. The solution is to use some library functions (pgm_read_byte and pgm_read_word) and a macro (PROGMEM) that come with the C compiler. With these in place the firmware will not transfer brainwaveTab to RAM, but leaves it in program ROM – and the firmware works great! (This is described in the actual firmware, on page 12.)

The following pages show the complete firmware. You can download this firmware in its own file (with the makefile), called "slmFirmware.zip", from the makezine.com website where you downloaded this document.

```
/*
Sound & Light Machine
Firmware
for use with ATtiny2313
Make Magazine issue #10
Mitch Altman
19-Mar-07
*/

#include <avr/io.h>             // this contains all the IO port definitions
#include <avr/interrupt.h>      // definitions for interrupts
#include <avr/sleep.h>          // definitions for power-down modes
#include <avr/pgmspace.h>       // definitions or keeping constants in program memory

#define TIMER0_PRESCALE_1 1
#define TIMER0_PRESCALE_8 2
#define TIMER0_PRESCALE_64 3
#define TIMER0_PRESCALE_256 4
#define TIMER0_PRESCALE_1024 5
#define TIMER1_PRESCALE_1 1
#define TIMER1_PRESCALE_8 2
#define TIMER1_PRESCALE_64 3
#define TIMER1_PRESCALE_256 4
#define TIMER1_PRESCALE_1024 5


/*
The hardware for this project is very simple:
     ATtiny2313 has 20 pins:
       pin 1   connects to serial port programming circuitry
       pin 10  ground
       pin 12  PB0 - Left eye LED1
       pin 13  PB1 - Right eye LED1
       pin 14  OC0A - Left ear speaker (base-frequency)
       pin 15  OC1A - Right ear speaker (Offset Frequencies for binaural beats)
       pin 17  connects to serial port programming circuitry
       pin 18  connects to serial port programming circuitry
       pin 19  connects to serial port programming circuitry
       pin 20  +3v
    All other pins are unused
    This firmware requires that the clock frequency of the ATtiny
      is the default that it is shipped with:  8.0MHz
*/


/*
The C compiler creates code that will transfer all constants into RAM when the microcontroller
resets.  Since this firmware has a table (brainwaveTab) that is too large to transfer into RAM,
the C compiler needs to be told to keep it in program memory space.  This is accomplished by
the macro PROGMEM (this is used, below, in the definition for the brainwaveTab).  Since the
C compiler assumes that constants are in RAM, rather than in program memory, when accessing
the brainwaveTab, we need to use the pgm_read_byte() and pgm_read_dword() macros, and we need
to use the brainwveTab as an address, i.e., precede it with "&".  For example, to access
brainwaveTab[3].bwType, which is a byte, this is how to do it:
     pgm_read_byte( &brainwaveTab[3].bwType );
And to access brainwaveTab[3].bwDuration, which is a double-word, this is how to do it:
     pgm_read_dword( &brainwaveTab[3].bwDuration );
*/


// table of values for meditation
//   start with lots of Beta (awake / conscious)
//   add Alpha (dreamy / trancy to connect with subconscious Theta that'll be coming up)
//   reduce Beta (less conscious)
//   start adding Theta (more subconscious)
//   pulse in some Delta (creativity)
//   and then reverse the above to come up refreshed
struct brainwaveElement {
  char bwType;  // 'a' for Alpha, 'b' for Beta, 't' for Theta, or 'd' for Delta ('0' signifies last entry in table
  unsigned long int bwDuration;  // Duration of this Brainwave Type (divide by 10,000 to get seconds)
} const brainwaveTab[] PROGMEM = {
  { 'b', 600000 },
  { 'a', 100000 },
  { 'b', 200000 },
  { 'a', 150000 },
  { 'b', 150000 },
  { 'a', 200000 },
  { 'b', 100000 },
  { 'a', 300000 },
  { 'b',  50000 },
  { 'a', 600000 },
  { 't', 100000 },
  { 'a', 300000 },
  { 't', 200000 },
  { 'a', 300000 },
  { 't', 300000 },
  { 'a', 150000 },
  { 't', 600000 },
  { 'a', 150000 },
  { 'b',  10000 },
  { 'a', 150000 },
  { 't', 600000 },
  { 'd',  10000 },
  { 't', 100000 },
  { 'd',  10000 },
  { 't', 100000 },
  { 'd',  10000 },
  { 't', 300000 },
  { 'a', 150000 },
  { 'b',  10000 },
  { 'a', 150000 },
  { 't', 300000 },
  { 'a', 150000 },
  { 'b',  10000 },
  { 'a', 200000 },
  { 'b',  50000 },
  { 'a', 200000 },
  { 'b', 150000 },
  { 'a', 150000 },
  { 'b', 200000 },
  { 'a', 100000 },
  { 'b', 250000 },
  { 'a',  50000 },
  { 'b', 600000 },
  { '0',      0 }
};
```

```
// This function delays the specified number of 1/10 milliseconds
void delay_one_tenth_ms(unsigned long int ms) {
  unsigned long int timer;
  const unsigned long int DelayCount=87;  // this value was determined by trial and error

  while (ms != 0) {
    // Toggling PD0 is done here to force the compiler to do this loop, rather than optimize it away
    for (timer=0; timer <= DelayCount; timer++) {PIND |= 0b0000001;};
    ms--;
  }
}


// This function blinks the LEDs (connected to PB0, PB1 - for Left eye, Right eye, respectively)
//   at the rate determined by onTime and offTime
//   and keeps them blinking for the Duration specified (Duration given in 1/10 millisecs)
// This function also acts as a delay for the Duration specified
void blink_LEDs( unsigned long int duration, unsigned long int onTime, unsigned long int offTime) {
  for (int i=0; i<(duration/(onTime+offTime)); i++) {
    PORTB |= 0b00000011;          // turn on LEDs at PB0, PB1
    delay_one_tenth_ms(onTime);   //   for onTime
    PORTB &= 0b11111100;          // turn off LEDs at PB0, PB1
    delay_one_tenth_ms(offTime);  //   for offTime
  }
}
```

```
// This function starts the Offset Frequency audio in the Right ear through output OC1A  (using Timer 1)
//   to create a binaural beat (between Left and Right ears) for a Brainwave Element
//   (the base-frequency of 400.641Hz is already assumed to be playing in the Left ear before calling this function)
//   and blinks the LEDs at the same frequency for the Brainwave Element
//   and keeps it going for the Duration specified for the Brainwave Element
// The timing for the Right ear is done with 16-bit Timer 1 (set up for CTC Mode, toggling output on each compare)
//   Output frequency = Fclk / (2 * Prescale * (1 + OCR1A) ) = 8,000,000 / (2 * (1 + OCR1A) )
void do_brainwave_element(int index) {
    char brainChr = pgm_read_byte(&brainwaveTab[index].bwType);
    if (brainChr == 'b') {
         // PORTB &= 0b00001100;  // (for debugging purposes only -- commented out for SLM)
         // PORTB |= 0b10000000;
      // Beta
      // start Timer 1 with the correct Offset Frequency for a binaural beat for the Brainwave Type
      //   to Right ear speaker through output OC1A (PB3, pin 15)
      OCR1A = 9637;  // T1 generates 415.024Hz, for a binaural beat of 14.4Hz
      // delay for the time specified in the table while blinking the LEDs at the correct rate
      //   onTime = 34.7ms, offTime = 34.7ms --> 14.4Hz
      blink_LEDs( pgm_read_dword(&brainwaveTab[index].bwDuration), 347, 347 );
      return;   // Beta
    }

    else if (brainChr == 'a') {
         // PORTB &= 0b00001100;  // (for debugging purposes only -- commented out for SLM)
         // PORTB |= 0b01000000;
      // Alpha
      // start Timer 1 with the correct Offset Frequency for a binaural beat for the Brainwave Type
      //   to Right ear speaker through output OC1A (PB3, pin 15)
      OCR1A = 9714;  // T1 generates 411.734Hz, for a binaural beat of 11.1Hz
      // delay for the time specified in the table while blinking the LEDs at the correct rate
      //   onTime = 45.1ms, offTime = 45.0ms --> 11.1Hz
      blink_LEDs( pgm_read_dword(&brainwaveTab[index].bwDuration), 451, 450 );
      return;   // Alpha
    }
    else if (brainChr == 't') {
         // PORTB &= 0b00001100;  // (for debugging purposes only -- commented out for SLM)
         // PORTB |= 0b00100000;
      // Theta
      // start Timer 1 with the correct Offset Frequency for a binaural beat for the Brainwave Type
      //   to Right ear speaker through output OC1A (PB3, pin 15)
      OCR1A = 9836;  // T1 generates 406.628Hz, for a binaural beat of 6.0Hz
      // delay for the time specified in the table while blinking the LEDs at the correct rate
      //   onTime = 83.5ms, offTime = 83.5ms --> 6.0Hz
      blink_LEDs( pgm_read_dword(&brainwaveTab[index].bwDuration), 835, 835 );
      return;   // Theta
    }

    else if (brainChr == 'd') {
         // PORTB &= 0b00001100;  // (for debugging purposes only -- commented out for SLM)
         // PORTB |= 0b00010000;
      // Delta
      // start Timer 1 with the correct Offset Frequency for a binaural beat for the Brainwave Type
      //   to Right ear speaker through output OC1A (PB3, pin 15)
      OCR1A = 9928;  // T1 generates 402.860Hz, for a binaural beat of 2.2Hz
      // delay for the time specified in the table while blinking the LEDs at the correct rate
      //   onTime = 225.3ms, offTime = 225.3ms --> 2.2Hz
      blink_LEDs( pgm_read_dword(&brainwaveTab[index].bwDuration), 2253, 2253 );
      return;   // Delta
    }

    // this should never be executed, since we catch the end of table in the main loop
    else {
         // PORTB &= 0b00001100;  // (for debugging purposes only -- commented out for SLM)
         // PORTB |= 0b00000010;
      return;      // end of table
    }
}
```

```
int main(void) {

  TIMSK = 0x00;  // no Timer interrupts enabled
  DDRB = 0xFF;   // set all PortB pins as outputs
  PORTB = 0x00;  // all PORTB output pins Off

  // start up Base frequency = 400.641Hz on Left ear speaker through output OC0A (using Timer 0)
  //   8-bit Timer 0 OC0A (PB2, pin 14) is set up for CTC mode, toggling output on each compare
  //   Fclk = Clock = 8MHz
  //   Prescale = 256
  //   OCR0A = 38
  //   F = Fclk / (2 * Prescale * (1 + OCR0A) ) = 400.641Hz
  TCCR0A = 0b01000010;  // COM0A1:0=01 to toggle OC0A on Compare Match
                        // COM0B1:0=00 to disconnect OC0B
                        // bits 3:2 are unused
                        // WGM01:00=10 for CTC Mode (WGM02=0 in TCCR0B)
  TCCR0B = 0b00000100;  // FOC0A=0 (no force compare)
                        // F0C0B=0 (no force compare)
                        // bits 5:4 are unused
                        // WGM2=0 for CTC Mode (WGM01:00=10 in TCCR0A)
                        // CS02:00=100 for divide by 256 prescaler
  OCR0A = 38;  // to output 400.641Hz on OC0A (PB2, pin 14)

  // set up T1 to accept Offset Frequencies on Right ear speaker through OC1A (but don't actually start the Timer 1 here)
  //   16-bit Timer 1 OC1A (PB3, pin 15) is set up for CTC mode, toggling output on each compare
  //   Fclk = Clock = 8MHz
  //   Prescale = 1
  //   OCR0A = value for Beta, Alpha, Theta, or Delta (i.e., 9520, 9714, 9836, or 9928)
  //   F = Fclk / (2 * Prescale * (1 + OCR0A) )
  TCCR1A = 0b01000000;  // COM1A1:0=01 to toggle OC1A on Compare Match
                        // COM1B1:0=00 to disconnect OC1B
                        // bits 3:2 are unused
                        // WGM11:10=00 for CTC Mode (WGM13:12=01 in TCCR1B)
  TCCR1B = 0b00001001;  // ICNC1=0 (no Noise Canceller)
                        // ICES1=0 (don't care about Input Capture Edge)
                        // bit 5 is unused
                        // WGM13:12=01 for CTC Mode (WGM11:10=00 in TCCR1A)
                        // CS12:10=001 for divide by 1 prescaler
  TCCR1C = 0b00000000;  // FOC1A=0 (no Force Output Compare for Channel A)
                        // FOC1B=0 (no Force Output Compare for Channel B)
                        // bits 5:0 are unused
  // loop through entire Brainwave Table of Brainwave Elements
  //   each Brainwave Element consists of a Brainwave Type (Beta, Alpha, Theta, or Delta) and a Duration
  // Seeing the LEDs blink and hearing the binaural beats for the sequence of Brainwave Elements
  //   synchs up the user's brain to follow the sequence (hopefully it is a useful sequence)
  int j = 0;
  while (pgm_read_byte(&brainwaveTab[j].bwType) != '0') {  // '0' signifies end of table
    do_brainwave_element(j);
    j++;
  }

  // Shut down everything and put the CPU to sleep
  TCCR0B &= 0b11111000;  // CS02:CS00=000 to stop Timer0 (turn off audio in Right ear speaker)
  TCCR1B &= 0b11111000;  // CS12:CS10=000 to stop Timer1 (turn off audio in Left ear speaker)
  MCUCR |= 0b00100000;   // SE=1 (bit 5)
  MCUCR |= 0b00010000;   // SM1:0=01 to enable Power Down Sleep Mode (bits 6, 4)
  delay_one_tenth_ms(10000);  // wait 1 second
  PORTB = 0x00;          // turn off all PORTB outputs
  DDRB = 0x00;           // make PORTB all inputs
  sleep_cpu();           // put CPU into Power Down Sleep Mode
}
```