Morse Code Tutor -
from the ground up

Part 2: Simple Morse

Bruce E. Hall, W8BH

This is part 2 of a series about a $20 device that helps you learn Morse Code. It is inspired by Jack Purdum's "Morse Code Tutor", using a Blue Pill microcontroller board and the Arduino IDE.

Now that we have the Blue Pill working within the Arduino IDE and have successfully loaded the Blink sketch, it is time to do something more interesting. The monotonous blinking LED gets boring after a while. Let's make it do something more "ham-like". Let's make it do Morse!

Dits and Dahs.

It doesn't get much simpler than a dit in morse code. Let's create a routine to make the LED do a dit. We will define a constant DITPERIOD to equal the length of one dit, in milliseconds. Here is the code to send the character "S":

```
#define DITPERIOD   120                         // period of dit, in milliseconds
#define LED         PC13

void dit() {
  digitalWrite(LED,0);                          // turn on LED
  delay(DITPERIOD);                             // wait 1 dit period
  digitalWrite(LED,1);                          // turn off LED
  delay(DITPERIOD);                             // space between code elements
}

void setup() {
  pinMode(LED,OUTPUT);
  dit(); dit(); dit)                            // Send "S"
}
```

It looks similar to the blink sketch, doesn't it?

In Morse, each character is made up of elements: a dit is 1 element and a dah is three elements. In addition, the "intracharacter" space between each audible dit/dah is also one element. For example, the character 'S' dit-dit-dit is exactly 5 elements in length: 3 dits + 2 spaces.

Now let's write the code for dah(), and rearrange things to allow for spaces of different lengths. Now we have enough code to send anything we want:

```
void ditSpaces(int spaces) {                        // user specifies #dits to wait
  for (int i=0; i<spaces; i++)                      // count the dits...
    delay(DITPERIOD);                               // no action, just mark time
}

void dit() {
  digitalWrite(LED,0);                              // turn on LED
  ditSpaces(1);
  digitalWrite(LED,1);                              // turn off LED
  ditSpaces(1);                                     // space between code elements
}

void dah() {
  digitalWrite(LED,0);                              // turn on LED
  ditSpaces(3);                                     // length of dah = 3 dits
  digitalWrite(LED,1);                              // turn off LED
  ditSpaces(1);                                     // space between code elements
}

void setup() {
  pinMode(LED,OUTPUT);
  dit(); dah(); dit)                                // Send "R" as an example.
}
```

Try compiling this yourself. Can you make the LED flash an "R"? Notice that dah() is very similar to dit(), differing only in the length of time the LED stays lit. Make the code slower or faster by changing the DITPERIOD define.

Elements, word length, and Paris.

It is more convenient to specify the code speed in words per minute (WPM), rather than milliseconds-per-dit! To do this we need to know how many dits are in a word. But words are all different lengths, you say. To calculate Morse speed, the standard word is "PARIS " (including the space after the word). The spaces between characters is 3 elements long, and the space between words is 7 elements long:

Character	Intracharacter Elements	Extracharacter Elements
P (.--.)	11	3 (space after P)
A (.-)	5	3 (space after A)
R (.-.)	7	3 (space after R)
I (..)	3	3 (space after I)
S (...)	5	7 (space after S)

"Paris " has a total of 31 intra-character and 19 extra-character elements = 50 elements altogether. Since one minute = 60000 milliseconds and the number of elements in one minute is (WPM * 50 elements), we can rewrite the duration of one element (in milliseconds) as 60000/(50*wpm) = 1200/wpm. Change the definition of DITPERIOD to the following:

```
#define CODESPEED   13                              // speed in Words per Minute
#define DITPERIOD   1200/CODESPEED                  // period of dit, in milliseconds
```

A code table.

Now we can specify the speed, in words per minute, and send any character we want as a combination of dit() and dah(). Let's create a table for all of the Morse characters, so that we don't have to specify dit() dah() dah() dit() every time we want to send the letter 'p'. It is possible and perfectly fine to create a huge switch/case statement to do exactly that:

```
switch (character) {
  case 'A': dit(); dah(); break;
  case 'B': dah(); dit(); dit(); dit(); break;
  case 'C': dah(); dit(); dah(); dit(); break;
  // and so on
}
```

A more compact way is to create an array of binary numbers, one per character, with all of the character's elements encoded into the binary number. This method is not nearly as readable as the example above, but works equally well:

```
byte morse[] = {                                  // char encoded into an 8-bit byte
  0b00000101,          // A
  0b00011110,          // B
  0b00011010,          // C
  0b00001110,          // D
  0b00000011,          // E
  0b00011011,          // F
  0b00001100           // G
  //etc
};

void sendElements(int x) {                        // send string of bits as Morse
  while (x>1) {                                   // stop when value is 1 (or less)
    if (x & 1) dit();                             // right-most bit is 1, so dit
    else dah();                                   // right-most bit is 0, so dah
    x >>= 1;                                      // shift bits right
  }
  characterSpace();                               // add inter-character spacing
}
```

That code is more complicated. Let's study it.

The Morse table, abbreviated here, contains a representation for all of the letters, numbers, and punctuation used in Morse code. It is used by the routine sendElements(), which converts a table entry into the required dit()s and dah()s. Take the array entry for letter 'C' for example, which is binary 00011010. The routine reads the bits from RIGHT to LEFT, starting with bit 0, the right-most bit. This bit is 0, so a dah() is sent:

	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
'C'	0	0	0	1	1	0	1	**0**

Each bit is then shifted right so that the old bit1 is now in the bit0 position. This is 1, so a dit() is sent:

	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
Shift right #1	0	0	0	0	1	1	0	**1**

After the second shift, bit 0 contains a 0, so a dah() is sent:

	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
Shift right #2	0	0	0	0	0	1	1	**0**

After the third shift, bit 0 contains a 1, so a dit is sent:

	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
Shift right #3	0	0	0	0	0	0	1	**1**

On the fourth and final shift, all of bits have been read except the last bit, a stop bit, which is always 1:

	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
Shift right #4	0	0	0	0	0	0	0	**1**

Confusing? Yes, it feels like something only a programmer can love. This method is based on an add-on I did in 2010 for W8DIZ's DDS VFO project (see http://w8bh.net/avr/MemoryKeyer.pdf), which was based on earlier work by Jeff Otterson N1KDO in 1997 and David Robinson WW2R/G4FRE. If it makes no sense, feel free to create your table using the switch/case statement instead.

Other Morse characters.

The supplied code table covers most of the International Morse code characters. But what if you want to send a different code? For example, Japanese text may be sent using Wabun (Kana) code, where each character represents a Japanese kana symbol. The kana letter 'mi' is sent as "di-di-dah-di-dah". How would we send that code? Here are the steps:

1. Convert the rhythm into a series of 1 and 0's, where 1 is dit and 0 is dah	di-di-dah-di-dah = 11010
2. Add a sentinel '1' to the end	110101
3. Reverse the order	101011
4. Add the binary prefix '0b'	0b101011
5. Call sendElements to send it	sendElements(0b101011)

Sending characters and strings.

With a table in place, all we need to do is look up the character in the table and send it. Check out the sendCharacter routine():

```
void sendCharacter(char c) {                    // send single ASCII character in Morse
  if (c==32) wordSpace();                       // space between words
  else sendElements(morse[c-33]);               // send the character
```

```
        }
```

This routine accepts a character, looks it up in the table, and converts it to dits and dahs. So where do the 32 and 33 numbers come from? 32 is the ASCII code for a space, and ASCII 33 ‘!’ is the first entry in the morse code table. “C-33” converts the ASCII value of the character to an index for the table. Purists and those who love strongly-typed languages will cringe at mixing character and integer types, but it works nicely and isn’t hard to understand.

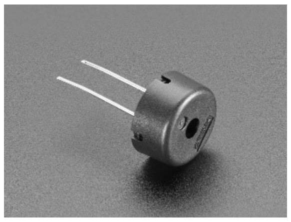
Now we have everything we need to send an entire string, which could be a word, sentence, paragraph, or even a whole book:

```
        void sendString (char *ptr) {
          while (*ptr)                                  // send the entire string
            sendCharacter(*ptr++);                      // one character at a time
        }

        void loop() {
          sendString("CQ DE W8BH");
          delay(5000);
        }
```

Audio.

It is time to add the first piece of hardware. We want to hear the code, not just see it. A small piezo element will do, like the PS1240 from Adafruit. Connect the (+) lead to pin B12 and the other lead to ground. Then add some define statements, and a few lines of audio code to the dit() and dah() routines:



```
        #define PITCH       1200                        // pitch in Hz of morse audio
        #define AUDIO       PB12                        // pin attached to piezo element

        void dit() {
          digitalWrite(LED,0);                          // turn on LED
          tone(AUDIO,PITCH);                            // and turn on sound
          ditSpaces();
          digitalWrite(LED,1);                          // turn off LED
          noTone(AUDIO);                                // and turn off sound
          ditSpaces();                                  // space between code elements
        }

        void dah() {
          digitalWrite(LED,0);                          // turn on LED
          tone(AUDIO,PITCH);                            // and turn on sound
          ditSpaces(3);                                 // length of dah = 3 dits
          digitalWrite(LED,1);                          // turn off LED
          noTone(AUDIO);                                // and turn off sound
          ditSpaces();                                  // space between code elements
        }
```

The tone() and noTone() routines are provided by the standard Arduino tone library. They are built-in and do not require any explicit include directive.

More Sound. These go up to 11.

The advantages of piezo elements are that they are small and don't use a lot of current.   You can hook them directly to a microcontroller pin. The big disadvantage is that they don't make much sound   A simple 3" 8-ohm speaker, like the Adafruit 1313 (pictured) will make a much nicer sound indeed.   If you connect it directly to the blue pill, you will enjoy its booming sound until you fry the microcontroller. It uses too much current!   Add a series current-limiting resistor of 100 ohms (louder) to 200 ohms (safer). Even better, a single NPN transistor driver using an 2N3904 or 2N4401 will keep the microcontroller safe and provide good sound output.

Part 2 Summary.

Using a Blue Pill microcontroller and a piezo element, we can now listen to Morse code. It can be of any length and play at any speed and pitch. In Part 3 we will create useful routines that generate number patterns, word patters, and callsigns.

See my github account for the full source code.

73, Bruce.