Morse Code Tutor -
from the ground up

Part 7: A simple matter of programming

Bruce E. Hall, W8BH

This is the 7th part of a series about an inexpensive 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.

I have selected a few details about the remaining code. It would be rather boring to review every single routine, especially if we have already covered similar ones. I will concentrate on new or interesting topics that help complete the project.

EEPROM, or lack thereof.

Small microcontrollers, like the one in the Arduino UNO, typically provide a small area of EEPROM (Electrically Erasable and Programmable Read Only Memory) where nonvolatile data can be stored. Unlike RAM, where program data is usually stored, EEPROM retains its contents when the device loses its power. This makes EEPROM useful for storing things like user preferences.

Strangely enough, the Blue Pill microcontroller doesn't have any EEPROM at all. So how can we store user preferences, such as code speed?

Thankfully, the smart people who wrote the Arduino-STM32F103C, interface created a substitute. It uses a portion of memory normally reserved for program code. To use it, we add a single include to the top of our sketch:

```
#include "EEPROM.h"
```

It creates an EEPROM object that we can use to read and write byte-sized variables. For example, EEPROM.write(address, data) will write date to the EEPROM and EEPROM.read(address,data) will read the data. You can also write with EEPROM.update(address,data), which minimizes wear and tear on the memory. It writes only if the data is different from what is already stored.

The concept is to load the data on program startup and to save it whenever the user changes a preference. I save 4 variables: two for code speed, one for pitch, and one to indicate which paddle is used for 'dit'.

Here are two simple routines to read and write our configuration data:

```
void saveConfig()
{
  EEPROM.update(0,42);                          // the answer to everything
  EEPROM.update(1,charSpeed);                   // save the character speed in wpm
  EEPROM.update(2,codeSpeed);                   // save overall code speed in wpm
  EEPROM.update(3,pitch/10);                    // save pitch as 1/10 of value
  EEPROM.update(4,ditPaddle);                   // save pin corresponding to 'dit'
}

void loadConfig()
{
  int flag = EEPROM.read(0);                    // values have been saved before?
  if (flag==42)                                 // yes, so load saved parameters
  {
     charSpeed  = EEPROM.read(1);
     codeSpeed  = EEPROM.read(2);
     pitch      = EEPROM.read(3)*10;
     ditPaddle  = EEPROM.read(4);
  }
}
```

One issue to consider: how do you know, when the program starts, if *anything* has been previously stored at these locations? You wouldn't want to set the code speed to some absurd random value, like 124 words per minute. I check the first byte to see if it matches a predetermined value. If it does, I can be reasonably sure that the EEPROM is working and that the data represents my data.

What did the user send?

Part 2 reviewed how to make the device send all sorts of Morse code, and Part 3 added paddle inputs so the user can send code. But I did not review how to decode Morse input. How do we decode what the user is sending? It would be instructive for the user to see if his code matches what he intended to send.

Jack's code implements a very nice binary search through the morse code tree. I chose a simpler, brute-force approach: look at the pattern of dits and dahs that the user sent, and find the pattern in the morse code table. It scans the table from top to bottom, and stops when the code if found. Since there are less than 60 entries in the table, a decoded character is returned in a matter of microseconds:

```
int decode(int code)                            // convert code to morse table index
{
  int i=0;
  while (i<ELEMENTS(morse) && (morse[i]!=code))   // search table for the code
    i++;                                        // ...from beginning to end.
  if (i<ELEMENTS(morse)) return i;              // found the code, so return index
  else return -1;                               // didn't find the code, return -1
}
```

It actually takes more effort to package the paddle input into a bit pattern. Remember the code we used to let the user send?

```
void doPaddles()
{
  while (!button_pressed) {
    if (ditPressed())
    {
       dit();                   // user wants a dit, so do it.
    }
    if (dahPressed())
```

```
    {
       dah();                    // user wants a dah, so do it.
    }
  }
}
```

Let's modify it so that we build a bit pattern as the user operates the paddles. The original code and pattern-creation code are highlighted below. The remaining code just monitors the time taken to receive input. When the user pauses, the character is assumed complete and is decoded:

```
char receivedChar()                          // monitor paddles, return decoded char
{
  int bit = 0;
  int code = 0;
  unsigned long start = millis();
  while (!button_pressed)
  {
    if (ditPressed())                        // user pressed dit paddle:
    {
      dit();                                 // so sound it out
      code += (1<<bit++);                    // add a '1' element to code
      start = millis();                      // and reset timeout.
    }
    if (dahPressed())                        // user pressed dah paddle:
    {
      dah();                                 // so sound it tout
      bit++;                                 // add '0' element to code
      start = millis();                      // and reset the timeout.
    }
    if (bit && (millis()-start > ditPeriod))  // waited for more than a dit
    {                                        // so that must be end of character:
      code += (1<<bit);                      // add stop bit
      int result = decode(code);             // look up code in morse array
      if (result<0) return ' ';              // oops, didn't find it
      else return '!'+result;                // found it! return result
    }
    if (millis()-start > 5*ditPeriod)        // long pause = word space
      return ' ';
  }
  return ' ';                                // return on button press
}
```

Notice the use of the left-shift operator "<<". This is the same bit-shifting technique we used to send a Morse pattern, only in reverse.

Farnsworth Encoding.

The Farnsworth method encourages the user to listen to characters sent at a fast "target" rate. Initially, the spaces between the characters and words are lengthened such that the overall speed is slow. The spacing between characters and words is gradually shortened as the student gains proficiency.

To implement Farnsworth timing, I created two user settings: code speed and character speed. Code speed is the overall rate of text transmission, and character speed is the rate at which each character is sent. If both are equal, the code is spaced normally. If the character speed is higher, the inter-character and inter-word spacings are increased to maintain the selected code speed.

It's tricky to implement! Here are a few details for the mathematically inclined; feel free to skip.

Go back to Part 2 and review Morse code timing. Recall that code speed is based on the 50 element word "PARIS ", which contains 31 intra-character elements and 19 extra-character elements. For a given speed, the duration of an element in milliseconds = 1200/wpm. If we set all of the intra-character elements to be at (a higher) characterSpeed, the total intracharacter time is:

Intra-character time = 31 elements * (1200/characterSpeed) = 37200/characterSpeed

The time to send the complete word, including intra-character and extra-character time, is:

Total time = 50 elements * 1200/codeSpeed = 60000/codeSpeed

The extra-character time is equal to the difference between them:

Extra-character time = (Total time) – (Intra-character time)
= (60000/codeSpeed) - (37200/characterSpeed)

To get length of an extra-character element, divide the above by the number of extra-character elements (19):

Extra-character element duration = (3158/codeSpeed) – (1958/characterSpeed)

The characterSpace() and wordSpace() routines are delays based on the extra-character element duration, so they use this formula to determine the proper delay. The intra-character element duration remains at 1200/characterSpeed.

Notice that when codeSpeed and characterSpeed are equal, the above equation simplifies to 1200/speed, as expected.

Part 7 Summary.

I hope that you have a better understanding of the project code, and have the tools and confidence to create something new. I would love to hear from you directly. Drop me a line if you learned something new, read something inspiring, or created your own version.

Many thanks to Jack Purdum W8TEE for inspiring me to learn from his project and create my own.

In Part 8 I add an SD card reader so that we can listen to any text that we want, including books.

The full source code for this project is available on my github account.

73, Bruce.