Morse Code Tutor - from the ground up

Part 3: More Morse

Bruce E. Hall, W8BH

This is part 3 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.

Now that we are able to send Morse Code, it's time to create routines that are useful in a learning environment. For example, routines that send random letters, numbers, and words. How about a random callsign generator? We will round out this part by adding paddles so that the user can practice sending, too.

Random thoughts.

You cannot learn Morse Code by listening to the alphabet over and over. Once you hear 'A' you will anticipate B, C, D and the rest without fully listening and comprehending them. So we need a way of jumping around and presenting random characters. Arduino includes the random() function. It takes two parameters, a & b, and outputs an integer result that lies between a and b. For example, random(0,10) will output a number between 0 to 10. The minimum value is included as a possible result but the maximum value is not. For example, possible results of random(0,4) are 0, 1, 2, and 3.

The numbers and alphabet are arranged in ascending order 0..9 and A..Z in the ASCII table, making our job relatively easy. The alphabet starts with 'A', so start there and add a random offset:

```
char randomLetter()                         // returns a random uppercase letter
{
  return 'A'+random(0,26);
}

char randomNumber()                         // returns a random single-digit # 0-9
{
  return '0'+random(0,10);
}
```

Creating a routine to send random Letters is only a few lines of code:

```
void sendLetters()                          // send random letters forever...
{
  while (true)
      sendCharacter(randomLetter());        // send the letter
}
```

Listen to it and see how long you last. It is a bit tiring, isn't it? The code never pauses. To make it more natural sounding, it helps to break the characters up into groups or "words" of 5 characters each. Adding the space between character groups makes it less tiresome:

```
#define WORDSIZE    5                       // number of chars per random word

void sendLetters()                          // send random letters forever...
{
  while (true) {
    for (int i=0; i<WORDSIZE; i++)          // break them up into "words"
      sendCharacter(randomLetter());        // send the letter
    sendCharacter(' ');                     // send a space between words
  }
}
```

Real Words.

Now let's present the user with real words, not fake ones. At higher speeds it becomes very helpful to learn the sound of the entire word, not just the letters. You aren't reading this text one letter at a time; you are reading the words. The same is true for (faster) Morse. Here are two ways for presenting words: one is just a long list of non-randomized words in a single string, the other is a randomized list of individual words:

```
char *commonWords = "a an the this these that some all any every who which what such
other I me my we us our you your he him his she her it its they them their man men people
time work well May will can one two great little first at by on upon over before to from
with in into out for of about up when then now how so like as well very only no not more
there than and or if but be am is are was were been has have had may can could will would
shall should must say said like go come do made work";

void sendCommonWords()
{
  while (true) {
    sendString(commonWords);                // one long string of 100 words
  }
}


char *hamWords[]  = {"DE", "TNX FER", "BT", "WX", "HR", "TEMP", "ES", "RIG", "ANT",
                     "DIPOLE", "VERTICAL", "BEAM", "HW", "CPI", "WARM", "SUNNY",
                     "CLOUDY", "COLD", "RAIN", "SNOW", "FOG","RPT", "NAME", "QTH",
                     "CALL", "UR", "SLD", "FB", "RST”
                    };

void sendHamWords()                         // send some common ham words
{
  while (true) {
    int index=random(0, ELEMENTS(hamWords)); // eeny, meany, miney, moe
    sendString(hamWords[index]);            // send the word
    sendCharacter(' ');                     // and a space between words
  }
}
```

I hope that you find these routines simple enough to modify yourself. The only new concept is a macro called ELEMENTS(), which takes the name of an array and returns the number of items in that array. It must be defined in your code:

```
#define ELEMENTS(x) (sizeof(x) / sizeof(x[0]))
```

You do not need to use this macro. For instance, you could substitute the number 28 for ELEMENTS(hamWords), since there are 28 words in that array. But what happens if you miscounted? Or, imagine that you add 3 words to the list next month. If you forget to change the 28 to a 31, you will never see the last three words in the list. Using ELEMENTS reduces errors and creates more easily modifiable code.

Callsigns.

Callsigns are tricky to hear correctly, because they are (almost) random. So, practicing listening to them is important. A random callsign generator is harder to program than a random word generator, but not too hard. Let's look at the code first and then parse it out:

```
char prefix[]     = {'A', 'W', 'K', 'N'};

void createCallsign(char* call)              // returns with random US callsign in "call"
{
  strcpy(call,"");                           // start with empty callsign
  int i = random(0, 4);                      // 4 possible start letters for US
  char c = prefix[i];                        // Get first letter of prefix
  addChar(call,c);                           // and add it to callsign
  i = random(0,3);                           // single or double-letter prefix?
  if ((i==2) or (c=='A'))                    // do a double-letter prefix
  {                                          // allowed combinations are:
    if (c=='A') i=random(0,12);              // AA-AL, or
    else i=random(0,26);                     // KA-KZ, NA-NZ, WA-WZ
    addChar(call,'A'+i);                     // add second char to prefix
  }
  addChar(call,randomNumber());              // add zone number to callsign
  for (int i=0; i<random(1, 4); i++)         // Suffix contains 1-3 letters
    addChar(call,randomLetter());            // add suffix letter(s) to call
}
```

The first issue is that handling strings in C or a C-like language is error-prone, and has led to all sorts of workarounds and gotchas. Without going into too much detail, I will stick to the original, efficient method of considering a string to be an array of characters, terminated at the end with a zero. Not the character "0", which has an ASCII value of 48, but the ASCII value '/0'. Using this method, the string "HELLO" is an array of <u>6</u> character bytes with the values: 72, 69, 76, 76, 79, 0.

Next, there is no good way of returning such a string as the result of a function call. Unfortunately you can't do "callsign = makeRandomCallsign()" Yes, there are workarounds but they aren't needed. The preferred method is to pass a pointer to the string instead, and then let the code modify the contents of the string. You must take care that the string has enough space to accommodate everything that your routine adds to it.

The two routines, "strcpy" (string copy) and "strcat" (string concatenation) are safe for copying to and adding to a string, respectively. Their first argument is the recipient string and the second argument is the donor string. In the code above, strcpy(call,””) clears the contents of the string by copying a blank string into call. There are other ways of doing it, but this is safe.

This routine creates random US callsigns. Feel free to modify it to your region of choice! US callsigns start with a 1- or 2-character prefix, with the first character always K, N, W, or A. The following lines of code randomize this choice:

```
int i = random(0, 4);                       // 4 possible start letters for US
char c = prefix[i];                         // Get first letter of prefix
addChar(call,c);                            // and add it to callsign
```

If the first character is A, it must be followed by another letter A through L. Otherwise, a second prefix letter may or may not be present. The following lines account for those details:

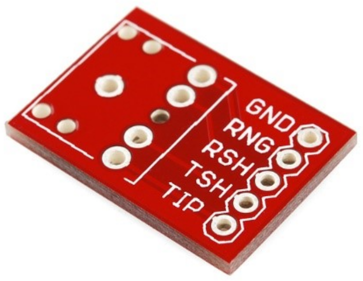
```
i = random(0,3);                            // single or double-letter prefix?
if ((i==2) or (c=='A'))                     // do a double-letter prefix
{                                           // allowed combinations are:
  if (c=='A') i=random(0,12);               // AA-AL, or
  else i=random(0,26);                      // KA-KZ, NA-NZ, WA-WZ
  addChar(call,'A'+i);                      // add second char to prefix
}
```

The rest is easy: a zone number, followed by a suffix of 1 to 3 letters:

```
addChar(call,randomNumber());               // add zone number to callsign
for (int i=0; i<random(1, 4); i++)          // Suffix contains 1-3 letters
  addChar(call,randomLetter());             // add suffix letter(s) to call
```

The strcat() function does not allow us to add a single character to a string, so I created a small helper function addChar() to do that.

Paddle input.



We have spent a lot of time learning how to generate Morse Code, because *listening to* and *understanding* code is usually harder than manually sending it. But now that the code-creation routines are relatively complete, it is time to add a set of paddles. Don’t have a set of paddles yet? No problem, you can easily test this code by touching the corresponding pins to ground. If you have a set of paddles that you want to use, and they are terminated with an audio plug, having a breakout board with a jack is very handy. I use the sparkfun version (PRT-10588), pictured at right. They sell the matching jack separately.

First, define which pins will be connected to the paddles:

```
#define PADDLE_A          PB8                    // Morse Paddle "dit"
#define PADDLE_B          PB7                    // Morse Paddle "dah"
```

Then, in your setup() routine, add both of those pins as inputs. They will be touched to ground by the paddle, so use the internal pullup resistor to keep them at logic 1 levels until the paddle is pressed:

```
    pinMode(PADDLE_A, INPUT_PULLUP);            // two paddle inputs, both active low
    pinMode(PADDLE_B, INPUT_PULLUP);
```

The keyer code is straightforward:

```
    bool ditPressed()
    {
      return (digitalRead(PADDLE_A)==0);             // pin is active low
    }

    bool dahPressed()
    {
      return (digitalRead(PADDLE_B)==0);             // pin is active low
    }

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

The keyer routine looks to see if either paddle is pressed, and issues dits and dahs accordingly. ditPressed() will be true whenever the PADDLE_A pin is grounded, and dahPressed() will be true whenever the PADDLE_B pin is grounded.

Part 3 Summary.

We now have a very capable set of routines that can send and receive Morse Code.   In Part 4 we will add a 2.2" LCD display, so that we can see the characters that are being sent and received.

See my github account for the source code.

73, Bruce.