Morse Code Tutor - from the ground up

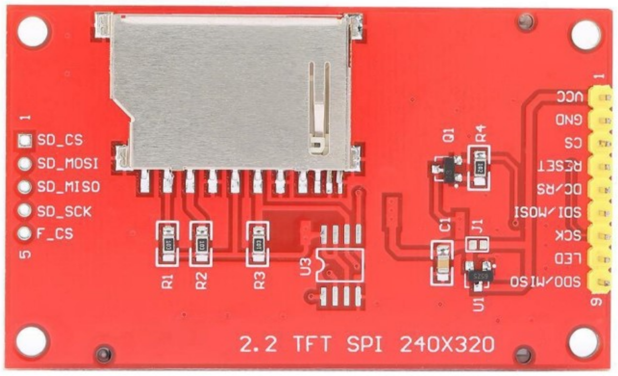
Part 8: Add an SD card

Bruce E. Hall, W8BH

This is Part 8 in a series of articles about my Morse Code Tutor, an inexpensive and easy-to-build device that can help you learn Morse code.

After using Morse Code Tutor for a while, I noticed things that I wanted to change. Have you? In my case, I wanted to be able to listen to more meaningful text. Not just Morse code letters and words, but full sentences and paragraphs. Wouldn't it be great to listen to a book or magazine? I came up with a few more add-ons that make the tutor more fun to use. Read on if you are interested.

The SD card reader.



The TFT display board includes an SD card connector. Let's see if we can hook it up. The photo on the left shows the shiny metal connector along the top and corresponding holes along the left edge, all labelled "SD_" something. These four pins provide the interface between the SD card and the microcontroller. Notice that all four lines (CS, MOSI, MISO, SCK) have corresponding pins on the opposite side of the board which are used by the display.

The interface is known as SPI (serial peripheral interface). MOSI sends data/commands to the device, MISO returns data from the device, and SCK is the data clock. Every device on an SPI bus also has a CS (chip select), so that multiple devices can all share MOSI, MISO, and SCK.

My board did not come with pins on the SD lines. The first order of business is to solder some in male header pins. Fortunately, the distance between the SD and TFT pins is breadboard compatible.

Here are the connections between the Display Board and the microcontroller board. The connections on the 9-pin side remain the same as before, and the connections on the 5-pin side are new. Also notice that PA7 (MOSI) and PA5 (SCK) of the microcontroller go to both sides of the display. This is because the SPI bus is being shared by the LCD and the SD card.

Display Board (9-pin TFT side)	Display Board (5-pin SD side)	Microcontroller Board
Vcc		3.3V
GND		GND
CS		PA1
RST		3.3V
DC		PA0
MOSI		**PA7**
SCK		**PA5**
LED		3.3V
MISO		-
	SD_CS	PA4
	SD_MOSI	**PA7**
	SD_MISO	PA6
	SD_SCK	**PA5**
	F-CS	-

Solder in the SD pins, re-insert the display on the breadboard, and connect as shown. The sketch from part 7 should still work; in fact, make sure that it still works.

Arduino comes with a build-in SD library, making it easy to add SD card functionality to a sketch. To use the library, include it at the top of the sketch, just after the EEPROM include:

```
#include "SD.h"
```

Next, initialize the SD library by telling it which pin is connected to the SD card chip select line. Put the define with the rest of your defines, and put the SD.begin statement in the setup() routine.

```
#define SD_CS              PA4                      // SD card "CS" pin

SD.begin(SD_CS);                                    // initialize SD library
```

We can now open and close files on the card, and read those files one character at a time with the read() function. For example,

```
void readBook()
{
  File book = SD.open("morse.txt");                 // look for book on sd card
  if (book) {                                       // find it?
    while (book.available()) {                      // do for all characters in book:
      char ch = book.read();                        // get next character
      sendCharacter(ch);                            // and send it
    }
    book.close();                                   // close the file
  }
}
```

SD.open opens the file, book.available() returns true if there are any characters left to read, and book.read() returns the next character in the file. Can you add this code to part 7, and find a good way to execute it? You could substitute it for an existing routine, or you could add a menu choice for it. If it doesn't work, try adding it to one of the previous tutorial parts. For example, the sketch "TestSD" on my github account is this routine attached to Part 2.

The SD card.

You will need an SD card to test your code. The SD connector on my display is the old "standard" 32 x 24mm size; yours may vary. A 2 GB card is ample enough to hold many large books in text format. Format your SD card and add a file to the root directory named "morse.txt". It is OK if other files are on the card; only "morse.txt" will play.

I have a dozen books loaded onto my SD card. The text of these books is found at Project Gutenberg, an online repository for copyright-free eBooks. The texts are also available on my github account.

Faster, faster.

There are things you notice after using the tutor for a while. For instance, while listening to a book, I really wanted to speed things up. You can do this by leaving the book, going to the config menu, increasing the code speed, then reopening the book. But by then you've lost your rhythm. Wouldn't it be great to just increase the code speed *on the fly*? The same goes for slowing it down. Slowing down might not seem like a good strategy, but why not? Especially if you had just increased the speed too high and need to bring it back down a notch or two.

The rotary encoder knob is a good way to change the speed, especially since it is not doing anything else while you are listening to code. The idea is this: every time you send a character in the sketch, check the encoder knob to see if it has been rotated. If so, increment/decrement the speed. In fact, we can put this check inside the sendCharacter() routine, incorporating it as a part of all sending routines:

```
void sendCharacter(char c) {                        // send single ASCII character in Morse
  // .. some code here … //
  checkForSpeedChange();                            // allow change in speed while sending
  addCharacter(c);                                  // display character on LCD
  if (c==32) wordSpace();                           // space between words
  else sendElements(morse[c-33]);                   // send the character
}
```

The code for checkForSpeedChange() looks at the encoder and changes the code speed accordingly, almost exactly the same as it does in config menu routine:

```
void checkForSpeedChange()
{
  int dir = readEncoder();
  if (dir!=0)
  {
    codeSpeed += dir;
    if (codeSpeed<MINSPEED)
      codeSpeed=MINSPEED;
    if (codeSpeed>MAXSPEED)
        codeSpeed=MAXSPEED;
    charSpeed = codeSpeed;
    }
    ditPeriod = intracharDit();
  }
}
```

That works great! You can also adjust for Farnsworth encoding, which I do in the Part8 final code.

Pause It.

Another thing I've noticed: You can quit a practice session by pressing the encoder button, but sometimes what you really want to do is PAUSE. For example, you are listening to your book and the telephone rings. You exit the book, but it was just getting interesting and now you've lost your spot. It would be great to be able to pause the book, take care of the interruption, and resume when you are ready. Let's add a pause.

We have already used our encoder for speed changes, so what other input devices are left? Answer: the paddles! We aren't using them when we are listening to code. I decided on the following:

D<u>AH</u> = P<u>AU</u>SE
D<u>IT</u> = RE<u>T</u>URN

A bit corny but it works for me. I also tried using the dah paddle by itself as a sort of Pause/Unpause toggle, but DAH/DIT work better.

Just like checkForSpeedChange(), we can add pause to the sendCharacter() routine. By doing so it takes effect for all sending routines:

```
void sendCharacter(char c) {                        // send single ASCII character in Morse
  // .. some code here … //
  checkForSpeedChange();                            // allow change in speed while sending
  do
    checkPause();                                   // allow user to pause morse output
  while (paused);
  addCharacter(c);                                  // display character on LCD
  if (c==32) wordSpace();                           // space between words
  else sendElements(morse[c-33]);                   // send the character
}
```

How does that work? We add a new global variable called pause. The highlighted code is a do…while loop that loops while pause is true. In other words, don't do anything if we are paused. Don't go to the next character, just wait. And while we are waiting, call the checkPause() routine, which has the job of checking to see if we should continue to be paused or unpaused:

```
void checkPause()
{
  if (dahPressed())                                 // did user press "dah"?
    paused = true;                                  // yes, so pause output
  if (ditPressed() || button_pressed)               // did user press "dit"?
    paused = false;                                 // yes, so resume output
}
```

Pause is a really great feature for only a few lines of code. Why use the do…while loop rather than a simpler while loop? Because it forces the loop to run at least once, checking for a pause. If pause is found, it stays in the loop until unpaused.

Part 8 summary.

For a very modest amount of coding effort, we can now read any book we want, change speed on the fly, and pause the Morse output. You can also skip ahead in the text (about a page at a time).

The Part 8 code includes a few extras: I changed the words list so that words are sent in random order. I also added single letter and 2 letter sending practice. The sending practice sessions also display how many times the user sent the text correctly.

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

73, Bruce.