Morse Code Tutor -
from the ground up

Part 5: Add a rotary encoder

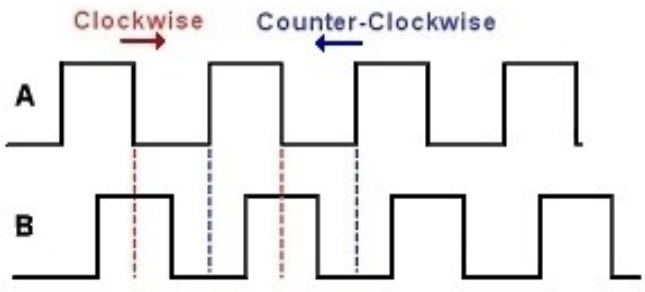
Bruce E. Hall, W8BH

This is part 5 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 we have all the tools needed to send and receive code, and we can see corresponding words on the display. And we have many different types of practice: letters, numbers, words, and callsigns. But the user has no way to select between these choices. It is time to add another input device to our project: the rotary encoder. The coding requires a little thought, so grab a cup of coffee and dig in.

The rotary encoder.

Encoders are the modern-day version of the potentiometer. Indeed, volume controls, which were often potentiometers in the pre-digital era, are often done with encoders today. So how do they work?



The encoder has two outputs, A and B, which are 90-degrees out of phase with each other. As the rotator knob is turned, each output cycles between high and low states. The first vertical dashed line in the graph above corresponds to the start of the red-shaded sequence (below); the green-shaded cycle starts at the third vertical-dashed line. Notice how the cycle repeats itself after 4 unique transitions.

B	0	1	**1**	**0**	**0**	**1**	1	0	0	1	1
A	1	1	**0**	**0**	**1**	**1**	0	0	1	1	0

We can determine the direction that the knob is turning by comparing the previous and current states of pins B and A. For example, if B becomes high after A we are turning clockwise; if B becomes low after A then we are turning counter-clockwise. Let's create a list of all of the possible transition states BA -> B'A' corresponding to the red-shaded area above.

- Clockwise: 10 to 00, 00 to 01, 01 to 11, and 11 to 10. After that the cycle repeats.
- Counter-clockwise: 11 to 01, 01 to 00, 00 to 01, and 01 to 11. After that the cycle repeats.

Abbreviate the transitions such that "01-to-11" becomes 0111, "11 to 10" becomes 1110, etc. These are 4-bit binary numbers, and can be further reduced to a decimal number equivalent. It results in the following table of transition states:

Clockwise Rotation Transition States	CCW Rotation Transition States
1000 = 8	**1101 = 13**
0001 = 1	**0100 = 4**
0111 = 7	**0010 = 2**
1110 = 14	**1011 = 11**

There are 8 valid transition states, 4 clockwise and 4 counterclockwise. Notice also that there are 8 additional transition states which cannot occur: 0000 (0), 0011 (3), 0101 (5), 0110 (6), 1001 (9), 1010 (10), 1100 (12), 1110 (14). This makes intuitive sense, because at each transition one and only one of the outputs changes. The outputs cannot both stay the same (0000,0101,1010,1111) and they cannot both change (0011,1100,0110,1001).

Transition State Table:

Transition State	Direction
0000 (0)	-
0001 (1)	CW
0010 (2)	CCW
0011 (3)	-
0100 (4)	CCW
0101 (5)	-
0110 (6)	-
0111 (7)	CW
1000 (8)	CW
1001 (9)	-
1010 (10)	-
1011 (11)	CCW
1100 (12)	-
1101 (13)	CCW
1110 (14)	CW
1111 (15)	-

Expressed in a slightly different way, we can create a table of all 16 transition states, including the impossible ones, and whether or not they indicate CW or CCW rotation. See table at left.

What's the point of all this? If we flag any time a rotary pin changes, we can check the state of *both* pins and compare them to the previous result. Each pin change will indicate that the knob is turning, and the transition state will indicate the direction of rotation.

Now, how do we check to see if the encoder outputs are changing? The easiest way is to poll the corresponding microcontroller pins, repeatedly checking to see if either of them goes low. This works as long as you aren't doing any other time-intensive chore in your code. Another method is to use a hardware interrupt. By using interrupts, you briefly leave your code to take stock of the pin change, then return to where you left off. The associated encoder data will be waiting for you whenever you are ready to check it.

Attaching the encoder.

So much for theory – it is time to add this encoder. The encoder has 3 pins. Usually the center pin is ground and the outside pins are outputs A and B. The encoder output pins are connected to microcontroller pins PA8 and PA9. On the other side of the encoder are 2 pins for the pushbutton switch. One of these goes to ground, and the other is connected to microcontroller pin PA4. The following defines must then be added to the code:

```
#define ENCODER_A          PA9                  // Rotary Encoder output A
#define ENCODER_B          PA8                  // Rotary Encoder output B
#define ENCODER_BUTTON     PB15                 // Rotary Encoder pushbutton
```

All three inputs must be declared as inputs, and are active low. Therefore the internal pullup resistors should be enabled. Additionally, each pin must be configured to be a hardware interrupt. Here is the initialization code:

```
void initEncoder()
{
  pinMode(ENCODER_A, INPUT_PULLUP);
  pinMode(ENCODER_B, INPUT_PULLUP);
  pinMode(ENCODER_BUTTON, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(ENCODER_BUTTON),buttonISR,CHANGE);
  attachInterrupt(digitalPinToInterrupt(ENCODER_A),rotaryISR,CHANGE);
  attachInterrupt(digitalPinToInterrupt(ENCODER_B),rotaryISR,CHANGE);
}
```

The last three statements attach these interrupts to interrupt service routines, named buttonISR() and rotaryISR(), which are called when the corresponding pins change state.

Need an encoder? Adafruit #377 is a simple but dependable Bournes encoder. There are plenty of cheaper alternatives.

The encoder interrupt service routine (ISR).

The microcontroller immediately suspends whatever its doing and jumps to the interrupt service routine when the designated pin changes state. It is important that the ISR work quickly so that the microcontroller can return to the previously running code.

We need two ISR routines: one for the rotary encoder AB outputs, and one for the pushbutton. Let's look at the encoder routine first. Since we are in the interrupt routine, we know that the encoder has just moved, and a change of the pins was detected. Recalling the 'theory' above, we can determine the transition state (and therefore direction) by comparing the output pins to their previous levels.

Let's create a variable called rotary_state, and put the value of both encoder pins into it. This is a byte variable which means that it contains 8 individual bits, labelled b7 through b0. We will store the value of encoder_A into the right-most bit (b0), and the value of encoder_B in b1:

rotary_state bits: after storing encoder outputs

b7	b6	b5	b4	b3	b2	b1	b0
0	0	0	0	0	0	Encoder_B	Encoder_A

To do this in code we use the 'OR' operator, represented by '|=':

```
rotary_state |= (digitalRead(ENCODER_A));       // put encoder_A on bit 0
rotary_state |= (digitalRead(ENCODER_B) << 1);  // put encoder_B on bit 1
```

Notice that we also use the bit-shift operator "<<" to put encoder_B's value into bit 1 instead of bit 0. Now we need to compare the current logic levels with the previous levels. You could set up another variable for this and call it previous_rotary_state, but a trickier way is to just shift the rotary_state to the left by two bits:

rotary_state bits: after shifting twice to the left

b7	b6	b5	b4	b3	b2	b1	b0
0	0	0	0	Encoder_B	Encoder_A	0	0

Now bits b3 and b2 contain the encoder values. And we have created space at b1 and b0 to store the next transition. On the next transition we store the new values B' and A', just the same as before:

rotary_state bits: after OR'ing in new data

b7	b6	b5	b4	b3	b2	b1	b0
0	0	0	0	Encoder_B	Encoder_A	Encoder_B'	Encoder_A'

Voila, rotary_state contains all the information for our transition state in its lower 4 bits. Now skip back two pages and look at the transition state table. rotary_state corresponds to the column on the left. We can create a small array that will give the direction (1 for CW, -1 for CCW) for each entry in the table:

```
int states[] = {0,1,-1,0,-1,0,0,1,1,0,0,-1,0,-1,1,0};
```

Here is the full code. "rotary_counter" increments/decrements according to the encoder movement

```
void rotaryISR()
{
  const int states[] = {0,1,-1,0,-1,0,0,1,1,0,0,-1,0,-1,1,0};
  static byte rotary_state = 0;                    // current and previous encoder states

  rotary_state <<= 2;                              // shift previous state up 2 bits
  rotary_state |= (digitalRead(ENCODER_A));        // put encoder_A on bit 0
  rotary_state |= (digitalRead(ENCODER_B) << 1);   // put encoder_B on bit 1
  rotary_state &= 0x0F;                            // zero upper 4 bits

  int change = states[rotary_state];               // map transition to CW vs CCW rotation
  if (change!=0)                                   // make sure transition is valid
  {
    rotary_counter += change;                      // update rotary counter +/- 1
  }
}
```

The pushbutton interrupt service routine (ISR).

Who knew that the rotary encoder was so complicated? By comparison the pushbutton is easy: just check the state of the button and set a variable accordingly:

```
button_state = digitalRead(ENCODER_BUTTON);
```

You can use this and it works, just not very well. The problem is, at the time of pressing and releasing the button, there are a few milliseconds of time when the logic level rapidly fluctuates. This is called switch "bouncing". If you write a routine to count the number of button presses, it might register 14 presses instead of 1 actual press. A hardware solution to the problem is to attach an RC debounce circuit to the switch. We can do the same in software, reducing the component count. As a side benefit, we also can track the length of the last button press -- useful if you want to distinguish 'short' from 'long' button presses.

Similar to the rotary routine, we need to keep tabs on the previous state and compare it to the new one. It doesn't involve any bit manipulation since there is only one pin to keep track of. button_state will be the previous state and pinState will be the new state:

```
  static boolean button_state = false;
  boolean pinState = digitalRead(ENCODER_BUTTON);
```

The word 'static' indicates that the variable, while local in scope, retains its value after the routine ends. This is very useful for keeping track of the state. A global variable could also be used, but keeping it local means that only this routine may modify its value. Here is the outline of our buttonISR(), marking the button transitions:

```
void buttonISR()
{
  static boolean button_state = false;
  boolean pinState = digitalRead(ENCODER_BUTTON);

  if ((pinState == LOW) && (button_state == false))
  {
        // Button was up, but is currently down
  }
  else if ((pinState == HIGH) && (button_state == true))
  {
        // Button was down, but now released
  }
}
```

To implement a software debounce, we need to record the time, using millis(), when the button was pressed and when it was released. If the button had been in the previous state for at least 10mS, count it as a valid transition. Here is the full routine:

```
void buttonISR()
{
  static boolean button_state = false;
  static unsigned long start, end;
  boolean pinState = digitalRead(ENCODER_BUTTON);

  if ((pinState==LOW) && (button_state==false))
  {                                          // Button was up, but is now down
    start = millis();                        // mark time of button down
    if (start > (end + 10))                  // was button up for 10mS?
    {
      button_state = true;                   // yes, so change state
      button_pressed = true;
    }
  }
```

```
  else if ((pinState==HIGH) && (button_state==true))
  {                                                 // Button was down, but now up
    end = millis();                                 // mark time of release
    if (end > (start + 10))                         // was button down for 10mS?
    {
      button_state = false;                         // yes, so change state
      button_released = true;
      button_downtime = end - start;                // and record how long button was down
    }
  }
}
```

Do something.

After a long discussion about rotary encoder programming, it is finally time to add something to our project. In the previous tutorial we had several different Morse practice routines but no way to switch between them. Let's modify each of those routines, so they run until a button is pressed. Now we can listen to the routines, one after another, using the encoder button to switch between them.

For each of the routines, change the while() loop so that it exits if the button has been pressed. For example, in sendLetters():

```
void sendLetters()
{
  while (!button_pressed) {                         // used to say "while(true)"
    for (int i=0; i<WORDSIZE; i++)                  // group letters into "words"
      sendCharacter(randomLetter());                // send the letter
    sendCharacter(' ');                             // send a space between words
  }
}
```

Now the main program loop can cycle through these routines. Do not forget to reset the button_pressed flag: it is set by the ISR but needs to be reset when the event is handled.

```
void loop() {
  sendLetters();      button_pressed = false;
  sendNumbers();      button_pressed = false;
  sendHamWords();     button_pressed = false;
  sendMixedChars();   button_pressed = false;
  sendCallsigns();    button_pressed = false;
}
```

Part 5 Summary.

We can send Morse Code, display the corresponding text, and cycle through the different practice sessions using the encoder button. In Part 6 we will put the rotary encoder to better use, adding a full menu system to the project.

See my github account for the source code.

73, Bruce.