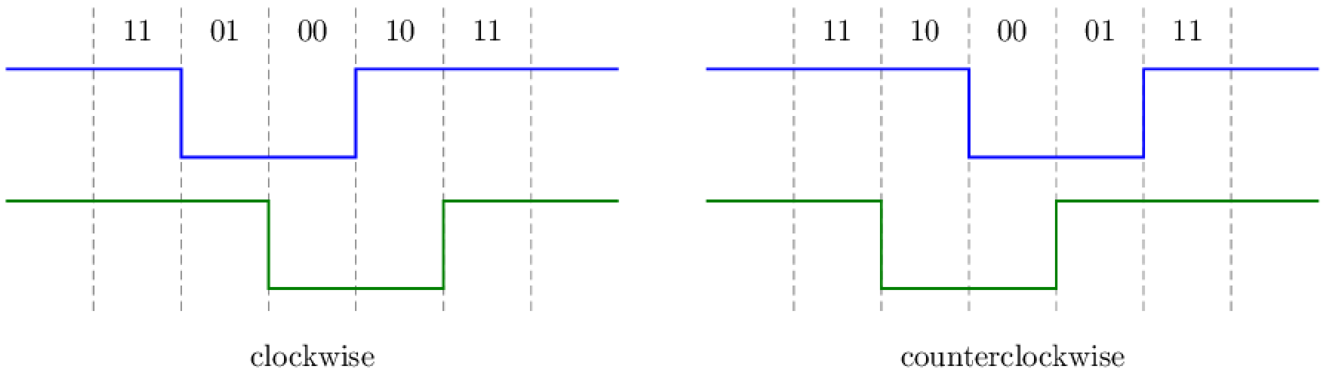
MP Electronic Devices: Yet another algorithm for rotary encoder control

This is a part of MP Electronic Devices site.

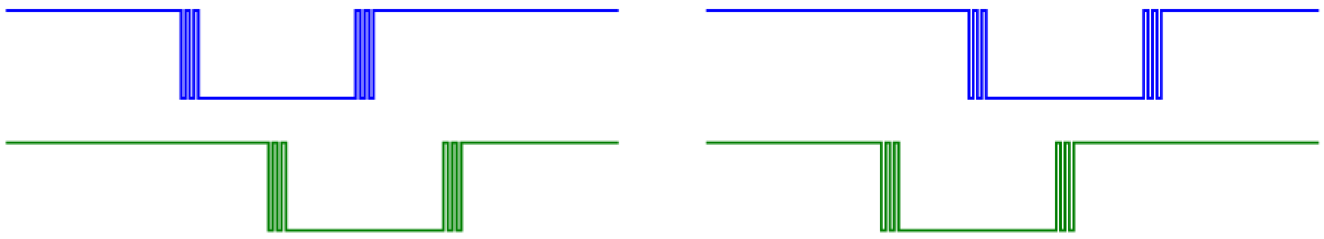
Principles of operation

Rotary encoders are devices that typically convert the angular motion of a shaft into digital output signals. There are many web resources that explain the mechanisms of these devices and how to connecting them to a microcomputer, but this page is not intended to be a complete overview. The idea is to present another algorithm to properly read the output of mechanical rotary encoders.



In principle, common rotary encoders generate the signal by alternately closing two switches, as shown in the figure above. Depending on the connection, the neutral state can be either High (1) or Low (0). We assume here the most common connection where the neutral state is High. For clockwise movement, the action on the first switch precedes the action on the second switch, while this is reversed for counterclockwise movement. If we denote each state as a binary number with two digits representing the states of the two switches, the neutral state is represented as `11`, clockwise rotation is represented by the chain of states `11` → `01` → `00` → `10` → `11` and counterclockwise rotation is represented by the chain of states `11` → `10` → `00` → `01` → `11`.

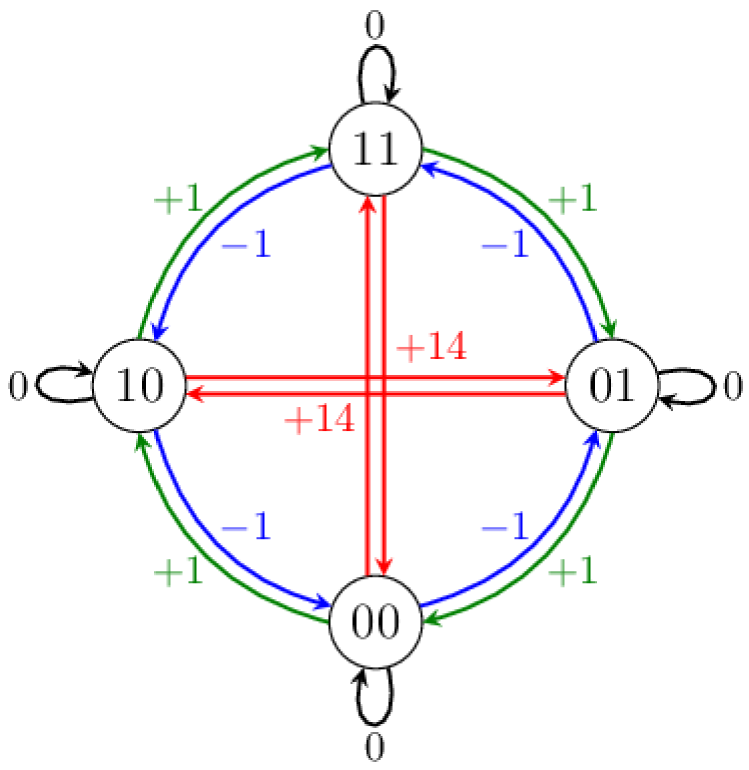
Switch bouncing



The problem is that mechanical electrical switches are not perfect, so the phenomenon **switch bouncing** occurs. When a switch is toggled, the contacts must physically move from one position to another. As the components of the switch settle into their new position, they bounce mechanically, causing the underlying circuit to open and close several times. The signal obtained is shown in the figure above.

This problem can be fixed either by hardware or software. In the case of software, this involves a certain subjectively determined hold time during which the activity of the switch is ignored. However, in case of the rotary switch we can avoid the hold time altogether. Since the pair of switches passes through four successive states, we can in principle define exactly when the operation is completed.

Algorithm



The microprocessor can read out a total of four different states, and two successive readouts give a total of sixteen possible transitions, as shown in the figure above. Four transitions represent no movement, four transitions represent a quarter movement to the right, four transitions represent a quarter movement to the left, and four transitions represent theoretically impossible transitions.

Let's define a **full cycle** as a process that begins and ends in the neutral state. By assigning 0 for no movement, +1 for a quarter movement to the right and −1 for a quarter movement to the left, we get the aggregate of +4 for a full cycle to the right, the aggregate of −4 for a full cycle to the left and the aggregate of 0 for no movement. In particular, each individual switch bouncing, whether to the left or to the right, totals 0 and has no influence to the result. Also, the modulo of the aggregate and 4 gives us the exact state. For example, if the modulo is zero, the current state is the neutral state.

Special care must be taken with "impossible" transitions. Due to technical inadequacies, they are indeed possible. If we assign the value 0 to the impossible transitions, the aggregate of a full cycle with exactly one impossible transition is −6, −2, +2 or +6. To recognise that the full cycle contained one (or more) impossible transitions, we must assign a number larger than 10 to the impossible transitions, so that the aggregate of the full cycle will be larger than 4. However, if we assign the number 14, we not only recognise the existence of the impossible transition(s), but also the modulo of the aggregate and 4 is zero again when and only when the neutral state is reached.

Implementation

All these considerations taken together can be observed in the following Python script, intended for use on Raspberry Pi. The `rotary` function first updates `lrmem`, a binary variable with four digits, where the first two digits represent the previous state and the second two digits represent the current state, which is essentially the last transformation. The list `TRANS` contains the values for each transformation, as shown in the following table.

`lrmem`	**Transformation**	`TRANS`	**Comment**
0b0000	`00` → `00`	0	no movement
0b0001	`00` → `01`	−1	movement to the left
0b0010	`00` → `10`	+1	movement to the right
0b0011	`00` → `11`	+14	impossible movement
0b0100	`01` → `00`	+1	movement to the right
0b0101	`01` → `01`	0	no movement
0b0110	`01` → `10`	+14	impossible movement
0b0111	`01` → `11`	−1	movement to the left
0b1000	`10` → `00`	−1	movement to the left
0b1001	`10` → `01`	+14	impossible movement

0b1010	`10` → `10`	0	no movement
0b1011	`10` → `11`	+1	movement to the right
0b1100	`11` → `00`	+14	impossible movement
0b1101	`11` → `01`	+1	movement to the right
0b1110	`11` → `10`	−1	movement to the left
0b1111	`11` → `11`	0	no movement

The `rotary` function then updates `lrsum`, the variable containing the current aggregate. When the neutral state is reached, the function acts accordingly.

Python code for Raspberry Pi

```
import pigpio

pi = pigpio.pi()

# -1: left transition, +1: right transition, 0: no transition and 14:
impossible transition
TRANS = [0, -1, 1, 14, 1, 0, 14, -1, -1, 14, 0, 1, 14, 1, -1, 0]
LEFT = 16
RIGHT = 20
PUSH = 21

def rotary():
    global lrmem
    global lrsum

    l = pi.read(LEFT)
    r = pi.read(RIGHT)
    lrmem = (lrmem % 4)*4 + 2*l + r
    lrsum = lrsum + TRANS[lrmem]
    # encoder not in the neutral state
    if(lrsum % 4 != 0): return(0)
    # encoder in the neutral state
    if (lrsum == 4):
        lrsum=0
        return(1)
    if (lrsum == -4):
        lrsum=0
        return(-1)
    # lrsum > 0 if the impossible transition
    lrsum=0
    return(0)

pi.set_mode(LEFT, pigpio.INPUT)
pi.set_mode(RIGHT, pigpio.INPUT)
pi.set_mode(PUSH, pigpio.INPUT)
pi.set_pull_up_down(LEFT, pigpio.PUD_UP)
pi.set_pull_up_down(RIGHT, pigpio.PUD_UP)
pi.set_pull_up_down(PUSH, pigpio.PUD_UP)
```

```
lrmem = 3
lrsum = 0
num = 0
print(num)

while(True):
    res = rotary()
    if (res!=0):
        num=num + res
        print(num)
    if(pi.read(PUSH)==0):
        break
```

Note that no additional electronic components are necessary, as the script uses built-in Raspberry Pi pull-up resistors.

C code for Arduino

```
#define LEFT 2
#define RIGHT 3
#define PUSH 4

uint8_t lrmem = 3;
int lrsum = 0;
int num = 0;

int8_t rotary()
{
   static int8_t TRANS[] = {0,-1,1,14,1,0,14,-1,-1,14,0,1,14,1,-1,0};
   int8_t l, r;

   l = digitalRead(LEFT);
   r = digitalRead(RIGHT);

   lrmem = ((lrmem & 0x03) << 2) + 2*l + r;
   lrsum = lrsum + TRANS[lrmem];
   /* encoder not in the neutral state */
   if(lrsum % 4 != 0) return(0);
   /* encoder in the neutral state */
   if (lrsum == 4)
      {
      lrsum=0;
      return(1);
      }
   if (lrsum == -4)
      {
      lrsum=0;
      return(-1);
      }
   /* lrsum > 0 if the impossible transition */
   lrsum=0;
   return(0);
}

void setup()
{
   pinMode(LEFT, INPUT);
   pinMode(RIGHT, INPUT);
   pinMode(PUSH, INPUT);
   pinMode(LEFT, INPUT_PULLUP);
   pinMode(RIGHT, INPUT_PULLUP);
   pinMode(PUSH, INPUT_PULLUP);
   Serial.begin(9600);
   Serial.println(num, DEC);
}

void loop()
{
   int8_t res;

   res = rotary();
   if (res!=0)
   {
      num = num + res;
      Serial.println(num);
   }
   if (digitalRead(PUSH) == 0)
   {
      Serial.println(num);
      delay(250);
   }
}
```

Links to other useful resources

Better Rotary Encoder - no switch-bounce - Ralph Bacon's Arduino interrupt implementation of this code

Rotary Encoder: H/W, S/W or No Debounce?

Rotary Encoder : How to use the Keys KY-040 Encoder on the Arduino

Please use the feedback form to tell me what else you would like to see here, suggestions, new tricks, etc.

Created by Marko Pinteric: feedback form.

Updated December 25, 2021. Web page has been read by 3924 visitors since March 2021.