mathertel.de

A Library for the Arduino environment for using a rotary encoder as an input.

Here you can find an Arduino compatible library for using rotary encoders.

I was searching a library for using a rotary encoder in my latest project and found a lot of information on this topic but none of the existing libraries did immediately match my expectations so I finally built my own.

This article likes to explain the software mechanisms used in detail so you can understand the coding and might be able to adjust it to your needs if you like. There are various aspects when writing a library for rotary encoders and you can also find a lot of the sources I analyzed at the bottom of this article.

Download

You can download the library and examples directly from the github repository that you can find at:

- https://github.com/mathertel/RotaryEncoder

Use the "Download zip file" button to get all the files and put them into your Sketches / libraries folder.

Rotary Encoder signals

The signals a rotary encoder produces are based on a 2-bit gray code available on 2 digital data signal lines. Typical encoders use 3 output pins: 2 for the signals and one for the common signal usually GND.

There are 4 possible combinations of HIGH and LOW on these 2 signal lines that can be seen on this video:

Rotary Encoder signal generation

- The rotary encoder has a detent mechanism that **holds the knob in place when both contacts are open**. On the latch [detent] position [3]; both signals are HIGH.

- When rotating the knob the two contacts will be connected to the common ground one by one.
  On position **3** the TOP signal is **HIGH** and the BOTTOM signal is **HIGH** = detent
  On position **1** the TOP signal is **LOW** and the BOTTOM signal is **HIGH**.
  On position **0** the TOP signal is **LOW** and the BOTTOM signal is **LOW.**
  On position **2** the TOP signal is **HIGH** and the BOTTOM signal is **LOW.**
- Common practice is to use ground as the common signal and use pull-up resistors to have a HIGH signal when the contacts are open.
- The Arduino processor has internal pull-up resistors on all input pins that can be enabled by software. Have a look at the inline comments of the library.

The bouncing problem

When using contacts there is no exact ON and OFF condition while switching; this is because the contacts never close exactly and especially older contacts or dirty contacts generate a lot of fast ON/OFF sequences.

When you stop rotating the knob the contact position should be fixed. If not throw it away.

By using a capacitor it is possible to build a low pass filter that *reduces* the bouncing effect and eliminates very small spikes. If you ever use this solution, put the capacitor as near as possible to the contacts. 100nF will do here.

Another solution is to detect changes of the signal in a very small timeframe and ignore them. You can find a solution using this approach on the Arduino playground article.

The solution I found in some of the libraries around is to exactly follow all 4 possible states of the signals that a switch will pass when being rotated from one position to the next and detect the next position only when reaching a detent state. Even when a bouncing will occur it will not change the position counting more than once.

High Frequency

There are library approaches that can handle high frequency rotary encoders. I only use the rotary encoder with my fingers and have not found any problems for that speed.

However with the library it is important to have a look at the signals every few microseconds or at least when a signal has changed. So give the library often a try to detect the situation or use an interrupt.

You can tweak reading the ports by using the bitRead function or a direct read of the Port, but the library looses some of the compatibility with the Arduino style of programming because you have to know the exact hardware definitions.

If you need more advanced support for decoding signals from encoders have a look at the optical solutions and processors like the ATXMEGA that includes hardware support for reading position from Quadrature Encoders.

Counting States

Many sources use two variables to store the levels of the 2 signal lines.

I found it more practical to use the numbers 0, 1, 2 and 3 for a specific state.

The first signal-line [say: pin SW] counts for the value 1 and the second [say: pin DT] counts for the value 2. We use the addition of both values. Because the digitalRead function returns 0 or 1 this can be done by using a bit-shift and addition:

```
thisState = sig1 + 2 * sig2;
```

or more efficiently:

```
thisState = sig1 | (sig2 << 1);
```

Explanation:

curState	= sig_1	+ 2 * sig_2	= curState
	= 0	+ 2 * 0	= 0
	= 0	+ 2 * 1	= 2
	= 1	+ 2 * 0	= 1
	= 1	+ 2 * 1	= 3

Transition from State to State

Detecting a movement of the knob is now identical to detect a transition from one state to the next one. Not all transitions are valid and when both signals change at once there is something wrong with the signals.

Here is a straight forward implementation for calculating the position of the knob from an old (last known) state and the current state:

```
if ((oldState == 3) && (thisState == 1)) knobPosition++;
else
if ((oldState == 1) && (thisState == 0)) knobPosition++;
else
if ((oldState == 0) && (thisState == 2)) knobPosition++;
else
if ((oldState == 2) && (thisState == 3)) knobPosition++;

else
if ((oldState == 3) && (thisState == 2)) knobPosition--;
else
if ((oldState == 2) && (thisState == 0)) knobPosition--;
else
if ((oldState == 0) && (thisState == 1)) knobPosition--;
else
if ((oldState == 1) && (thisState == 3)) knobPosition--;
```

This solution uses a lot of programming instructions and also seems to be slow (especially when turning left). A small static array can help reducing time and space.

- *The implementation from Brian Low also uses a table and almost the same trick: calculate a unique index from the oldState and the curState and make a lookup into an array of integers:*

```
index = curState + 4 * oldState;
```

- *or more efficiently:*

```
index = curState | (oldState << 2);
```

Explanation;

```
index = 4*oldState + curState = index
  13  =  4 * 3     +   1        1101
   4  =  4 * 1     +   0        0100
   2  =  4 * 0     +   2        0010
  11  =  4 * 2     +   3        1011

  14  =  4 * 3     +   2        1110
   8  =  4 * 2     +   0        1000
   1  =  4 * 0     +   1        0001
   7  =  4 * 1     +   3        0111
```

The KNOBDIR array holds values to deduct the rotating direction:

- –1 for the click's where the rotor position was decremented; rotated CCW
- 1 for the clicl's where the rotor position was incremented; rotated CW
- 0 in all other cases; a position change from 0 to 3 is impossible for instance

```
const int8_t KNOBDIR[] = {
      0, -1,  1,  0,
      1,  0,  0, -1,
     -1,  0,  0,  1,
      0,  1, -1,  0  };
```

Updating the position counter of the rotary encoder now is the result of a simple formula:

```
knobPosition += KNOBDIR[curState | (oldPos << 2)];
```

Every time the current state is equal to 3 the rotary encoder is in the next valid position and a new official position can be made available by dividing the internal counter by 4 (or shifting right by 2):

```
if (curState == 3) knobPositionExtern = (knobPosition >> 2);
```

The resulting implementation now is very small and obviously very fast. Also situations where the knob is not turned completely from one official position to another is handled correctly.

The only thing that is critical is that every state change has to be detected and the check routine implementing this functionality has to be called often at minimum once per state change.

The Return

Most implementations I found calculate an integer number for the position of the knob others return a "right" or "left". I find the support for a [counter] position value very useful in my projects and therefore stay with this approach.

To read the current position just use the getPosition() function.

I also added the possibility to change the position from within a sketch by calling setPosition(newPosition); you can use that for relative movement detection.

Using Polling

The simplest approach for checking the rotary encoder state is to poll the signals as often as you can. Just call the tick() function as often as you can and then use the position value for your needs.

The SimplePollRotator example just does this and prints out the current position when it has changed.

Using interrupts

Constantly polling the current state of the rotary encoder is sometimes a void time and resource consumption. Especially when dealing with other topics in your sketch that can take a while you might not have the time for polling.

A good approach is to use the pin change interrupt feature available in the ATMega processors. The Arduino environment has no direct support for this kind of interrupts so programming an Interrupt Service Routine (ISR) is required.

It is important to keep ISR implementations as small and fast as possible. The only thing that has to be done is to call the tick() function. The logic of your sketch should be implemented in the loop function.

Here is the implementation for the pin change interrupt in the case you have used the A2 and A3 lines for the encoder.

Enable the Pin Change Interrupt 1 in general that covers the Analog input pins or Port C:

```
PCICR |= (1 << PCIE1);
```

Now enable the interrupt for pin 2 and 3 of Port C:

```
PCMSK1 |= (1 << PCINT10) | (1 << PCINT11);
```

Now the Interrupt Service Routine only has to call the tick function:

```
ISR(PCINT1_vect) {
  encoder.tick(); // just call tick() to check the state.
}
```

You can also find this in the InterruptRotator example.

The Examples

SimplePollRotator

This example checks the state of the rotary encoder in the loop() function. The current position is printed on output when changed.

SimplePollRotatorLCD

The same example as SimplePollRotator but outputs to an LCD display. You may need to include another LCD library when not using the LiquidCrystal_PCF8574 library. (LCD using SPI)

InterruptRotator

This example checks the state of the rotary encoder by using interrupts as described above.

In this example, when reaching position 66 the output will freeze for 6.6 seconds. The further turned positions will be recognized anyway because the interrupt still works in the background.

The output is correct 6.6 seconds later.

Links

Some discussions on simple rotary approaches:

- http://playground.arduino.cc/Main/RotaryEncoders

A datasheet of a typical rotary encoder:

- http://www.hobbytronics.co.uk/datasheets/TW-700198.pdf

The gray code explained:

- http://en.wikipedia.org/wiki/Gray_code.

Some other rotary decoder implementations:

- https://github.com/brianlow/Rotary/blob/master/Rotary.cpp
- http://www.buxtronix.net/2011/10/rotary-encoders-done-properly.html

IO Performance:

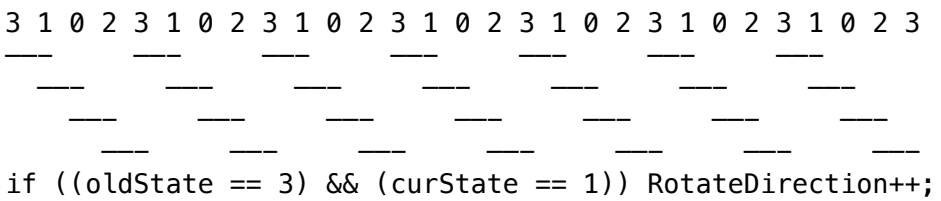
- http://www.instructables.com/id/Arduino-is-Slow-and-how-to-fix-it/
- http://forum.arduino.cc/index.php/topic,68656.0.html
- http://jeelabs.org/2010/01/06/pin-io-performance/
- http://garretlab.web.fc2.com/en/arduino/inside/arduino/wiring_digital.c/digitalRead.html

Interrupts

- http://playground.arduino.cc/Main/PcInt

More:

- http://www.atmel.com/Images/doc8109.pdf




```
if ((oldState == 3) && (curState == 1)) RotateDirection++;
else
if ((oldState == 1) && (curState == 0)) RotateDirection++;
else
if ((oldState == 0) && (curState == 2)) RotateDirection++;
} else
if ((oldState == 2) && (curState == 3)) RotateDirection++;
else
```

```
if ((oldState == 3) && (curState == 2)) RotateDirection--;
else
if ((oldState == 2) && (curState == 0)) RotateDirection--;
else
if ((oldState == 0) && (curState == 1)) RotateDirection--;
else
if ((oldState == 1) && (curState == 3)) RotateDirection-;
```

Brian Low
The operation is a careful arrangement of conductive tracks. There are mostly 3 pins: two for the "bit" outputs, and a common (wiper). As the encoder turns, the bit pins change according to a Gray Code sequence as follows:

Position	Bit A	Bit B
0	0	0
1/4	1	0
1/2	1	1
3/4	0	1
1	0	0

The basic way of decoding these is to watch for which bits changes.
For example:
Change from "00" to "10" indicates one direction: right
Change from "00" to "01" indicates the other direction; left.

The Gray code actually follows a simple state machine. At any given state, it can only change to one of two other values. Let's take a look at the state table again:

Position	Bit A	Bit B
0	0	0
1/4	1	0
1/2	1	1
3/4	0	1
1	0	0

For example:
At position 1/4, the only valid next states are either 00, or 11;
At position 1/2, the only valid next states are either 10, or 01;
At position 3/4, the only valid next states are either 11, or 00;

Any other state is invalid and should be ignored. The algorithm should know this.