SwitchPack LIBRARY

TUTORIAL

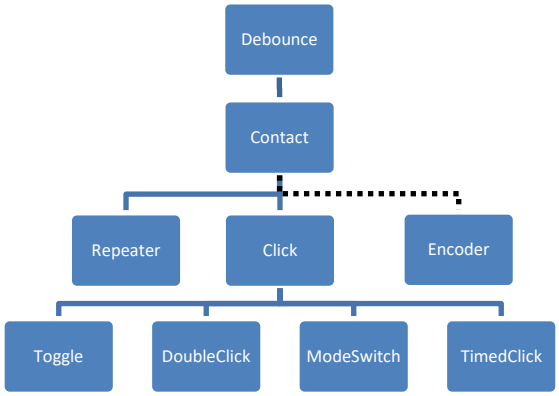
What is the SwitchPack Library?

The SwitchPack Library is a package of 9 Classes designed to easily use switches in different contexts.

1. **Debounce:** This is a super-fast switch debouncer and EMF filter. All the other classes are debounced by this class.
2. **Contact:** This Class will debounce the switch and can answer the following questions:
   Is it .open()?
   Is it .closed()?
   Did it just .rose()? (Just saw the signal rising)
   Did it just .fell()? (Just saw the signal falling)
   All the other classes inherits those methods from Contact
3. **Click:** This Class will debounce the switch and can answer the following question:
   Has it .clicked()? (closed and re-opened)
4. **DoubleClick:** This Class will debounce the switch and can answer the following question:
   What is the .clickCount()?
   .setLimit() will adjust the time limit to qualify a double-click valid.
   .setMaxClicks() will permit to validate triple-clicks, quadruple-clicks,...
5. **Toggle:** This Class will debounce the switch and check if it has been clicked. If it did, its status will toggle between ON and OFF
   .readStatus() reads the switch and return its status.
   .getStatus() returns the current switches' status
   It is also possible, at any time to .setStatus()
6. **TimedClick:** This Class has 3 chronometers. It can answer the following questions:
   When the switch .wasLastRead()?.The time when it was read.
   What was the .clickTime()? (How long was it pressed?)
   What is the .timeSinceLastClick()? (Between now and the time it was last released)
7. **Repeater:** This Class operates exactly as the keys on the keyboard. The first action is taken immediately after the switch is closed. After some time, the action is repeated periodically. You know that it is time to take action with this question:
   Is a .repeatRequired()?
   .setStart() allows to specify the time required after the switch is closed to start repeating
   .setBurst() allows to set the time to wait between repeats.
8. **ModeSwitch:** This class has a mode counter. Each click increases the mode, up until the switch is in its last mode. The next click will revert to mode 0.
   Ex.: For a 4 modes switch, successive clicks will place the switch in modes {0, 1, 2, 3, 0, 1,...}
   .readMode() reads the switch and returns its new mode.
   .getMode() returns the current mode without reading the switch.
   It can also automatically .resetAfter(sometime)

9. **Encoder:** This Class uses two Contact class objects to read a hand turned rotary quadrature encoder.
   .getRotation() reads the encoder and returns zero if it was not turned. It will return CW (1) or CCW (-1) if it was.
   It has also a counter that increases and decreases when the encoder is turned:
   .getCount() reads the encoder and returns the current count.
   .setCount() allows to set the counter to any value
   .resetCount() resets the counter to zero.

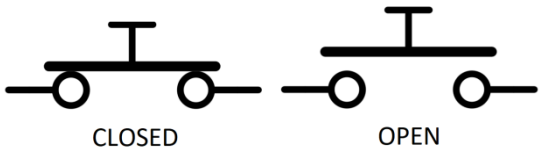
The following diagram shows which Class inherits from which Class. The Encoder Class is special since it **uses** two Contact class objects rather than **inheriting** form this class.



About switches

Anatomy of a mechanical switch

A mechanical switch (opposed to an optical switch) is made of two (or more) parts that can conduct electricity. One part can be moved to make a contact with the other one, thus conduct electricity. We say that the switch is **closed**. It can also be moved so that there is no contact with the other, thus not conducting electricity. We say that the switch is **open.**

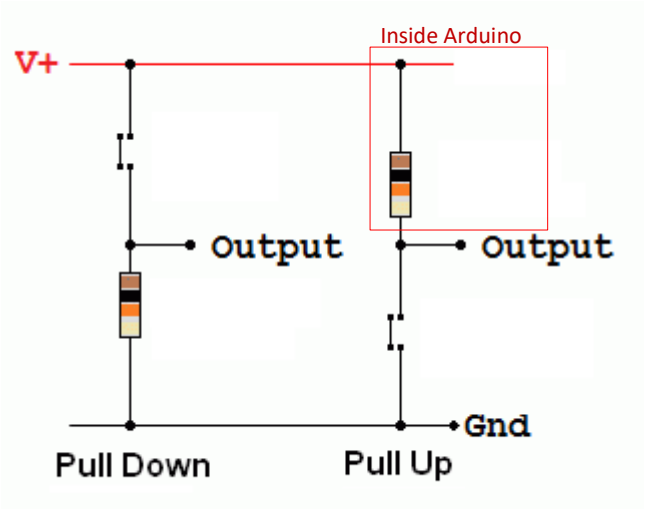


Connecting a switch to an Arduino

Arduinos' digital pins can report if there is current flowing thru them or not. To know if there is current on pin number 4, we call:

```
digitalRead(4);
```

If there is current flowing, it will return HIGH, and if not, it will return LOW.
There are two ways to connect a switch to an Arduino. They are called PULLUP and PULLDOWN.



On the left, we have a switch that is connected in PULLDOWN mode. When the switch is **open**, the resistor that is connected to ground makes sure that the output pin stays **LOW**. When the switch is **closed**, current from V+ flows thru the switch, bringing the output pin **HIGH**.

On the right, we have a switch that is connected in PULLUP mode. When the switch is **open**, current will flow thru the resistor to the output pin, bringing it **HIGH.** When the switch is **closed**, the output pin is tied to ground, bringing it **LOW**.

In PULLDOWN mode, an **open** switch brings the pin **LOW,** and a **closed** switch brings the pin **HIGH.**
In PULLUP mode, an **open** switch brings the pin **HIGH,** and a **closed** switch brings the pin **LOW.**

If we opt for the PULLDOWN mode, we will have to supply the resistor. 10KΩ is OK. In the setup part of our sketch, we will initialise the pin with the following line:

```
pinMode(2, INPUT);
```

If we opt for the PULLUP mode, we don't have to supply the resistor, since all Arduino's pins have pullup resistors that we can activate in the setup part of our sketch with the following line:

```
pinMode(2, INPUT_PULLUP);
```

Using SwitchPack[1]

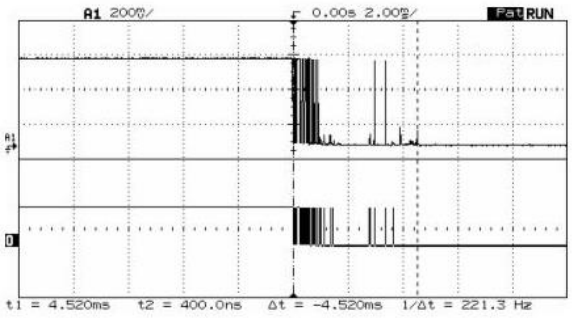
To use any of the classes that SwitchPack offers, we need to include it at the beginning of our sketch.

```
#include <SwitchPack.h>
```

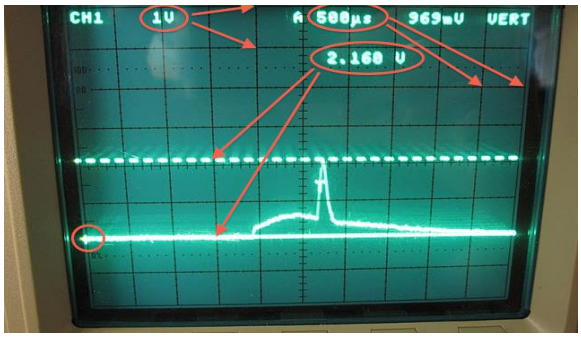
[1] You can find, in Appendix A, a setup that will allow you to try all the different classes of this Library

The Debounce Class

So, a switch is either open or closed? Most of the times it is, but at the moment it is pressed or released (or switched on or off) the transition is not so smooth. The switch bounces on and off several times before settling to its new state.



Here is an example of what happens when a switch is released. As long as the switch was pressed, the signal was HIGH. When it was released, it began bouncing between HIGH and LOW for a while, until it settled to the LOW state. The top diagram shows the real voltages going thru the switch, and the bottom diagram shows what a call to digitalRead() would return.



Another problem that can arise is that when the current flows thru the wires, it can be perturbed by electromagnetic fields (EMF). Those fields are everywhere. Even our fingers can generate some. We all experienced taking a shock touching some metal when the air is dry (especially when standing on a carpet). Those can be transmitted to the wires and bring a signal from high to low (and vice versa) easily if they are not properly shielded.

The Debounce class is designed to get rid of all those disturbances. And it is really fast

To debounce any pin of your setup, you only need to instantiate one object:

```
Debounce debouncer;
```

Once the object has been instantiated, you can debounce any switch with this line (here pin 2):

```
byte aSwitch = debouncer.debouncePin(2);
```

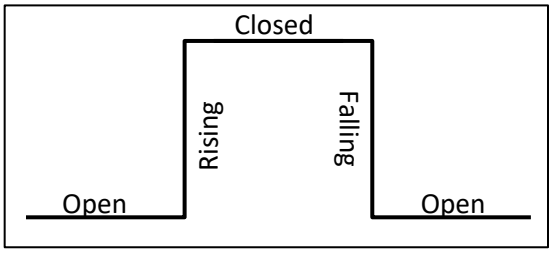
Remember:

In PULLDOWN mode, an **open** switch brings the pin **LOW,** and a **closed** switch brings the pin **HIGH.**
In PULLUP mode, an **open** switch brings the pin **HIGH,** and a **closed** switch brings the pin **LOW.**

In the examples directory, there is a sketch that debounces a switch with a pulldown resistor (DebouncePulldown.ino), and another one for a switch that uses the internal pullup resistor (DebouncePullup.ino).

The Contact Class

When a button is pressed and released, its signal goes thru 4 phases:



Open: The voltage is 0

Rising: The voltage rises from 0 to Vcc

Closed: The voltage is at Vcc

Falling: The voltage falls from Vcc to 0

Notice that we don't see the bouncing anymore. They are cleaned by the Debounce class.

To use a switch with the Contact class, we have to instantiate it:

```
Contact aSwitch(6, PULLDOWN);
Contact aSwitch(2, PULLUP);
Contact aSwitch(2); //defaults as PULLUP
```

In the setup() part of the sketch, we have to make it begin: It will do the appropriate `pinMode(x, INPUT)` or `pinMode(x, INPUT_PULLUP)` for us

```
aSwitch.begin();
```

Now that the switch knows in which mode it is connected, it can take care of translating HIGH and LOW of either mode, to a more exact description of the state that it is presently in:

```
if (aSwitch.closed()) { Do things; }
if (aSwitch.open()) { Do things; }
```

There are times when we want to do special things if we detect the rising or falling edge of the signal. (Some examples will show that in other classes.) We can do that with:

```
if (aSwitch.closed()) {
  if (aSwitch.rose()) { Do things; }
  Do other things;
}
```

We can also:

```
if (aSwitch.open()) {
  if (aSwitch.fell()) { Do things; }
  Do other things;
}
```

If we are only interested in the rising or falling edge, we can:

```
if (a.Switch.closed() && aSwitch.rose()) { Do things; }
or
if (a.Switch.open() && aSwitch.fell()) { Do things; }
```

In the examples directory, there are 3 sketches that demonstrate how Contacts works:

- (Contact.ino) uses both a switch in a pullup configuration and another one in a pulldown configuration to show that either one will respond correctly to .closed(), no matter which one is pressed.
- (ContactSpeed.ino) highlights the fact that .closed() and .open() are extremely fast, and that we will need the Click class if we need the sketch to respond only once to a press of a button. We can see how many times .closed() respond when we try to press and release the button really, really fast. (.open() is just a few nanoseconds slower). It uses the Serial monitor. Don't forget to open it.
- (SwitchCycles.ino) uses 3 LEDs to show the four states of a switch:
    - Open : All LEDs are off;
    - Rising: LED 11 flashes;
    - Closed: LED 13 is on;
    - Falling: LED 12 flashes.

The Click Class

The Click class inherits from the Contact class. So anything a Contact object can do, a Click object can also do.

The Contact class is well suited to control electronic parts. They can react really fast. Using a contact switch like the one on the left to stop a stepper motor before it bangs into something is a good example of when a Contact object should be used.

But when human hands get involved, Arduino is too fast. We need to have a human way to fire an action. The most common way to do so is the click method. When the switch is pressed and released, we have clicked. (Click returns true on the falling edge of the signal, false otherwise)

We create a Click object just the same way that we did for a Contact object:

```
Click aSwitch(2, PULLUP)   //or PULLDOWN
void setup() {
  aSwitch.begin()
}
```

To find out if we need to do some action, we write:

```
if (aSwitch.clicked()) { Do things; }
```

In the examples directory, there is a sketch (Click.ino) that demonstrates how to use a Click object.

There is also another sketch (ArrayOfClicks.ino) that shows how to setup many Click objects.

The DoubleClick Class

The DoubleClick class inherits from the Click class. So anything a Click object can do, a DoubleClick object can also do. And that includes what a Contact object can do.

To double-click, is to click the button twice within a certain time limit. So, we will need to specify the time we give the user to double-click and consider it valid. If we consider that half a second is valid, then we will specify 500 milliseconds. Of course, a DoubleClick object will also report if the switch was only clicked once.

We instantiate a DoubleClick object with the following lines:

```
DoubleClick aSwitch(2, PULLDOWN, 500);
DoubleClick aSwitch(2, PULLUP, 500);
DoubleClick aSwitch(2, 500)   //If in PULLUP mode
```

In the setup() section of the sketch, we add:

```
aSwitch.begin();
```

To take action according to how the switch was pressed we write:

```
Switch(aSwitch.clickCount()) {
  Case 1: { Do things; break; }
  Case 2: { Do things; break; }
}
```

DoubleClick can do more than double-click. It can triple-click, quadruple-click… To change the maximum allowable clicks per reads, in the Setup() section of our sketch, after the aSwitch.begin() instruction, we write:

```
aSwitch.setMaxClicks(3);
```

In the examples directory, there is a sketch (DoubleClick.ino) that demonstrates how to use a DoubleClick object. You can also try the (TripleClick.ino) sketch.

The Toggle Class

The Toggle class inherits from the Click class. So anything a Click object can do, a Toggle object can also do. And that includes what a Contact object can do.

The toggle class allows us to transform a momentary button into an on/off switch. Pressing the button once will turn the switch on; pressing it another time will turn it off.

We create a Toggle object just the same way that we did for a Click object:

```
Toggle aSwitch(2, PULLUP)    //or PULLDOWN
void setup() {
  aSwitch.begin()
}
```

The method that follows reads the pin and, if necessary updates the Toggle status:

```
byte theStatus = aSwitch.readStatus();
```

This method just returns the current status of Toggle, without reading the pin:

```
byte theStatus = aSwitch.getStatus();
```

Finally, this method allows us to set the Toggle to a new status:

```
aSwitch.setStatus(ON);
or
aSwitch.setStatus(OFF);
```

In the examples directory, there is a sketch (Toggle.ino) that demonstrates how to use a Toggle object.

The TimedClick Class

The TimedClick class inherits from the Click class. So anything a Click object can do, a TimedClick object can also do. And that includes what a Contact object can do.

This class just adds three chronometers to the Click class:

- The first one remembers when the switch was last read;
- The second one remembers the last time the pin rose;
- The third one remembers the last time the pin fell;

We create a TimedClick object just the same way that we did for a Click object:

```
TimedClicked aSwitch(2, PULLUP)    //or PULLDOWN
void setup() {
  aSwitch.begin()
}
```

To find out if we need to do some action, we write:

```
if (aSwitch.clicked()) { Do things; }
```

To find out when the switch was last read, we write:

```
unsigned long wasRead = aSwitch.wasLastRead();
```

To find out how long the switch was closed, we write:

```
unsigned long timeClosed = aSwitch.clickTime();
```

To find out how long ago the switch was released, we write:

```
Unsigned long released = aSwitch.timeSinceLastClick();
```

In the examples directory, there is a sketch (TimeClicked.ino) that demonstrates how to use a TimeClick object.

The Repeater Class

The Repeater class inherits from the Contact class. So anything a Contact object can do, a Repeater object can also do.

A Repeater object acts like the keys of a computer keyboard. Instead of just sending characters, it can spark any action that you want (including sending characters of course).

The first action is taken immediately after the switch is closed. After a predetermined time, if the switch is still closed, the action is repeated over and over at a predetermined speed.

We instantiate a Repeater object with the following lines:

```
Repeater aSwitch(2, PULDOWN, 500, 50);
Repeater aSwitch(2, PULLUP, 500, 50);
Repeater aSwitch(2, 500, 50);  //if in PULLUP mode
```

500 mean that the action will start repeating after 500 milliseconds (½ second).
50 mean that the action will be repeated every 50 milliseconds (1/20th second)

In the setup() section of the sketch we add the following line:

```
aSwitch.begin();
```

To know if we need to act, we write:

```
if (aSwitch.repeatRequired()) { Do things; }
```

We can also change the start delay and the repeat delay with those lines:

```
aSwitch.setStart(1000);
aSwitch.setBurst(25);
```

In the examples directory, there is a sketch (Repeater.ino) that demonstrates how to use a Repeater object.

The ModeSwitch Class

The ModeSwitch class inherits from the Click class. So anything a Click object can do, a ModeSwitch object can also do. And that includes what a Contact object can do.

A ModeSwitch object has a counter that changes when the switch is clicked. Each time the switch is clicked, the counter is increased by 1 until it meets a value that we can set. At that point, it is reset to 0.

The possible values always start at 0. Suppose that we want to have a switch that can take 4 different modes. Clicking the switch repeatedly, the ModeSwitch will return: 0, 1, 2, 3, 0, 1, 2…

To create a ModeSwitch object with four modes, we can write:

```
ModeSwitch aSwitch(2, PULLDOWN, 4);
ModeSwitch aSwitch(2, PULLUP, 4);
ModeSwitch aSwitch(2, 4);   //If in PULLUP mode
```

In the setup() section of the sketch we add the following line:

```
aSwitch.begin();
```

The following method will read the switch and return the current mode:

```
switch (aSwitch.readMode()) {
  case 0: { Do something; break; }
  case 1: { Do something; break; }
  case 2: { Do something; break; }
  case 3: { Do something; break; }
}
```

If we just want to know what the current mode is without reading the switch, we write:

```
byte theMode = aSwitch.getMode();
```

We can also have the ModeSwitch object to reset (return to 0) automatically after some delay in milliseconds:

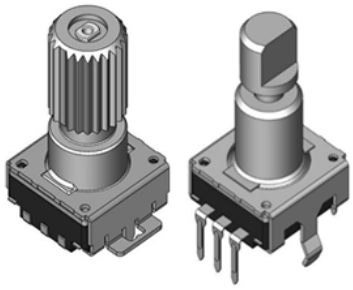
```
aSwitch.resetAfter(5000);
```

In the examples directory, there is a sketch (ModeSwitch.ino) that demonstrates how to use a ModeSwitch object. There is also (ModeWithReset.ino) that shows how the reset features works.

There is also a sketch (MiniKeyboard.ino) that uses an array of Repeater objects, a Contact object and a ModeSwitch object

The Encoder Class

This is the last class of the SwitchPack Library. It inherits from no other class. Instead, it uses two Contact objects.



A rotary encoder is a knob that can be turned clockwise and counter clockwise. Usually it has detents (or clicks) that you can feel (and sometimes hear). Contrary to a potentiometer, it can be turned in either direction indefinitely. It is also called a quadrature encoder.

Inside, it has two switches that opens and closes in a specific way that enables us to determine the direction the encoder has been turned.

Since an encoder has two switches, we create an Encoder object this way:

```
Encoder anEncoder(7, 8);
```

In the setup() section of the sketch, we will write:

```
anEncoder.begin();
```

If our switches are not wired in PULLUP mode, right after .begin() we can write:

```
anEncoder.setMode(PULLDOWN, PULLDOWN);
```

If we realize that the encoder only responds to every other click, we most likely have an encoder with 2 steps per click (as opposed to 4 steps per click, which is the most common). Then, in the setup() section of the sketch, after .begin(), we can write:

```
anEncoder.stepsPerClick(2);
```

To read the encoder, we will write:

```
int rotation = anEncoder.getRotation();
if (rotation == CW) { Do something; }
if (rotation == CCW) {Do something else; }
```

The Encoder object also has a counter that increases (if turned CW) and decreases (if turned CCW) automatically. To get the current count, we write:

```
int counter = anEncoder.getCount();
```

The counter can be set or reset with:

```
anEncoder.setCount(50);  //Sets the counter to 50
anEncoder.resetCount();  //Sets the counter to 0
```

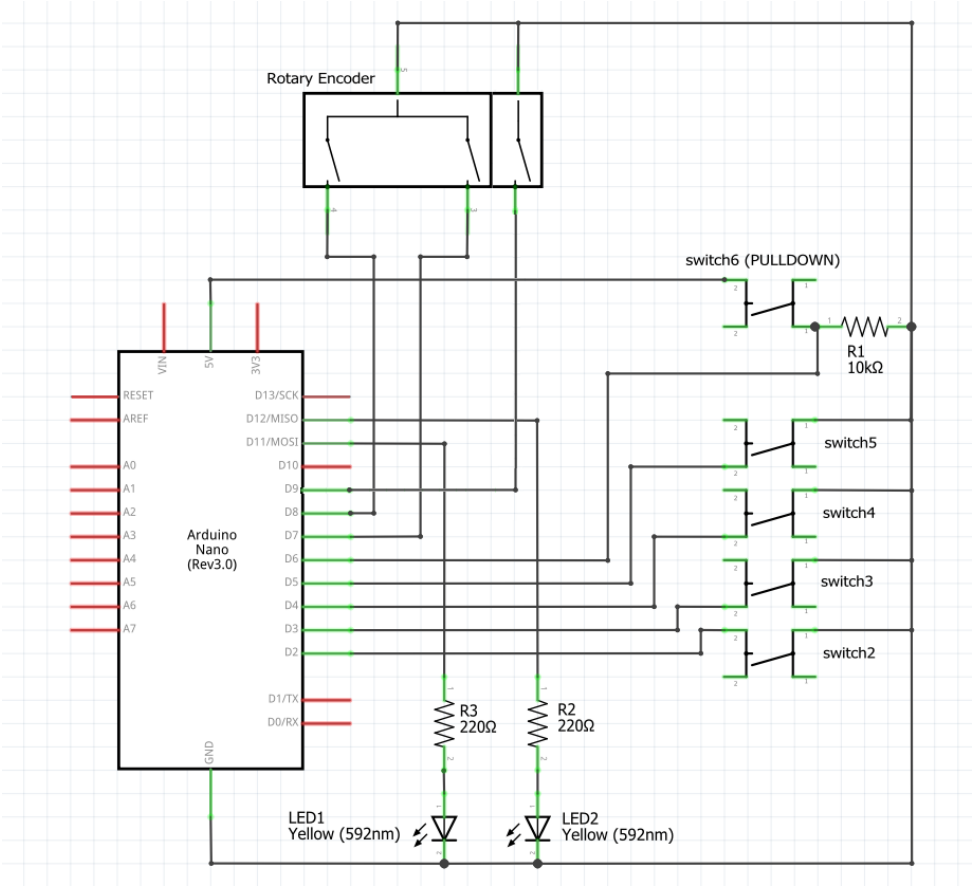
In the examples directory, there is a sketch (Encoder.ino) that demonstrates how to use an Encoder object.

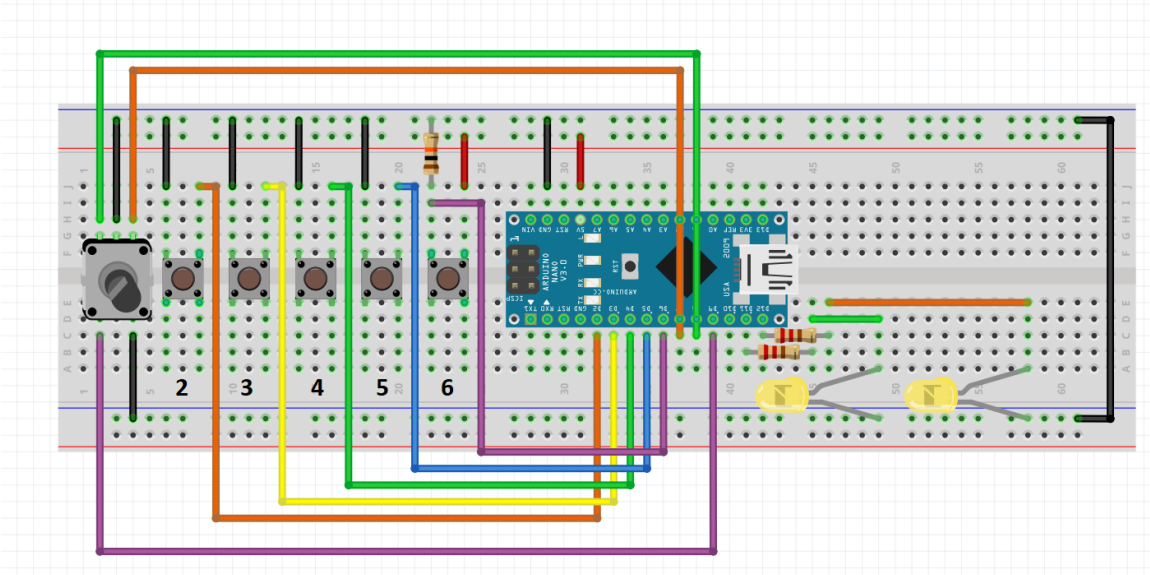
Some encoders come with a switch (you press the encoder to click). You can use that switch to do whatever you like. There is a sketch in the example directory that uses that switch as a ModeSwitch. (EncoderWithMode.ino).

I sincerely hope that you will find this SwitchPack Library useful for your projects.
Jacques Bellavance

APPENDIX A: Setup to try all the example sketches

Designed with Fritzing





Switches 2..5 are in PULLUP mode. Switch 6 is in PULLDOWN mode. LEDs have to be placed short pin to ground. The resistor on Switch 6 is 10KΩ. The two resistors for the LEDs are 220Ω.