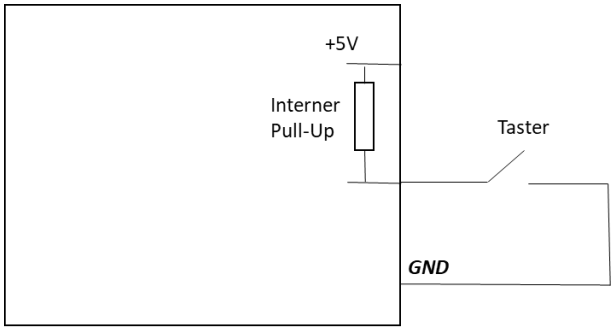
Lösung 03: Interrupts, EEPROM

Aufgabe 1: Externe Interrupts mit der Arduino Library

a) Aufgrund der Anforderung, dass der Zustand bei geschlossenem Taster HIGH sein soll, muss die rechte Schaltungsvariante verwendet werden. Ist der Taster offen, dann wäre der Zustand am Digital Pin 21 ohne weiteren Widerstand undefiniert. Es muss ein ***Pull-Down*** Widerstand hinzugefügt werden.

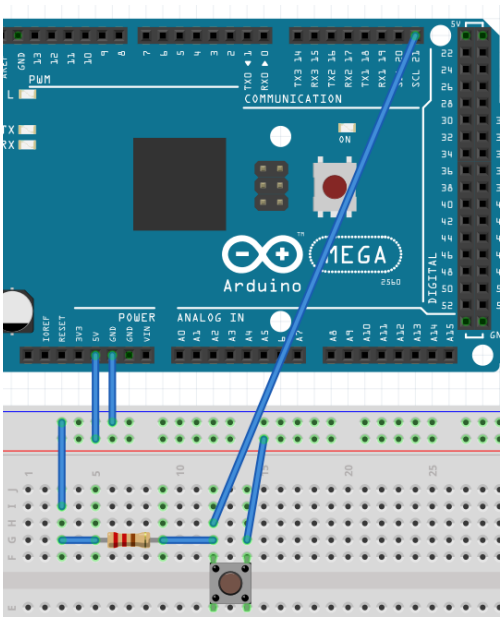
Weiterführender Hinweis: Die linke Variante ist beim ATmega2560 einfacher zu implementieren, da bereits ***interne Pull-Up Widerstände*** vorhanden sind. Man benötigt also keinen extra Widerstand. Die internen Pull-Up Widerstände sind normalerweise automatisch „eingeschaltet“, sobald man den betreffenden Pin als Eingang konfiguriert. Schreibt man ins PORT Register HIGH (LOW) während der entsprechende Pin als Eingang konfiguriert ist (Handbuch: 13.2.1), so aktiviert (deaktiviert) man den entsprechenden internen Pull-Up Widerstand.



Interner Pull-Up

b) Problematisch ist, dass das Kommando `delay(3000)` ein sogenanntes „Busy Waiting“ implementiert. Während des Wartens kann ein Drücken der Taste nicht erkannt werden. Somit zählt die Variable `counter` sicher zu wenige Tastendrücke. Es ist sogar recht unwahrscheinlich, dass ein Tastendruck überhaupt erkannt wird, da man genau die Pause zwischen den 3 Sekunden Wartezeit erwischen muss. .

c) Verkabelung, siehe Zeichnung.



d) Man schlägt im Pin Mapping (https://www.arduino.cc/en/Hacking/PinMapping2560) nach, dass der Digital Pin 21 mit dem Port D, Pin #0 (PD0) verbunden ist. Neben GPIO hat dieser Pin des Mikrocontrollers die Zweitfunktionen SCL (benötigt für Kommunikationsschnittstelle I2C, siehe später) und externer Interrupt INT0.

e) Wichtig ist, dass in der loop-Methode eigentlich nichts passiert. Dort findet nur Busy-Waiting statt. Man kann sich gut vorstellen, dass stattdessen der Mikrocontroller noch eine andere Aufgabe ausführt z.B. irgendetwas ansteuern muss und nur unterbrochen wird, wenn der Taster gedrückt wird (oder in realer Anwendung eine Lichtschranke unterbrochen wird).

Irritierend bei Interrupts ist, dass die Methode isr() scheinbar nie aufgerufen wird. Die Methode wird nämlich von der Hardware selbst aufgerufen. Man muss die Hardware nur passend konfigurieren, was in der setup()-Routine geschieht.

```
volatile int counter = 0;

void isr() {
   counter++;
}

void setup() {
   Serial.begin(9600);
   pinMode(21, INPUT);
   attachInterrupt(digitalPinToInterrupt(21), isr, RISING);
}

void loop() {
   Serial.println(counter);
   delay(3000); // do some other work
}
```

f) Es passiert (im Test des Dozenten) hin und wieder dass ein Tastendruck zweimal erkannt wird. Eine SW-Entprellung in einer ISR könnte man wie folgt machen. Bitte aber den folgenden Hinweis auf der Angabe beachten.

```
volatile int counter = 0;
volatile long time_prev_rising_edge;

void isr() {
      if (millis() - time_prev_rising_edge > 250) {
      // minimal time between 2 button switches 250 ms
      counter++;
      }
      time_prev_rising_edge = millis();
}

void setup() {
      Serial.begin(9600);
      pinMode(21, INPUT);
      time_prev_rising_edge = millis();
      delay(250);
      attachInterrupt(digitalPinToInterrupt(21), isr, RISING);
}

void loop() {
```

```
        Serial.println(counter);
        delay(3000); // do some other work
}
```

Aufgabe 2: Externe Interrupts mit der AVR-Libc

Die AVR-Libc definierte feste Makros für die verschiedenen Interrupttypen, hier ISR (INT0_vect). Die Liste findet man hier: http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html

Auch hier wird die ISR scheinbar nie aufgerufen, siehe Kommentar oben.

```
#include "Arduino.h"

volatile int counter = 0;

void setup()  {
        Serial.begin(9600);
        // pin is by default an input pin, setting it as input is optional
        EIMSK |= (1 << INT0);                   // turn on INT0
        EICRA |= (1 << ISC00) | (1 << ISC01);  // set INT0 to trigger an rising edge
        sei();                                  // globally activate interrupts SREG
}

void loop() {
        Serial.println(counter);
        delay(3000); // do some other work

}

ISR (INT0_vect) {
        counter++;
}
```

Aufgabe 3: Theorie – Betriebsstundenzähler und Wear Leveling

a) Laut Datenblatt verträgt jede EEPROM-Zelle mindestens 100000 Schreibzyklen. Für mehr garantiert der Hersteller des Speichers nicht. Man muss deshalb darauf achten, dass man nicht versehentlich zu oft auf die gleiche Speicherzelle schreibt!
https://en.wikipedia.org/wiki/Wear_leveling

b) 20 Jahre lang jede Stunde schreiben ergibt: 20 * 365 Tage * 24 Stunden = 175200 Schreibzugriffe
Bei so einer langen Lebenszeit des Produktes sollte man ruhig eine Art Wear Leveling einsetzen. Im Embedded Bereich muss man sich als Programmiere selbst darum kümmern. Bei Flash-Festplatten übernehmen das HW-Controller.

c) Eine gute Erläuterung der Application Note gibt es hier:
https://stackoverflow.com/questions/10667491/is-there-a-general-algorithm-for-microcontroller-eeprom-wear-leveling/10669628
Zusatz: Im konkreten Fall würde es sogar ausreichen, nur den Parameter Buffer zu verwenden. Das geht aber nur, da ein Betriebsstundenzähler mit jedem Schreiben genau um 1 erhöht wird. Bei Anwendungen, die den Wert variabel erhöhen (z.B. man möchte sich merken, wann ein bestimmtes Ereignis das letzte Mal eingetreten ist und schreibt immer die „Uhrzeit“ bei Eintritt des Ereignisses in den EEPROM), geht das nicht.

d) Man wechselt sich beim Schreiben ab. Erst beim 7. Schreibvorgang schreibt man wieder die gleiche Speicherzelle -> 600 000 Schreibzyklen sind möglich.
Der aktuelle gültige Wert ist *c*. Als nächstes müsste man *d* beschreiben, da 98 + 1 != 93.
(siehe Link von c)