www.mikrocontroller.net

Entprellung

Mechanische Schalter prellen beim Ein- und Ausschalten, d.h sie schalten schnell aus und ein, verursacht durch mechanisches Vibrationen des Schaltkontaktes. Vereinfacht dargestellt sieht eine von einem Schalter oder Taster geschaltete Spannung beim Schalten wie folgt aus:

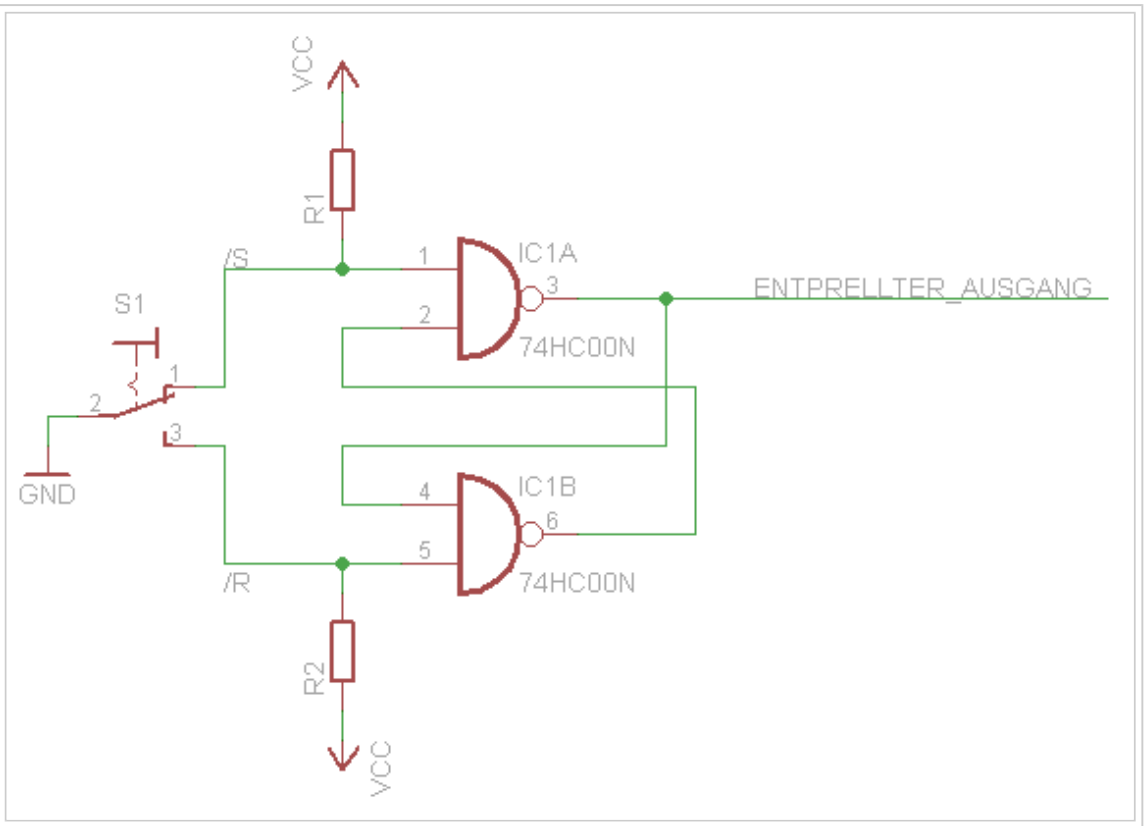
Für die Auswertung dieses unsauberen Signals gibt es verschiedene Ansätze:

Inhaltsverzeichnis

Hardwareentprellung

Wechselschalter

Für die Entprellung von Wechselschaltern (engl. Double Throw Switch) kann ein klassisches RS-Flipflop genutzt werden. Bei dieser Variante werden neben zwei NAND-Gattern nur noch zwei Pull-Up Widerstände benötigt.

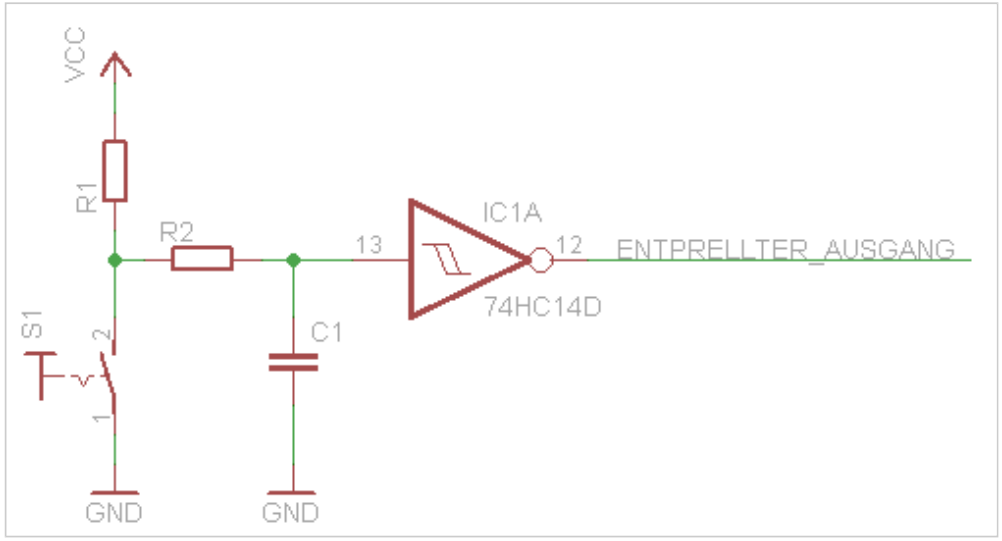


Taster entprellen mit NAND-RS-Flipflop

In der gezeigten Schalterstellung liegt an der Position /S der Pegel 0 an. Damit ist das Flipflop gesetzt und der Ausgang auf dem Pegel 1. Schließt der Schalter zwischen den Kontakten 2 und 3, liegt an der Postion /R der Pegel 0 an. Dies bedeutet, dass der Ausgang des Flipflops auf den Pegel 0 geht. Sobald der Schalter von einem zum anderen Kontakt wechselt, beginnt er in der Regel zu prellen. Während des Prellens wechselt der Schalter zwischen den beiden Zuständen "Schalter berührt Kontakt" und "Schalter ist frei in der Luft". Der Ausgang des Flipflops bleibt in dieser Prellzeit aber stabil, da der Schalter während des Prellens nie den gegenüberliegenden Kontakt berührt und das RS-Flipflop seinen Zustand allein halten kann. Die Prellzeit ist stark vom Schaltertyp abhängig und liegt zwischen 0,1 und 10ms. Die Dimensionierung der Widerstände ist relativ unkritisch. Als Richtwert können hier 100kOhm verwendet werden.

Einfacher Taster

Auch wenn das RS-Flipflop sehr effektiv ist, wird diese Variante der Entprellung nur selten angewendet. Grund dafür ist, dass in Schaltungen häufiger einfache Taster eingesetzt werden. Diese sind oft kleiner und preisgünstiger. Um einfache Taster (engl. Single Throw Switch) zu entprellen, kann ein einfacher RC -Tiefpass eingesetzt werden. Hierbei wird ein Kondensator über einen Widerstand je nach Schalterstellung auf- oder entladen. Das RC-Glied bildet einen Tiefpass, sodass die Spannung über den Kondensator nicht von einen Pegel auf den anderen springen kann.



Taster entprellen mit RC-Entpreller

Wenn der Schalter geöffnet ist, lädt sich der Kondensator langsam über die beiden Widerstände R_1 und R_2 auf V_{cc} auf. Beim Erreichen der Umschaltschwelle springt der Ausgang auf den Pegel 0. Wird der Schalter geschlossen, entlädt sich der Kondensator langsam über den Widerstand R_2. Demnach ändert sich der Ausgang des Inverters auf den Pegel 1. Während der Taster prellt, kann sich die Spannung über dem Kondensator nicht sprunghaft ändern, da das Auf- und Entladen eher langsam über die Widerstände erfolgt. Außerdem sind die Schaltschwellen für den Übergang LOW->HIGH und HIGH->LOW stark verschieden (Hysterese, siehe Artikel Schmitt-Trigger). Bei richtiger Dimensionierung der Bauelemente wird somit der Ausgang des Inverters prellfrei.

Zu beachten ist, dass der Inverter **unbedingt** einer mit Schmitt-Trigger Eingängen sein muss, weil bei Standard-Logikeingängen im Bereich von üblicherweise 0,8V - 2,0V der Ausgang nicht definiert ist. Als Inverter kann zum Beispiel der 74HC14 eingesetzt werden. Alternativ kann auch ein CD4093 eingesetzt werden. Hierbei handelt es sich um NAND-Gatter mit Schmitt-Trigger-Eingängen. Um aus einem NAND -Gatter einen Inverter zu machen, müssen einfach nur die beiden Eingänge verbunden werden oder ein Eingang fest auf HIGH gelegt werden.

Für eine geeignete Dimensionierung muss man etwas mit den Standardformeln für einen Kondensator jonglieren. Die Spannung über den Kondensator beim Entladen berechnet sich nach:

$$U_C(t) = U_0 \cdot e^{\frac{-t}{R_2 C_1}}$$

Damit der Ausgang des Inverters stabil ist, muss die Spannung über den Kondensator und damit die Spannung am Eingang des Inverters über der Spannung bleiben, bei welcher der Inverter umschaltet. Diese Schwellwertspannung ist genau die zeitabhängige Spannung über den Kondensator.

$$U_C(t) = U_{th}$$

Durch Umstellen der Formel ergibt sich nun:

$$R_2 = \frac{-t}{C_1 \cdot ln\left(\frac{U_{th}}{U_0}\right)}$$

Ein Taster prellt üblicherweise etwa 10ms. Zur Sicherheit kann bei der Berechnung der Widerstandes eine Prellzeit von 20ms angenommen werden. U_0 ist die Betriebsspannung also Vcc. Die Schwellwertspannung muss aus dem Datenblatt des eingesetzten Schmitt-Triggers abgelesen werden. Beim 74HC14 beträgt der gesuchte Wert 2,0V. Nimmt man für den Kondensator 1µF und beträgt die Betriebsspannung 5V, ergibt sich für den Widerstand ein Wert von etwa 22kOhm.

Wird der Schalter geöffnet, lädt sich der Kondensator nach folgender Formel auf:

$$U_C(t) = U_0 \cdot \left(1 - e^{\frac{-t}{(R_1+R_2)\cdot C_1}}\right)$$

Mit U_th=U_C ergibt das Umstellen nach (R_1+R_2):

$$R_1 + R_2 = \frac{-t}{C_1 \cdot ln\left(1 - \frac{U_{th}}{U_0}\right)}$$

Für die Schwellspannung lässt sich aus dem Datenblatt ein Wert von 2,3V ablesen. Mit diesem Wert und den Annahmen von oben ergibt sich für R_1+R_2 ein Wert von 32kOhm. Somit ergibt sich für R_1 ein Wert von etwa 10kOhm.

Anmerkung: Beim 74LS14 von Hitachi z. B. sind die oberen und unteren Schaltschwellwerte unterschiedlich. Es muss darauf geachtet werden, dass U_{th} beim Entladen die untere Schwelle und U_{th} beim Laden die obere Schwelle einnimmt.

Softwareentprellung

Bei Verwendung eines Mikrocontrollers kann man sich die zusätzliche Hardware sparen, da die Entprellung genauso gut in Software funktioniert. Dabei ist zu beachten, dass zusätzliche Rechenleistung und je nach Umsetzung auch Hardwareressourcen benötigt werden (z. B. Timer).

Flankenerkennung

Bei einem Taster gibt es insgesamt 4 Zustände:

- 1. war nicht gedrückt und ist nicht gedrückt
- 2. war nicht gedrückt und ist gedrückt (steigende Flanke)
- 3. war gedrückt und ist immer noch gedrückt
- 4. war gedrückt und ist nicht mehr gedrückt (fallende Flanke)

Diese einzelnen Zustände lassen sich jetzt bequem abfragen/durchlaufen. Die Entprellung geschieht dabei durch die ganze Laufzeit des Programms.

Die Taster werden hierbei als Active-Low angeschlossen um die internen Pull-Ups zu nutzen.

Diese Routine gibt für den Zustand "steigende Flanke" den Wert "1" zurück, sonst "0"

```
#define TASTERPORT PINC
#define TASTERBIT PINC1

char taster(void)
{
    static unsigned char zustand;
    char rw = 0;

    if(zustand == 0 && !(TASTERPORT & (1<<TASTERBIT)))   //Taster wird gedrueckt (steigende Flanke)
    {
        zustand = 1;
        rw = 1;
    }
    else if (zustand == 1 && !(TASTERPORT & (1<<TASTERBIT)))   //Taster wird gehalten
    {
         zustand = 2;
         rw = 0;
    }
    else if (zustand == 2 && (TASTERPORT & (1<<TASTERBIT)))   //Taster wird losgelassen (fallende Flanke)
    {
        zustand = 3;
        rw = 0;
    }
    else if (zustand == 3 && (TASTERPORT & (1<<TASTERBIT)))   //Taster losgelassen
    {
        zustand = 0;
        rw = 0;
    }

    return rw;
}
```

Warteschleifen-Verfahren

Alle mechanischen Kontakte, sei es von Schaltern, Tastern oder auch von Relais, haben die unangenehme Eigenschaft zu prellen. Dies bedeutet, dass beim Schalten eines Kontaktes derselbe nicht direkt den endgültigen Zustand aufweist, sondern zwischenzeitlich möglicherweise mehrfach ein- und ausschaltet.

Soll nun mit einem Mikrocontroller gezählt werden, wie oft ein solcher Kontakt geschaltet wird, muss das Prellen des Kontakts berücksichtigt werden, da sonst pro Schaltvorgang möglicherweise mehrfache Impulse gezählt werden. Diesem Phänomen muss beim Schreiben des Programms unbedingt Rechnung getragen werden.

Beim folgenden einfachen Beispiel für eine Entprellung ist zu beachten, dass der AVR im Falle eines Tastendrucks 200ms wartet, also brach liegt. Bei zeitkritische Anwendungen sollte man ein anderes Verfahren nutzen (z. B. Abfrage der Tastenzustände in einer Timer-Interrupt-Service-Routine).

```
#include <avr/io.h>
#include <inttypes.h>
#ifndef F_CPU
#warning "F_CPU war noch nicht definiert, wird nun mit 3686400 definiert"
#define F_CPU 3686400UL     /* Quarz mit 3.6864 Mhz  */
#endif
#include <util/delay.h>     /* bei alter avr-libc: #include <avr/delay.h> */

/* Einfache Funktion zum Entprellen eines Tasters */
inline uint8_t debounce(volatile uint8_t *port, uint8_t pin)
{
    if ( !(*port & (1 << pin)) )
    {
        /* Pin wurde auf Masse gezogen, 100ms warten   */
        _delay_ms(50);   // Maximalwert des Parameters an _delay_ms
        _delay_ms(50);   // beachten, vgl. Dokumentation der avr-libc
        if ( *port & (1 << pin) )
        {
            /* Anwender Zeit zum Loslassen des Tasters geben */
            _delay_ms(50);
            _delay_ms(50);
            return 1;
        }
    }
    return 0;
}

int main(void)
{
    DDRB &= ~( 1 << PB0 );        /* PIN PB0 auf Eingang Taster)  */
    PORTB |= ( 1 << PB0 );        /* Pullup-Widerstand aktivieren */
    ...
    if (debounce(&PINB, PB0))
    {
        /* Falls Taster an PIN PB0 gedrueckt     */
        /* LED an Port PD7 an- bzw. ausschalten: */
        PORTD = PORTD ^ ( 1 << PD7 );
    }
```

```
    ...
}
```

Die obige Routine hat leider mehrere Nachteile:

- sie detektiert nur das Loslassen (unergonomisch)
- sie verzögert die Mainloop immer um 100ms bei gedrückter Taste
- sie verliert Tastendrücke, je mehr die Mainloop zu tun hat.

Eine ähnlich einfach zu benutzende Routine, aber ohne all diese Nachteile findet sich im Forenthread Entprellung für Anfänger

Der *DEBOUNCE* Befehl in dem BASIC-Dialekt BASCOM für AVR ist ebenfalls nach dem Warteschleifen-Verfahren programmiert. Die Wartezeit beträgt standardmäßig 25 ms, kann aber vom Anwender überschrieben werden. Vgl. BASCOM Online-Manual zu DEBOUNCE.

Eine C-Implementierung für eine Tastenabfrage mit Warteschleife ist im Artikel AVR-GCC-Tutorial: IO-Register als Parameter und Variablen angeben.

Der Nachteil dieses Verfahrens ist, dass der Controller durch die Warteschleife blockiert wird. Günstiger ist die Implementierung mit einem Timer-Interrupt.

Warteschleifenvariante mit Maske und Pointer (nach Christian Riggenbach)

Hier eine weitere Funktion, um Taster zu entprellen: Durch den zusätzlichen Code kann eine Entprellzeit von durchschnittlich 1-3ms (mindestens 8*150µs = 1ms) erreicht werden. Grundsätzlich prüft die Funktion den Pegel der Pins auf einem bestimmten Port. Wenn die/der Pegel 8 Mal konstant war, wird die Schleife verlassen. Diese Funktion kann sehr gut eingesetzt werden, um in einer Endlosschleife Taster anzufragen, da sie, wie erwähnt, eine kurze Wartezeit hat.

```
void entprellung( volatile uint8_t *port, uint8_t maske ) {
  uint8_t   port_puffer;
  uint8_t   entprellungs_puffer;

  for( entprellungs_puffer=0 ; entprellungs_puffer!=0xff ; ) {
    entprellungs_puffer<<=1;
    port_puffer = *port;
    _delay_us(150);
    if( (*port & maske) == (port_puffer & maske) )
      entprellungs_puffer |= 0x01;
  }
}
```

Die Funktion wird wie folgt aufgerufen:

```
// Bugfix 20100414
// http://www.nongnu.org/avr-libc/user-manual/FAQ.html#faq_port_pass
entprellung( &PINB, (1<<PINB2) ); // ggf. Prellen abwarten
if( PINB & (1<<PINB2) )           // dann stabilen Wert einlesen
{
  // mach was
}
else
{
  // mach was anderes
}
```

Als Maske kann ein beliebiger Wert übergeben werden. Sie verhindert, dass nichtverwendete Taster die Entprellzeit negativ beeinflussen.

Debounce-Makro von Peter Dannegger

Peter Dannegger hat in "Entprellen für Anfänger" folgende vereinfachtes Entprellverfahren beschrieben. Das Makro arbeitet in der Originalversion mit *active low* geschalteten Tastern, kann aber einfach für *active high* geschaltete Taster angepasst werden (Tasty Reloaded).

```
/************************************************************************/
/*                                                                      */
/*                      Not so powerful Debouncing Example              */
/*                      No Interrupt needed                             */
/*                                                                      */
/*              Author: Peter Dannegger                                 */
/*                                                                      */
/************************************************************************/
// Target: ATtiny13

#include <avr/io.h>
#define F_CPU 9.6e6
#include <util/delay.h>


#define debounce( port, pin )                                   \
({                                                              \
  static uint8_t flag = 0;     /* new variable on every macro usage */  \
  uint8_t i = 0;                                                \
                                                                \
  if( flag ){                  /* check for key release: */     \
    for(;;){                   /* loop ... */                   \
      if( !(port & 1<<pin) ){  /* ... until key pressed or ... */     \
        i = 0;                 /* 0 = bounce */                 \
        break;                                                  \
      }                                                         \
      _delay_us( 98 );         /* * 256 = 25ms */               \
      if( --i == 0 ){          /* ... until key >25ms released */     \
        flag = 0;              /* clear press flag */           \
        i = 0;                 /* 0 = key release debounced */  \
        break;                                                  \
```

```
      }                                                          \
    }                                                            \
  }else{                        /* else check for key press: */  \
    for(;;){                    /* loop ... */                   \
      if( (port & 1<<pin) ){    /* ... until key released or ... */ \
        i = 0;                  /* 0 = bounce */                 \
        break;                                                   \
      }                                                          \
      _delay_us( 98 );          /* * 256 = 25ms */               \
      if( --i == 0 ){           /* ... until key >25ms pressed */ \
        flag = 1;               /* set press flag */             \
        i = 1;                  /* 1 = key press debounced */    \
        break;                                                   \
      }                                                          \
    }                                                            \
  }                                                              \
  i;                            /* return value of Macro */      \
})

/*
   Testapplication
 */
int main(void)
{
  DDRB  &= ~(1<<PB0);
  PORTB |=   1<<PB0;
  DDRB  |=   1<<PB2;
  DDRB  &= ~(1<<PB1);
  PORTB |=   1<<PB1;
  DDRB  |=   1<<PB3;
  for(;;){
    if( debounce( PINB, PB1 ) )
      PORTB ^= 1<<PB2;
    if( debounce( PINB, PB0 ) )
      PORTB ^= 1<<PB3;
  }
}
```

Wenn das Makro für die gleiche Taste (Pin) an mehreren Stellen aufgerufen werden soll, muss eine Funktion angelegt werden, damit beide Aufrufe an die gleiche Zustandsvariable *flag* auswerten [1]:

```
// Hilfsfunktion
uint8_t debounce_C1( void )
{
  return debounce(PINC, PC1);
}

// Beispielanwendung
int main(void)
{
  DDRB  |=   1<<PB2;
  DDRB  |=   1<<PB3;
  DDRC  &= ~(1<<PC1);
  PORTC |=   1<<PC1; // Pullup für Taster

  for(;;){
```

```
    if( debounce_C1() )  // nicht: debounce(PINC, PC1)
      PORTB ^= 1<<PB2;
    if( debounce_C1() )  // nicht: debounce(PINC, PC1)
      PORTB ^= 1<<PB3;
  }
}
```

Timer-Verfahren (nach Peter Dannegger)

Grundroutine (AVR Assembler)

Siehe dazu: Forum

Vorteile

- besonders kurzer Code
- schnell

Außerdem können 8 Tasten (aktiv low) gleichzeitig bearbeitet werden, es dürfen also alle exakt zur selben Zeit gedrückt werden. Andere Routinen können z. B. nur eine Taste verarbeiten, d.h. die zuerst oder zuletzt gedrückte gewinnt, oder es kommt Unsinn heraus.

Die eigentliche Einlese- und Entprellroutine ist nur 8 Instruktionen kurz. Der entprellte Tastenzustand ist im Register *key_state*. Mit nur 2 weiteren Instruktionen wird dann der Wechsel von *Taste offen* zu *Taste gedrückt* erkannt und im Register *key_press* abgelegt. Im Beispielcode werden dann damit 8 LEDs ein- und ausgeschaltet. Jede Taste entspricht einem Bit in den Registern, d.h. die Verarbeitung erfolgt bitweise mit logischen Operationen. Zum Verständnis empfiehlt es sich daher, die Logikgleichungen mit Gattern für ein Bit = eine Taste aufzumalen. Die Register kann man sich als Flipflops denken, die mit der Entprellzeit als Takt arbeiten. D.h. man kann das auch so z. B. in einem GAL22V10 realisieren.

Als Kommentar sind neben den einzelnen Instruktionen alle 8 möglichen Kombinationen der 3 Signale dargestellt.

Beispielcode für AVR (Assembler):

```
.nolist
.include "c:\avr\inc\1200def.inc"
.list
.def  save_sreg         = r0
.def  iwr0              = r1
.def  iwr1              = r2

.def  key_old           = r3
.def  key_state         = r4
.def  key_press         = r5

.def  leds              = r16
.def  wr0               = r17
```

```
.equ  key_port          = pind
.equ  led_port          = portb

      rjmp   init
.org OVF0addr           ;timer interrupt 24ms
      in     save_sreg, SREG
get8key:                              ;/old      state     iwr1      iwr0
      mov    iwr0, key_old            ;00110011  10101010            00110011
      in     key_old, key_port        ;11110000
      eor    iwr0, key_old            ;                              11000011
      com    key_old                  ;00001111
      mov    iwr1, key_state          ;                    10101010
      or     key_state, iwr0          ;          11101011
      and    iwr0, key_old            ;                              00000011
      eor    key_state, iwr0          ;          11101000
      and    iwr1, iwr0               ;                    00000010
      or     key_press, iwr1          ;store key press detect
;
;                     insert other timer functions here
;
      out    SREG, save_sreg
      reti
;-----------------------------------------------------------------------
init:
      ldi    wr0, 0xFF
      out    ddrb, wr0
      ldi    wr0, 1<<CS02 | 1<<CS00    ;divide by 1024 * 256
      out    TCCR0, wr0
      ldi    wr0, 1<<TOIE0             ;enable timer interrupt
      out    TIMSK, wr0

      clr    key_old
      clr    key_state
      clr    key_press
      ldi    leds, 0xFF
main: cli
      eor    leds, key_press           ;toggle LEDs
      clr    key_press                 ;clear, if key press action done
      sei
      out    led_port, leds
      rjmp   main
;---------------------------------------------------------------
```

Komfortroutine (C für AVR)

Siehe dazu: Forum

Anmerkung Wenn statt active-low (Ruhezustand High) active-high (Ruhezustand Low) verwendet wird muss eine Zeile geändert werden siehe: gesamter Beitrag im Forum, Stelle 1 im Beitrag, Stelle 2 im Beitrag

Funktionsprinzip wie oben plus zusätzliche Features:

- Kann Tasten sparen durch unterschiedliche Aktionen bei kurzem oder langem Drücken

- Wiederholfunktion, z. B. für die Eingabe von Werten

Das Programm ist für avr-gcc/avr-libc geschrieben, kann aber mit ein paar Anpassungen auch mit anderen Compilern und Mikrocontrollern verwendet werden. Eine Portierung für den AT91SAM7 findet man hier (aus dem Projekt ARM MP3/AAC Player).

```
/************************************************************************/
/*                                                                      */
/*                      Debouncing 8 Keys                               */
/*                      Sampling 4 Times                                */
/*                      With Repeat Function                            */
/*                                                                      */
/*              Author: Peter Dannegger                                 */
/*                      danni@specs.de                                  */
/*                                                                      */
/************************************************************************/

#include <stdint.h>
#include <avr/io.h>
#include <avr/interrupt.h>

#ifndef F_CPU
#define F_CPU           1000000                 // processor clock frequency
#warning kein F_CPU definiert
#endif

#define KEY_DDR         DDRB
#define KEY_PORT        PORTB
#define KEY_PIN         PINB
#define KEY0            0
#define KEY1            1
#define KEY2            2
#define ALL_KEYS        (1<<KEY0 | 1<<KEY1 | 1<<KEY2)

#define REPEAT_MASK     (1<<KEY1 | 1<<KEY2)       // repeat: key1, key2
#define REPEAT_START    50                        // after 500ms
#define REPEAT_NEXT     20                        // every 200ms

#define LED_DDR         DDRA
#define LED_PORT        PORTA
#define LED0            0
#define LED1            1
#define LED2            2

volatile uint8_t key_state;                             // debounced and inverted key state:
                                                // bit = 1: key pressed
volatile uint8_t key_press;                             // key press detect

volatile uint8_t key_rpt;                               // key long press and repeat


ISR( TIMER0_OVF_vect )                          // every 10ms
{
  static uint8_t ct0, ct1, rpt;
  uint8_t i;
```

```
  TCNT0 = (uint8_t)(int16_t)-(F_CPU / 1024 * 10e-3 + 0.5);  // preload for 10ms

  i = key_state ^ ~KEY_PIN;                       // key changed ?
  ct0 = ~( ct0 & i );                             // reset or count ct0
  ct1 = ct0 ^ (ct1 & i);                          // reset or count ct1
  i &= ct0 & ct1;                                 // count until roll over ?
  key_state ^= i;                                 // then toggle debounced state
  key_press |= key_state & i;                     // 0->1: key press detect

  if( (key_state & REPEAT_MASK) == 0 )            // check repeat function
     rpt = REPEAT_START;                          // start delay
  if( --rpt == 0 ){
    rpt = REPEAT_NEXT;                            // repeat delay
    key_rpt |= key_state & REPEAT_MASK;
  }
}

///////////////////////////////////////////////////////////////////
//
// check if a key has been pressed. Each pressed key is reported
// only once
//
uint8_t get_key_press( uint8_t key_mask )
{
  cli();                                          // read and clear atomic !
  key_mask &= key_press;                          // read key(s)
  key_press ^= key_mask;                          // clear key(s)
  sei();
  return key_mask;
}

///////////////////////////////////////////////////////////////////
//
// check if a key has been pressed long enough such that the
// key repeat functionality kicks in. After a small setup delay
// the key is reported being pressed in subsequent calls
// to this function. This simulates the user repeatedly
// pressing and releasing the key.
//
uint8_t get_key_rpt( uint8_t key_mask )
{
  cli();                                          // read and clear atomic !
  key_mask &= key_rpt;                            // read key(s)
  key_rpt ^= key_mask;                            // clear key(s)
  sei();
  return key_mask;
}

///////////////////////////////////////////////////////////////////
//
uint8_t get_key_short( uint8_t key_mask )
{
  cli();                                          // read key state and key press atomic !
  return get_key_press( ~key_state & key_mask );
}

///////////////////////////////////////////////////////////////////
//
uint8_t get_key_long( uint8_t key_mask )
```

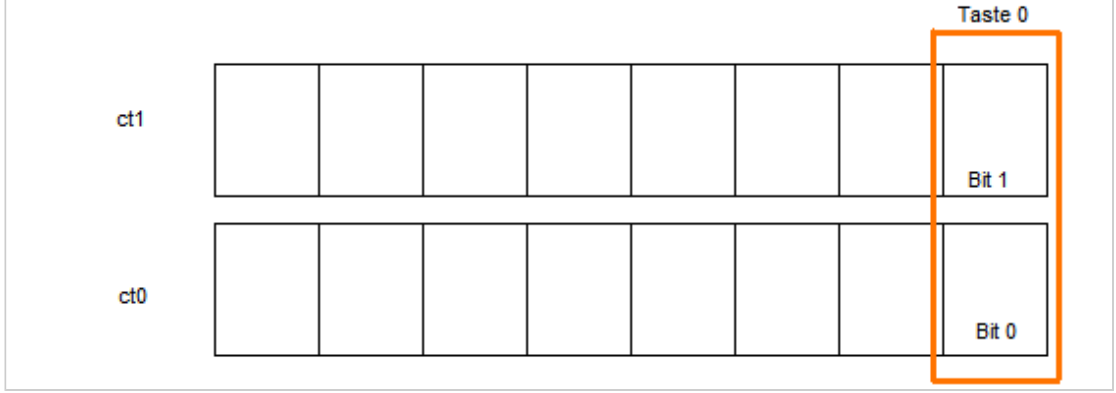
```
{
  return get_key_press( get_key_rpt( key_mask ));
}

int main( void )
{
  LED_PORT = 0xFF;
  LED_DDR = 0xFF;

  // Configure debouncing routines
  KEY_DDR &= ~ALL_KEYS;                 // configure key port for input
  KEY_PORT |= ALL_KEYS;                 // and turn on pull up resistors

  TCCR0 = (1<<CS02)|(1<<CS00);          // divide by 1024
  TCNT0 = (uint8_t)(int16_t)-(F_CPU / 1024 * 10e-3 + 0.5);  // preload for 10ms
  TIMSK |= 1<<TOIE0;                    // enable timer interrupt

  sei();

  while(1){
    if( get_key_short( 1<<KEY1 ))
      LED_PORT ^= 1<<LED1;

    if( get_key_long( 1<<KEY1 ))
      LED_PORT ^= 1<<LED2;

    // single press and repeat

    if( get_key_press( 1<<KEY2 ) || get_key_rpt( 1<<KEY2 )){
      uint8_t i = LED_PORT;

      i = (i & 0x07) | ((i << 1) & 0xF0);
      if( i < 0xF0 )
        i |= 0x08;
      LED_PORT = i;
    }
  }
}
```

Das single-press-und-repeat-Beispiel geht nicht in jeder Beschaltung; folgendes Beispiel sollte universeller sein (einzelne LED an/aus):

```
// single press and repeat
if( get_key_press( 1<<KEY2 ) || get_key_rpt( 1<<KEY2 )){
    LED_PORT ^=0x08;
```

Funktionsweise

Der Code basiert auf 8 parallelen vertikalen Zählern, die über die Variablen ct0 und ct1 aufgebaut werden



8 vertikale Zähler in 2 8-Bit Variablen

wobei jeweils ein Bit in ct0 mit dem gleichwertigen Bit in ct1 zusammengenommen einen 2-Bit-Zähler bildet. Der Code der sich um die 8 Zähler kümmert, ist so geschrieben, daß er alle 8 Zähler gemeinsam parallel behandelt.

```
i = key_state ^ ~KEY_PIN;                       // key changed ?
```

i enthält an dieser Stelle für jede Taste, die sich im Vergleich mit dem vorhergehenden entprellten Zustand verändert hat, ein 1 Bit.

```
ct0 = ~( ct0 & i );                             // reset or count ct0
ct1 = ct0 ^ (ct1 & i);                          // reset or count ct1
```

Diese beiden Anweisungen erniedrigen den 2-Bit Zähler ct0/ct1 für jedes Bit um 1, welches in i gesetzt ist. Liegt an der entsprechenden Stelle in i ein 0 Bit vor (keine Änderung des Zustands), so wird der Zähler ct0/ct1 für dieses Bit auf 1 gesetzt. Der Grundzustand des Zählers ist als ct0 == 1 und ct1 == 1 (Wert 3). Der Zähler zählt daher mit jedem ISR Aufruf, bei dem die Taste als verändert erkannt wurde

```
ct1   ct0
  1     1   // 3
  1     0   // 2
  0     1   // 1
  0     0   // 0
  1     1   // 3
```

```
i &= ct0 & ct1;                                 // count until roll over ?
```

in i bleibt nur dort ein 1-Bit erhalten, wo sowohl in ct1 als auch in ct0 ein 1 Bit vorgefunden wird, der betreffenede Zähler also bis 3 zählen konnte. Durch die zusätzliche Verundung mit i wird der Fall abgefangen, dass ein konstanter Zählerwert von 3 in i ein 1 Bit hinterlässt. Im Endergebnis bedeutet dass, dass nur ein Zählerwechsel von 0 auf 3 zu einem 1 Bit an der betreffenden Stelle in i führt. Das heißt, ein Tastendruck wird erkannt, wenn die Taste 4 mal hintereinander in einem anderen Zustand vorgefunden wurde als dem zuletzt bekannten entprellten Tastenzustand.

An dieser Stelle ist i daher ein Vektor von 8 Bits, von denen jedes einzelne der Bits darüber Auskunft gibt, ob die entsprechende Taste mehrmals hintereinander im selben Zustand angetroffen wurde, der nicht mit dem zuletzt bekannten Tastenzustand übereinstimmt.

Damit ist der Tasteneingang entprellt. Und zwar sowohl beim Drücken einer Taste als auch beim Loslassen. Der weitere Code beschäftigt sich dann nur noch damit, diesen entprellten Tastenzustand weiter zu verarbeiten.

Links

- AVR-Tutorial: Tasten
- AVR-GCC-Tutorial Tastenentprellung
- Beitrag im Forum, AVR Assembler
- A guide to debouncing (engl.), praktische Erläuterungen zum Entprellen in Soft- und Hardware
- Understanding Destructive LC Voltage Spikes

Von „http://www.mikrocontroller.net/articles/Entprellung“
Kategorien: AVR | Signalverarbeitung