

Mikroprozessorpraktikum WS 2011/12  
Aufgabenkomplex: 4

Teilnehmer:

Marco Träger, Matr. 4130515  
Alexander Steen, Matr. 4357549

Gruppe: Freitag, Arbeitsplatz: HWP 1

## A 4.1 Taster als Interruptquelle

**A 4.1.1** Beide Taster an Port1 (P1.0 und P1.1) sollen interruptfähig sein und auf eine LH-Flanke den Interrupt auslösen. Zunächst soll die Toggle-Funktionalität implementiert werden.

Zunächst definieren wir uns einige Variablen: `counter` wird im folgenden Code (weiter unten) die Anzahl der Tastendrucke am Taster P1.0 zählen. `waittime` wird ebenfalls weiter unten im Code genutzt, um in dem Interrupt mittels `wait()` eine Abarbeitungszeit zu simulieren; damit minimieren wir das Prellen der Tasterkontakte und verhindern, dass mehrfach in den Interrupt gesprungen wird.

```
1  int counter = 0;
2  int waittime = 15000;
3
4  void init411() {
5      // LEDs vorbereiten
6      P4DIR |= (0x07);
7      P4SEL &= ~(0x07);
8      P4OUT |= 0x07;
9
10     // Taster vorbereiten
11     P1DIR &= ~(0x03);
12     P1SEL &= ~(0x03);
13     // Interrupt einschalten
14     P1IE |= (0x03);
15     // Auf LH-Flanke stellen
16     P1IES &= ~(0x03);
17 }
18
19 void aufgabe411() {
20     wait(50000);
21     P4OUT ^= 0x04; //LED4.2 toggeln
22 }
```

Von der `main.c`-Datei wird die `init411`-Funktion vor der Endlosschleife und dem darin enthaltenen Funktionsaufruf von `aufgabe411` ausgeführt. Es werden zunächst alle notwendigen Einstellungen für die LEDs und die Taster vorgenommen. Danach wird in einer Endlosschleife die LED 4.2 getoggelt.

Als nächstes wird ein Interrupt für den Port 1 definiert. Die Vorgehensweise wurde aus der Datei `interrupt.c` aus dem Startprojekt übernommen.

```
23 #pragma vector = PORT1_VECTOR
24 __interrupt void PORT1 (void) {
25     // P1.0 wurde gedrueckt
26     if((P1IN & 0x01)) {
27         counter++;
28         // Zehnte Mal gedrueckt
29         if(counter == 10) {
30             counter = 0;
31             P4OUT ^= 0x01; //toggle
32         }
33         wait(waittime);
34     }
35     // P1.1 wurde gedrueckt
36     if((P1IN & 0x02)) {
37         P4OUT ^= 0x02; //toggle
38         wait(waittime);
39     }
40
41     // Interrupt-Flag zuruecksetzen
42     CLEAR(P1IFG, 0xFF);
43 }
```

In dem Interrupt wird überprüft, welche der beiden Taster gedrückt wurde und danach entschieden, ob entweder die LED 4.1 sofort oder die LED 4.0, falls der zehnte Druck in Folge, getoggelt wird. Zusätzlich wird in beiden Fällen mittels `wait(waittime)` ein wenig Arbeitslast simuliert, sodass ein Prellen der Tastkontakte vermieden wird. Am Ende der ISR wird die Interrupt-Flag zurückgesetzt und somit der Interrupt als abgearbeitet markiert.

Um nun nur bei leuchtender LED 4.1 den Taster P1.0 interruptfähig zu schalten, wird zwischen Zeile 37 und 38 die Anweisung `P1IE ^= 0x01;` eingefügt. Damit wird bei jedem Druck auf die Taste P1.1 ebenfalls die Interruptfähigkeit von P1.0 getoggelt. Da wir mit ausgeschalteten LEDs starten, müssen wir nur noch die Initialisierung von P1IE in Zeile 14 von `P1IE |= (0x03);` auf `P1IE |= (0x02);` ändern. Damit startet das Programm ohne Interruptfähigkeit für P1.0.

## A 4.2 Totmannschaltung

**A 4.2.1** Der Watchdog und ein Tasten-Interrupt sollen in dieser Aufgabe gleichzeitig genutzt werden. In der Technik kommt so etwas in Form einer sogenannten Totmannschaltung vor. Betätigt der Fahrer eines Fahrzeugs innerhalb eines festgelegten Zeitraumes nicht eine Taste, erfolgt eine Notbremsung.

Die Initialisierung erfolgt analog zu A 4.1.1. Allerdings wird hier nur Taster P1.0 interruptfähig geschaltet und zusätzlich der Watchdog eingerichtet. Aus Aufgabe A 3.1.1 wissen wir, dass bei einem Takt von 32768 Hz auf Basis der ACLK-Taktquelle der Watchdog nach einer Sekunde auslöst, falls `WDTIS1 = 0` und `WDTIS0 = 0`. Da wir vier Sekunden Zeit vor dem Watchdog-Interrupt haben wollen, müssen wir den Takt-Vorteiler im `BCSCTL1`-Register auf 4 stellen. Zusätzlich wird der Watchdog mit dem `WDTTMSSEL`-Bit gestartet, damit wir den Watchdog-Timer nutzen. Die Funktion `aufgabe421` wird in einer Endlosschleife ausgeführt und toggelt die LED 4.0.

```
1 void init421() {
2     // LEDs vorbereiten
3     P4DIR |= (0x07);
4     P4SEL &= ~(0x07);
5     P4OUT |= 0x07; LEDs aus
6     LEDOFF;
7
8     // Watchdog einrichten
9     BCSCTL1 |= DIVA1; // Divisor /4
10    WDTCTL = WDTPW + WDTTMSSEL + WDTSSSEL; //PW, Timer Interval, ACLK
11    IE1 = WDTIE; // Watchdog interrupt enable
12
13    LED2ON;
14
15    // taster vorbereiten
16    P1DIR &= ~(0x01);
17    P1SEL &= ~(0x01);
18    P1IE |= (0x01);
19    P1IES &= ~(0x01);
20 }
21
22 void aufgabe421() {
23     wait(50000);
24     P4OUT ^= 0x01;
25 }
```

Des Weiteren brauchen wir nicht nur eine ISR auf dem Taster, sondern auch für den Watchdog-Timer-Interrupt. Das Schlüsselwort `WDT_VECTOR` wurde der Interrupttabelle in der Datei `interrupt.c` entnommen.

```
#pragma vector = PORT1_VECTOR
__interrupt void PORT1 (void) {
    // Pin gedrueckt
    if((P1IN & 0x01)) {
        // Watchdog zuruecksetzen
        WDTCTL = WDTPW + WDTSEL + WDTCTL;
        LED2OFF;
        wait(1500);
    }
    CLEAR(P1IFG, 0xFF);
}

#pragma vector = WDT_VECTOR
__interrupt void WDT (void) {
    if(P4OUT & 0x02) { // LED4.1 ist aus
        P4OUT &= ~(0x02); // LED anschalten
    } else {
        // Watchdog ausschalten
        WDTCTL = WDTPW + WDTHOLD;
        IE1 &= ~WDTIE;
        // Ampelsequenz
        while(1) {
            P4OUT = ROT;
            wait(WAITTIME);
            P4OUT = ROTGELB;
            wait(WAITTIME);
            P4OUT = GRUEN;
            wait(WAITTIME);
            P4OUT = GELB;
            wait(WAITTIME);
            P4OUT = ROT;
            wait(WAITTIME);
        }
    }
    // WDTIFG is reset automatically by servicing the interrupt
}
```

Die ISR für den Taster setzt den Watchdog-Zähler zurück und schaltet die LED 4.1 aus. In der Watchdog-Timer-ISR wird anhand der LED 4.1 entschieden, was gemacht wird: Ist die LED aus, so wird sie eingeschaltet. Ist sie allerdings bereits eingeschaltet, wird der Watchdog ausgeschaltet und in eine endlose Ampelsequenz übergegangen.

Die groß geschriebenen Konstanten beziehen sich auf die Definitionen aus Aufgabe A 1.1.6.