Interrupts

Schon im BASCOM-Teil wurde das Interrupt-Konzept vorgestellt und angewandt. Nun betrachten wir die Interrupts aus der Assemblerperspektive und können so ein noch differenzierteres Bild vom Interruptmechanismus erhalten. Wie immer wollen wir uns von einem einfache, aber lehrreichen Beispiel leiten lassen.

Der Mikrocontroller soll in seinem Hauptprogramm einen Ton erzeugen. Dazu wird PortB.6 mit einem Kabel an den Speaker angeschlossen und am Ausgang PortB.6 ein rasch wechselndes Signal erzeugt. Dazu dient folgender Programmteil:

```
.def zz   = r18

sbi ddrd, 6              ; PortD.6 als Ausgang

schleife:                ; Schleife zur Signalerzeugung
  sbi portd, 6           ; PortD.6 high
  rcall warte            ; warten
  cbi portd, 6           ; PortD.6 low
  rcall warte            ; warten
  rjmp schleife          ; usw.

warte:                   ; Zeit schinden
  ldi zz, 0              ; Zeitzähler zz auf 0
  warte0:
    inc zz               ; zz um 1 erhöhen
     cpi zz, 200         ; zz mit 200 vergleichen
     brne warte0         ; falls zz noch nicht gleich 200, dann
                         zur Marke warte0 springen
  ret                    ;
```

Die Kommentare sollten die Funktionsweise des Programms hinreichend erläutern. Neu ist hier lediglich:

Mnemonic	Abkürzung für	Bedeutung
`inc rx`	increment	erhöht den Inhalt des Registers rx um 1

Zusätzlich soll der Mikrocontroller noch auf Tastendruck reagieren und über LEDs an PortB anzeigen, wie oft der Taster Ta0 betätigt worden ist. Dazu muss der Mikrocontroller bei jedem Tastendruck den Inhalt einer Zählvariablen um 1 erhöhen und das Ergebnis über PortB ausgeben. Ta0 ist an PortD.2 angeschlossen. Damit der Tasterzustand abgefragt werden kann, muss PortD.2 als Eingang konfiguriert werden:

```
cbi ddrd, 2                    ; PortD.2 als Eingang
sbi portd, 2                   ; PortB.2 pull-up
```

Nun muss der Schalterszustand fortwährend kontrolliert werden; wenn der Taster gedrückt ist, dann ist PortD.2 auf 0 und unsere Zählvariable muss um 1 erhöht werden. Es ist aber klar, dass es nicht ausreicht, irgendwo in die Schleife zur Signalerzeugung einen solchen Befehl einzubauen, der den Zustand von PortD.2 abfragt. Es könnte ja sein, dass unser Taster gerade nur betätigt ist, wenn der Mikrocontroller die `warte0`-Schleife durchläuft. In diesem Fall würde der Mikrocontroller den Tastendruck übersehen.

Hier kommt unser Interrupt-Konzept ins Spiel: Wenn das INT0-Interrupt aktiviert ist, kontrolliert der Mikrocontroller selbstständig, d. h. ohne Steuerung durch das arbeitende Programm, ob ein bestimmtes Signal an PortD.2 vorliegt, z. B. ob der Eingangspegel von high auf low wechselt. Ist dies der Fall, dann werden folgende Aktionen ausgelöst:

- der aktuelle Befehl wird noch zu Ende ausgeführt
- die Adresse des folgenden Befehls wird auf den Stack gelegt
- das Interrupt-Flag SREG.I wird zurückgesetzt (mehr dazu weiter unten)
- die Einsprungadresse für die Interrupt-Serviceroutine wird in den Programmzähler (PC) geladen

Der Attiny kennt insgesamt 19 verschiedene Interrupts. All diesen Interrupts ist eine eigene Adresse im Programmspeicher zugeordnet, die so genannte Einsprungadresse. Die folgende Tabelle aus dem Manual gibt eine Liste sämtlicher Interrupts des Attiny mit ihren Einsprungadressen wieder.

Vector No.	Program Address	Source	Interrupt Definition
1	0x0000	RESET	External Pin, Power-on Reset, Brown-out Reset, and Watchdog Reset
2	0x0001	INT0	External Interrupt Request 0
3	0x0002	INT1	External Interrupt Request 1
4	0x0003	TIMER1 CAPT	Timer/Counter1 Capture Event
5	0x0004	TIMER1 COMPA	Timer/Counter1 Compare Match A
6	0x0005	TIMER1 OVF	Timer/Counter1 Overflow
7	0x0006	TIMER0 OVF	Timer/Counter0 Overflow
8	0x0007	USART0, RX	USART0, Rx Complete
9	0x0008	USART0, UDRE	USART0 Data Register Empty
10	0x0009	USART0, TX	USART0, Tx Complete
11	0x000A	ANALOG COMP	Analog Comparator
12	0x000B	PCINT	Pin Change Interrupt
13	0x000C	TIMER1 COMPB	Timer/Counter1 Compare Match B
14	0x000D	TIMER0 COMPA	Timer/Counter0 Compare Match A
15	0x000E	TIMER0 COMPB	Timer/Counter0 Compare Match B
16	0x000F	USI START	USI Start Condition
17	0x0010	USI OVERFLOW	USI Overflow
18	0x0011	EE READY	EEPROM Ready
19	0x0012	WDT OVERFLOW	Watchdog Timer Overflow

Abbildung 1

Der Attiny ist nun so konstruiert, dass er bei einem Interrupt-Ereignis die zugehörige Einsprungadresse in den PC schreibt. Für unser INT0-Ereignis bedeutet das, dass nach dem Interrupt-Ereignis der Befehl in der Programmzelle 0001 ausgeführt wird. Wäre z. B. ein Overflow-Ereignis beim Timer/Counter0 eingetreten, dann würde als nächstes der Befehl in der Programmzelle 0006 ausgeführt.

In diese Zelle schreiben wir nun einen Befehl, welcher den Mikrocontroller zu der zugehörigen Interrupt-Serviceroutine führt. Im Falle unseres INT0-Interrupts müssen wir also in die Zelle 0001 den Befehl

```
rjmp zaehlen
```

schreiben; dabei ist `zaehlen` eine Marke für unsere INT0-Serviceroutine:

```
zaehlen:
  inc zaehler
  out portb, zaehler
  reti                    ; Rückkehr (s. u.)
```

Die Aktionen, die vom Mikrocontroller bis dahin beim INT0-Interrupt ausgeführt werden, sind in der Abb. 2 rot gekennzeichnet. Wir gehen dabei davon aus, dass ein Interrupt erfolgt, wenn gerade der Befehl in der Programmzeile x ausgeführt wird. Dann wird die Adresse x+1 auf den Stapel gelegt und der Mikrocontroller gelangt über den Befehl `rjmp zaehlen` zum Beginn unserer INT0-Serviceroutine. Beachten Sie, dass dieser Sprung nicht mit einem `rcall`-Befehl ausgeführt wird. Das wäre auch nicht sinnvoll, weil die Rücksprungadresse bereits hardwaremäßig durch das Interruptereignis auf dem Stapel gesichert worden ist.

Wenn alle Befehle der Serviceroutine ausgeführt worden sind, dann wird durch den `reti`-Befehl die Rücksprungadresse x+1 wieder vom Stapel geholt und in den PC geladen. Dadurch kehrt der Mikrocontroller nach der kurzen "Unterbrechung" an die richtige Stelle des Programms zurück. Warum übrigens ein normaler `ret`-Befehl nicht ausreicht, werden wir gleich noch klären.

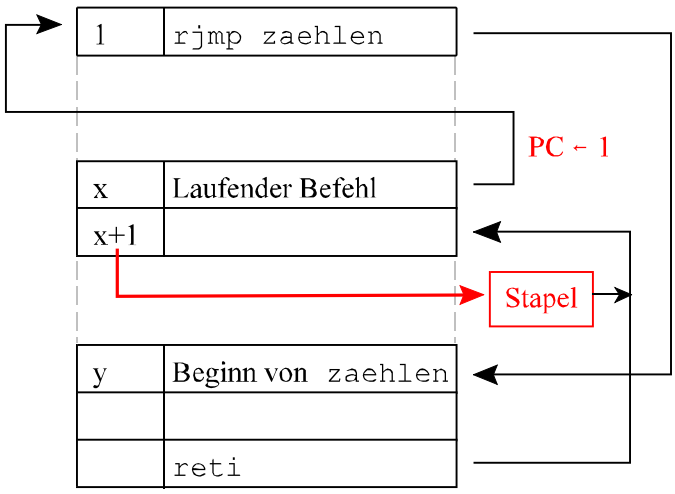

Abbildung 2

Zuvor müssen wir uns noch einem kleinen technischen Problem widmen. Wir haben gesehen: Wenn wir mit Interrupts arbeiten, dann werden die ersten Zellen des Programmspeichers für Interrupt-Aktivitäten benutzt; sie stehen deswegen nicht mehr für das Hauptprogramm zur Verfügung. Einige Befehle unseres Programms dürfen also nicht irgendwo stehen. Wir müssen also beim

Schreiben des Programms auch darauf achten, an welcher Stelle im Programmspeicher die Befehle zu stehen haben. Glücklicherweise sind solche Überlegungen nur für den Anfang des Programms zu machen.

Soll ein Befehl in einer bestimmten Programmzelle stehen, so lässt sich dies mit der Assemblerdirektive

```
.org xxxx
```

erreichen. Der Befehl, der gleich nach dieser Direktiven steht, wird in die Programmzelle mit der Adresse xxxx geschrieben; alle folgenden Befehle werden dann wie üblich der Reihe nach angefügt. In unserem Fall schreiben wir

```
.org 0001
rjmp zaehlen
```

Allerdings müssen wir noch bedenken, dass der Mikrocontroller bei jedem Neustart oder Reset zur Programmzelle 0000 geht. Diese Aktion kann auch als ein Interrupt mit der Einsprungadresse 0000 angesehen werden; es wird hierbei aber keine Rücksprungadresse gesichert. In diese Zelle 0000 müssen wir einen Sprung zum Hauptprogramm einfügen. Unser gesamtes Programm sieht nun also so aus:

```
.include "tn2313def.inc"
.def zaehler = r16
.def temp = r17
.def zz = r18

rjmp init
.org 1
  rjmp zaehlen                ;Einsprungadresse des Interrupts
                              INT0 bei 1
init:                         ; Beginn des Hauptprogramms
  ldi temp, 255               ; PortB als Ausgang
  out ddrb, temp
  cbi ddrd, 2                 ; PortD.2 als Eingang
  sbi portd, 2                ; PortB.2 pull-up
  sbi ddrd, 6                 ; PortD.6 als Ausgang
  ldi zaehler, 0

schleife:
  sbi portd, 6
  rcall warte
  cbi portd, 6
  rcall warte
  rjmp schleife

warte:
  ldi zz, 0
  warte0:
```

```
    inc zz
     cpi zz, 200
     brne warte0
  ret

zaehlen:                  ; Beginn der Interrupt-Serviceroutine
  inc zaehler
  out portb, zaehler
  reti
```

Leider ist das Programm so noch nicht lauffähig: Was noch fehlt, ist die Initialisierung des INT0-Interrupts. Aus dem BASCOM-Kapitel zu Interrupts wissen wir bereits, dass dazu noch einige Einstellungen erforderlich sind:

- INT0-Interrupt konfigurieren (in unserem Fall: fallende Flanke)
- INT0-Interrupt freigeben
- Interrupts global freigeben

INT0-Interrupts können durch unterschiedliche Ereignisse am PortD.2 ausgelöst werden, z. B. eine fallende oder steigende Flanke. Durch welches dieser Ereignisse nun tatsächlich ein Interrupt ausgeführt werden soll, wird durch die Bits ISC01 und ISC00 des I/O-Registers MCUCR gesteuert (Abb. 3). Für eine fallende Flanke muss ISC01 = 1 und ISC00 = 0 sein. Da die anderen Bits im Augenblick für uns unwichtig sind, reicht es daher aus, in das MCUCR-Register den Wert 2 zu schreiben:

```
ldi temp, 2
out mcucr, temp
```

Bit	7	6	5	4	3	2	1	0	
	PUD	SM1	SE	SMD	ISC11	ISC10	ISC01	ISC00	MCUCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Abbildung 3

Die Freigabe des INT0-Interrupts erfolgt über das INT0-Bit des I/O-Registers GIMSK (Abb. 4):

```
ldi temp, 64
out gimsk, temp
```

Bit	7	6	5	4	3	2	1	0	
	INT1	INT0	PCIE	–	–	–	–	–	GIMSK
Read/Write	R/W	R/W	R/W	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

Abbildung 4

Um die Interrupts global freizugeben, muss das I-Bit des I/O-Registers SREG gesetzt werden (Abb. 5). Am einfachsten geht das mit dem `sei`-Befehl:

Mnemonic	**Abkürzung für**	**Bedeutung**
`sei`	set global interrupt flag	setzt das Interruptflag I des I/O-Registers SREG

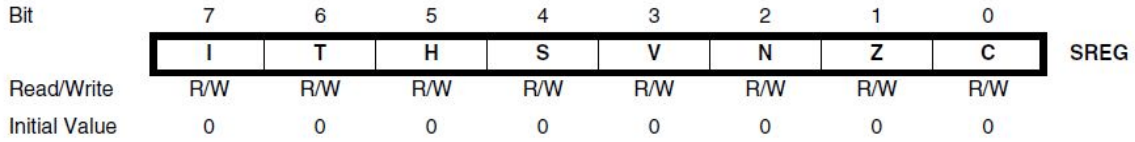

Abbildung 5

In dem Augenblick, in dem der Interrupt ausgelöst wird, wird dieses I-Bit automatisch zurückgesetzt; dadurch werden weitere Interrupt-Auslösungen verhindert. Erst durch den Befehl `reti` werden das I-Bit wieder gesetzt und damit Interrupts erneut zugelassen.

Mnemonic	**Abkürzung für**	**Bedeutung**
`reti`	return from interrupt	setzt das Interruptflag I des I/O-Registers SREG zurück, holt die Rückkehradresse vom Stapel in den PC

Würde man statt des `reti`-Befehls den `ret`-Befehl benutzen, so würde der Mikrocontroller zwar an die richtige Stelle zurückkehren; der Mikrocontroller wäre nun aber für weitere Interrupts gesperrt - es sei denn, das I-Bit würde explizit wieder gesetzt werden.

Interrupt-Serviceroutinen können also standardmäßig nicht durch weitere Interruptereignisse unterbrochen werden. In den meisten Fällen - so wie hier - ist das auch sinnvoll so. Nur in Ausnahmefällen wird man auch Interrupt-Serviceroutinen unterbrechen lassen. Dies kann man erreichen, indem man am Anfang dieser Routine den `sei`-Befehl einfügt.

Bei dem folgenden Programm sind nun alle diese Erkenntnisse zusammengeflossen; außerdem haben wir noch einen weiteren Interrupt zugelassen: Über den Taster Ta1wird ein INT1-Interrupt ausgelöst; dieser sorgt dafür, dass der Zählerstand um 1 erniedrigt wird. Mit dem Taster Ta0 zählen wir also vorwärts, mit dem Taster Ta1 rückwärts.

```
.include "tn2313def.inc"

.def zaehler = r16
.def temp = r17
.def zz = r18
```

```
rjmp init

.org 1
  rjmp aufwaertszaehlen       ;Einsprungadresse des Interrupts
int0 bei 1
  rjmp abwaertszaehlen        ;Einsprungadresse des Interrupts
int1 bei 2

init:

  ldi temp, 255
  out ddrb, temp
  cbi ddrd, 2                 ; PortD.2 als Eingang
  sbi portd, 2                ; PortB.2 pull-up
  sbi ddrd, 6                 ; PortD.6 als Ausgang

  ldi temp, 2+8
  out mcucr, temp             ; INT0 und INT1 fallende Flanke
  ldi temp, 64+128
 out gimsk, temp              ; INT0 und INT1 Interrupt zulassen
  sei

  ldi zaehler, 0

schleife:
  sbi portd, 6
  rcall warte
  cbi portd, 6
  rcall warte
  rjmp schleife

warte:
  ldi zz, 0
  warte0:
    inc zz
     cpi zz, 200
     brne warte0
  ret

aufwaertszaehlen:
  inc zaehler
  out portb, zaehler
  reti

abwaertszaehlen:
  dec zaehler            ; zaehler <- zaehler - 1
  out portb, zaehler
  reti
```

Jetzt unterziehen wir unser Programm einem ausführlichen Test. Zunächst schließen wir den Speaker an PortD.6 an und stecken die Leuchtdioden in die Anschlüsse von PortB. Nach dem Starten des Programms hören wir einen hohen Ton am Speaker; die LEDs sind aus. Nun betätigen wir den Taster Ta0: Die LEDs zählen im Zweiersystem mit: &B00000000, &B00000001, &B00000010, &B0000011, &B0000100, ... Während des Tastendrucks ist keine Veränderung beim Ton festzustellen. (Dass auch ohne Tastendruck manchmal geringe Schwankungen in der Tonhöhe festzustellen sind, liegt daran, dass unsere Attiny-Platine standardmäßig ohne Quarz arbeitet und damit der Taktgeber nicht ganz frequenzstabil ist.) Offensichtlich ist die Zeit, die die Interrupt-Serviceroutine benötigt, so klein, dass sie gegenüber der Wartezeit der `warte0`-Schleife nicht ins Gewicht fällt.

Nun betätigen wir auch den Taster Ta1. An den LEDs erkennen wir, dass nun zurückgezählt wird. Allerdings werden dabei scheinbar einige Zahlen übersprungen. Wir kennen bereits den Grund dafür: Taster Ta1 ist im Gegensatz zu Ta0 nicht durch einen Kondensator entprellt. Ein einziger Tastendruck kann dann mehrere Interrupts in rascher Folge auslösen. Dabei wird so schnell nach unten gezählt, dass wir die einzelnen Zählschritte nicht mehr wahrnehmen können.

Irgendwann sind wir schließlich bei negativen Zählständen angekommen. Wenn wir nun wieder den Taster Ta0 betätigen, können wir beim Erreichen des Zählstandes 0 manchmal - aber nicht immer - ein merkwürdiges Phänomen wahrnehmen: ein kleiner Knackser im Speaker zeigt an, dass es eine Unregelmäßigkeit im Schwingungsvorgang gibt. Wie lässt sich diese Unregelmäßigkeit erklären? Und: Lässt sie sich beheben?

Zunächst zur Erklärung: Schuld ist hier das Zero-Flag Z des Registers SREG (Abb. 5). Dieses Flag wird gesetzt, wenn das Ergebnis einer Rechnung gleich 0 ist. In unserem Fall wird das Zero-Flag auf 1 gesetzt, wenn der Zählstand 0 erreicht wird. Das Zero-Flag wird aber, wie wir wissen, auch vom `brne`-Befehl benutzt: Die `warte0`-Schleife wird abgebrochen, wenn das Zero-Flag gesetzt ist. Normalerweise ist das der Fall, wenn in dem vorangegangenen Vergleichsbefehl `cpi` festgestellt worden ist, dass `zz` = 200 ist. Wenn nun genau zwischen diesem `cpi`-Befehl und dem nachfolgenden `brne`-Befehl ein Interrupt erfolgt, dann kann die `warte0`-Schleife abgebrochen werden, auch wenn `zz` noch nicht den Wert 200 erreicht hat. Und dies geschieht gerade dann, wenn in der Interrupt-Serviceroutine `zaehler` den Wert 0 erhält und somit das Zero-Flag auf 1 gesetzt wird. Durch einen solchen vorzeitigen Abbruch kann die Wartezeit deutlich verringert werden, dies hört man als Unregelmäßigkeit im Ton!

Das Problem ist hier also, dass die Interrupt-Serviceroutine einen ungewünschten Einfluss auf den Ablauf des Hauptprogramms hat. Dies kann man unterbinden, indem man direkt am Anfang der Interrupt-Serviceroutine den Inhalt des I/O-Registers SREG mit dem `push`-Befehl auf dem Stapel rettet und am Ende mit dem `pop`-Befehl zurückholt. Die Routine für das Aufwärtszählen sieht dann z. B. so aus:

```
.def savesreg = r19

aufwaertszaehlen:
  in savesreg, sreg
  push savesreg
```

```
inc zaehler             ; kein Problem, wenn Zero-Flag = 1
out portb, zaehler
pop savesreg
out sreg, savesreg
reti
```

Hierbei wird natürlich vorausgesetzt, dass das Register r19 nur in der Interrupt-Serviceroutine benutzt wird; sollte man sich hierin nicht sicher sein, müsste man auch dieses Register zunächst mithilfe des Stapel retten.

Compiler wie z. B. BASCOM berücksichtigen diese Problematik: Sie retten vorsichtshalber bei jeder Interrupt-Serviceroutine sämtliche Rechenregister und das I/O-Register SREG. Das sorgt dafür, dass die Interrupt-Serviceroutinen bei Compilern meist deutlich länger und damit auch langsamer werden als es in Wirklichkeit nötig wäre. (Bei BASCOM z. B. kann aus diesem Grunde das Retten der Register mit der Option Nosave unterbunden werden; das ist aber sehr riskant, weil man meistens nicht genau weiß, ob und welche Register der Compiler gerade einsetzt.) Der Assemblerprogrammierer hingegen kann genau Obacht geben und sich bei den Rettungsaktionen auf die entscheidenden Register beschränken - ein weiterer Vorteil der Assemblerprogrammierung!

Aufgaben

1. Wie viele Zyklen dauert die Interrupt-Serviceroutine `zaehlen`?

2. Wie viele Zyklen dauert das Unterprogramm `warte` (ohne rcall)? Wie lange dauert eine vollständige Schwingung des Speakers, wenn die Taktfrequenz des Mikrocontrollers 4,0 MHz beträgt?

3. Ein Programm macht nur Gebrauch von dem EEPROM-Ready-Interrupt; die zugehörige Serviceroutine möge die Marke `eepromfertig` haben. Wie müssen die ersten Programmzeilen aussehen?

4. Am Ende einer Interrupt-Serviceroutine wurde ein `pop`-Befehl vergessen. Beschreiben Sie das Problem, das sich daraus ergibt.

5. Erläutern Sie, wie man das I-Bit des SREG-Registers (ohne den `sei`-Befehl) setzen kann. Achten Sie darauf, dass die anderen Bits des SREG-Registers nicht verändert werden sollen.
   Hinweise: Der `sbi`-Befehl lässt sich hier nicht benutzen. Ggf. kann der `ori`-Befehl helfen.