B4 Interrupts (Unterbrechungen)

Einführung

Eine ankommende SMS unterbricht kurz die eben begonnene Arbeit. Ein kurzer Blick genügt um die SMS in ihrer Priorität einzuordnen und zum Beispiel die gelieferte Information zu behalten.

In den Mikrocontrollern sind neben dem Prozessor viele periphere Bausteine integriert die gleichzeitig (parallel) zum Prozessor arbeiten können. In einem solchen Ein-Chip-Computersystem[10] müssen diese peripheren Bausteine auch die Möglichkeit besitzen ein laufendes Programm zu unterbrechen. Diese Unterbrechung (*Interrupt*) muss kurzfristig erfolgen können und dauert meist nur kurz.

Das Ereignis (*service*), das die Unterbrechung auslöst, wird Unterbrechungsanforderung (*Interrupt Request, IRQ*) genannt und es bewirkt, dass eine Unterbrechungsroutine (*Interrupt Service Routine, ISR*) aufgerufen wird. Bei der Unterbrechungsroutine handelt es sich um ein spezielles Unterprogramm, das es ermöglicht angemessen auf die Anforderung zu reagieren. Nach der Ausführung des ISR-Codes wird das Hauptprogramm an der Unterbrechungsstelle fortgesetzt.

Interrupts sind ein mächtiges Werkzeug, da sie erlauben praktisch in Echtzeit auf Ereignisse zu reagieren. Ohne Interrupts ist ein gleichzeitiges paralleles Arbeiten mehrerer Bausteine nicht möglich. Auch Multitasking-Betriebssysteme sind ohne Interrupts nicht möglich.

Interrupts sind allerdings auch gefährlich. Der Programmierer muss seine Denkweise beim Programmieren ändern. Hardware-Interrupts können Programme (auch die Unterprogramme!) zu jedem beliebigen Zeitpunkt unterbrechen und damit deren Variablen und Zeiger (Arbeitsregister, SF-Register und SRAM-Daten) verändern. Hardware-Interrupts erfolgen absolut asynchron gegenüber dem unterbrochenen Programm. Die Interrupt-Programmierung erfolgt eine große Sorgfalt.

Die Alternative zu den Interrupts ist das zyklischen Abfragen (*Polling*), um den Status von zum Beispiel Schaltern oder Ein-/Ausgabebausteinen zu erfahren. Diese Methode ist zwar einfacher, aber wesentlich ineffizienter, da der Prozessor beim Polling für keine anderen Aufgaben mehr zur Verfügung steht.

Die Interrupts der AVR-Controller

Die AVR-Controller benutzen **maskierbare Hardware-Interrupts**. Software-Interrupts[11] und so genannte *Exceptions* zur strukturierte Ausnahmebehandlung sind nicht vorhanden.

10 Manchmal wird hier der englische Begriff „*System on a Chip*“ oder SoC verwendet.

11 Externe Interrupts können bei Bedarf softwaremäßig durch Setzen des entsprechenden Portpins (das dann als Ausgang konfiguriert sein muss) ausgelöst werden. Dies stellt allerdings einen Spezialfall dar, der hier nicht behandelt wird.

Die AVR-Controllern besitzen mit dem Hardware-**RESET** einen einzigen nicht-maskierbaren Interrupt (***N**on **M**askable **I**nterrupt **NMI***). Ein Reset kann im Programm nicht abgeschaltet werden. Er erfolgt immer zwingend. Beim Reset wird der Programmzeiger automatisch auf die Flash-Adresse **0x0000** gesetzt. An dieser Adresse befindet sich dann Sprungbefehl (**rjmp** oder **jmp**) zum eigentlichen Programm (siehe Assemblervorlage).

Die Hardware-Interrupts der AVR-Controller sind maskierbar, d.h. sie können abgeschaltet werden. Das Abschalten (bzw. Einschalten) erfolgt mit einzelnen Bits, sogenannten Flags, in unterschiedlichen SF-Registern.

Hardware-Interrupts können global geschaltet werden mit dem **I**-Flag in Statusregister **SREG**. Dies geschieht mit den Befehlen **cli** (***cl**ear **i**-flag*) und **sei** (***se**t **i**-flag*). Diese Flag dient als Hauptschalter um also alle maskierbaren Interrupts zu verbieten oder zu erlauben.

Daneben kann aber auch jedes einzelne Interrupt mit dem ihm eigenen Bit maskiert werden. Zum Beispiel ist dies beim externen Interrupt **INT0** Bit 6 im SF-Register **GICR**. Mit einer ODER-Maskierung (**ori Tmp1,0b01000000**) kann das Interrupt freigegeben werden. Eine UND-Maskierung (**andi Tmp1,0b10111111**) sperrt das Interrupt[12].

Bsp. für das Freigeben des Interrupt:

```
        in      Tmp1,GICR               ;General Interrupt Control Register GICR einlesen
        ori     Tmp1,0b01000000         ;INT0 = 1
        out     GICR,Tmp1               ;Wert ins GICR Register zurueckschreiben
```

Welche maskierbaren Interrupts von welchem Baustein ausgelöst werden können ist aus der Interrupt-Vektortabelle ersichtlich. Jeder Controller hat seine eigene Vektortabelle! Damit der Programmcode portierbar bleibt sollen nicht die absoluten Adressen, sondern die von ATMEL® in den Definitionsdateien (Bsp. **m32def.inc**) vorgeschlagenen Namen für die Adressen verwendet werden.

Die Interrupt-Vektortabelle

Tritt ein Hardware-Interrupt auf, so wird eine zu diesem Interrupt fest zugehörige Adresse in den Befehlszähler (PC) geladen und der Controller springt auf diese Flash-Adresse (Einsprungadresse). Diese nicht veränderbare Adresse wird als Interrupt-Vektor bezeichnet (Vektor steht hier für Adresszeiger) und die hardwaremäßig festgelegte Tabelle als Interrupt-Vektortabelle.

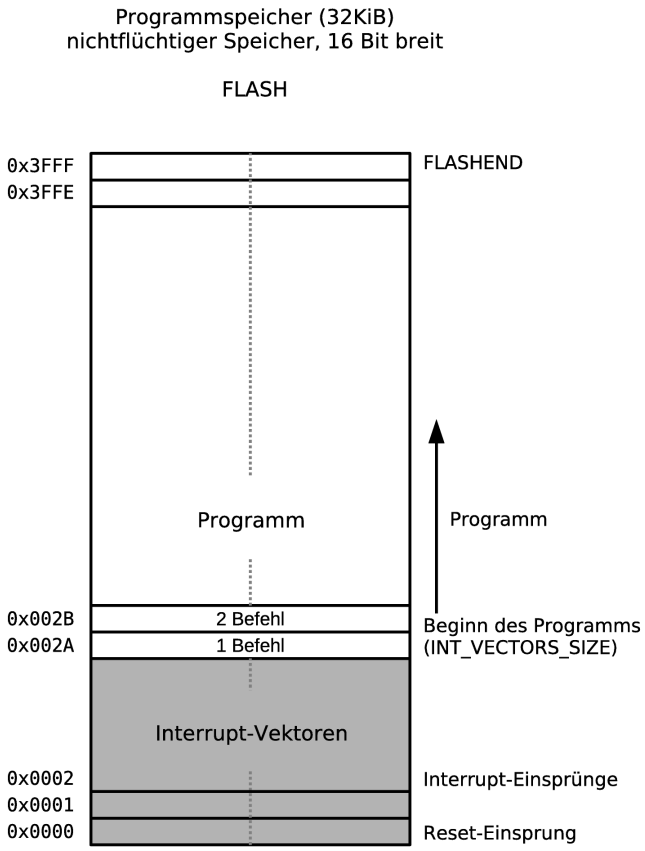
Der ATmega32 besitzt 21 solcher Adressen. Da ein nicht relativer Sprung-Befehl (**jmp**) zwei Flash-Speicherplätze benötigt, sind 42 Flash-Speicherplätze durch die Vektortabelle belegt. Das eigentliche Programm kann also erst an der Adresse **0x002A**[13] (= 42d) beginnen.

Da der Platz in der Interrupt-Vektortabelle natürlich nicht für den gesamten Code der Unterbrechungsroutine (ISR) reicht, wird hier mit einem Sprungbefehl auf die ISR

12 Befindet sich ein Interrupt-Flag in einem der unteren 32 SF-Register (wie zum Beispiel das Flag für den AD-Wandler) so können natürlich statt einer Maskierung die Befehle **sbi** und **cbi** verwendet werden

13 Der in der Definitionsdatei vorgesehene Name für diese Adresse heißt: **INT_VECTORS_SIZE**

verzweigt. Dieser Sprungbefehl ist veränderbar und muss am Anfang des Programms initialisiert werden, damit ein entsprechender Interrupt erfolgen kann[14].



Da die Interrupt-Vektortabelle hardwaremäßig festgelegt wurde, ist die Reihenfolge der Interrupt-Vektoren nicht veränderbar. Die Reihenfolge bestimmt die Prioritäten der Interrupts. Umso niedriger die Adresse, desto höher die Priorität. Der externe Interrupt **`INT0`** hat also nach dem Reset die höchste Priorität. Treten zum Beispiel gleichzeitig ein externer Interrupt und ein Interrupt an der seriellen Schnittstelle auf, so wird zuerst der externe Interrupt ausgeführt.

Bemerkungen:

Bei anderen Controllern oder Prozessoren werden Interrupts oft anders behandelt als bei der AVR-Familie. Hier sind eventuell die Vektortabellen frei programmierbar und ein Interrupt-Controller übernimmt einen Teil der Interrupt-Steuerung.

Das eigentliche Programm kann erst an der Adresse **`0x002A`** beginnen. Der in der Definitionsdatei vorgesehene Name für diese Adresse heißt: **`INT_VECTORS_SIZE`**. Der Sprungbefehl an der Adresse **`0x0000`** verzweigt also normalerweise auf diese Adresse.

Die Vektortabelle beginnt mit der Adresse **`0x0000`**, (Einsprungadresse des **`RESET`**s). Wenn aber das **`BOOTRST`**-Fuse-Bit programmiert wurde

14 Da der veränderbare Sprungbefehl auf die ISR zeigt, kann man hier von einer zweiten softwaremäßigen Interrupt-Vektortabelle sprechen.

springt der Controller nach einem **RESET** automatisch in den Bootbereich (Bootloader-Programm) im oberen Adressbereich des Flash-Speichers. Mit Hilfe des **IVSEL**-Bit im SF-Register **GICR** kann dann die Interrupt-Vektortabelle in den Anfang des Bootbereichs verlegt werden.

Die Interrupt-Vektortabellen des ATmega8 und des ATmega16 unterscheiden sich von der Interrupt-Vektortabelle des ATmega32!

Interrupt-Vektor-Tabelle des ATmega32:

Vektor-nummer	Adresse im Flash	Name in "m32def.inc"	Quelle des Interrupts	verantwortlich für den Interrupt
21	0x0028	SPMRaddr	SPM_RDY	Interface für Programmspeicher
20	0x0026	TWIaddr	TWI	I²C-Interface
19	0x0024	ACIaddr	ANA_COMP	Analog-Komparator
18	0x0022	ERDYaddr	EE_RDY	EEPROM-Interface
17	0x0020	ADCCaddr	ADC	A/D-Wandlung vollständig
16	0x001E	UTXCaddr	USART, TXC	USART, Senderegister leer
15	0x001C	UDREaddr	USART, UDRE	USART, UDR-Register leer
14	0x001A	URXCaddr	USART, RXC	USART, Empfangsregister voll
13	0x0018	SPIaddr	SPI, STC	Serielle Übertragung beendet
12	0x0016	OVF0addr	TIMER0 OVF	Overflow von Timer 0
11	0x0014	OC0addr	TIMER0 COMP	Compare Match von Timer 0
10	0x0012	OVF1addr	TIMER1 OVF	Overflow von Timer 1
9	0x0010	OC1Baddr	TIMER1 COMPB	Compare Match B von Timer 1
8	0x000E	OC1Aaddr	TIMER1 COMPA	Compare Match A von Timer 1
7	0x000C	ICP1addr	TIMER1 CAPT	Capture Event von Timer 2
6	0x000A	OVF2addr	TIMER2 OVF	Overflow Match von Timer 2
5	0x0008	OC2addr	TIMER2 COMP	Compare Match von Timer 2
4	0x0006	INT2addr	INT2	externer Interrupt-Eingang 2
3	0x0004	INT1addr	INT1	externer Interrupt-Eingang 1
2	0x0002	INT0addr	INT0	externer Interrupt-Eingang 0
1	0x0000		RESET	externer Pin, Power-On-Reset, Brown-Out-Reset, Watchdog Reset, ...

Interrupt-Behandlung

Tritt ein Interrupt auf, so wird das zu diesem Interrupt zugehörige Flag (Bit in einem SF-Register) gesetzt. Sind Interrupts noch nicht zugelassen, so behält (speichert) dieses Flag das Auftreten des Interrupts, so dass dieses zu einem späteren Zeitpunkt ausgelöst werden kann.

Drei Bedingungen müssen erfüllt sein, damit der Controller Interrupts ausführt:

1. **Interrupts müssen global freischaltet sein.**
   (Hauptschalter, Befehle **`sei`** und **`cli`**).
2. **Interrupts müssen zusätzlich einzeln freigeschaltet werden.**
   (Setzen des spezifischen Interrupt-Bits (Flags) im entsprechenden SF-Register)
3. **Es muss ein Ereignis auftreten, das einen Interrupt auslöst** (und ein zum Interrupt gehörendes spezifisches Flag setzt).

Sind die drei Bedingungen erfüllt, so wird das Programm unterbrochen. Die Hardware (Interruptsteuerung) des AVR-Controllers übernimmt folgende Aufgaben:

- Der gerade abgearbeitete **Befehl wird beendet**.
- Die **Rücksprungadresse** (Adresse des folgenden Befehls) wird automatisch **auf den Stapel** gerettet[15].
- Weitere **Interrupts werden unterbunden** indem das globale Interrupt-Flag gelöscht wird (entspricht dem Befehl **`cli`**)
- Das **spezifische Interrupt-Flag** der Baugruppe **wird gelöscht**, damit der Interrupt nicht erneut auftritt.
- Der Befehlszähler (PC[16]) wird mit der dem Interrupt entsprechenden Adresse der Vektortabelle geladen und der sich an dieser Adresse befindende **Sprungbefehl zur Unterbrechungsroutine (ISR) wird ausgeführt**.

Nach dem die Unterbrechungsroutine abgearbeitet wurde übernimmt die hardwaremäßige Interruptsteuerung wieder:

- Die **Rücksprungadresse** wird vom Stapel **in den Befehlszähler** geladen.
- **Interrupts werden wieder zugelassen** indem das globale Interrupt-Flag gesetzt wird (entspricht dem Befehl **`sei`**).
- Mindestens **ein Befehl des Hauptprogramms wird ausgeführt**[17].

15 Dies setzt voraus, dass der Stapel initialisiert wurde!!

16 *Program Counter* oder auch noch *Instruction Pointer*

17 Ein erneuter Interrupt kann erst auftreten, wenn mindestens ein Befehl des laufenden Programms ausgeführt wurde.

sei

Setze globales Interrupt-Flag im SREG-Register I = 1 (*set global interrupt flag*).

1	0	0	1	0	1	0	0	0	1	1	1	1	0	0	0

Der Befehl nach dem sei-Befehl wird noch ausgeführt bevor Interrupts zugelassen sind.
Beeinflusste Flags: I **Taktzyklen: 1**

cli

Lösche globales Interrupt-Flag im SREG-Register I = 0 (*clear global interrupt flag*).

1	0	0	1	0	1	0	0	1	1	1	1	1	0	0	0

Interrupts werden sofort unterbunden. Tritt ein Interrupt gleichzeitig mit den cli-Befehl auf, so wird dieses Interrupt nicht mehr ausgeführt.
Beeinflusste Flags: I **Taktzyklen: 1**

Die Behandlungsroutine (ISR)

Die Interruptsteuerung kümmert sich nicht um das Retten wichtiger Registerinhalte, bzw. der Statusflags. Alle von der Behandlungsroutine veränderten Register (Arbeitsregister, SF-Register (z.B.: Statusflags im **SREG**-Register) und SRAM-Speicherzellen) müssen unbedingt gleich nach dem Eintritt in die Behandlungsroutine auf den Stapel gerettet werden und vor dem Verlassen der ISR wiederhergestellt werden.

Alle Registerinhalte, die in der Behandlungsroutine (ISR) verändert werden, müssen am Anfang der ISR auf den Stapel gerettet werden. Dies gilt auch für das Statusregister SREG.
Am Ende der ISR werden die Registerinhalte wiederhergestellt.

Der Befehl zum Rücksprung aus einer ISR unterscheidet sich vom Rücksprungbefehl der Unterprogramme durch ein angehängtes „i“ (`reti`).

Der Code für eine typische ISR, welche die Register **r16** und **r17** als interne Variablen benutzt, könnte folgendermaßen aussehen:

```
;-------------------------------------------------------------------------
;       ISR mit zwei internen Variablen
;-------------------------------------------------------------------------

ISR_I1: push    r16             ;benutzte Reg. retten (r16 = Zwischenspeicher)
        in      r16,SREG        ;Statusregister einlesen
        push    r16             ;Statusregister retten
        push    r17

        ;Code der ISR

        pop     r17             ;Achtung !! Umgekehrte Reihenfolge !!
        pop     r16             ;Werte der geretteten Register wieder-
        out     SREG,r16        ;herstellen
        pop     r16
        reti                    ;Ruecksprung ins Hauptprogramm aus einer
                                ;Interrupt-Behandlungsroutine
```

reti

Rücksprung aus einer ISR (*return from interrupt*).

1	0	0	1	0	1	0	1	0	0	0	1	1	0	0	0

Die Rücksprungadresse wird vom Stapel genommen. Interrupts werden global wieder zugelassen (I = 1).
Beeinflusste Flags: I **Taktzyklen: 4**

Weitere Informationen zur Interruptbehandlung

- Tritt ein Interrupt auf, so wird das spezifische Interruptflag gesetzt. Dieses Flag kann nicht manuell gesetzt werden. Das Löschen des Interrupt-Flags ist durch Schreiben einer Eins in das entsprechende Bit des Registers möglich!!
- Treten Interrupts auf, während sie global oder spezifisch gesperrt sind, so wird mittels des jeweiligen Flags behalten, dass das Interrupt auftrat. Sobald die Sperren aufgehoben sind, werden die Interrupts nach ihrer Priorität ausgeführt. Es besteht aber die Möglichkeit das Flag manuell zu löschen bevor die Sperren aufgehoben sind. So kann man zum Beispiel verhindern, dass ein Interrupt zweimal ausgeführt wird.
- Treten mehrere Interrupts gleichzeitig auf, so wird das Interrupt mit der höchsten Priorität als erstes ausgeführt. Die Interruptflags der anderen Interrupts werden gesetzt, und somit ihr Auftreten gespeichert. Nach der Abarbeitung des ersten Interrupts wird ein Befehl des Programms ausgeführt, bevor dann das Interrupt mit der zweit höchsten Priorität ausgeführt wird usw.
- Befindet sich der Controller in einer ISR, so sind Interrupts global gesperrt und auch das zu diesem Interrupt gehörende Flag wurde gelöscht. Tritt jetzt das gleiche Interrupt zu diesem Zeitpunkt noch einmal auf, so wird die gleiche ISR nach dem Verlassen und dem Abarbeiten eines Befehls noch einmal ausgeführt. Dies kann

durch das Löschen des Flags (durch Schreiben einer Eins) kurz vor Verlassen der ISR unterbunden werden.

- Es besteht die Möglichkeit Interrupts mittels des **sei**-Befehls innerhalb einer ISR wieder zuzulassen. So können verschachtelte Interrupts erfolgen. Hierbei ist allerdings allergrößte Vorsicht geboten. Insbesondere dürfen die Stapeloperationen nicht unterbrochen werden.
- Tritt der **cli**-Befehl gemeinsam mit einem Interrupt auf, so überwiegt der **cli**-Befehl. Der auftretende Interrupt wird nicht mehr ausgeführt.
- Wird ein Interrupt zugelassen, und tritt auf, so muss der Sprungbefehl in der Vektortabelle für diesen Interrupt vorhanden sein und es muss eine ISR zu diesem Interrupt mit mindestens einem **reti**-Befehl existieren. Ist dies nicht der Fall, so durchläuft der Controller die Vektor-Tabelle weiter bis er auf den nächsten Sprungbefehl trifft und führt diesen aus. Die Wirkung wäre, dass die ISR des folgenden Interrupt fälschlicherweise ausgeführt wird.

Externe Interrupts

Der ATmega32 verfügt über drei externe Interrupts, d.h. an drei speziellen Pins des Controllers können Signale ein Interrupt auslösen[18].

Es handelt sich hierbei um die Pins **PD2** (**INT0**), **PD3** (**INT1**)und **PB2** (**INT2**).

Diese Pins sollten, falls sie für externe Interrupts benutzt werden, auch als Eingang initialisiert werden[19]. Die Pull-Up-Widerstände können auch bei Bedarf zugeschaltet werden!

INT2 hat die geringste Priorität und kann nur auf fallende oder steigende Flanken des Eingangssignals reagieren. Die beiden anderen Interrupts **INT0** und **INT1** können zusätzlich auf einen Low-Pegel (Zustand), und auf beliebige Pegeländerung (beide Flanken) reagieren.

Die Initialisierung

Zur **Initialisierung und Statusabfrage** der externen Interrupts dienen vier SF-Register:

- **MCUCR = MCU Control Register**
- **MCUCSR = MCU Control and Status Register**

18 Reichen drei externe Interrupts nicht aus, so bietet sich als Alternative zum ATmega32 der pinkompatible ATmega324PA an mit 32 externen Interrupts.

19 Sie dürfen, falls eine Interrupt-Quelle angeschlossen ist, keinesfalls als Ausgang initialisiert werden, da sonst ein Kurzschluss entstehen kann. Sind sie ohne Interrupt-Quelle als Ausgang initialisiert, so kann durch Schalten des Ausgangs ein Interrupt ausgelöst werden, und somit ein Software-Interrupt simuliert werden.

- **GICR = General Interrupt Control Register**
- **GIFR = General Interrupt Flag Register**

Damit ein externes Interrupt genutzt werden kann sind folgende Initialisierungsschritte notwendig:

1. Der richtige Sprungbefehl zur ISR muss in die Vektortabelle eingetragen werden.
2. Interrupts müssen global erlaubt werden (**sei**).
3. Im SF-Register **MCUCR** (für **INT0** und **INT1**) bzw. **MCUCSR** (für **INT2**) muss festgelegt werden auf welche Flanke bzw. auf welchen Pegel der Interrupt reagieren soll.
4. Der spezifische Interrupt muss im SF-Register **GICR** freigeschaltet werden.

Zusätzlich dazu muss eine ISR geschrieben werden, in der alle benutzten Register und Flags gerettet und wiederhergestellt werden. Im SF-Register **GIFR** befinden sich die Interrupt-Flags, die bei Bedarf durch Schreiben einer 1 gelöscht werden können.

Die SF-Register für externe Interrupts

Das Interrupt-Kontrollregister GICR

GICR-Register: SF-Register-Adresse **0x3B** (SRAM-Adresse **0x005B**)
Befehle: **in**, **out** (**sbi**, **cbi**, **sbic**, **sbis** nicht da Adresse > 32 (0x1F)!)
Eine Maskierung beim Zugriff ist sinnvoll um **IVSEL** nicht zu verstellen. Es werden hier nur die drei oberen Bits benötigt um den jeweiligen Interrupt zu aktivieren.

GICR = General Interrupt Control Register

Bit	7	6	5	4	3	2	1	0
GICR 0x3B	**INT1**	**INT0**	**INT2**	**-**	**-**	**-**	**IVSEL**	**IVCE**
Startwert	0	0	0	0	0	0	0	0
Read/Write	R/W	R/W	R/W	R	R	R	R/W	R/W

INTn ***External INTerrupt Request n Enable***

0 Das betreffende Interrupt ist nicht aktiviert.
1 Das betreffende Interrupt ist aktiviert, wenn ebenfalls das **I**-Bit in **SREG** gesetzt ist (Interrupts global erlaubt, **sei**). Ein Interrupt wird auch erkannt, wenn das betreffende Pin als Ausgang initialisiert wurde.

Das Interrupt-Flagregister GIFR

GIFR-Register: SF-Register-Adresse **0x3A** (SRAM-Adresse **0x005A**)
Befehle: **in**, **out** (**sbi**, **cbi**, **sbic**, **sbis** nicht da Adresse > 32 (0x1F)!)

Die Flags zeigen an ob ein Interrupt aufgetreten ist. Sind Interrupts zum Zeitpunkt des Auftretens gesperrt, so speichert das Flag das Ereignis.

GIFR = General Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0
GIFR 0x3A	**INTF1**	**INTF0**	**INTF2**	**-**	**-**	**-**	**-**	**-**
Startwert	0	0	0	0	0	0	0	0
Read/Write	R/W	R/W	R/W	R	R	R	R	R

INTFn ***External INTerrupt Flag n***

0 Es trat kein Interrupt auf. Das Flag wird automatisch gelöscht wenn die Interruptroutine aufgerufen wird. Es ist beim **INT0** bzw. **INT1** dauernd gelöscht, wenn die Interrupts auf Pegel reagieren sollen.

1 Es wurde ein Interrupt erkannt und dies wird im Flag gespeichert. Passiert dies innerhalb einer Interruptroutine (wo standardmäßig Interrupts global gesperrt sind), so wird das Interrupt nach dem Verlassen der Routine ausgeführt.
Mit dem **Schreiben einer Eins!** kann das Interrupt-Flag manuell gelöscht werden.

Das MCU-Kontrollregister MCUCR

MCUCR-Register: SF-Register-Adresse **0x35** (SRAM-Adresse **0x0055**)
Befehle: **in**, **out** (**sbi**, **cbi**, **sbic**, **sbis** nicht da Adresse > 32 (0x1F)!)
Eine Maskierung beim Zugriff ist sinnvoll. Es werden hier nur die vier niederwertigsten Bit zur Flankensteuerung vom INT0 (Bit 0 und 1) und INT1 (Bit 2 und 3) benötigt.

MCUCR = MCU Control Register

Bit	7	6	5	4	3	2	1	0
MCUCR 0x35	**SE**	**SM2**	**SM1**	**SM0**	**ISC11**	**ISC10**	**ISC01**	**ISC00**
Startwert	0	0	0	0	0	0	0	0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

ISCn ***Interrupt Sense Control n ISCn1, ISCn0***

Mit je zwei Bit kann bei **INT0** und bei **INT1** ausgewählt werden, ob der Interrupt auf einen Zustand (Pegel) oder auf eine Flanke reagieren soll (zustandsgesteuert bzw. flankengesteuert).

00 Der Low-Zustand löst einen Interrupt aus.
01 Jede Zustandsänderung löst einen Interrupt aus.
10 Eine fallende Flanke löst einen Interrupt aus.
11 Eine steigende Flanke löst einen Interrupt aus.

Das MCU-Kontrollregister und Statusregister MCUCSR

MCUCSR-Register: SF-Register-Adresse **0x34** (SRAM-Adresse **0x0054**)
Befehle: **in**, **out** (**sbi**, **cbi**, **sbic**, **sbis** nicht da Adresse > 32 (0x1F)!)

Eine Maskierung beim Zugriff ist sinnvoll, da hier nur Bit 6 (für INT2) benötigt wird.

MCUCSR = MCU Control and Status Register

Bit	7	6	5	4	3	2	1	0
MCUCSR 0x34	**JTD**	**ISC2**	**-**	**JTRF**	**WDRF**	**BORF**	**EXTRF**	**PORF**
Startwert	0	0	0	-	-	-	-	-
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W

ISC2 ***Interrupt Sense Control 2***

INT2 kann nur flankengesteuert betrieben werden.

0 Eine fallende Flanke löst einen Interrupt aus.
1 Eine steigende Flanke löst einen Interrupt aus.

Programmierbeispiel:

Das folgende Programmierbeispiel soll aufzeigen wie externe Interrupts eingesetzt werden können.

Zwei Interrupt-Signale an **INT0** (**PD2**) und **INT1** (**PD3**) steuern eine LED am Ausgang **PC0** des Controllers. Zur Erzeugung der Interrupt-Signale werden zwei entprellte Taster verwendet. Eine steigende Flanke an **INT0** soll die LED einschalten. Eine fallende Flanke schaltet die LED wieder aus.

Im Hauptprogramm soll eine zweite LED (**PC1**) im Sekundentakt blinken.

```
;*****************************************************************************
;*
;*      Titel:  Beispiel zur Interrupt-Steuerung (B400_int_example.asm)
;*      Datum:  30/12/09                Version:        0.4 (03/01/13)
;*      Autor:  www.weigu.lu
;*
;*      Informationen zur Beschaltung:
;*      Prozessor:      ATmega32        Quarzfrequenz:  16MHz
;*      Eingaenge:      entprellter Taster an PORTD2 (INT0)
;*                      entprellter Taster an PORTD3 (INT1)
;*      Ausgaenge:      LED an PORTC0
;*
;*      Informationen zur Funktionsweise:
;*
;*      Eine steigende Flanke an INT0 soll LED einschalten.
;*      Eine fallende Flanke an INT1 soll die LED ausschalten.
;*      Im Hauptprogramm blinkt eine LED
;*
;*****************************************************************************

;-----------------------------------------------------------------------------
;       Einbinden der controllerspezifischen Definitionsdatei
;-----------------------------------------------------------------------------
.NOLIST                         ;List-Output ausschalten
.INCLUDE "m32def.inc"           ;AVR-Definitionsdatei einbinden
.LIST                           ;List-Output wieder einschalten

;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
;       Programmspeicher (FLASH)     Programmstart nach RESET ab Adr. 0x0000
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
.CSEG                           ;was ab hier folgt kommt in den FLASH-Speicher
.ORG    0x0000                  ;Programm beginnt an der FLASH-Adresse 0x0000
RESET:  rjmp    INIT            ;springe nach INIT (ueberspringe ISR Vektoren)

;-------------------------------------------------------------------------------
;       Sprungadressen fuer die Interrupts organisieren (ISR VECTORS)
;-------------------------------------------------------------------------------
; Vektortabelle (im Flash-Speicher)
.ORG    INT0addr                ;interner Vektor für INT0 (alt.: .ORG 0x0002)
        rjmp    ISR_I0          ;Springe zur ISR von INT0
.ORG    INT1addr                ;interner Vektor für INT1 (alt.: .ORG 0x0004)
        rjmp    ISR_I1          ;Springe zur ISR von INT1

;-------------------------------------------------------------------------------
;       Initialisierungen und eigene Definitionen
;-------------------------------------------------------------------------------
.ORG    INT_VECTORS_SIZE        ;Platz fuer ISR Vektoren lassen
INIT:
.DEF    Zero = r15              ;Register 1 wird zum Rechnen benoetigt
        clr     r15             ;und mit Null belegt
.DEF    Tmp1 = r16              ;Register 16 dient als erster Zwischenspeicher
.DEF    Tmp2 = r17              ;Register 17 dient als zweiter Zwischenspeicher
.DEF    Cnt1 = r18              ;Register 18 dient als Zaehler
.DEF    WL = r24                ;Register 24 und 25 dienen als universelles
.DEF    WH = r25                ;Doppelregister W und zur Parameteruebergabe
.DEF    W = r24

.EQU    LEDI = 0                ;LED ISR wird an Pin 0 angeschlossen
.EQU    LEDM = 1                ;LED MAIN an Pin 1 angeschlossen

        ;Stapel initialisieren (fuer Unterprogramme bzw. Interrupts)
        ldi     Tmp1,LOW(RAMEND) ;RAMEND (SRAM) ist in der Definitions-
        out     SPL,Tmp1         ;datei festgelegt
        ldi     Tmp1,HIGH(RAMEND)
        out     SPH,Tmp1

        ;externe Interrupts initialisieren
        in      Tmp1,MCUCR              ;MCU Control Register einlesen
        andi    Tmp1,0b11111011         ;ISC11=1, ISC10=0 => fallende Flanke
        ori     Tmp1,0b00001011         ;ISC01=1, ISC00=1 => steigende Flanke
        out     MCUCR,Tmp1              ;Wert ins MCUCR zurückschreiben

        in      Tmp1,GICR               ;General Int. Control Reg. einlesen
        ori     Tmp1,0b11000000         ;INT1=1, INT0=1 => INT0+INT1 aktivieren
        out     GICR,Tmp1               ;GICR setzen

        ;Port C und Port D konfigurieren
        cbi     DDRD,2                  ;INT0 = Eingang
        cbi     DDRD,3                  ;INT1 = Eingang
        sbi     DDRC,LEDI               ;Richtungsbit fuer LEDI auf Ausgang
        sbi     DDRC,LEDM               ;Richtungsbit fuer LEDM auf Ausgang

        ;globales Interrupt-Flag setzen (Int. erlauben)
        sei

;-------------------------------------------------------------------------------
;       Hauptprogramm
;-------------------------------------------------------------------------------
MAIN:   sbi     PORTC,LEDM              ;LED im Sekundentakt toggeln
        rcall   W500MS
        cbi     PORTC,LEDM
        rcall   W500MS
        rjmp    MAIN                    ;Endlosschleife

;-------------------------------------------------------------------------------
;       Unterprogramme und Interrupt-Behandlungsroutinen
;-------------------------------------------------------------------------------
; Interrupt-Behandlungsroutine von INT0
ISR_I0: push    r16             ;benutzte Reg. retten (r16 = Zwischenspeicher)
```

```
        in      r16,SREG        ;Statusregister einlesen
        push    r16             ;Statusregister retten

        sbi     PORTC,LEDI      ;LED ein

        pop     r16             ;Werte der geretteten Register wieder-
        out     SREG,r16        ;herstellen
        pop     r16
        reti                    ;Ruecksprung ins Hauptprogramm aus einer
                                ;Interrupt-Behandlungsroutine

; Interrupt-Behandlungsroutine von INT1
ISR_I1: push    r16             ;benutzte Reg. retten (r16 = Zwischenspeicher)
        in      r16,SREG        ;Statusregister einlesen
        push    r16             ;Statusregister retten

        cbi     PORTC,LEDI      ;LED aus

        pop     r16             ;Werte der geretteten Register wieder-
        out     SREG,r16        ;herstellen
        pop     r16
        reti                    ;Ruecksprung ins Hauptprogramm aus einer
                                ;Interrupt-Behandlungsroutine

.INCLUDE "lib/SR_TIME_16M.asm"  ;Zeitschleifenbibliothek einbinden
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
.EXIT                           ;Ende des Quelltextes
```

Bemerkung: Bei den Steuerregistern **MCUCR**, **GICR** und dem Portregister **PORTC** müssen nur einige Bits gesetzt oder gelöscht werden. Im Programm wird das durch eine Maskierung erreicht (siehe Modul A Kapitel 3). Die Befehle **sbi** und **cbi** können hier leider nicht eingesetzt werden, da nur die unteren 32 SF-Register mit diesen Befehlen erreicht werden können. Jeweils das ganze Register mit **ldi** zu setzen ist keine gute Idee, da die nicht benutzte, allerdings nicht unbedingt bedeutungslose Bits verstellt werden, und zu schwer auffindbaren Fehlern führen können.

Aufgaben

✎ **B400** Teste das obige Programm.
Nenne das Programm **"B400_int_example".**

✎ **B401** Ein Programm soll die positiven Flanken eines prellenden Schalters an **PD2** (**INT0**) zählen (Pull-Up-Widerstand zuschalten!) und an 8 LEDs ausgeben. Die Zählervariable (**COUNT1**) soll sich im **SRAM** befinden!

a) Zeichne das entsprechende Flussdiagramm.
b) Schreibe das kommentierte Assemblerprogramm.
Nenne das Programm **"B401_int_counter_1"**.

✎ **B402** Das vorige Programm soll erweitert werden. Eine positive Flanke an **PD3** (**INT1**) soll bewirken, dass der die Zählrichtung verändert wird (aus einem Vorwärtszähler wird ein Rückwärtszähler). Eine positive Flanke an **INT2** setzt den Zähler auf Null zurück. Treten **INT1** oder **INT2** auf, so soll deren

Auftreten jeweils durch eine Flagvariable[20] im SRAM markiert werden (z.B.: **COUNTD** und **COUNTZ**). Die ISR von **INT0** muss dann diese Flags auswerten. Nenne das Programm **"B402_int_counter_2"**.

✎ **B403** Interessanterweise reicht es das **MCUCR**-Register (bzw. **MCUCSR**) zu initialisieren, damit die Interruptsflags bei zum Beispiel einer fallenden Flanke am entsprechenden Eingang gesetzt werden. So kann, **ohne die Benutzung von Interrupts**, ein Ereignis über einen längeren Zeitraum gespeichert werden. Teste diese Eigenschaft, indem du ein Programm schreibst, das alle 10 Sekunden das Flag von **INT0** abfragt. **MCUCR** wurde für eine fallende Flanke initialisiert. Das Flag muss danach mit einer Eins gelöscht werden. An **PD2** wird ein Taster (Pull-Up aktivieren) angeschlossen.
Nenne das Programm **"B403_noint_flagtest"**.

Bemerkungen: Wird der Pull-Up-Widerstand erst nach der Initialisierung des **MCUCR**-Register zugeschaltet, so kann nach einem **RESET** durch Störstrahlung bereits eine Interruptanforderung gespeichert werden, und somit ein Interrupt ausgelöst werden, der nicht auftreten soll.

Auch ist es wichtig nach dem Zuschalten des Pull-Up ungefähr 1μs zu warten (NOPs oder Zeitschleife) bevor das **MCUCR**-Register zugeschaltet wird, da es bei ungünstigem Layout bis zu 10 Taktzyklen (bei 16 MHz) dauern kann bis der Pull-Up wirklich aktiv ist.

20 Unter Flagvariable ist hier ein Byte zu verstehen, dass nur zwei verschiedene Zustände kennt. Dazu kann dann zum Beispiel Bit 0 verwendet werden. Die Variable wird also als Wert entweder 0x00 oder 0x01 enthalten.