Skriptum Mikrocontroller

Teil 5 Interrupts

DI Wolfgang Zukrigl, HTL Litec
DI(FH) Andreas Pötscher, HTL Litec

Inhaltsverzeichnis

Datum	Beschreibung der durchgeführten Änderung	Autor
12.05.2014	V1.0 . . . Korrektur kleinerer Fehler	Wolfgang Zukrigl, HTL Linz–Paul-Hahn-Straße (LiTec)
19.09.2014	V1.1 . . . Übernahme in Markdown und allgemeine Überarbeitung	Wolfgang Zukrigl, HTL Linz–Paul-Hahn-Straße (LiTec)
24.02.2019	V2.0 . . . : Einfügen von Merksätzen, Beispielen, Aufgaben. Viele Teile neu überarbeitet.	Wolfgang Zukrigl, HTL Linz–Paul-Hahn-Straße (LiTec)

Datum	Beschreibung der durchgeführten Änderung	Autor
01.04.2020	V2.1 ... Fehler bei Beispiel ausgebessert (PE6/INT6 auf PE5/INT5 geändert)	Wolfgang Zukrigl, HTL Linz–Paul-Hahn-Straße (LiTec)
19.03.2025	V2.2 ... Übernahme in Markdown und allgemeine Überarbeitung	Andreas Pötscher, HTL Linz–Paul-Hahn-Straße (LiTec)

Polling vs Interrupts

Polling

Polling nennt man das zyklische Abfragen eines „Hardware-Zustandes“ (z. B. des Zustands eines GPIO-Eingangs). Der aktuelle Zustand ist dabei nicht interessant, sondern es wird darauf gewartet, ob sich der Zustand geändert hat.

> **Beispiel:**
>
> Ein Sensor an der rotierenden Achse eines Motors liefert bei jeder Umdrehung einen High-Impuls (ähnlich wie bei einem Fahrrad-Tacho, bei dem ein Magnet in den Speichen bei jeder Umdrehung einen Reed-Kontakt schließt). Der Motor dreht sich mit bis zu 6.000 Umdrehungen pro Minute.
>
> Das Sensorsignal ist an einen GPIO-Pin eines Mikrocontrollers angeschlossen, sodass dieser die Anzahl der bisherigen Umdrehungen des Motors erfassen kann. Ein konkretes Anwendungsbeispiel sind Aufwickelvorgänge (z. B. für Fäden, Folien oder Transformatorwicklungen), bei denen eine bestimmte Anzahl an Windungen erreicht werden soll.
>
> Das Signal am GPIO-Pin ist für eine halbe Motorumdrehung High und für die restliche halbe Umdrehung Low. Wie oft muss der Mikrocontroller den Zustand des GPIO-Pins abfragen?

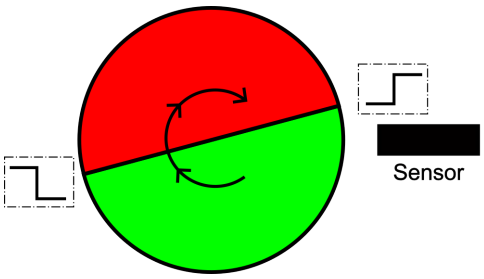

Abbildung 1: Schema des Beispiels

Lösung:

Motor-Drehzahl:

$$n = 6.000 \frac{\text{Umdrehungen}}{\text{Minute}} = 100 \frac{\text{Umdrehungen}}{\text{Sekunde}}$$

Dauer einer Umdrehung:

$$T = \frac{1}{100\,\text{s}^{-1}} = 10\,\text{ms}$$

Da das Signal für eine halbe Umdrehung High und für die andere Hälfte Low ist, wechselt es seinen Zustand nach:

$$\frac{T}{2} = 5\,\text{ms}$$

Der Mikrocontroller muss daher mindestens alle **5 Millisekunden** den Zustand des GPIO-Pins abfragen („pollen“), da sonst ein gesamter High-Impuls und damit eine komplette Motorumdrehung verloren gehen könnte.

Polling ist sehr rechenintensiv, vor allem, wenn Zustände in kurzen Zeitabständen abgefragt werden müssen.

Wenn der Mikrocontroller in jedem Programmzyklus neben dem Polling noch eine weitere Aufgabe ausführen muss, darf diese maximal so lange dauern, wie es die Zykluszeit des Pollings erlaubt. Ein Durchlauf der Endlosschleife im Hauptprogramm des Mikrocontrollers darf also höchstens so lang sein wie die Polling-Zykluszeit.

Länger andauernde Aufgaben (z. B. die Ausgabe eines Strings über die serielle Schnittstelle) wären dann nicht mehr möglich.

> **Wichtig!**
>
> Polling bedeutet zyklisches Abfragen eines Hardware-Zustands (z.B: Pegel eines GPIO-Pins), um Änderungen des Hardware-Zustands erkennen, und darauf reagieren zu können.

Interrupts

Polling ist sehr rechenintensiv. Manche Aufgaben lassen sich mit Polling nur schwer oder gar nicht lösen. Wie bereits erwähnt, ist dies insbesondere dann der Fall, wenn sowohl „schnelle“ als auch selten auftretende, aber länger andauernde Aufgaben gleichzeitig ausgeführt werden müssen.

Aus diesem Grund verfügen Prozessoren und Mikrocontroller über sogenannte **Interrupts**. Dabei erkennt die Peripherie unabhängig von der CPU und dem laufenden Programm, dass sich ein Hardware-Zustand verändert hat.

Die Peripherie signalisiert der CPU, dass ein solches Ereignis aufgetreten ist, indem sie ein sogenanntes **Interrupt-Flag** (ein Bit in einem Special-Function-Register) auf eins setzt. Sobald die CPU erkennt, dass ein Interrupt-Flag gesetzt wurde, unterbricht sie das aktuelle Programm und führt das zugehörige Unterprogramm aus.

Ein solches Unterprogramm wird als **Interrupt-Service-Routine (ISR)** oder **Interrupt-Handler** bezeichnet. Nachdem die Interrupt-Service-Routine vollständig abgearbeitet wurde, setzt die CPU das Hauptprogramm genau an der Stelle fort, an der es zuvor unterbrochen wurde.

In der folgenden Abbildung wird von der Hardware ein Interrupt beispielhaft in Zeile 2 des While-Loops ausgelöst. Der Interrupt kann aber auch in jeder anderen Zeile ausgelöst werden. Daraufhin wird der While-Loop unterbrochen und die ISR ausgeführt. Dannach wird der While-Loop wieder an der vorherigen Stelle weiter ausgeführt.

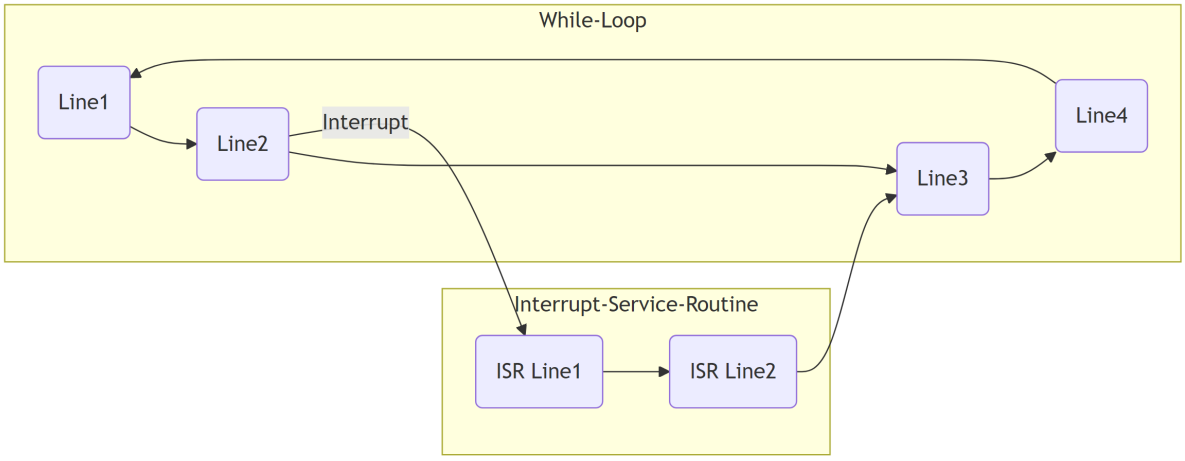

Abbildung 2: Ablauf eines Interrupt mit zugehöriger Interrupt Service Routine

> **Beispiel:**
>
> Das vorige Beispiel soll nicht mit Polling, sondern mit einem Interrupt gelöst werden.

Lösung: Der Sensorausgang muss mit einem interrupt-fähigen GPIO-Eingang des Mikrocontrollers verbunden werden. In der Setup-Phase des Programms muss der Interrupt aktiviert („enabled“) werden. Der Interrupt soll bei steigenden Flanken am GPIO-Eingang (also bei einem Pegelwechsel von Low nach High) ausgelöst werden. In der Interrupt-Service-Routine (ISR) wird eine globale Variable inkrementiert, die die Anzahl der bisher vom Motor ausgeführten Umdrehungen speichert. Im Hauptprogramm kann dieser Wert jederzeit ausgelesen werden. Beispielsweise könnte der Motor gestoppt werden, sobald 1.000 Umdrehungen (Windungen) erreicht sind.

> **Wichtig!**
>
> Bei der Verwendung von Interrupts überlässt man das Erkennen von Hardware-Zustandsänderungen der Peripherie des Mikrocontrollers. Diese signalisiert der CPU mittels eines Interrupt-Flags, dass eine Änderung aufgetreten ist. Daraufhin unterbricht die CPU das laufende Hauptprogramm, führt die Interrupt-Service-Routine (ISR) aus und setzt anschließend das Hauptprogramm fort.

Funktionsweise von Interrupts

Die folgenden Ausführungen gelten nicht nur für AVR, sondern generell für Prozessoren und Mikrocontroller. Die konkreten Beispiele und Register beziehen sich jedoch auf den ATMega2560.

Interrupt-Flags, Interrupt-Enable-Flags, Interrupt-Vektoren

Jedes Mal, wenn die Peripherie des Mikrocontrollers ein Interrupt-Ereignis erkennt, wird das zugehörige Interrupt-Flag gesetzt. Der ATmega2560 verfügt über etwa 56 verschiedene Hardware-Ereignisse, die einen Interrupt auslösen können. Entsprechend gibt es 56 Interrupt-Flags und 56 Interrupt-Service-Routinen (ISRs).

Ein Interrupt wird aber nur ausgelöst, wenn:

- Der zugehörige Interrupt muss aktiviert („enabled“) sein. Dies ist der Fall, wenn das entsprechende Interrupt-Enable-Flag in der Initialisierungsphase des Programms gesetzt wurde. Ist dieses Bit nicht gesetzt, wird der Interrupt nicht aktiviert. Dies bezeichnet man als „Ausmaskieren“ des Interrupts.
- Interrupts global erlaubt sind. Bei den AVR-Mikrocontrollern gibt es den C-Befehl `sei();` zum globalen Erlauben, sowie den Befehl `cli();` zum globalen Sperren von Interrupts.

> **Wichtig!**
>
> Damit eine Interrupt-Service-Routine (ISR) tatsächlich ausgeführt wird, muss das zugehörige Interrupt-Enable-Flag vom Programm gesetzt worden sein. Zudem müssen Interrupts global aktiviert sein. Das Interrupt-Flag selbst wird von der Peripherie gesetzt und signalisiert der CPU, dass ein Interrupt-Ereignis aufgetreten ist.

Ist ein Interrupt sowohl durch das zugehörige Interrupt-Enable-Flag als auch global aktiviert, unterbricht das Setzen des Interrupt-Flags durch die Peripherie das Hauptprogramm. Die CPU springt dann zu einer festen Adresse im Programmspeicher.

Für die 56 Interrupt-Ereignisse im ATmega2560 existieren entsprechend 56 feste Adressen am Beginn des Programmspeichers. Dieser Bereich wird als Interrupt-Vektor-Tabelle bezeichnet, und die 56 festen Adressen nennt man Interrupt-Vektoren.

An jeder dieser festen Adressen befindet sich ein Sprungbefehl, der zur eigentlichen, vom Programmierer erstellten Interrupt-Service-Routine (ISR) führt.

Beispiele aus der Interruptvektortabelle des ATmega 2560 sind:

Vektornummer	Quelle	Beschreibung
1	Reset	Auch ein Reset wird zu den Interrupts gezählt
2	Int0	Pegeländerung an Pin PD0/INT0
3	Int1	Pegeländerung an Pin PD1/INT1
...	.....	....
26	USART0 RX	Serielle Schnittstelle Nr. 0: Zeichen wurde empfangen
...	.....	....

> **Wichtig!**
>
> Interrupt-Vektoren sind fixe Adressen im Programmspeicher, an die die CPU bei Auftreten eines (erlaubten) Interrupts verzweigt.

Interrupt-Ereignisse während des Abarbeitens von Interrupts

Gleichzeitig mit der Unterbrechung des Hauptprogramms und dem Sprung zur Interrupt-Service-Routine (ISR) passieren drei Dinge:

- Wie bei jedem anderen Unterprogrammaufruf wird die Rücksprungadresse (die Adresse, an der das Hauptprogramm unterbrochen wurde) auf dem Stack gespeichert. Dadurch kann die CPU das Hauptprogramm nach der Abarbeitung der ISR an der richtigen Stelle fortsetzen.
- Interrupts werden global gesperrt und erst nach Beendigung der ISR wieder aktiviert. Dadurch wird verhindert, dass weitere Interrupt-Ereignisse die aktuell laufende ISR unterbrechen.

- Das von der Peripherie gesetzte Interrupt-Flag wird von der CPU gelöscht. Dadurch kann, sobald die Interrupt-Service-Routine (ISR) beginnt, bereits ein weiteres Interrupt-Ereignis auftreten, ohne dass es verloren geht.

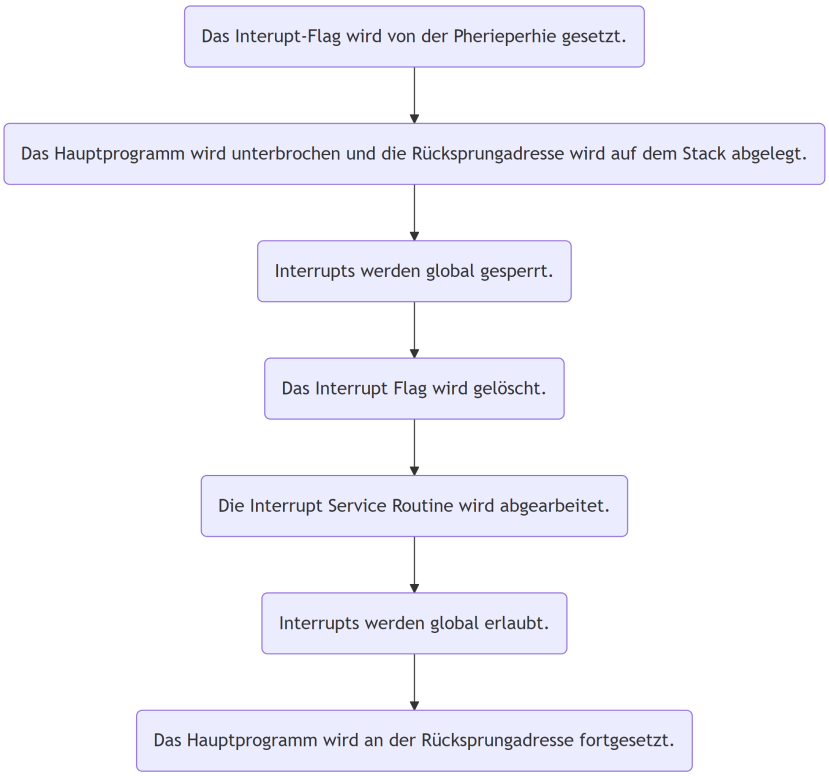

Abbildung 3: Genauer Ablauf eines Interrupts

Wenn während der Abarbeitung einer Interrupt-Service-Routine (ISR) ein weiteres Interrupt-Ereignis auftritt, geht dieses nicht verloren, da die Peripherie das entsprechende Interrupt-Flag setzt (dieses wurde zuvor von der CPU beim Sprung zur ISR gelöscht).

Allerdings bleiben Interrupts während der aktuellen Abarbeitung der ISR global gesperrt. Sobald die ISR vollständig abgearbeitet ist, springt die CPU zurück ins Hauptprogramm und aktiviert die Interrupts wieder global. Danach wird zunächst ein einzelner Maschinenbefehl des Hauptprogramms ausgeführt, bevor die CPU das gesetzte Interrupt-Flag erkennt und erneut in die ISR verzweigt.

Danach erkennt die CPU, dass das Interrupt-Flag erneut gesetzt ist, und verzweigt direkt wieder in die Interrupt-Service-Routine. Das Interrupt-Ereignis geht also nicht verloren, aber es wird verspätet abgearbeitet.

Wenn mehrere Interrupts gleichzeitig auftreten, wird der Interrupt mit der höchsten Priorität (niedrigste Vektor-Adresse) zuerst ausgeführt.

Falls ein Interrupt-Flag erneut gesetzt wird, während die ISR für denselben Interrupt noch läuft, wird das Ereignis nicht mehrfach gezählt, sondern geht verloren.

Hinweis:

Andere Mikrocontroller und Prozessoren haben ausgereiftere Mechanismen, z.B. Prioritäten für Interupts. Hier können wichtigere (höher priorisierte) Interrupts andere, unwichtigere Interrupts unterbrechen.

Externe Interrupts beim ATmega2560

Externe Interrupts werden durch eine Änderung des Spannungspegels an speziellen Interrupt-Pins ausgelöst. Der Name rührt daher, dass diese Pegeländerungen von externer Hardware verursacht werden. Im Gegensatz dazu entstehen interne Interrupts durch Zustandsänderungen in den integrierten Peripherie-Modulen des Mikrocontrollers.

Der ATmega2560 verfügt über acht externe Interrupts, die auf den folgenden Pins liegen:

- INT0 bis INT3 → Port D (Pins PD0 bis PD3)
- INT4 bis INT7 → Port E (Pins PE4 bis PE7)

> **Wichtig!**
>
> Der ATmega2560 verfügt über acht Pins für externe Interrupts, die sich auf die Ports D und E verteilen. Diese Pins besitzen die alternative Funktionsbezeichnung INT0 bis INT7. Beispielsweise hat der Pin PD0 neben seiner Hauptfunktion auch die alternative Funktion INT0 (laut Datenblatt: PD0 / SCL / INT0).

Initialisieren der Interrupt-Pins INT0 bis INT7

Zuallererst muss der zu verwendende Interrupt-Pin als Eingang konfiguriert werden. Ob der interne Pullup-Widerstand aktiviert werden muss, hängt von der jeweiligen Beschaltung des Pins ab.

Für jeden Interrupt-Pin gibt es vier Möglichkeiten, wie der Interrupt ausgelöst werden kann:

- Ein Low-Pegel löst einen Interrupt aus (Pegel-getriggerter Interrupt, auch level-triggered interrupt genannt).
- Eine fallende Flanke am Pin löst einen Interrupt aus (Pegeländerung von High nach Low).
- Eine steigende Flanke am Pin löst einen Interrupt aus (Pegeländerung von Low nach High).
- Sowohl eine steigende als auch eine fallende Flanke am Pin lösen einen Interrupt aus.

Jeder Interrupt-Pin hat ein zwei Bit breites Bitmuster, das aus den Bits ISCn1 und ISCn0 besteht. Das „n“ steht dabei für die Interrupt-Nummer (0 bis 7). Diese acht mal zwei Bits verteilen sich auf die beiden Register EICRA (External Interrupt Control Register A) und EICRB (External Interrupt Control Register B).

Bit	7	6	5	4	3	2	1	0
EICRA:	ISC31	ISC30	ISC21	ISC20	ISC11	ISC10	ISC01	ISC00

Bit	7	6	5	4	3	2	1	0
EICRB:	ISC71	ISC70	ISC61	ISC60	ISC51	ISC50	ISC41	ISC40

ISC steht für Interrupt Sense Control. Die Bits ISC71 und ISC70 gehören zum externen Interrupt des Pins PE7/INT7, die Bits ISC61 und ISC60 zum Pin PE6/INT6, und so weiter. Übrigens spricht man bei dem Bit ISC61 vom Bit „ISC sechs eins“ und nicht vom Bit „ISC einundsechzig“. Die Bits haben folgende Bedeutung:

Bit ISCn1	Bit ISCn1	Bedeutung:
0	0	Low-Level: Der Low-Pegel am Pin INTn erzeugt den Interrupt (level-triggered Interrupt).
0	1	Any Edge: Sowohl eine steigende als auch eine fallende Flanke am Pin INTn erzeugen einen Interrupt.
1	0	Falling edge: Die fallende Flanke am Pin INTn erzeugt einen Interrupt
1	1	Rising edge: Die steigene Flanke am Pin INTn erzeugt einen Interrupt.

Low-level-getriggerte Interrupts stellen einen Sonderfall dar, der nur in Verbindung mit spezieller Hardware und sogenanntem Interrupt-Handshaking verwendet wird. Dieser Fall wird hier jedoch nicht weiter behandelt.

> **Beispiel:**
>
> Der Pin PE5/INT5 soll bei fallenden Flanken Interrupt-Ereignisse auslösen. Zu Testzwecken werden die Interrupts durch einen Taster ausgelöst, der zwischen dem Pin und GND geschaltet wird. Welche Schritte sind dafür notwendig?

Lösung:

Header einbinden für sei() und ISR Macro.

```
#include <avr/interrupt.h>
```

Pin initialsieren. PE5/INT5 wird ein Eingang und der interne Pull-Up wird aktiviert.

```
DDRE &= ~(0x01 << 5);
PORTE |= 0x01 << 5;
```

Fallende Flanken sollen Interrupts auslösen: Die beiden Bits ISC51 und ISC50 müssen auf das Bitmuster `10` gesetzt werden. Beide Bits sind im Register `EICRB` an den Bitpositionen 3 und 2. Für diese können die Macros ISC51 (`#define 3`) und ISC50 (`#define 2`) verwendet werden.

```
EICRB |= 0x01 << ISC51;
EICRB &= ~(0x01 << ISC50);
```

> **Wichtig!**
>
> Interrupt-Pins müssen als Eingänge konfiguriert werden (mit oder ohne internen Pullup-Widerstand). Mit zwei Bits aus den Registern EICRA bzw. EICRB muss für jeden verwendeten Interrupt festgelegt werden, welche Art von Spannungsänderung ein Interrupt-Ereignis auslösen soll (steigende Flanke, fallende Flanke oder beide Flanken).

Register für Interrupt-Flags und Interrupt-Enable-Flags

Die Interrupt-Flags für diese acht externen Interrupts befinden sich im Special-Function-Register EIFR.

Bit	7	6	5	4	3	2	1	0
EIFR:	INTF7	INTF6	INTF5	INTF4	INTF3	INTF2	INTF1	INTF0

Da die Interrupt-Flags von der GPIO-Peripherie gesetzt und von der CPU beim Verzweigen in eine Interrupt-Service-Routine automatisch gelöscht werden, muss der Programmierer dieses Register nur selten direkt verwenden.

In der Initialisierungsphase des Programms kann, unmittelbar vor der globalen Interrupt-Freigabe, der Befehl `EIFR = 0x00;` ausgeführt werden. Dadurch werden sporadische Pegelwechsel, die während des Resets oder der Initialisierung auftreten, gelöscht, sodass sie nicht sofort einen Interrupt auslösen.

Die Interrupt-Enable-Flags für die acht externen Interrupts befinden sich im Special-Function-Register EIMSK.

Bit	7	6	5	4	3	2	1	0
EIMSK:	INT7	INT6	INT5	INT4	INT3	INT2	INT1	INT0

> **Beispiel:**
>
> Spannungspegeländerungen am Pin PE5/INT5 sollen Interrupts auslösen können, während Änderungen an den anderen sieben Interrupt-Pins keine Interrupts auslösen dürfen. Mit welchem Wert muss das EIMSK-Register beschrieben werden?

Das Bit „INT5“, das ist das Bit Nr. 5 im EIMSK-Register muss auf eins gesetzt werden, die anderen auf null. Der Befehl lautet:

```
EIMSK = (0x01 << INT5);
```

Globales Erlauben/Sperren von Interrupts

Jeder Prozessor und jeder Mikrocontroller verfügt über eine einfache Möglichkeit, Interrupts global zu sperren oder zu erlauben.

Beim ATmega2560 werden Interrupts global aktiviert, indem das sogenannte „I“-Bit auf 1 gesetzt wird. Sind Interrupts global gesperrt, ist dieses Bit auf 0 gesetzt.

Das „I“-Bit befindet sich an Position 7 im Special-Function-Register SREG (Status-Register). Theoretisch könnte man Interrupts mit den folgenden Befehlen freigeben oder sperren:

```
SREG |= (0x01 << 7);  // Interrupts aktivieren
SREG &= ~(0x01 << 7); // Interrupts deaktivieren
```

In der C-Programmiersprache stehen dafür die Funktionen

- `sei();` zum globalen Erlauben (enable) von Interrupts.
- `cli();` zum globalen Sperren (disable) von Interrupts.

zur Verfügung (Header-Datei: `#include <avr/interrupt.h>`). Der C-Compiler übersetzt diese Funktionen direkt in die entsprechenden Assembler-Anweisungen, sodass sie möglichst schnell ausgeführt werden.

Das globale Erlauben von Interrupts bedeutet nicht, dass automatisch alle 56 Interrupts aktiviert sind.

Zwischen den Interrupt-Enable-Flags und dem „I"-Bit besteht eine logische UND-Verknüpfung. Eine Interrupt-Service-Routine wird nur dann ausgeführt, wenn beide Bedingungen erfüllt sind:

- Das zugehörige Interrupt-Enable-Flag ist auf 1 gesetzt.
- Das „I"-Bit im SREG-Register ist auf 1 gesetzt.

> **Wichtig!**
>
> - Wenn Interrupts mit `cli();` global gesperrt werden, sind alle Interrupts deaktiviert – unabhängig von den einzelnen Interrupt-Enable-Flags.
> - Wenn Interrupts mit `sei();` global erlaubt werden, sind nur jene Interrupts aktiv, deren zugehöriges Interrupt-Enable-Flag ebenfalls auf 1 gesetzt ist.

Das ISR()-Makro

Bei jedem erlaubten Interrupt-Ereignis wird das Hauptprogramm automatisch unterbrochen, und die zugehörige Interrupt-Service-Routine (ISR) wird ausgeführt.

Eine Interrupt-Service-Routine ähnelt einer Funktion und weist folgende Eigenschaften auf:

- Wie bei regulären Funktionen wird die ISR außerhalb anderer Funktionen definiert und darf nicht innerhalb der `main()`-Funktion verschachtelt werden.
- Der Funktionskopf wird mit dem ISR()-Makro gebildet, das erst nach dem Einbinden der Bibliothek `#include <avr/interrupt.h>` verwendet werden kann.
- Innerhalb der runden Klammern des ISR-Makros muss der korrekte Interrupt-Vektor angegeben werden. Da es insgesamt 56 Interrupt-Ereignisse gibt, muss dem Compiler mitgeteilt werden, zu welchem Ereignis die ISR gehört.
- Nach dem Funktionskopf ISR(interrupt-vector) folgt in geschweiften Klammern der Anweisungsblock, also der Code, der innerhalb der ISR ausgeführt wird.
- Eine ISR erhält keine Übergabeparameter und gibt keinen Wert zurück.
- Falls erforderlich, kann `return;` ohne Rückgabewert genutzt werden, um die ISR vorzeitig zu verlassen.

Die folgende Tabelle listet die Namen einiger Interrupt-Vektoren, wie sie in der Programmiersprache C für das ISR-Makro verwendet werden müssen. Die Namen aller Interrupt-Vektoren enden mit _vect:

Interrupt-Vektor	**Interrupt-Quelle**
`INT0_vect`	Externer Interrupt 0 (Pin PD0/INT0)
`INT1_vect`	Externer Interrupt 1 (Pin PD1/INT1)
...	...
`TIMER1_COMPA_vect`	Timer1, Comparator A
`TIMER1_OVF_vect`	Timer1, Overflow

Interrupt-Vektor	Interrupt-Quelle
...	...

Ein konkreter Aufruf des ISR()-Makros sieht so aus:

```
int main()
{
    //main
}

ISR(INT2_vect)
{
    //Interrupt Service Routine
}
```

> **Wichtig!**
>
> Mit dem ISR-Makro und dem zugehörigen Interrupt-Vektor wird eine Interrupt-ServiceRoutine programmiert.

Datenaustausch zwischen main und der ISR

In den meisten Fällen müssen zwischen dem Hauptprogramm und der Interrupt-Service-Routine Daten ausgetauscht werden. Dazu werden globale Variablen genutzt.

```
volatile int counter = 0;

int main()
{
    ...
    printf("%d\n", counter);
}

ISR(INT2_vect)
{
    ...
    counter++;
}
```

Der Typ-Qualifizierer `volatile` weist den Compiler an, dass sich der Wert einer Variable jederzeit unerwartet ändern kann. Dies ist besonders wichtig für Variablen, die durch Hardware oder Interrupt-Service-Routinen modifiziert werden.

Wird `volatile` vergessen, kann der Compiler anstelle eines direkten Speicherzugriffs einen zuvor zwischengespeicherten Wert aus einem Register verwenden. Dies geschieht häufig, wenn der Code mit Optimierungsoptionen kompiliert wird (z. B. mit dem Compiler-Flag -Os beim avr-gcc).

In solchen Fällen bemerkt das Hauptprogramm nicht, dass die Variable durch eine Interrupt-Service-Routine (ISR) verändert wurde. Dies führt zu schwer zu findenden Fehlern – es scheint, als würde die ISR gar nicht ausgeführt, obwohl sie tatsächlich aktiv ist.

> **Wichtig!**
>
> Globale Variable, die in der Interrupt-Service-Routine verändert und im Hauptprogramm ausgelesen werden, müssen mit `volatile` deklariert sein.

Gesamter Programmaufbau

Bei der Programmierung einer Mikrocontroller-Anwendung mit Interrupts sollten folgende Punkte beachtet werden:

- **Kurze ISR-Ausführung**: Die Interrupt-Service-Routine (ISR) sollte möglichst kurz sein, da während ihrer Ausführung sowohl das Hauptprogramm als auch andere Interrupts blockiert sind.
- **Datenaustausch über globale Variablen**: Der Austausch von Daten zwischen ISR und Hauptprogramm erfolgt über globale Variablen.
- **volatile für ISR-Variablen**: Globale Variablen, die von einer ISR verändert und vom Hauptprogramm ausgelesen werden, müssen mit volatile deklariert werden, um Fehler durch Compiler-Optimierungen zu vermeiden.
- **Aufwendige Berechnungen ins Hauptprogramm auslagern**: Falls ein Interrupt eine aufwändige Verarbeitung erfordert, sollten in der ISR nur die wichtigsten Aufgaben erledigt werden. Beispielsweise können Daten von der Hardware abgeholt und in globalen Variablen gespeichert werden.

> **Beispiel:**
>
> An Pin 2 von Port D (PD2/INT2) ist ein Taster gegen Masse geschaltet. Dieser Pin wird als Eingang mit aktiviertem Pull-up-Widerstand konfiguriert. Der externe Interrupt 2 (INT2) wird so eingestellt, dass eine steigende Flanke (das Loslassen des Tasters) einen Interrupt auslöst. In der Interrupt-Service-Routine (ISR) wird bei jedem Loslassen des Tasters eine globale Variable namens counter inkrementiert. Diese Variable ist mit volatile deklariert, um sicherzustellen, dass der Compiler sie nicht optimiert und immer den aktuellen Wert berücksichtigt. Im Hauptprogramm wird in einer Endlosschleife der aktuelle Wert von `counter` über die serielle Schnittstelle mittels `printf()` ausgegeben.

```
#include "USART.h"
#define PINBTN 2
volatile int counter = 0;

int main()
{
    DDRD = 0x00;
    PORTD |= 0x01 << PINBTN;

    USARTInit(0, 9600, 1, 0, 1, 0);

    //Interrupt 2 on Falling edge
    EICRA |= 0x01 << ISC21;
    EICRA &= ~(0x01 << ISC20);

    //Enable External Interrupt 2
    EIMSK |= 0x01 << INT2;
```

```
    //Enable Interrupts global
    sei();

    while(1)
    {
        printf("Counter: %d\n", counter);
        _delay_ms(1000);
    }
}

//Interrupt Service Routine
ISR(INT2_vect)
{
  counter++;
}
```