Universität Koblenz-Landau
Institut für Physik
Universitätsstr. 1
56072 Koblenz

Seminar Mikrocontroller

Interrupts & I/O-Ports

Arne Baldauf abaldauf@uni-koblenz.de

26. Juni 2006

Inhaltsverzeichnis

1 Interrupts - Allgemein

In diesem Kapitel soll zunächst einmal in das Thema eingeführt und die allgemeine Interrupt-Funktionalität erläutert werden, bevor im nächsten Teil auf die spezifische Funktionalität der Atmel AVR 8bit RISC Serie eingegangen wird.

1.1 Einführung

Bei Interrupts handelt es sich um einen *Benachrichtigungsmechanismus*, welcher es einer ALU[1] erlaubt, mit minimaler Verzögerung aufgetretene Ereignisse zu erkennen und zu verarbeiten. Dabei kann es sich sowohl um intern in der ALU auftretende als auch von externen Bauteilen signalisierte Ereignisse handeln. Die Anzahl und Art der Interrupts ist plattformspezfisch, sie sind jedoch immer als fest in Hardware vorhandene Schaltung realisiert. Neben der schnellen Reaktion auf Ereignisse besitzt die Interrupt-Funktionalität den Vorteil, dass sie *Multiprocessing*[2] ermöglicht.
Betrachtet man die regelmäßige Abfrage auf Veränderungen (auch *Polling* genannt) als alternative Methode zur Ereigniserkennung, so werden die Vorteile des Interruptmechanismus noch deutlicher: Die Reaktionszeit beim Polling ist schlechtestenfalls die Länge des Abfrageintervalls. Wählt man für diesen einen kleinen Wert, so ist der Prozessor zu einem hohen Anteil nur mit den Abfragen beschäftigt; wählt man einen großen Wert, so sinkt die Reaktionsgeschwindigkeit. Unabhängig davon ermöglicht Polling kein Multiprocessing.
Interrupts werden in drei verschiedene Kategorien unterteilt. Zunächst einmal existiert die Gruppe der *externen Interrupts*, zu welcher beispielsweise das Reset-Signal und die I/O-Interrupts gehören. Als zweites ist die Gruppe der *internen Interrupts* zu nennen, unter welche z.B. der *Supervisor Call* oder die Division durch null (*Divide-by-zero error*) fallen. Als dritte und letzte Kategorie bleiben die *Software-Interrupts*, auf welche hier aufgrund der Irrelevanz für Mikrocontroller nicht weiter eingegangen werden soll.

1.2 Interrupt-Verarbeitung

Wurde ein Interrupt ausgelöst, müssen verschiedene Schritte vor dem Beginn und nach dem Ende der eigentlichen Interruptbehandlungsroutine abgearbeitet werden. Der Ablauf geschieht typischerweise wie folgt:

1. Interrupts sperren
2. Program Counter (PC) und Statusregister sichern
3. Privilege Level prüfen
4. PC und Statusregister der Interrupt-Routine laden

[1] Arithmetic Logical Unit, siehe Vortrag „ALU und Speicher“ in dieser Seminarreihe.
[2] Das gleichzeitige Ausführen mehrerer Programme auf einem Prozessor, quasi-parallel durch Umschalten zwischen den einzelnen Programmen; hiermit ist nicht echtes *Parallel processing* gemeint.

5. Interrupt freigeben
6. Ausführen der Interruptbehandlungsroutine
7. Interrupts sperren
8. PC und Statusregister wiederherstellen
9. Interrupts freigeben

Das Speichern von *Program Counter*[3] und *Statusregister* geschieht auf einen Stack[4] und ist notwendig, damit das unterbrochene Programm nach Abarbeitung der Interruptbehandlung korrekt fortgesetzt werden kann. Da die Schritte 2, 3, 4 und 8 kritisch für die korrekte Ausführung der Programme sind, dürfen diese nicht unterbrochen werden, weshalb hierfür alle Interrupts global gesperrt werden. Im Gegensatz dazu kann die eigentliche Interruptbehandlungsroutine selbst wieder durch einen weiteren Interrupt unterbrochen werden. Wird ein Interrupt signalisiert, solange die Interruptsperrung gesetzt ist, so löst die Verarbeitung unmittelbar nach Freigabe der Sperrung aus (das Interrupt-Signal „geht nicht verloren“). Auf das Privilege Level wird noch später in diesem Kapitel eingegangen.

1.3 Interrupt-Vektor-Tabelle

Die Interruptbehandlungsroutinen selbst sind frei programmierbar und können (theoretisch[5]) beliebig lang sein. Damit der Prozessor an die korrekte Adresse (die Speicherstelle mit dem ersten Befehl) dieser Routine springen kann, werden die Adressen in der *Interrupt-Vektor-Tabelle* gespeichert. Da die Adressen (beinahe[6]) beliebig sein können, ist der Inhalt der Tabelle logischerweise ebenfalls programmierbar. Die Interrupt-Vektor-Tabelle selbst befindet sich jedoch immer an einer festen, für die jeweilige Prozessorarchitektur spezifischen Speicheradresse (*Interrupt base address*).
Ein Eintrag in der Tabelle besteht lediglich aus einer Adresse ohne weitere Informationen. Die Größe jedes Eintrags entspricht also der Adressgröße der Architektur, auf einem 32Bit-System z.B. folglich 4Byte[7]. Die Interrupts sind durchnummeriert (0-basiert, *Interrupt vector number*). Ein 32Bit-Prozessor findet also die gespeicherte Adresse der eigentlichen Interrupt-Routine an der mittels der einfachen Berechnung
*Interrupt_base_address + 4 * Interrupt_vector_number* auffindbaren Adresse.

[3] Beim Program Counter (PC) handelt es sich um einen Zeiger, welcher auf die Speicheradresse zeigt, an der der nächste zu verarbeitende Befehl gespeichert ist.

[4] Stack, auch *Stapel* oder *Keller* genannt, ist eine Datenstruktur, bei der immer nur auf das oberste Element zugegriffen werden kann.

[5] Begrenzt werden diese natürlich durch den verfügbaren Programmspeicher.

[6] bestimmte Speicherbereiche werden bei den meisten Prozessorfamilien zur Adressierung von z.B. externen Geräten verwendet, können also nicht für das Speichern von Programmen oder Daten verwendet werden.

[7] Bei den aktuellen 64Bit-erweiterten Prozessoren ist ein Eintrag ebenfalls 4Byte groß und verweist (nur im 64Bit-Modus) auf einen Eintrag in einer zweiten Tabelle, welche die eigentlichen 64Bit-Adressen enthält, also 8Byte pro Eintrag.

Vector No.	Address displacement (in bytes)	Interrupt event	Source	Priority
0	0	Reset	extern	0.0
1	4	I/O	extern	0.1
2	8	Bus error	extern	0.2
3	12	Address error	intern	0.3
4	16	Trace	intern	1.0
5	20	Illegal instruction	intern	1.2
6	24	Privilege violation	intern	1.3
7	28	Zero divide	intern	2.0
8	32	TRAPV instruction	intern	2.0
9	36	OPC Emulation 1	intern	2.0
10	40	OPC Emulation 2	intern	2.0
11-24	44...	- not used -	-	-

Tabelle 1: Einträge der Interrupt-Vektor-Tabelle der Intel x86-Architektur

Die Tabelle 1 stellt die Zugehörigkeit der einzelnen Einträge der Interrupt-Vektor-Tabelle der Intel x86-Architektur dar. Wie zu erkennen ist, besitzen Prozessoren dieser Familie[8] lediglich einen externen Interrupt für I/O-Geräte. Die einzelnen Interrupts dieser Geräte münden in den PIC oder APIC[9], welcher die externe Interrupt-Leitung des Prozessors ansteuert[10].
Wird ein externer Interrupt ausgelöst, so muss der Prozessor in der Behandlungsroutine zunächst das eigentliche auslösende Gerät beim PIC abfragen (dies geschieht wie bei anderen Geräten auch mittels Adress- / Datenbus) und kann dann in eine gerätespezifische Subroutine springen[11]. Abbildung 1 stellt die Verbindungen zwischen CPU, PIC und Geräten graphisch dar.

1.4 Prioritäten und Privilege Levels

Wie in der Tabelle 1 ersichtlich, besitzt jeder Interrupt eine *Priorität*. Diese dient zum einen dazu, eine Ausführungsreihenfolge für die Interruptbehandlungsroutinen festzulegen, wenn zwei oder mehr Interrupts im gleichen Takt ausgelöst werden. Zum anderen bestimmt die Priorisierung die Reihenfolge der Verarbeitung, wenn nach Aufhebung einer globalen Interruptsperrung mehrere signalisierte Interrupts anstehen.

[8] Dazu zählen alle typischen aktuellen PC-Prozessoren seit dem i386 bis zu den aktuellen Intel Core2 und AMD Athlon64 Prozessoren bzw. deren Servervarianten Xeon und Opteron. Neue Befehle o.ä. stellen lediglich eine Erweiterung der Grundarchitektur dar. Nicht zu dieser Familie gehören z.B. IBM PowerPC, Sun UltraSPARC oder Intel Itanium.

[9] Programmable Interrupt Controller (PIC) bzw. Advanced Progammable Interrupt Controller (APIC). Diese unterscheiden sich im wesentlichen durch die Anzahl ihrer Interrupt-Leitungen (16 bzw. 24).

[10] Bei den in der Geräteverwaltung aktueller Betriebssysteme angezeigten Interrupts handelt es sich um die Interrupt-Leitungen des PIC / APIC, *nicht* um die Interrupts des Prozessors.

[11] An dieser Stelle setzt meistens die gerätespezifische Treibersoftware an.

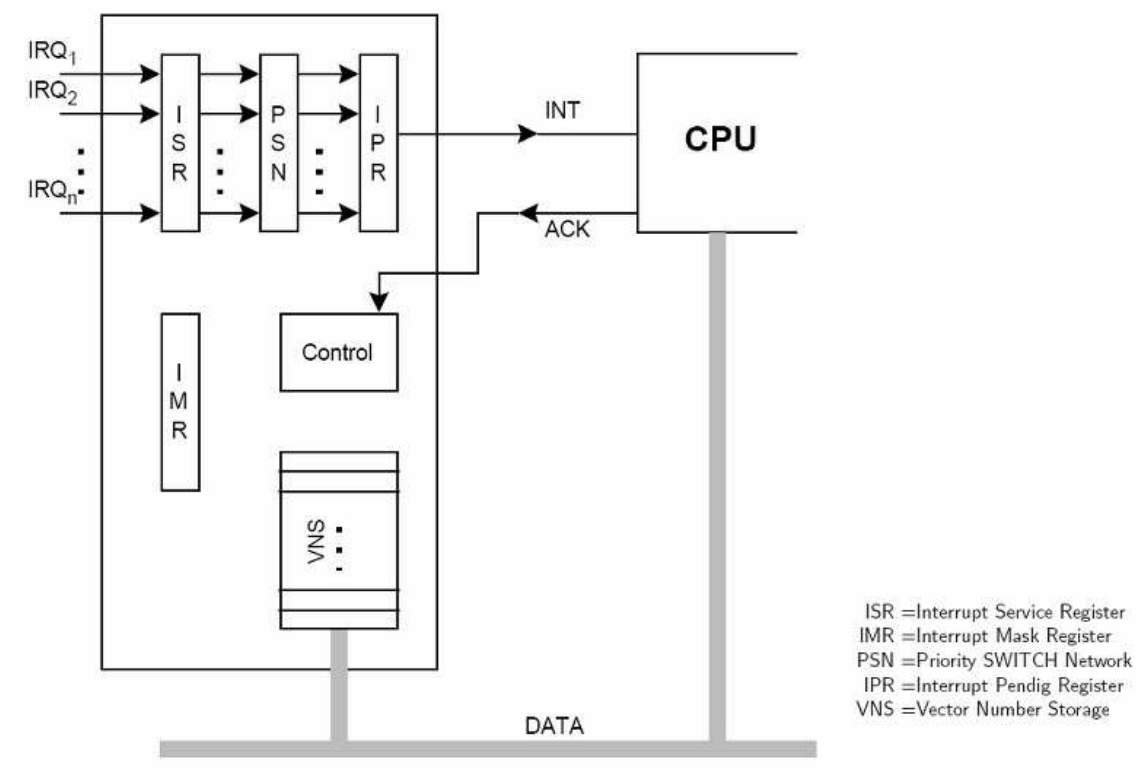

Abbildung 1: Ansteuerung der Geräteinterrupts und der CPU mittels eines PIC

Die *Privilege Levels* bestimmen vor allem bei den internen und den Software-Interrupts, durch welchen Prozess diese ausgelöst werden dürfen. Da dies nur für ausgewachsene Betriebssysteme von Bedeutung ist, sollen an dieser Stelle nur kurz die Möglichkeiten hiervon aufgeführt werden. Durch die Privilege Levels wird eine komplette Trennung von *User Mode* und *Supervisor Mode* (auch *System Mode* oder *Kernel Mode* genannt) erreicht. Sind alle Routinen des System Mode nur durch Interrupts erreichbar, so kann der Kernel die Laufzeiten und die Umschaltung der User-Mode-Prozesse verwalten, während sein eigener Programm- und Datenspeicher vor Zugriffen aus dem *User Space* geschützt ist. Ferner kann der Kernel dafür sorgen, dass auch die Speicherbereiche der einzelnen Prozesse voreinander geschützt sind.

2 Interrupts - Atmel AVR

Nachdem nun ein Überblick über die allgemeine Funktionsweise geschaffen wurde, wird in diesem Kapitel auf die konkrete Funktionalität und die Besonderheiten der Atmel AVR Mikrocontroller am Beispiel des AtMega16 eingegangen.

2.1 Interrupt-Verarbeitung

Der Ablauf der Interruptverarbeitung dieses Controllers ähnelt dem Ablauf aus Kapitel 1.3, jedoch mit einigen kleineren Abweichungen. Zunächt einmal existieren keinerlei Privilege Levels, so dass deren Überprüfung komplett entfällt. Desweiteren erfolgt durch die Hardware lediglich die globale Interruptsperrung und das Sichern des Program Counters auf den Stack zu Beginn, sowie das Zurückschreiben und die globale Interruptfreigabe zum Ende der Verarbeitung. Ein Speichern und Wiederherstellen des Zustandes des Statusregisters muss in der eigentlichen Interruptbehandlungsroutine durch die Software erfolgen. Ebenso bleiben Interrupts während der kompletten Verarbeitung gesperrt, wenn keine Freigabe in der Behandlungsroutine erfolgt (Wichtig: In diesem Fall muss auch wieder eine Sperrung erfolgen, bevor der *Return from Interrupt*[12] erfolgt).
Wird während der Verarbeitung eines Befehls über mehrere Takte (wie z.B. des Sprungbefehls JMP mit 3 Takten) ein Interrupt signalisiert, so wird die Interruptverarbeitung erst nach dem letzten Takt des Befehls begonnen. Der Controller benötigt vier weitere Takte, bevor der erste Befehl der Interruptbehandlungsroutine verarbeitet wird. In diesen Takten erfolgen die Interrupt-Sperrung, das Löschen des jeweiligen Interrupt Flags (dazu später mehr), das Sichern des PC sowie der Sprung an die Startadresse der Interrupt-Routine. Befindet sich der Controller in einem der Sleep-Modi (Stromspar-Modi[13]), so werden acht anstatt vier Takte für diesen Vorgang benötigt, zuzüglich der modusspezifischen Startup-Dauer.
Wird der *Return from Interrupt* ausgelöst, so werden wiederum vier Takte benötigt, in denen der urspüngliche PC vom Stack wiederhergestellt wird, der Sprung an die ursprüngliche Adresse erfolgt und die Interrupts freigegeben werden. Danach wird die Ausführung der unterbrochenen Routine fortgesetzt. Steht noch die Verarbeitung eines weiteren Interrupts an, so wird noch ein Befehl der unterbrochenen Routine verarbeitet, bevor die Interruptbehandlung beginnt.

2.2 Interrupt-Typen

Grundsätzlich existieren bei Atmel AVR Controllern zwei verschiedene Interrupt-Typen[14].
Jeder Interrupt des Typ 1[15] besitzt ein zugehöriges Interrupt Flag, wobei es sich um ein einfaches Merkerbit handelt, das bei der Erkennung des Interrupts gesetzt und bei Beginn der

[12] Dies bezeichnet sowohl das Ende der Interruptbehandlungsroutine, an der der Rücksprung in die unterbrochene Routine erfolgt, als auch den Befehl, der diesen Vorgang auslöst.
[13] Details hierzu siehe Vortrag „Clock, Power und Watchdog“ in dieser Seminarreihe.
[14] Hiermit sind nicht die Interrupt-Arten (intern, extern, Software) gemeint.
[15] Bei externen Interrupts spricht man hier von *Interrupt by edge*.

Interruptbehandlung gelöscht wird. Die Verarbeitung setzt sofort ein, wenn Interrupts global freigegeben sind, bzw. setzt unmittelbar nach der Freigabe ein, wenn diese global gesperrt sind. Nach Ende der Behandlung wird der Interrupt erst dann erneut ausgelöst, wenn der Interrupt nochmals signalisiert wird (die Bedingung also zwischenzeitlich nicht mehr und dann wieder erfüllt wird, was bereits während der laufenden Verarbeitung sein kann).
Interrupts des Typ 2[16] besitzen dagegen kein Interrupt Flag. Sind Interrupts global gesperrt, und wird die Bedingung für einen solchen Interrupt erfüllt, so setzt nach der globalen Interruptfreigabe die Interruptbehandlung nur ein, wenn die Bedingung zu diesem Zeitpunkt immer noch erfüllt ist. Ist nach Verarbeitung der Interruptbehandlungroutine der Interrupt weiterhin signalisiert, so beginnt die Interruptbehandlung erneut. Eine normale Verarbeitung des unterbrochenen Programmes erfolgt erst, wenn nach dem Durchlauf der Interruptbehandlungsroutine die Bedingung nicht mehr erfüllt ist[17].
Die meisten Interrupts des AtMega16 gehören zum Typ 1, lediglich die Interrupts *Reset*[18], *EEPROM Ready* und *Store Program Memory Ready* sind Interrupts des Typ 2. Einen Sonderfall stellen die externen Interrupts 0 und 1 dar: Diese verhalten sich wie Typ 1, wenn sie auf Auslösung bei einer Signalflanke (*Interrupt by egde*) eingestellt sind. Sind sie auf Auslösung bei einem bestimmten Signalpegel (*Interrupt by level*) konfiguriert, so verhalten sie sich wie Typ 2. Mehr zu den möglichen Modi der externen Interrupts später in diesem Kapitel.

2.3 Interrupt-Vektor-Tabelle

Die Interrupt-Vektor-Tabelle des AtMega 16 befindet sich grundsätzlich immer am Anfang des Programmspeichers (Interrupt base address = $000). Jeder Eintrag ist 2 Byte groß, dabei handelt es sich jedoch nicht um eine Adressangabe für die Interruptbehandlungroutine wie im ersten Kapitel beschrieben. Stattdessen ist in dem Eintrag ein Befehl gespeichert, welcher beim Auslösen des Interrupts ausgeführt wird. Üblicherweise handelt es sich dabei um einen Sprungbefehl in die eigentliche Interruptbehandlungsroutine. Wird ein Interrupt nicht verwendet, so ist es üblich, zur Sicherheit den *Return from Interrupt*-Befehl (Assembler: *reti*) an der entsprechenden Stelle der Tabelle einzutragen.
Tabelle 2 zeigt eine Übersicht der Einträge der Interrupt-Vektor-Tabelle. Die Einträge sind in der Tabelle und im Speicher aufsteigend nach ihrer Priorität geordnet, d.h. *Reset* besitzt die höchste und *Store Program Memory Ready* die niedrigste Priorität.

[16] Bei externen Interrupts spricht man hier von *Interrupt by level*.
[17] Zwischen jedem Durchlauf der Interruptbehandlungsroutine wird allerdings immer der einzelne, obligatorische Befehl des unterbrochenen Programmes ausgeführt.
[18] Man beachte, dass Reset invertiert ist, die „Verarbeitung“ (das Hauptprogram) also stattfindet, wenn dieser nicht ausgelöst ist.

Vector No.	Speicheradresse	Quelle	Auslöser
0	$000	RESET	External Pin, Power-On Reset, Brown-out Reset, Watchdog Reset, JTAG AVR Reset
1	$002	INT0	External Interrupt Request 0
2	$004	INT1	External Interrupt Request 1
3	$006	TIMER2_COMP	Timer/Counter2 Compare Match
4	$008	TIMER2_OVF	Timer/Counter2 Overflow
5	$00A	TIMER1_CAPT	Timer/Counter1 Capture Event
6	$00C	TIMER1_COMPA	Timer/Counter1 Compare Match A
7	$00E	TIMER1_COMPB	Timer/Counter1 Compare Match B
8	$010	TIMER1_OVF	Timer/Counter1 Overflow
9	$012	TIMER0_OVF	Timer/Counter0 Overflow
10	$014	SPI_STC	Serial Transfer Complete
11	$016	USART_RXC	USART, Rx Complete
12	$018	USART_UDRE	USART Data Register Empty
13	$01A	USART_TXC	USART, Tx Complete
14	$01C	ADC	ADC Conversion Complete
15	$01E	EE_RDY	EEPROM Ready
16	$020	ANA_COMP	Analog Comparator
17	$022	TWI	Two-Wire Serial Interface
18	$024	INT2	External Interrupt Request 2
19	$026	TIMER0_COMP	Timer/Counter0 Compare Match
20	$028	SPM_RDY	Store Program Memory Ready

Tabelle 2: Einträge der Interrupt-Vektor-Tabelle des Atmel AtMega16

Einen typischen Programmkopf in AVR-Assembler mit allen Interruptvektoren zeigt Listing 1. Das Schlüsselwort *.org* bewirkt, dass der darauf folgende Befehl an die angegebene Speicheradresse geschrieben wird und stellt somit die Korrektheit der Einträge der Interruptvektortabelle sicher. Mit Ausnahme des Beginns der ersten Interruptbehandlungsroutine ist es im weiteren Programmcode aber eher unüblich (da fehleranfällig), das Schlüsselwort zu verwenden.
Die einzelnen Sprungbefehle verweisen auf die entsprechenden Sprungmarken, welche natürlich alle im weiteren Programmcode (am Beginn der jeweiligen Interruptbehandlungsroutine) vorhanden sein müssen. In Listing 1 ist lediglich der Beginn der Reset-Routine aufgeführt. Da diese meistens auch das Hauptprogramm ist, muss sie mit einigen nötigen Befehlen zur Initialisierung beginnen. So ist es zur Verwendung von Interrupts unumgänglich, den Stackpointer auf eine Adresse festzulegen, was hier auf die höchste Speicheradresse des SRAM

erfolgt[19]. Um weitere Interrupts zu ermöglichen, werden Interrupts global freigegeben (*sei*), denn diese sind noch vom Beginn der Reset-Interruptbehandlung gesperrt. Vor Beginn des eigentlichen Hauptprogrammes sind natürlich auch noch beliebige weitere Initalisierungsbefehle möglich. Am Ende der Interruptroutine ist es außerdem üblich, wieder an den Beginn des Hauptprogrammes zu springen (und somit die nur einmal benötigten Initialisierungsbefehle auszulassen[20]).

```
1  .org 0x000    jmp  RESET        ; Reset Handler
2  .org 0x002    jmp  EXT_INT0     ; IRQ0 Handler
3  .org 0x004    jmp  EXT_INT1     ; IRQ1 Handler
4  .org 0x006    jmp  EXT_INT2     ; IRQ2 Handler
5  .org 0x008    jmp  TIM2_COMP    ; Timer2 Compare Handler
6  .org 0x00A    jmp  TIM2_OVF     ; TIM2_OVF ; Timer2 Overflow Handler
7  .org 0x00C    jmp  TIM1_CAPT    ; Timer1 Capture Handler
8  .org 0x00E    jmp  TIM1_COMPA   ; Timer1 CompareA Handler
9  .org 0x010    jmp  TIM1_COMPB   ; Timer1 CompareB Handler
10 .org 0x012    jmp  TIM1_OVF     ; Timer1 Overflow Handler
11 .org 0x014    jmp  TIM0_COMP    ; Timer0 Compare Handler
12 .org 0x016    jmp  TIM0_OVF     ; Timer0 Overflow Handler
13 .org 0x018    jmp  SPI_STC      ; SPI Transfer Complete Handler
14 .org 0x01A    jmp  USART_RXC    ; USART RX Complete Handler
15 .org 0x01C    jmp  USART_UDRE   ; UDR Empty Handler
16 .org 0x01E    jmp  USART_TXC    ; USART TX Complete Handler
17 .org 0x020    jmp  ADCI         ; ADC Conversion Complete Handler
18 .org 0x022    jmp  EE_RDY       ; EEPROM Ready Handler
19 .org 0x024    jmp  ANA_COMP     ; Analog Comparator Handler
20 .org 0x026    jmp  TWI          ; Two-wire Serial Interface Handler
21 .org 0x028    jmp  SPM_RDY      ; Store Program Memory Ready Handler
22
23 .org 0x02A  RESET:
24               ldi  r16, high(RAMEND)
25               out  SPH,r16      ; Set Stack Pointer to top of RAM
26               ldi  r16, low(RAMEND)
27               out  SPL,r16
28               ...  ...          ; Possible other initalizations
29               sei               ; Enable interrupts
30             MAIN_START:
31               ...  ...          ; Main program
32               rjmp MAIN_START   ; Goto beginning of main program
33
```

[19] Beim Beschreiben des Stack werden beim AVR die Adressen heruntergezählt und nicht heraufgezählt, so dass hier kein Adressüberlauf erfolgt. Die Adresse ist ein 16-Bit-Wert, so dass Low Byte und High Byte initialisiert werden müssen.

[20] Wenn dies erwünscht ist, schließt man die Routine einfach mit *reti* ab. Da Reset bekanntlich konstant auslöst, beginnt danach der nächste Durchlauf.

```
34      ...             ; Other interrupt routines, called routines, etc.
```

Listing 1: Typischer Programmkopf mit Interrupt-Vektoren

2.4 Sonderfälle

Neben dem aufgezeigten Standard-Setup kann man auch noch einige nicht ganz so gebräuchliche Einstellungen vornehmen, welche sich auf die Interruptbehandlung auswirken. Diese sollen hier jedoch nur kurz aufgezeigt und nicht tiefer behandelt werden.
Wird die *BOORST-Fuse* programmiert, so erfolgt nach einem Reset ein Sprung an die Adresse der *Boot Loader*-Routine. Diese Adresse ist unter anderem von der eingestellten Größe des Loaders abhängig.
Ist das *Interrupt Vector Select* Bit (IVSEL) im *Global Interrupt Control Register* (GICR) gesetzt, so wird die Interrupt-Vektor-Tabelle an der Startadresse des *Boot Flash*-Speicherbereichs erwartet.
Verwendet man den Controller völlig ohne Interrupts[21], so kann die Interrupt-Vektor-Tabelle am Anfang des Programmspeichers komplett entfallen und direkt mit dem Hauptprogramm begonnen werden.

2.5 Externe Interrupts

Wie in der Tabelle 2 ersichtlich, besitzt der AtMega16 drei allgemeine externe Interrupts. Diese können auf unterschiedliche Signale an dem jeweiligen Portpin einen Interrupt auslösen.
INT0 und INT1 können jeweils (unabhängig voneinander) auf eine steigende, eine fallende oder eine beliebige Signalflanke, sowie auf den 0-Pegel (*on low*) einen Interrupt auslösen. Im letztgenannten Fall handelt es sich um den Typ des *Interrupt by level*.
INT2 kann hingegen nur auf eine steigende oder eine fallende Signalflanke konfiguriert werden. Da dieser Interrupt asynchron ist, kann er auch auf Signale reagieren, welche kürzer als ein Taktzyklus des Mikrocontrollers sind. Für eine zuverlässige Erkennung darf die Pulsweite des Signals aber auch nicht kürzer als fünfzig Nanosekunden sein[22]. Ein wenig Vorsicht ist allerdings beim Konfigurieren von INT2 geboten. Wird der Signalerkennungsmodus umgeschaltet, während der Interrupt eingeschaltet ist, so kann dieser unbeabsichtigt auslösen.
Ist der Portpin eines externen Interrupts als Ausgang geschaltet und der Interrupt gleichzeitig eingeschaltet, so löst dieser auch aus, wenn durch die Software das Signal des Pin entsprechend dem Interrupt geschaltet wird. Dies kann einerseits eine leicht zu übersehende Fehlerquelle darstellen, andererseits handelt es sich dabei um eine Möglichkeit, Software-Interrupts zu bewerkstelligen.
Die Konfiguration der externen Interrupts erfolgt über spezielle Register. Der Signalmodus des INT0 wird mittels der Bits 0 und 1 (*ISC00* / *ISC01*) des *MCU Control Register* (*MCUCR*)

[21] Hiervon ausgenommen ist Reset, welcher sich natürlich nicht sperren oder ausschalten lässt.
[22] Die Taktdauer beim Maximaltakt von 16MHz liegt bei 62,5ns; dies ist also vor allem bei langsamerer Taktung des Controllers von Vorteil.

eingestellt. Äquivalent erfolgt dies für INT1 mittels der Bits 2 und 3 (*ISC10* / *ISC11*) im gleichen Register. Die Bitbelegung des *MCUCR* zeigt Abbildung 2.
Der Modus von INT2 kann mittels Bit 6 (*ISC2*) im *MCU Control and Status Register* (*MCUCSR*) konfiguriert werden, Abbildung 3 zeigt dessen Belegung.
Durch Setzen bzw. Löschen der Bits 5 (*INT2*), 6 (*INT0*) und 7 (*INT1*) im *General Interrupt Control Register* (*GICR*) können die externen Interrupts einzeln ein- bzw. ausgeschaltet werden. Die Bitbelegung des Registers wird in Abbildung 4 dargestellt.
Im *Statusregister* (*SREG*), dessen Bits in Abbildung 5 beschrieben werden, ist lediglich das Bit 7 (*I*) für Interrupts von Belang. Hierüber werden Interrupts global ein- (*Global Interrupt Enable*) oder ausgeschaltet (*Global Interrupt Disable*).
Die Interrupt Flags befinden sich in den Bits 5 (*INTF2*), 6 (*INTF0*) und 7 (*INTF1*) des *General Interrupt Flag Register* (*GIFR*), siehe Abbildung 6. Auch mittels Setzen dieser Flags durch die Software kann ein Interrupt künstlich ausgelöst werden[23].

Bit	7	6	5	4	3	2	1	0	
	SM2	SE	SM1	SM0	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 2: Bitbelegung des MCU Control Register (MCUCR)

Bit	7	6	5	4	3	2	1	0	
	JTD	ISC2	–	JTRF	WDRF	BORF	EXTRF	PORF	MCUCSR
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	See Bit Description					

Abbildung 3: Bitbelegung des MCU Control and Status Register (MCUCSR)

Bit	7	6	5	4	3	2	1	0	
	INT1	INT0	INT2	–	–	–	IVSEL	IVCE	GICR
Read/Write	R/W	R/W	R/W	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Abbildung 4: Bitbelegung des General Interrupt Control Register (GICR)

[23] Das Speichern erfolgt bei INT0 und INT1 nur in den Interrupt-by-edge Modi, ebenso ist ein Software-Interrupt durch Setzen der Flags nur in diesen Modi möglich (im Interrupt-by-level Modus ist dies nur mittels Schreiben der Portpins möglich).

Bit	7	6	5	4	3	2	1	0	
	I	T	H	S	V	N	Z	C	SREG
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 5: Bitbelegung des Statusregisters (SREG)

Bit	7	6	5	4	3	2	1	0	
	INTF1	INTF0	INTF2	–	–	–	–	–	GIFR
Read/Write	R/W	R/W	R/W	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

Abbildung 6: Bitbelegung des General Interrupt Flag Register (GIFR)

3 I/O-Ports - Atmel AVR

In diesem Kapitel werden primär die allgemeinen I/O-Ports des Mikrocontrollers behandelt, bevor eine kurze Übersicht über die Alternativfunktionen der Ports gezeigt wird. Auf die Einzelheiten der Alternativfunktionen wird an dieser Stelle jedoch nicht eingegangen, da dies Bestandteil anderer Vorträge dieses Seminars ist.

3.1 Allgemeine I/O-Ports

Der Atmel AtMega16 Controller besitzt vier I/O-Ports mit jeweils acht Pins, welche standardmäßig als *allgemeine digitale I/O-Ports* geschaltet sind. Auf jeden Portpin besteht voller Schreib-/Lesezugriff, und dieser erfolgt komplett unabhängig von allen anderen Pins. Jeder Pin kann also zur Laufzeit gelesen, geschrieben oder umkonfiguriert werden, ohne dass dies andere Pins beeinflussen würde. Die Portpins können sowohl als digitaler Eingang als auch als digitaler Ausgang geschaltet werden, nach dem Einschalten des Controllers ist ersteres der Fall.
Alle Portpins besitzen *Treiber*[24], welche logischerweise nur aktiv sind, wenn der zugehörige Pin als Ausgang konfiguriert ist. Jeder dieser Treiber ist ausreichend stark dimensioniert zur direkten Ansteuerung einer Standard-LED.
Ist ein Pin als Eingang konfiguriert, so kann für diesen ein interner *Pull-Up-Widerstand*[25] geschaltet werden, standardmäßig ist dieser aber deaktiviert. Der Widerstand besitzt einen festen, von der Versorgungsspannung des Controllers unabhängigen Wert.
Zur Absicherung des Controllerkerns vor Spannungsspitzen an den Ports besitzt jeder Pin jeweils eine *Zenerdiode*[26] gegen die Versorgungsspannung (VCC) und gegen Masse (GND).
Da alle Portpins ein Synchronisierungsglied[27] besitzen, erfolgt die Schaltung des Signals an einem Ausgang erst einen Controllertakt nach dem Schreiben des jeweiligen Wertes. Ebenso existiert an einem Eingang eine Verzögerung zwischen der Veränderung des Signals und der Auslesbarkeit des Wertes. Da ein externes Signal nicht synchron zum Controllertakt geschaltet sein muss, beträgt die Verzögerung bestenfalls 0,5 Takte, schlechtestenfalls 1,5 Takte (abhängig vom Zeitpunkt der Signaländerung relativ zum Takt).

Auch die Konfiguration und Steuerung der I/O-Ports erfolgt mittels spezieller Register, über die hier nun ein Überblick geschaffen werden soll. Im folgenden werden lediglich die Regis-

[24] Treiber sind Operationsverstärker, üblicherweise eine Transistorschaltung.

[25] Ein Pull-Up-Widerstand schaltet den Pegel eines Eingangs auf *high* (logisch 1), so dass daran angeschlossene Bauteile ein *low*-Signal (logisch 0) durch Schaltung der Leitung auf Masse erzielen. Ist kein Pull-Up-Widerstand aktiviert, liegt der Pegel auf *low* und Bauteile müssen selbst auf *high* schalten.

[26] Zenerdioden, auch *Z-Dioden* genannt, sind in *Sperrrichtung* unterhalb ihrer spezifischen *Durchbruchspannung* undurchlässig, oberhalb dieser schalten diese jedoch durch (ohne dabei wie gewöhnliche Dioden Schaden zu nehmen), so dass man sie u.a. zur Spannungsbegrenzung einsetzen kann.

[27] Das Synchronisierungsglied sorgt dafür, daß das Signal nur an der relevanten Taktflanke in die zugehörigen Register übertragen wird und umgekehrt. Es stellt dadurch sicher, daß der Controller nicht durch Umspringen des Signals mitten im Takt in einen undefinierten Zustand gerät.

ter von Port A besprochen, die Register der Ports B bis D sind in der Funktion und Belegung äquivalent dazu.
Mittels des *Data Direction Register A* (*DDRA*), dessen Bitbelegung in Abbildung 7 gezeigt ist, lassen sich die Pins als Eingänge (Bit gelöscht) oder Ausgänge (Bit gesetzt) schalten. Die Bitnummer entspricht dabei der Pinnummer des Ports.
In Abbildung 8 ist die Belegung von *Pin Data Register A* (*PINA*) dargestellt. In diesem Register kann das anliegende Signal der Pins ausgelesen, jedoch nicht geschrieben werden (nur Lesezugriff). Auch hier entsprechen Bit- und Pinnummer einander.
Die Funktion der einzelnen Bits des *Port Data Register A* (*PORTA*), die Belegung zeigt Abbildung 9, sind abhängig von den Werten in DDRA. Für die als Ausgänge geschalteten Pins kann in den entsprechenden Bits das Signal geschrieben und auch gelesen werden. Für die als Eingänge geschalteten Pins jedoch wird mittels der entsprechenden Bits der zugehörige Pull-Up-Widerstand ein- (Bit gesetzt) oder ausgeschaltet (Bit gelöscht). Dies erklärt auch die Existenz der Pin Data Register, da es ansonsten bei Eingängen nicht möglich wäre, das aktuelle Signal abzufragen.
Von Bedeutung für die I/O-Ports ist auch noch das Bit 2 (*PUD*) im *Special Function I/O Register* (*SFIOR*). Ist dieses gesetzt, so bleiben alle Pull-Up-Widerstände ausgeschaltet, unabhängig der Werte in den Port Data Registern. Die Position dieses und der restlichen Bits des Registers kann Abbildung 10 entnommen werden.

Bit	7	6	5	4	3	2	1	0	
	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0	DDRA
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 7: Bitbelegung des Data Direction Register A (DDRQ)

Bit	7	6	5	4	3	2	1	0	
	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0	PINA
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

Abbildung 8: Bitbelegung des Pin Data Register A (PINA)

3.2 Alternative Funktionen der I/O-Ports

Neben der Funktion als allgemeine digitale I/O-Ports können die Portpins auch für verschiedene Spezialfunktionen verwendet werden. Dabei sollte beachtet werden, dass auch bei aktivierten Alternativfunktionen der Zugriff über die Register der Standardports möglich ist. Da dies ein Fehlerpotential darstellt, empfiehlt es sich, eine Doppelverwendung zu vermeiden.

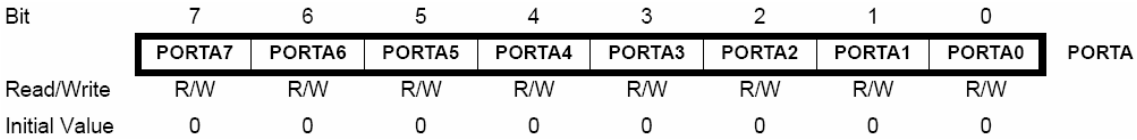

Abbildung 9: Bitbelegung des Port Data Register A (PORTA)

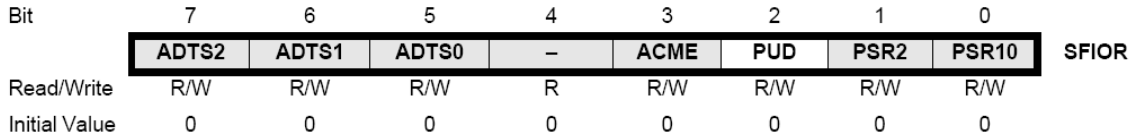

Abbildung 10: Bitbelegung des Special Function I/O Register (SFIOR)

Im folgenden werden die Spezialfunktionen dargestellt, die genaue Pinbelegung ist in Tabelle 3 aufgelistet.
Alle Anschlüsse des Port A können als Eingänge für die acht Kanäle des *Analog-Digital-Konverters* (*ADC*) verwendet werden, ansonsten existieren keine Alternativfunktionen.
An Port B können der Bus der *seriellen Programmierschnittstelle* (*SPI*), die beiden Eingänge der *analogen Vergleichseinheit* (*AC*), der *externe Interrupt 2* (*INT2*), der Takt der *seriellen Schnittstelle* (*USART*) sowie optionale Signale des *Timer/Counter 0* und *1* angeschlossen werden.
Port C ist alternativ belegt mit den Anschlüssen der Schnittstelle des *JTAG Debuggers*, der optionalen *externen Taktquelle der Zählwerke* und des *Two-Wire-Interface* (*TWI*) für den I^2C- oder den *System-Management-Bus* (*SMBus*)[28].
Auf die Pins an Port D können auch die *externen Interrupts 0* (*INT0*) und *1* (*INT1*), die beiden Datenleitungen der *seriellen Schnittstelle* (*USART*) und optionale Signale der *Timer/Counter 1* und *2* geschaltet werden.

Port Pin	Bezeichnung	Beschreibung Alternative Funktion(en)
PA7	ADC7	ADC Input Channel 7
PA6	ADC6	ADC Input Channel 6
PA5	ADC5	ADC Input Channel 5
PA4	ADC4	ADC Input Channel 4
PA3	ADC3	ADC Input Channel 3
PA2	ADC2	ADC Input Channel 2
PA1	ADC1	ADC Input Channel 1
PA0	ADC0	ADC Input Channel 0

[28] Der Unterschied liegt lediglich in Details der Spezifikationen

PB7	SCK	SPI Bus Serial Clock
PB6	MISO	SPI Bus Master Input / Slave Output
PB5	MOSI	SPI Bus Master Output / Slave Input
PB4	ISS	SPI Slave Select Input
PB3	AIN1 OC0	Analog Comparator Negative Input Timer/Counter0 Output Compare Match Output
PB2	AIN0 INT2	Analog Comparator Positive Input External Interrupt 2 Input
PB1	T1	Timer/Counter1 External Counter Input
PB0	T0 XCK	Timer/Counter0 External Counter Input USART External Clock Input / Output
PC7	TOSC2	Timer Oscillator Pin 2
PC6	TOSC1	Timer Oscillator Pin 1
PC5	TDI	JTAG Test Data In
PC4	TDO	JTAG Test Data Out
PC3	TMS	JTAG Test Mode Select
PC2	TCK	JTAG Test Clock
PC1	SDA	Two-wire Serial Bus Data Input/Output Line
PC0	SCL	Two-wire Serial Bus Clock Line
PD7	OCP2	Timer/Counter2 Output Compare Match Output
PD6	ICP1	Timer/Counter1 Input Capture Pin
PD5	OC1A	Timer/Counter1 Output Compare A Match Output
PD4	OC1B	Timer/Counter1 Output Compare B Match Output
PD3	INT1	External Interrupt 1 Input
PD2	INT0	External Interrupt 0 Input
PD1	TXD	USART Output Pin
PD0	RXD	USART Input Pin

Tabelle 3: Alternativfunktionen der I/O-Ports des AtMega16

4 Beispiel

Beim hier verwendeten Beispiel handelt es sich um einen Ausschnitt aus einer real existierende Schaltung. Die Originalschaltung besitzt viele weitere Bauelemente, welche das Beispiel unnötig komplex machen würden und hier deshalb ausgelassen werden.
Das reduzierte Schaltbild ist in Abbildung 11 gezeigt. Wie zu erkennen, sind dabei sechs Taster an den Mikrocontroller angeschlossen. Mittels Pull-Up-Widerständen werden die Signalpegel der Anschlusspins auf *high* gelegt. Dies geschieht in diesem Beispiel zur Verdeutlichung[29], die internen Pull-Up-Widerstände des AtMega16 hätten ebenso gut funktioniert. Wird einer der Taster gedrückt, so wird der jeweilige Pin mit Masse verbunden und so auf *low* gezogen.
Damit der Zustand der Taster nicht ständig in der Software abgefragt werden muss, sind alle Tasterpins über Dioden mit dem externen Interrupt 0 verbunden. Der Anschluss des Interrupts muss selbst ebenfalls über einen Pull-Up-Widerstand auf *high* gelegt werden. Die Dioden stellen sicher, dass beim Drücken eines Tasters nicht alle anderen Signale ebenfalls auf *low* gezogen werden, wohl aber der Interrupt-Pin (Der Interrupt ist im Endeffekt ein logisches Und über die Taster). Sinn davon ist es, beim Drücken jedes Tasters den Interrupt auszulösen und in der folgenden Interruptbehandlungsroutine mittels Abfrage der Portpins den eigentlichen Taster zu identifizieren.

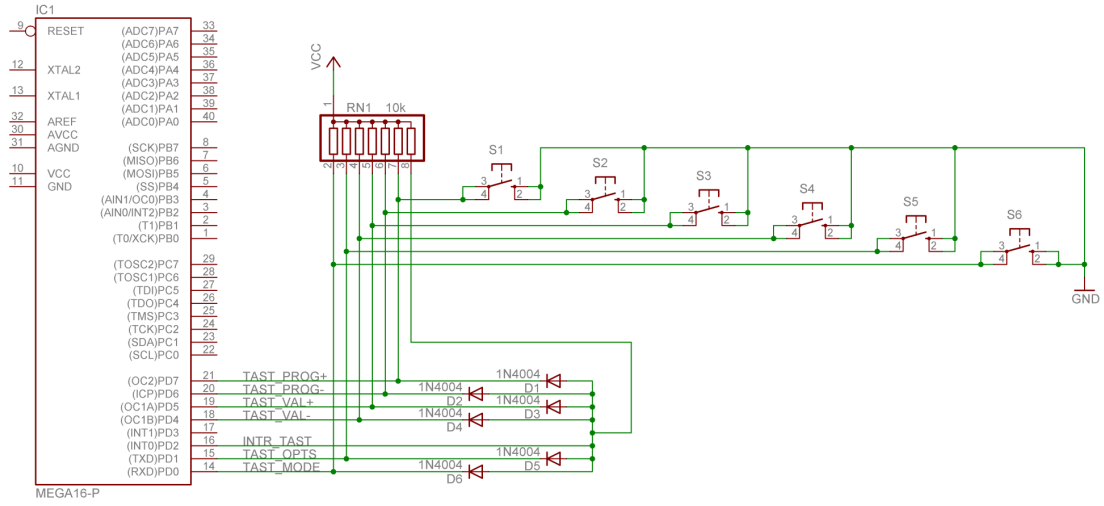

Abbildung 11: Schaltbild des verwendeten Beispiels

Die relevanten Ausschnitte der Tasterschaltung sind in Listing 2 aufgeführt. Vor dem eigentlich Codesegment werden dabei die Definitionen (Register etc.) des verwendeten Controllers importiert (Zeile 1). Desweiteren werden für einige Register neue Namen definiert, welche einen bestimmten Verwendungszweck verdeutlichen (Zeilen 4-5). Darüber hinaus werden Konstanten definiert, um die Abfrage der Pinbelegung der Taster zu vereinfachen (Zeilen

[29] Außerdem besaß der im allerersten Layout verwendete, ältere Controller keine eigenen Pull-Up-Widerstände.

8-14).
Der Programmcode beginnt mit der Interrupt-Vektor-Tabelle an der dafür vorgesehenen Programmspeicheradresse (Zeilen 19-21).
Im Initialisierungskopf des Hauptprogrammes werden der Stackpointer initalisiert (Zeilen 26-29), der Port D als Eingang konfiguriert (Zeilen 31-32), der externe Interrupt 0 auf die fallende Signalflanke eingestellt und eingeschaltet (Zeilen 19-21). Als letztes wird die globale Interruptsperrung aufgehoben (Zeile 36). Der eigentliche Teil des Hauptprogrammes kann außer acht gelassen werden.
Die Interruptbehandlungsroutine beginnt mit dem obligatorischen Sichern des Statusregisters und aller weiteren Register (Zeilen 44-45), welche in der Behandlungsroutine verwendet werden (in diesen könnten sich ebenfalls noch relevante Daten der unterbrochenen Routine befinden). Danach wird die erfolgte globale Interruptsperrung aufgehoben (Zeile 46). Als nächstes wird der Status der Pins an Port D in ein Arbeitsregister eingelesen[30] (Zeile 48). Anhand der Daten im Arbeitsregister wird nun bestimmt, welcher Taster gedrückt wurde (Zeilen 49-61), und dementsprechend an die Sprungmarke der jeweiligen Subroutine verzweigt (der Kürze halber ist nur der Rumpf der ersten aufgeführt).
In der jeweiligen Subroutine können die entsprechenden Aktionen bewirkt werden, danach erfolgt der Sprung zur letzten Subroutine der Interruptbehandlung (Zeilen 63-65). Am Ende der Interruptbehandlungsroutine folgen nun die obligatorischen Vorgänge der globalen Interruptsperrung, das Wiederherstellen der gesicherten Register und der Return-from-Interrupt.
Damit ist die Interruptbehandlung abgeschlossen.

```
1  .include "m16def.inc"          ; Definition für AtMega16
2
3  ;************* Register mit *richtigen* Namen ********
4  .def   templ=r18               ; Temporäre Variable (low byte)
5  ;...   ... ...
6
7  ;************* Pinbelegung(Taster) für Port D ********
8  .equ   tast_prog+=7            ; nächstes Anzeigeprogramm
9  .equ   tast_prog-=6            ; voriges Anzeigeprogramm
10 .equ   tast_val+=5             ; Wert erhöhen
11 .equ   tast_val-=4             ; Wert verringern
12 .equ   tast_opts=1             ; Option wechseln
13 .equ   tast_mode=0             ; Regelungsstufe umschalten
14 ;...   ... ...
15
16 ;***************************************************
17 .cseg                          ; CODE segment
18 ;************* Interrupt-Vektor-Tabelle **************
19 .org   0x000    jmp  RESET     ; Reset Handler
```

[30] In einer realen Schaltung wird zuvor eigentlich noch eine kleine Pause zum *Entprellen* des Tasters eingelegt (das Signal kann beim Drücken und Loslassen von Tastern kurzzeitig „flattern“).

```
20 .org   0x002    jmp   EXT_INT0 ; IRQ0 Handler
21 ;...   ...      ...   ...
22
23 ;************* Hauptprogramm ************************
24 .org 0x02A
25 RESET:
26   ldi   templ,high(RAMEND)
27   out   SPH,templ             ; Stack Pointer initialisieren
28   ldi   templ,low(RAMEND)
29   out   SPL,templ
30 ;  ...   ...
31   clr   templ
32   out   DDRD,templ            ; Port D alle Pins als Eingang (=0)
33   cbi   MCUCR,0
34   sbi   MCUCR,1               ; Interrupt 0 bei fallender Flanke
35   sbi   GICR,6                ; Interrupt 0 enable
36   sei                         ; Global Interrupt enable
37
38 MAIN_START:
39 ;  ...   ...                  ; eigentliches Hauptprogramm
40   rjmp  MAIN_START            ; in einer Endlosschleife durchlaufen
41
42 ;************* Interrupt-Routine Interrupt 0 *********
43 EXT_INT0:
44   in    r0,sreg               ; Status-Reg sichern
45 ;  ...   ...                  ; Hier verwendete Register sichern
46   sei                         ; Global Interrupt enable
47
48   in    templ,PIND            ; Welcher Taster ist gedrückt (=0)?
49   sbrs  templ,tast_prog+      ; Programm+ Taster?
50   rjmp  NEXTPROG
51   sbrs  templ,tast_prog-      ; Programm- Taster?
52   rjmp  PREVPROG
53   sbrs  templ,tast_val+       ; Wert+ Taster?
54   rjmp  VALINC
55   sbrs  templ,tast_val-       ; Wert- Taster?
56   rjmp  VALDEC
57   sbrs  templ,tast_opts       ; Optionen Taster?
58   rjmp  OPTCHANGE
59   sbrs  templ,tast_mode       ; Regelungsstufe Taster?
60   rjmp  MODECHANGE
61   rjmp  END_EXT_INT0          ; Unwahrscheinlich, dennoch abfangen!
62
63 NEXTPROG:                     ; Aktionen für die
```

```
64 ; ... ...                    ; Taste ausführen
65   rjmp END_EXT_INT0          ; zum Ende der Interruptverarbeitung
66
67 ; ...                        ; weitere Tastersubroutinen
68
69 END_EXT_INT0:
70   cli                        ; Global Interrupt disable
71 ; ... ...                    ; verwendete Register wiederherstellen
72   out sreg,r0                ; Status-Reg wiederherstellen
73   reti                       ; Rücksprung vom Interrupt
```

Listing 2: Quelltext des verwendeten Beispiels

5 Fazit

Nach den erfolgten Betrachtungen ist klar, dass große Teile der heutigen Computerarchitektur ohne den Interruptmechanismus nicht machbar wären, und dass die heutezutage erzielte Leistungsfähigkeit eines Prozessors im Zusammenspiel mit einer Vielzahl von Geräten undenkbar wäre.
Ebenso ist klar, dass Interrupts auch die Leistungsfähigkeit, Flexibilität und Möglichkeiten von modernen Mikrocontrollern steigern. Aufgrund der hardwarenahen Programmierung liegt es jedoch vor allem am Entwickler, diese Möglichkeiten zu kennen und zu nutzen.

Literatur

[1] *Vorlesungsskript Rechnerstrukturen*, WS 2000/2001, Prof. Dr. Christoph Steigner, Universität Koblenz-Landau, FB4

[2] *Vorlesungsskript Technische Informatik B*, Kapitel 8, WS 2005/2006, Prof. Dr. Christoph Steigner, Universität Koblenz-Landau, FB4
`https://www.uni-koblenz.de/~steigner/tib/TIB-8.ppt`

[3] *Datenblatt Atmel AtMega16*, Revision M, Atmel Corporation, April 2006
`http://www.atmel.com/dyn/resources/prod_documents/doc2466.pdf`