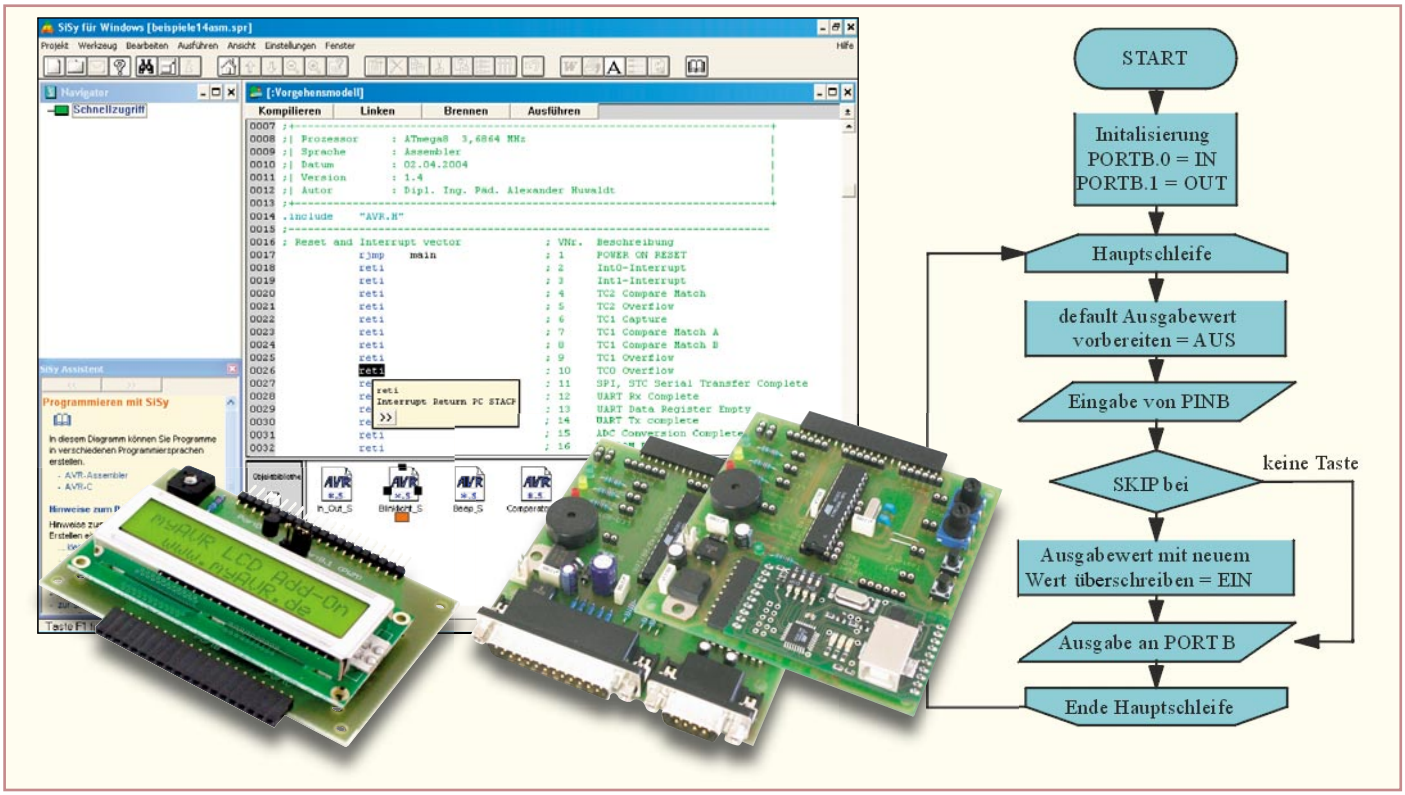


Mikrocontroller-Einstieg mit myAVR Teil 3

Keine Angst vor dem Einstieg in die Welt der Mikrocontroller-Programmierung! Die myAVR-Sets enthalten alles Nötige für den schnellen und fundierten Beginn der Programmierer-Karriere – Experimentier-Board mit ATMEL-Controller, Lehrbuch, Softwarepaket, Kabel, sämtliches Zubehör. Im dritten Teil unserer Serie zum Einstieg in die AVR-Programmierung fahren wir mit der I/O-Programmierung anhand eines weiteren Beispiels fort und befassen uns mit dem Thema Interrupt-Steuerung.

Rein wie raus – die I/O-Ports

Nachdem wir uns in der letzten Ausgabe sehr ausführlich der Funktion der I/O-Ports des AVR-Controllers gewidmet hatten, ist uns die Aussage noch gegenwärtig, dass es über das Steuerregister DDRx möglich ist, festzulegen, welcher Pin als Eingang oder als Ausgang wirken soll. Über das Register PORTx erfolgt entsprechend die Ausgabe, über PINx die Eingabe von Signalen über die entsprechenden I/O-Lines. Was man zunächst wissen muss, ist der Fakt, dass der Controller nach dem Einschalten oder einem Reset immer den Inhalt der Register DDRB/C/D komplett auf null setzt (0b00000000) – die „anhängenden“ I/O-Lines sind damit zunächst stets als Eingang gesetzt. Erst das Laden des entsprechenden Steuerregisters mit „1“, wie in unserem ersten Programmbeispiel, führt zu einer Konfiguration als Ausgabe-Linie.

Wir wollen aber nun den Fall der Konfiguration als Eingabe-Linie betrachten. Das fängt damit an, dass ein digitaler Signal-Eingang, noch dazu so ein empfindlicher wie der CMOS-Eingang des AVR-Controllers, stets definiert auf $+U_B$ (Pull-up) oder Masse (Pull-down) zu setzen ist. Warum? Ein digitaler Eingang wirkt „frei in der Luft hängend“ wie eine Empfangsantenne für alle möglichen elektromagnetischen Felder und liefert dem nachfolgenden Steuerregister undefinierbare Signale statt ordentlicher High- und Low-Signale, wie es in der Digitaltechnik üblich ist. Schließt man hingegen den Eingang definiert mit einem Pull-up-Widerstand ab, wie es in Abbildung 26 zu sehen ist, ist der Eingang nie offen und kann so die erwähnten Felder nicht mehr ohne weiteres „einfangen“. Ist

der hier als „Signalquelle“ dienende Taster offen, liegt der Eingang definiert auf +U_B, liefert also die digitale Information „1“. Ist der Taster hingegen geschlossen, liegt der Eingang auf Masse und liefert die digitale Information „0“. Die Information, ob der jeweilige interne Pull-up-Widerstand des Controllers aktiviert werden soll oder nicht, muss also, zusammen mit der Pin-Auswahl des IN-Pins, Bestandteil des Steuerprogramms werden, um eine I/O-Line als Eingang zu definieren. Dieser Programmschritt gehört also in das Initialisierungsprogramm. Dazu ist (in unserem Beispielprogramm beschäftigen wir uns wie im Ausgabeprogramm mit Port B) im Port-B-Register eine logische „1“ an der entsprechenden Position zu setzen.

Das Ein-/Ausgabeprogramm

Setzen wir die beiden eben besprochenen Schritte (wir wollen Port B 0 als Eingang definieren und den Pull-up-Widerstand setzen) also in Programmcode um.

Da wir natürlich wissen wollen, ob das funktioniert, verwenden wir einen der Taster auf dem myAVR-Board als Eingabeorgan und verbinden das schon in Teil 2 erarbeitete Ausgabeprogramm mit dieser Eingabeoperation, was nichts anderes heißt als: den an Port B 0 anzuschließenden Taster drücken, eine der LEDs mit Port B 1 verbinden und diese mit dem Taster ein- und ausschalten. Damit werden gleich zwei Programmroutinen miteinander verbunden, einer Aktion folgt eine Reaktion.

Dazu gehen wir dieses Mal den umgekehrten Weg und sehen uns zunächst das fertige Programm an, um den strukturierten Aufbau eines Programms weiter zu vertiefen (Abbildung 27). Hier erscheint zunächst die bereits bekannte und unverändert übernommene Reset- und Interrupt-Tabelle, darauf der Initialisierungsteil. Hier sind gegenüber der reinen Ausgabe-Initialisierung ein paar Zeilen hinzugekommen – die Eingabe-Initialisierung.

Zunächst ist hier ein neuer Befehl zu sehen: „cbi“.

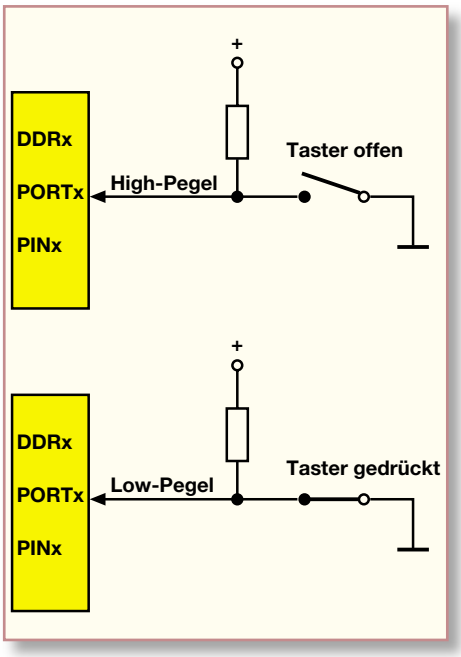


Bild 26: Die Pegelverhältnisse am Controller-Eingang mit aktiviertem Pull-up-Widerstand

```
;+------------------------------------------------------------------------+
;| Titel          : Beispiel IN/OUT für SiSy AVR-Board                    |
;+------------------------------------------------------------------------+
;| Funktion       : Solange Taster 1 gedrückt ist, wird eine LED          |
;|                : eingeschaltet.                                        |
;| Schaltung      : Taste an Port B.0, rote LED an Port B.1               |
;+------------------------------------------------------------------------+
;| Prozessor      : ATmega8  3,6864 MHz                                   |
;| Sprache        : Assembler                                             |
;| Datum          : 05.06.2004                                            |
;| Version        : 1.5                                                   |
;| Autor          : Dipl. Ing. Päd. Alexander Huwaldt                     |
;+------------------------------------------------------------------------+
.include    "AVR.H"
;-------------------------------------------------------------------------
; Reset and Interrupt vector              ; VNr.  Beschreibung
            rjmp    main                  ; 1     POWER ON RESET
            reti                          ; 2     Int0-Interrupt
            reti                          ; 3     Int1-Interrupt
            reti                          ; 4     TC2 Compare Match
            reti                          ; 5     TC2 Overflow
            reti                          ; 6     TC1 Capture
            reti                          ; 7     TC1 Compare Match A
            reti                          ; 8     TC1 Compare Match B
            reti                          ; 9     TC1 Overflow
            reti                          ; 10    TC0 Overflow
            reti                          ; 11    SPI, STC Serial Transfer Complete
            reti                          ; 12    UART Rx Complete
            reti                          ; 13    UART Data Register Empty
            reti                          ; 14    UART Tx complete
            reti                          ; 15    ADC Conversion Complete
            reti                          ; 16    EEPROM Ready
            reti                          ; 17    Analog Comparator
            reti                          ; 18    TWI (I²C) Serial Interface
            reti                          ; 19    Store Program Memory Ready
;-------------------------------------------------------------------------
; Start, Power ON, Reset
main:       ldi     r16,    hi8(RAMEND)
            out     SPH,    r16
            ldi     r16,    lo8(RAMEND)   ; Stack Initialisierung
            out     SPL,    r16           ; Init Stackpointer
            cbi     DDRB,   0             ; PortB.0 auf Eingang
            sbi     PORTB,  0             ; PortB.0 Pullup
            sbi     DDRB,   1             ; PortB.1 auf Ausgang
;-------------------------------------------------------------------------
mainloop:   wdr
            ldi     r16,    0b00000001    ; Wert bei Taste nicht gedrückt
            in      r17,    PINB          ; INPUT
            sbrs    r17, 0                ; Skip, wenn Taste nicht gedrückt, PORT B.0 = SET (1)
            ldi     r16,    0b00000011    ; Wert bei Taste gedrückt (r16 überschrieben)
            out     PORTB, r16            ; LED an/aus
            rjmp    mainloop
;-------------------------------------------------------------------------
```

Bild 27: Der Quellcode für unser I/O-Programm

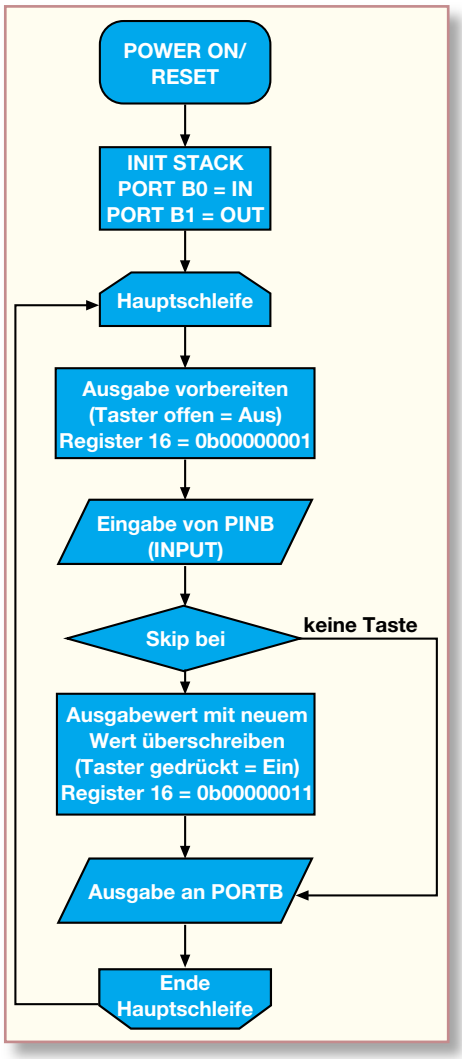


Bild 28: Das Flussdiagramm für die Lösung unserer I/O-Aufgabe

Der löscht an der definierten Stelle des Registers das nach dem Reset auf „0“ gesetzte Bit und setzt das Steuerregister für Port B 0 auf „Eingang“, also, wie erwähnt, auf „0“. Der folgende „sbi“-Befehl spricht direkt das Bit 0 im I/O-Register an und aktiviert damit die Pull-up-Funktion an diesem Pin.

Nun ist nur noch festzulegen, dass Port B 1 als Ausgang arbeiten soll, ergo ist das Steuerregister für Port B 1 auf „1“ zu setzen. Damit ist die Initialisierung erledigt und wir können uns dem Hauptprogramm zuwenden.

Nach dem Zurücksetzen des Watchdogs wird zuerst eines der allgemeinen Register, hier wieder das bereits bekannte r16, mit einem Ausgabewert für den Fall geladen, dass die Taste nicht gedrückt ist, denn auch dieser Fall muss definiert werden, um dem Rechner eindeutige Anweisungen für jeden Betriebsfall zu geben.

Das Laden von 0b00000001 bedeutet hier: Pin B 0 = 1 aktiviert den Pull-up-Widerstand, Pin B 1 = 0 heißt: LED ausgeschaltet.

Der nächste „in“-Befehl sorgt dafür, dass der Wert, der am Eingang B 0 über das Eingaberegister Pin B (siehe auch Abbildung 13 im Teil 2) durch den Taster definiert wird („0“ oder „1“), wiederum in einem allgemeinen Register, hier r17, zur Auswertung zwischengespeichert wird. Dies ist der an sich unauffällige Knackpunkt des Eingabeprogramms, der Input-Befehl.

Jetzt folgen die Festlegungen, was der Controller mit dem eben in r17 abgelegten Wert anfangen soll. Hier kommt mit „sbrs“ ein so genannter eleganter Sprungbefehl (engl. skip) zur Anwendung. „sbrs“ heißt „Skip if Bit in Register Set“, zu Deutsch, der Controller soll den dem Skip-Befehl folgenden Programmschritt auslassen, falls im folgend angegebenen Register (hier r17) das ebenfalls angegebene Bit (hier Bit „0“) gesetzt ist. Vereinfacht gesagt: Steht hier der zunächst aus Port B ausgelesene Wert von Bit 0 auf einer logischen „1“, heißt dies, der Taster ist nicht gedrückt und die LED darf nicht angesteuert werden. Damit springt das Programm sofort in die Out-Zeile. Hier verändert sich nichts, da in Register r16 ja immer noch die Ausgabedaten für einen offenen Taster definiert sind und die LED nicht angesteuert wird (Bit B 1 steht immer noch auf „0“). Über den „rjmp“-Befehl kehrt das Programm wieder zurück zum Beginn des Hauptprogramms.

Dieser beschriebene Ablauf setzt sich so lange fort, bis der Taster gedrückt wird. Was passiert jetzt? Bei der Abarbeitung des Skip-Befehls erkennt das Programm, dass nun im Register r17 für Port B für Bit 0 eine logische „0“ abgelegt ist, der Programmsprung wird also nicht ausgeführt und es geht weiter zum nächsten Register-Ladebefehl. Nun wird Register r16 mit dem neuen Inhalt geladen: Pin B 0 = 1 behält seinen Status als Eingang mit aktiviertem Pull-up-Widerstand, während Pin B 1 = 1 bedeutet, dass beim Auslesen des Registers

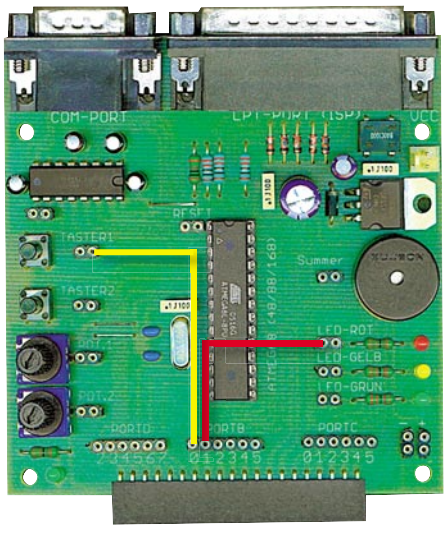


Bild 29: Die Verdrahtung zur I/O-Aufgabe auf dem myAVR-Board

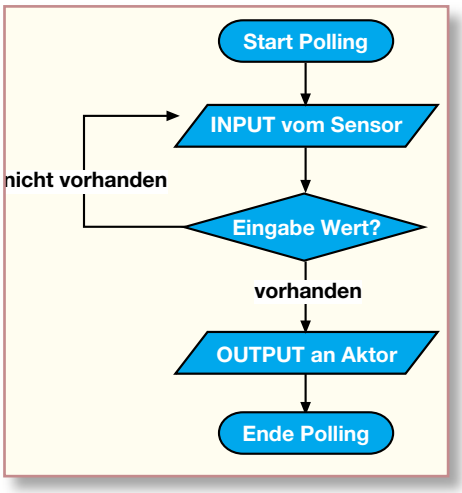


Bild 30: Der klassische Programmablauf beim Polling-Verfahren

beim folgenden „out“-Befehl die LED an Pin B 1 eingeschaltet wird. Auch diese Programmschleife wird jetzt so lange durchlaufen, bis die Taste losgelassen wird, das Programm dies feststellt und der Skip-Befehl wieder in Aktion tritt. Abbildung 28 zeigt noch einmal das Flussdiagramm des Programms, das die zu lösende Aufgabe übersichtlich dokumentiert.

Unsere Beschreibung quasi „von hinten“ sollte zeigen, dass es relativ einfach ist, einmal erarbeitete Algorithmen erneut anzuwenden und in neue Programmlösungen einzubinden. Hat man erst einmal die wesentlichen Befehle und ihre Auswirkungen im Hinterkopf, fällt es nicht mehr schwer, zunächst das Flussdiagramm anhand der zu lösenden Aufgabe zu entwerfen. Dass das seinen Sinn hat, merkt man spätestens dann, wenn man so weit ist, mehrere Programmteile, so genannte Unterprogramme, miteinander zu kombinieren, ohne sich im Quellcode zu „verlaufen“. Denn allein schon unser kleines Beispiel zeigt, dass man keine Aktivität, die im Verlaufe der Programmabarbeitung passieren kann, außer Acht lassen darf, um keine ungewollten Reaktionen zu erhalten. Weiterführendes zur Systematik des Programmentwurfs findet sich sehr ausführlich, nebst mehreren, auch umfangreicheren Programmierbeispielen, im Lehrgangsmaterial und auf der Programm-CD. Hier gibt es auch eine ausführliche und vor allem gut verständliche Unterweisung zur Programmierung mit Unterprogrammen und Sprunganweisungen, auf die wir im Rahmen unserer Serie ebenfalls nur dann näher eingehen, wenn sich die Gelegenheit dazu in unseren Beispielprogrammen ergibt.

Unser Programm wird nun in die Praxis umgesetzt – ein inzwischen bekannter Vorgang: Quellcode in den Editor schreiben, kompilieren, linken, auf den AVR brennen. Verbindet man nun nach Abbildung 29 einen Taster des myAVR-Boards mit dem

Bild 31:
Der Grundaufbau der Interrupt-Service-Routine

```
;------------------------------------------------------------------------
Grundaufbau Interrupt-Service-Routine (ISR)
EXT_INT0: cli                           ; weitere Interrupts unterbinden
          push    r16                   ; Register retten, hier als Beispiel r16
          ; ...                         ; Interruptbehandlung
          pop     r16                   ; Registerinhalt wieder zurückladen
          sei                           ; Interrupts wieder zulassen
          reti                          ; Interrupt beenden
;------------------------------------------------------------------------
```

Port-Pin B 0 und eine LED mit dem Port-Pin B 1, so muss nach Anschluss der Spannungsversorgung beim Drücken des Tasters die LED aufleuchten. Das war zu einfach? Dann erweitern Sie doch zum Training das Programm um einen zweiten Taster, der die nächste LED ansteuert. Da wird das Programm schon etwas komplexer!

Interrupt-Steuerung

Für Programmier-Einsteiger hat das Wort „Interrupt-Steuerung" einigen Schrecken – die Profis sprechen da von Interrupt-Vektoren, Interrupt-Service-Routinen, Interrupt-Quelle, -Maske usw. Der myAVR-Lehrgang schafft es dennoch, dieses etwas sperrig erscheinende Thema so zu vermitteln, dass man nach kurzer Zeit nicht nur mitreden, sondern auch entsprechend programmieren kann. Das wollen wir hier ausprobieren!

Was ist Interrupt-Steuerung eigentlich?

Bei unserem letzten Programmierbeispiel haben wir ein typisches Beispiel einer sequenziellen Abfrageroutine (Polling) kennen gelernt: Der Zustand des Tasters wird ständig abgefragt, bis dieser den erwarteten Zustand (gedrückt) einnimmt (Abbildung 30) und den Aktor (die LED) anspricht. Gleichermaßen könnte so auch ein längeres Programm so lange unterbrochen werden, bis das erwartete Ereignis eintritt. Solange kehrt das Polling-Programm immer wieder zur Abfrage zurück. Im ungünstigen Fall können aber dadurch weitere Verarbeitungsaufgaben des Controllers, etwa eine Ausgabe auf einem Display, gestört werden bzw. ganz ausbleiben. Und hat man das Abfrageprogramm als Unterprogramm eingebaut, kann es bei zeitkritischen Abläufen durchaus geschehen, dass das Programm gerade noch „woanders" arbeitet, wenn z. B. ein kurzer Steuerimpuls eintrifft. Der kann dann schlichtweg vom Programm „übersehen" werden – mit womöglich schlimmen Folgen. Also muss man hier einen anderen Weg gehen, um effektiv zu programmieren – man lässt das Programm nicht dauernd ein bestimmtes Ereignis abfragen, sondern steuert das Programm durch das Eintreffen des Ereignisses selbst! Dabei kann das Programm an jeder beliebigen Stelle (das muss man ausnahmsweise nicht vorher festlegen) bei Eintreffen des Ereignisses unterbrochen und ein Unterprogramm für die Verarbeitung des Ereignisses aufgerufen werden. Ist das erfolgt, setzt das Hauptprogramm an der Stelle fort, an der es unterbrochen wurde. Genau das bedeutet „Interrupt-Steuerung"!

Die besteht aus immer den gleichen Elementen:

Tabelle 1: Die Interrupt-Quellen des ATmega8

Beschreibung	Pin	Port
Extern RESET/Intern Watchdog	1	C6
Extern 0	4	D2
Extern 1	5	D3
Intern Timer 1/Eingabe	14	B0
Intern Timer 1, Vergleich	-	-
Intern Timer 1, Überlauf	-	-
Intern Timer 0, Überlauf	-	-
Intern SPI/Datenübertragung	17–19	B3–B1
Extern RX, UART Handshake receive	2	D0
Intern UDR, UART Senderegister	-	-
Extern TX, UART Handshake transmit	3	D1
Intern ADC	23–28	C0–C5
Intern EEPROM	-	-
Intern Analog Comparator	-	-
Extern TWI (I^2C), Serial Interface	27/28	C4/C5
Intern Store Program Memory	-	-

Interrupt-Quelle

Als Interrupt-Quelle können äußere Sensoren (Ports), Schwellwerte des A/D-Wandlers oder der integrierte Timer genauso auftreten wie spezielle Software-Befehle. Der AVR-Controller auf unserem myAVR-Board verfügt über zahlreiche dieser Quellen. Tabelle 1 zeigt einen Auszug der gängigsten Interrupt-Quellen dieses Controllers. Die Anzahl und Art der Interrupt-Quellen ist bei jedem Controller eine andere und dessen Aufgabenbereich angepasst.

Interrupt-Behandlungsroutine

Das ist eben jenes beschriebene Unterprogramm, das einer Interrupt-Quelle zugeordnet ist und als Reaktion auf deren Unterbrechungsanforderung gestartet wird, es wird auch Interrupt-Service-Routine (ISR) genannt. Wie sie grundsätzlich aufgebaut ist, zeigt Abbildung 31. Die hier aufgeführten Befehle werden wir später noch genauer erläutern. Man erkennt auf jeden Fall bereits die Systematik: Tritt ein Interrupt auf, wird die zugehörige Interrupt-Service-Routine (ISR) angesprungen. Weitere Interrupts werden zunächst zum geregelten Abarbeiten dieser ISR unterbunden, wenn nötig Speicherinhalte von Registern gerettet, die eigentliche ISR ausgeführt, die Ordnung im Speicher wiederhergestellt, das Steuerwerk für andere Interrupts wieder freigegeben, die ISR insgesamt beendet, und das Programm kehrt zum Hauptprogramm zurück.

Interrupt-Vektor

Den kennen wir schon, er steht in der Reset- und Interrupt-Tabelle jedes Quellprogramms. Hinter diesem Begriff verbirgt sich eine hier festgeschriebene Adresse, auf der ein Befehl zum Aufruf oder die Adresse der zugehörigen Interrupt-Behandlungsroutine gespeichert ist. Bei Eintreffen eines bestimmten Ereignisses springt das Programm also immer mit dem jeweils einzutragenden Befehl „rjmp" (+ selbst festzulegendem Unterprogramm-Namen) zu dem diesem Ereignis zugeordneten Unterprogramm. Nicht zur Nutzung vorgesehene Interrupt-Behandlungsroutinen sind mit dem Befehl „reti" zu versehen. Der sorgt dafür, dass ein hier dennoch aufgetretener Interrupt ordnungsgemäß beendet wird.

Da die Länge der Interrupt-Vektor-Tabelle von der Anzahl der jeweils verfügbaren

```
; Reset and Interrupt vector            ; VNr.  Beschreibung
            rjmp    main                ; 1     POWER ON RESET
            rjmp    EXT_INT0            ; 2     Int0-Interrupt
            reti                        ; 3     Int1-Interrupt
            reti                        ; 4     TC2 Compare Match
            reti                        ; 5     TC2 Overflow
            reti                        ; 6     TC1 Capture
            reti                        ; 7     TC1 Compare Match A
            reti                        ; 8     TC1 Compare Match B
            reti                        ; 9     TC1 Overflow
            reti                        ; 10    TC0 Overflow
            reti                        ; 11    SPI, STC Serial Transfer Complete
```

Bild 32: In diesem Ausschnitt aus der Reset- und Interrupt-Vektor-Tabelle ist bereits der Sprungbefehl zur Interrupt-Service-Routine mit dem Namen „EXT_INT0“ eingetragen.

Interrupt-Quellen abhängt, hat sie bei jedem Controller der ATmega-Reihe auch eine andere Länge. Die Vektoren sind in der Tabelle in einer festen Reihenfolge platziert, die nicht verändert werden darf. Gleichfalls darf man keine Vektoren aus der Tabelle löschen, die vor einem benutzten Vektor liegen.

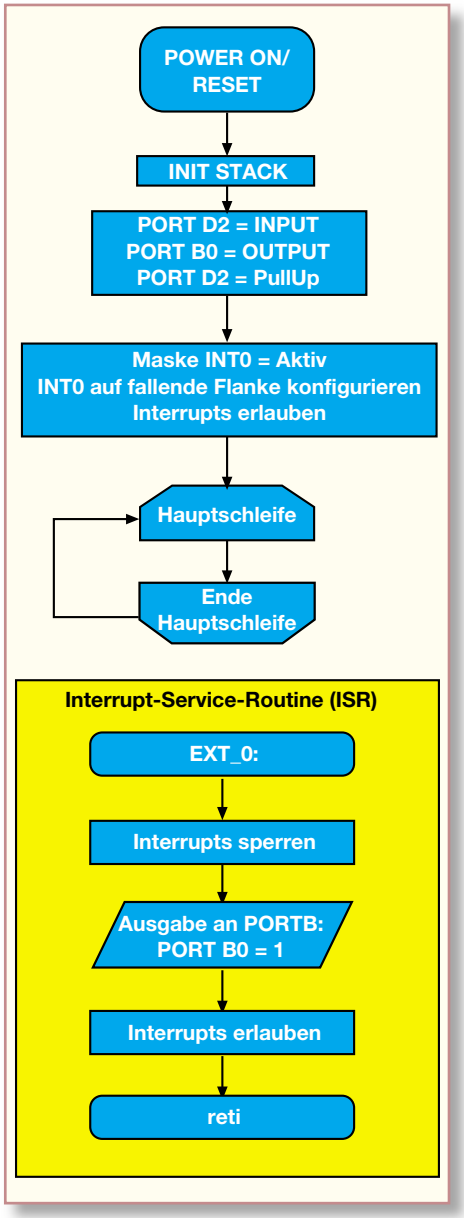


Bild 33: Flussdiagramm zur Interrupt-Steuerung

Interrupt-Kontroll-Logik

Sie ist fester Bestandteil des Prozessor-Steuerwerks und verantwortlich für das Erlauben oder Verbieten sowie Konfigurieren von Unterbrechungen, dem tatsächlichen Unterbrechen des laufenden Programms und für das Auslösen der korrekten Interrupt-Behandlungsroutine.

Wichtige Adressen

Dass bei der Verwaltung der Interrupt-Quellen und -Vektoren Ordnung herrschen muss, ist sicher jedem klar. Hier gibt es kein Variieren, ab der Adresse 0x0001 erwartet der AVR-Controller im Programmspeicher immer die Liste der Interrupt-Vektoren für die jeweiligen Interrupt-Quellen. Die Adresse 0x0000 ist immer fest für die Reset-Interrupt-Routine (ausgelöst durch ein Reset-Signal an Pin 1, Zuschalten der Betriebsspannung oder Auslösen des Watchdogs) reserviert.

Ein Beispiel für eine aktivierte Interrupt-Behandlungsroutine für die Interruptleitung INT0 (Pin 4, Port D 2) unseres AVR-Controllers zeigt Abbildung 32. Hier ist auch noch einmal die eben beschriebene Adressreservierung zu erkennen.

Interrupt-Befehle und -Register

Für das gesamte Handling der Interrupt-Quellen stehen Steuerregister und einige spezielle Befehle zur Verfügung. Die beiden wichtigen Befehle „sei“ und „cli“ dienen der Aktivierung und Deaktivierung der Interrupts:

sei – signalisiert dem Steuerwerk, dass Interrupts erlaubt sind. Er setzt das Interrupt-Flag „I“ (Bit 7) im Status-Register SREG (siehe Registerübersicht zu myAVR).

cli – signalisiert dem Steuerwerk, dass Interrupts verboten sind. Er löscht das Interrupt-Flag „I“ (Bit 7) im Status-Register SREG (siehe Registerübersicht zu myAVR).

Jetzt wird auch ein Blick in die eben genannte Registerübersicht interessant. Hier finden wir für jede Interrupt-Quelle mindestens ein spezifisches Steuerregister (Interrupt-Register und z. B. Register MCUCR), das die Konfiguration der Interrupt-Quelle (prinzipiell ähnlich wie bei den I/O-Registern) erlaubt.

Interrupts, die nicht abschaltbar sind, nennt man „nicht maskierbar“. Solch ein Interrupt ist z. B. die Interrupt-Quelle „RESET“ – logisch, würde man diese versehentlich abschalten, gelänge kein definierter Reset des Controllers mehr!

Hingegen sind die (externen) Interrupts, die man per Programm ein- und ausschalten kann, mit dem Begriff „maskierbar“ gekennzeichnet (bei unserem ATmega8 sind dies z. B. die allgemeinen externen Interrupts INT0 und INT1). Dazu stehen entsprechende Steuerregister als „Maske“ zur Verfügung, in denen man über die entsprechenden Bits die Interrupts ein- („1“) und ausschalten („0“) kann. Man teilt also dem Steuerwerk mit, welches Ereignis als Interrupt akzeptiert werden soll. Für unsere eben erwähnten INT0 und INT1 heißt das zuständige Register „GICR“. Je nachdem, wie dessen Bits 6 bzw. 7 gesetzt werden, sind die Interrupts erlaubt oder gesperrt.

Schließlich kann auch die Art der Interrupt-Auslösung festgelegt werden. Bei jedem digitalen Signal gibt es die unterschiedlichen Zustände High-Pegel, Low-Pegel, fallende und steigende Signalflanke, die die externen Interrupts auslösen können. Diese Konfiguration erfolgt über die Bits 0 bis 3 des Registers MCUCR (siehe Registertabelle). Bei Setzen des Bits 0 erfolgt eine fortlaufende Interrupt-Auslösung, solange Low-Pegel anliegt, bei Bit 1 löst jede Pegeländerung einen Interrupt aus, bei Bit 2 die fallende, bei Bit 3 die steigende Flanke.

Nach so viel Theorie wollen wir das Ganze zunächst wieder an einem ganz einfachen Beispiel praktizieren – wir schließen einen Taster an den Interrupt-Port INT0 (Port D 2) an und nutzen diesen als Ereignis-Quelle. Wird der Taster gedrückt, soll wieder die rote LED an Port B 0 aufleuchten. Die Auswertung wollen wir über die fallende Signalflanke (der für eine Auswertung interessante Moment also, wo der Taster tatsächlich gedrückt wird und den Eingang auf „low“ schaltet) vornehmen.

Dieses Mal machen wir uns die zu planenden Programmschritte zuerst anhand des Flussdiagramms klar (Abbildung 33):

- initialisieren
- Port D 2 als Eingang festlegen und Pull-up aktivieren
- Port B 0 als Ausgang festlegen
- Maskierung für INT0 aktivieren, INT0 für fallende Flanke konfigurieren, Interrupt erlauben
- bei Ereignis ISR starten, andere Interrupts sperren, Port B 0 auf 1 (LED an) schalten, Interrupts freigeben, zurück zum Hauptprogramm

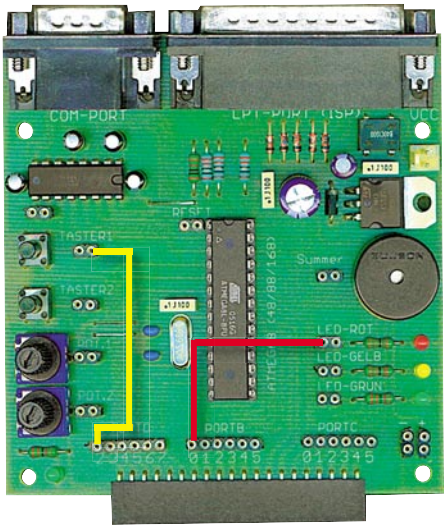
Im Quellcode-Editor wird wieder das Grundgerüst geladen und vorbereitet, dieses

Mal aktivieren wir aber den Vektor für den INT0-Interrupt, hier mit „EXT_0“.

Die Initialisierung wird hier etwas länger: Nach der Konfiguration des Eingangs und des Ausgangs mit „cbi“- und „sbi“-Befehlen wird zunächst im Register r16 eingetragen, dass INT0 maskiert werden soll (Bit 6 auf „1“, erlaubt den externen Interrupt über INT0). Die Ansprache dieses Registers erfolgt wie bei einem I/O-Register per „out“-Befehl). Danach erfolgt das Auslesen dieses Registerwertes in das Interrupt-Register GICR).

Die fallende Flanke als Auslöser wird jetzt ebenfalls ins Register r16 eingetragen (Bit 1 = 1, Bit 0 = 0; dabei wird der vorherige Wert gelöscht) und schließlich in das MCUCR-Kontroll-Register geladen.

Bild 34: Die Verdrahtung zur Interrupt-Steuerung auf dem myAVR-Board




```
;+----------------------------------------------------------------------+
;| Title         : Lichtschalter mit Interrupt mit SiSyAVR              |
;+----------------------------------------------------------------------+
;| Funktion      : Externer Taster als Interruptquelle schaltet LED ein |
;| Schaltung     : Taster an PD.2, LED an PB.0                          |
;+----------------------------------------------------------------------+
;| Prozssor      :AT90S4433  3,6864 MHz                                 |
;| Sprache       :Assembler                                             |
;| Datum         :12.05.2004                                            |
;| Version       :1.6                                                   |
;| Autor         :Dipl. Ing. Päd. Alexander Huwaldt                     |
;+----------------------------------------------------------------------+
.include    "AVR.H"
;----------------------------------------------------------------------
; Reset and Interrupt vectors
            rjmp    main                ;POWER ON RESET
            rjmp    EXT_0               ;Int0-Interrupt
            reti                        ;Int1-Interrupt
            reti                        ;TC1-Capture
            reti                        ;TC1-Compare
            reti                        ;TC1-Overflow
            reti                        ;TC0 Overflow, r
            reti                        ;SPI, STC Serial Transfer Complete
            reti                        ;UART Rx Complete
            reti                        ;UART Data Register Empty
            reti                        ;UART Tx complete
            reti                        ;ADC Conversion Complete
            reti                        ;EEPROM Ready
            reti                        ;Analog Comparator
;----------------------------------------------------------------------
; Start, Power ON, Reset
main:       ldi     r16 ,   lo8 (RAMEND)
            out     SPL ,   r16         ;Init Stackpointer LO
            ldi     r16 ,   hi8 (RAMEND)
            out     SPH ,   r16         ;Init Stackpointer Hi
            cbi     DDRD,   2           ;Port D2 Taster = IN
            sbi     PORTD,  2           ;PullUp
            sbi     DDRB,   0           ;Port B.0 = LED OUT
            ldi     r16,    0b01000000  ;Maskiere INT0
            out     GICR,   r16
            ldi     r16,    0b00000010  ;Konfiguriere
            out     MCUCR,  r16         ;fallende Fanke als Auslöser
            sei                         ;erlaube Interrupts
;......................................................................
mainloop:   wdr
            rjmp  mainloop
;======================================================================
;hier Unterprogramme und Interruptroutinen zufügen
;......................................................................
EXT_0:       cli
             sbi    PORTB, 0            ;Port B.0 = 1
             sei
             reti                       ;Rückkehr zum Hauptprogramm
;----------------------------------------------------------------------
```

Abschließend wird mit „sei“ die generelle Interrupt-Freigabe erteilt.

Die ISR finden wir ganz am Schluss unter „EXT_0“: Nach der Sperrung anderer Interrupts wird über Port B 0 die LED eingeschaltet, danach werden die Interrupts freigegeben, und der „reti“-Befehl führt wieder zurück ins Hauptprogramm.

Nun das Ganze übersetzen und auf den ATmega auf dem myAVR-Board übertragen, die beiden beteiligten Ports entsprechend verdrahten (Abbildung 34) und das Programm ausprobieren!

Wenn die LED nach dem Drücken des Tasters aufleuchtet, haben Sie soeben Ihr erstes Programm mit Interrupt-Steuerung erfolgreich getestet! Vorteil: Sie können das Ganze durch ein eigentliches Hauptprogramm ergänzen. Während dieses läuft, löst die Betätigung des Tasters einen Interrupt aus, nach dessen Abarbeitung (die LED leuchtet) das Hauptprogramm weiterläuft. In unserem Beispiel wird ja nur die Interrupt-Routine abgearbeitet, und da keine weitere Reaktion als die auf die fallende Flanke des Eingangssignals definiert ist, bleibt die LED an, sooft man auch den Taster drückt. Erst durch einen RESET (Buchse „RESET“ kurzschließen) gelangt der Controller wieder in den Startzustand und die LED verlischt.

Dieser Interrupt-Service-Routine werden wir in der Folge noch öfter begegnen, u. a. schon im nächsten Teil, wenn es um die Nutzung der integrierten Timer geht – wir entlocken dabei dem AVR Töne! ELV

Bild 35: Unser Beispielprogramm für die Nutzung von INT0 als Interrupt-Quelle