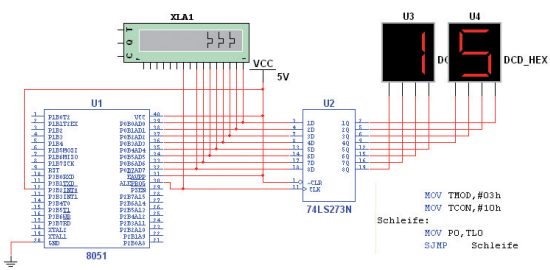
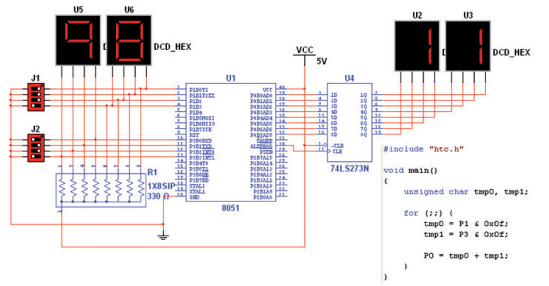
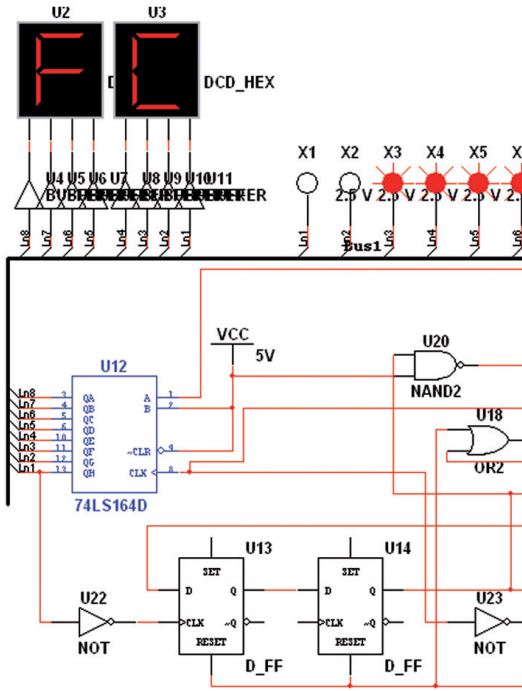
FRANZIS
PC & ELEKTRONIK

Herbert Bernstein







Mikrocontroller in der Elektronik

Mikrocontroller programmieren und in der Praxis einsetzen

Auf CD-ROM:

Alle Programme für ATtiny2313 und ATtiny26

Vorwort

ATMEL hat mehrere leistungsfähige 8-Bit- und 32-Bit-Mikrocontroller in seinem Programm. Für das Werkbuch der Mikrocontroller wurden der ATtiny2313 und der ATtiny26 verwendet. Beide Mikrocontroller verwenden für die Speicherung des Programms einen 2-Kbyte-Flash-Speicher, der über eine „In-System-Programmierung“ (3-Draht-ISP-Schnittstelle) programmiert wird. Die Programmierung übernehmen diese Bausteine selbstständig durch eine interne „High Voltage“ Programmierung.

Mit dem bekannten „digitalen“ 8-Bit-Mikrocontroller ATtiny2313, einem Industriestandard in einem praktischen 20-poligen DIL-Gehäuse, steht für den Elektroniker ein kostengünstiger Mikrocontroller zur Verfügung. Mit diesem Baustein lassen sich zahlreiche Aufgaben aus dem Bereich der Hobbyelektronik und aus der Praxis lösen.

Der „analoge“ 8-Bit-Mikrocontroller ATtiny26, ebenfalls ein Industriestandard in einem praktischen 20-poligen DIL-Gehäuse, hat mehrere 10-Bit-AD-Wandler zur Verfügung. Mit dem AD-Wandler lassen sich zahlreiche Versuche durchführen.

Die Versuche mit dem ATtiny2313 und ATtiny26 sind einfach auf einer Lochrasterplatine aufgebaut und man benötigt für die einzelnen Aufbauten etwa eine Stunde Lötzeit. Die Bauelemente erhalten Sie im Versandhandel.

Die Versuche werden alle im AVR-Assembler programmiert. Entwickler, Studenten und Schüler können auf komfortable Weise die ATMEL-Software für ihre Anwendungen auf dem PC implementieren und testen. Die sehr leistungsfähige Entwicklungsumgebung mit der Bezeichnung „AVR-Studio“ bündelt alle benötigten Funktionen – vom Assembler bis zum Simulator.

Wie bekommt man nun das Programm in Assemblersprache in den Mikrocontroller? Man muss es übersetzen, entweder in Hexadezimal- oder in Binärzahlen. Vor 20 Jahren musste man ein Programm in Assemblersprachen manuell übersetzen, Befehl für Befehl, und dies bezeichnete man als Hand-Assemblierung. Wie im Fall der Umwandlung hexadezimal in binär, ist die Hand-Assemblierung eine Routine-Aufgabe, die uninteressant, sich wiederholend und anfällig für zahllose kleine Irrtümer ist. Verwenden der falschen Zeile, Vertauschen von Ziffern, Auslassen von Befehlen und falsches Lesen der Codes sind nur einige der Fehler, die einem unterlaufen können. Die meisten Mikrocontroller machten früher die Aufgaben noch komplizierter, indem sie verschiedene Befehle mit unterschiedlichen Wortlängen

verwendeten. Manche Befehle sind nur ein Wort lang, während andere eine Länge von zwei oder drei Worten besitzen. Einige Befehle benötigen Daten im zweiten und dritten Wort und andere benötigen Speicheradressen, Registernummern oder andere Informationen.

Die Assemblierung ist eine weitere routinemäßige Aufgabe, die wir einem Mikrocontroller überlassen können. Der Mikrocontroller macht niemals Fehler beim Übersetzen von Codes. Er weiß immer, wieviele Worte und welches Format jeder Befehl benötigt. Ein Programm, das eine derartige Aufgabe ausführt, wird Assembler genannt. Das Assemblerprogramm übersetzt ein Anwenderprogramm oder Quellprogramm, das mit Mnemonics geschrieben wurde, in ein Programm in Maschinensprache (oder Objektprogramm), das der Mikrocontroller ausführen kann. Die Eingabe für den Assembler ist ein Quellprogramm und er gibt ein Objektprogramm aus.

Assembler besitzen eigene Regeln, die man erlernen muss. Diese enthalten die Verwendung bestimmter Markierungen oder Kennzeichen (wie Zwischenräume, Kommata, Strichpunkte oder Doppelpunkte) an den entsprechenden Stellen, korrekte Aussprache, die richtige Steuer-Information und vielleicht auch die ordnungsgemäße Platzierung von Namen und Zahlen. Diese Regeln stellen nur ein kleines Hindernis dar, das man leicht überwinden kann.

Ferner muss man beim Programmieren in Assemblersprache eine sehr detaillierte Kenntnis des verwendeten Mikrocontrollers besitzen. Man muss wissen, welche Register und Befehle der Mikrocontroller hat, wie die Befehle die verschiedenen Register genau beeinflussen, welche Adressierverfahren der Mikrocontroller verwendet und eine Unmenge weiterer Informationen. Keine dieser Informationen ist für die Aufgabe, die der Mikrocontroller letztlich ausführen muss, relevant.

Ich bedanke mich bei meinen Studenten für die vielen Fragen, die viel zu einer besonders eingehenden Darstellung wichtiger und schwieriger Fragen beigetragen haben. Meiner Frau Brigitte danke ich für die Erstellung der Zeichnungen, Herrn Gerhard Zwilling und der Firma ATMEL für die Unterstützung.

Inhalt

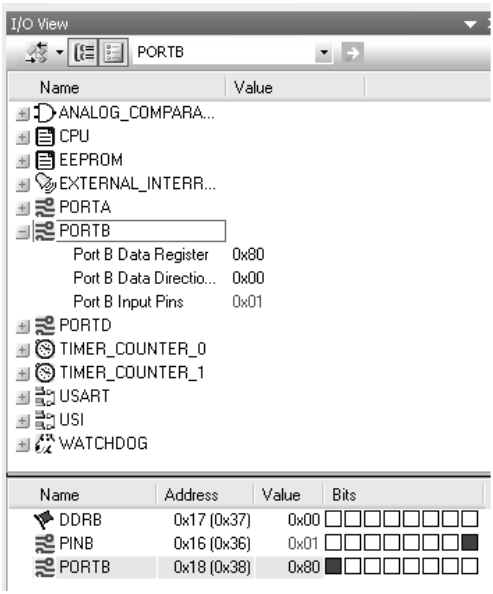


Abb. 1.25: Übersicht der I/O-Funktionen

1.4 Blinker mit interner Timerfunktion

Sie wollen den Assemblerkopf vom Programm „test2“ in das Programm „test3“ übernehmen. Wenn die Einstellungen von *Abb. 1.21* durchgeführt worden sind, klicken Sie „Minimieren“ an. Dann klicken Sie „Windows-Explorer“ an, klicken „Arbeitsplatz“ an und danach die Festplatte, wo Sie „Atmel Studio“ abgespeichert haben. Es erscheint ein Ordner mit „atmel2313“ und in dem Ordner sind zwei Ordner mit „test2“ und „test3“ vorhanden. Klicken Sie den Ordner „test2“ an und markieren Sie „test3.asm“.

In dem Programm soll der ATtiny2313 nur blinken, d. h. es ist keine Peripherie erforderlich, sondern nur eine Leuchtdiode am Anschluss PB7. *Abb. 1.26* zeigt das Programm.

Für das Programm ist wieder der gesamte Programmkopf erforderlich. Das Sperren des Interrupts ist am Anfang von *Abb. 1.26* gezeigt und das Statusregister wird durch den Befehl „in itmp, SREG“ in das Register 25 geladen. Danach ist mit dem Befehl „push itmp“ der Rücksprung aus einer Subroutine gewährleistet. Mit „reload Timer0“ werden die Daten zurückgeladen und mit „ldi itmp, CNTVAL“ wird der Zählerbereich geladen. Mit „out TCNT0“ wird das Register 25 auf den Ausgang zurückgeladen. Danach wird der Zählerwert von „TICKS“ geladen und um +1 in

```
; used Interrupts
TIM0_OVF:
    ; save StatusRegister
    in itmp,SREG
    push itmp

    ; reload Timer0
    ldi itmp,CNTVAL
    out TCNT0,itmp
    lds itmp, TICKS
    inc itmp
    sts TICKS, itmp
    cpi itmp,$0a
    brlo TIM0_OVF_DONE
    clr itmp
    sts TICKS,itmp
    sbr stavec,$01

TIM0_OVF_DONE:
    ; restore StatusRegister
    pop itmp
    out SREG,itmp
    reti


; Initialize
RESET:
    ; set stackpointer to end off RAM
    ldi acc0, low(RAMEND)
    out SPL,acc0

    ; set PortA to input with Pullup
    ldi acc0, $ff
    out PORTA, acc0
    ldi acc0, $00
    out DDRA, acc0

    ; set PortB to input with Pullup
    ; set PortB(7) to output with value 1
    ldi acc0, $ff
    out PORTB, acc0
    ldi acc0, $80
    out DDRB, acc0

    ; set PortD to input with Pullup
    ldi acc0, $ff
    out PORTD, acc0
    ldi acc0, $00
    out DDRD, acc0

    ; Timer0 normal mode with CK/256 (256us)
    ldi acc0, $04
    out TCCR0B, acc0

    ; Timer0 Overflow after $100 - $3D = $C3 = 195 (*256us) = (50ms)
    ldi acc0, CNTVAL
    out TCNT0, acc0

    ; Timer0 Overflow Interrupt enable
    ldi acc0, $02
    out TIMSK, acc0

    ; no button pressed
    ldi stavec, $00
    clr acc0
    sts TICKS, acc0

    sei

; mainprogram
main:
    sbrs stavec, 0
    rjmp main

    ; 500 ms interrupt occured
    cbr stavec,$01
    ; toggle PORTB(7)
    in acc0, PORTB
    ldi acc1, $80
    eor acc0, acc1
    out PORTB, acc0
    rjmp main
```

Abb. 1.26 (test2): Programm für einen Blinker mit der internen Timerfunktion

„inc itmp“ erhöht. Dann folgt die Speicherung im Schreib-Lese-Speicher mit „sts TICKS, itmp“ und der Vergleich mit „cpi itmp, $0A“, also mit dem Wert 10. Mit „brlo TIM0_OVF-DONE“ wird eine Verzweigung eingeschaltet. Ist die Bedingung erfüllt, wird das Programm auf „TIM0_OVF_DONE“ fortgesetzt. Andernfalls wird mit „sts TICKS, itmp“ der Wert im Register TICKS geladen und dann mit dem Statusvektor gesetzt.

Mit dem Programm „TIM0_OVF_DONE“ wird das Statusregister zurückgespeichert. Hier befindet sich der POP-Befehl. In das spezifizierte 8-Bit-Register „itmp“ oder Register 25 wird der Inhalt vom Register, das durch den Stackpointer adressiert wird, geladen. Bei einem 16-Bit-Register wird das Datenbyte mit der Adresse, das gleich dem Inhalt des Stackpointers ist, geladen. Das Datenbyte mit der Adresse <Stackpointer + 1> wird in das erste Register des Paares geladen. Wird das Registerpaar PSW spezifiziert, ersetzt das Byte, das durch den Stackpointer adressiert wird, die Bedienungsbits und das Byte, das durch den Stackpointer + 1 adressiert wird, den Akkumulator.

Mit dem PUSH-Befehl erfolgt die Datenübertragung in den Stack. Der Inhalt des spezifizierten 8-Bit-Registers oder des Registerpaars (16 Bit) wird in einem oder

zwei Speicherbytes sichergestellt, die durch den Stackpointer SP adressiert werden. Der Inhalt des ersten Registers wird in dem Byte mit der Adresse Stackpointer –1 abgespeichert. Arbeitet man im 16-Bit-Format, wird der Inhalt des zweiten Registers in dem Byte mit der Adresse –2 gespeichert. Wird das Registerpaar PSW spezifiziert, dann enthält das zweite Byte die acht Bedingungsbits. In diesem Fall wird der Stackpointer um 2 verringert, nachdem die Daten sichergestellt wurden.

Die Initialisierung beginnt mit dem Setzen des Stackpointers auf das Ende vom RAM mit „out SPL, acc0“, d. h. der Stackpointer SPL wird auf „low“ gesetzt.

Jetzt kann PortA gesperrt werden, während PortB auf seine Funktionen vorbereitet wird. Das Gleiche gilt auch für PortD.

Anschließend beginnt das Setzen des Timers 0 und zwar in „normaler Betriebsart“ und der intern erzeugte Takt von 1 MHz wird um 256 geteilt. Das Ergebnis ist 256 µs. Das Timerregister TCC0B wird auf „$04“ gesetzt. Danach wird weitergeteilt. Auf diese Weise leuchtet die LED für ca. 500 ms auf und ist dann ca. 500 ms dunkel.

Statt I/O-Port PB wird in *Abb. 1.27* eine Übersicht für den Timer_Counter 0 und 1 gezeigt. Jetzt sieht man den Unterschied zwischen den drei Einstellungen:

☐ Das leere Quadrat bedeutet, dass die Bitstelle ein 0-Signal hat.
▒ Das graue Quadrat zeigt, dass die Bitstelle nicht belegt ist.
■ Das schwarze Quadrat bedeutet, dass die Bitstelle ein 1-Signal hat.

Betrachtet man das TCCR0A-Register und vergleicht es mit Tabelle 1.4, erkennt man, dass Bitstelle 2 und 3 nicht belegt sind. Betrachtet man das TCCR0B-Register und vergleicht es mit Tabelle 1.4, erkennt man, dass die Bitstelle 4 und 5 nicht belegt sind, aber Bitstelle 2 gesetzt ist. Bei dem TCNT0-Register sind die Bitstellen 0, 2, 3, 4 und 5 gesetzt. Bei dem TIMSK-Register ist die Bitstelle 1 gesetzt.

Wird der Debugger im Einzelschritt weitergeschaltet, ändern sich die Bitstellen entsprechend. Der Projektleiter wird in dieser Phase bereits von den Entwicklern unterstützt. Es wird für Hardware (Platine oder Baugruppen) und Software (Programm in Assembler oder C) getrennt festgelegt, wie die einzelnen Funktionen des geplanten Produktes realisiert werden sollen. Bei der Software geschieht dies dadurch, dass mithilfe von Struktogrammen diverse Module definiert werden, die die einzelnen Aufgaben lösen. Dadurch werden die Gesamtaufgaben automatisch in einzelne, überschaubare Teilarbeiten untergliedert. Abhängig von der Größe des Softwareprojektes, teilt der Projektleiter die Aufgaben auf eine mehr oder weniger große Zahl von Entwicklern auf. Die Entwicklungswerkzeuge, die in dieser Phase den Projektleiter und die Entwickler unterstützen, sind ebenso wie in der Defini-

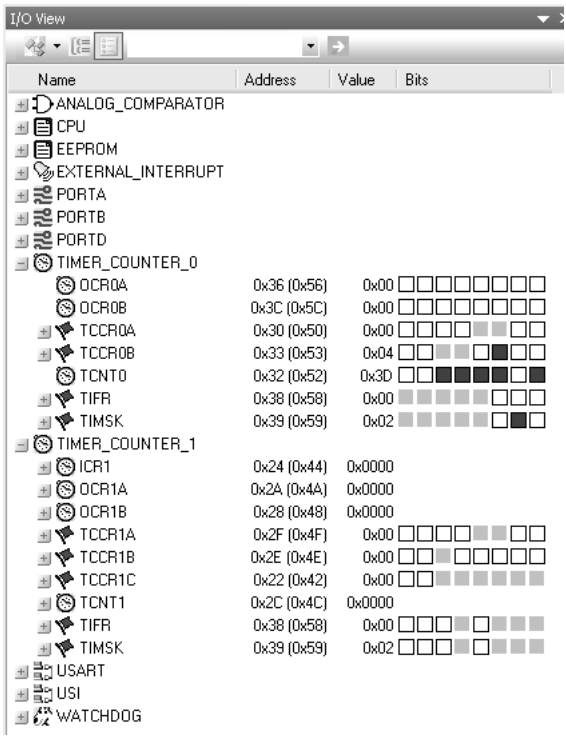


Abb. 1.27: Übersicht für den Timer_Counter_0 und 1

tionsphase die Software „Engineering Tools", z. B. unter Zuhilfenahme von Atmel Studio möglich.

Bei der Hardware werden die Bausteine und Komponenten ausgewählt, die die geforderten Funktionen ermöglichen und später die Software aufnehmen sollen. Es wird die Größe der Leiterplatten festgelegt und eventuell werden auch die technischen Zeichnungen für das Gehäuse, die Frontplatte, die Rückwand usw. erstellt.

In der Realisierungsphase werden, aufbauend auf den Ergebnissen der Planungsphase, Hard- und Software getrennt entwickelt. Auf der Softwareseite bedeutet dies, dass die geplanten Softwaremodule mithilfe geeigneter Hochsprachen oder aber auch mithilfe der Assemblersprache codiert (programmiert) werden. Dazu werden die in der Planungsphase erstellten verbalen Beschreibungen der einzelnen Module sukzessive durch Hochsprachen oder Assemblerquelltext ersetzt. Es folgen Compiler- und Assemblerläufe, um die so erstellten Programmteile testen zu können. Die Tests selbst werden dann mithilfe von Softwaresimulatoren oder mit Emulatoren durchgeführt. Meist sind nach den Tests Korrekturen an den einzelnen Modulen nötig. Diese Änderungen müssen natürlich am Quelltext durchgeführt werden.

Wenn von den Änderungen auch die Dokumente der Planungsphase betroffen sind, ist es erforderlich, auch an diesen Dokumenten die notwendigen Korrekturen durchzuführen. Wie *Abb. 1.22* zeigt, kann es erforderlich werden, bestimmte Softwaremodule sukzessive so lange zu korrigieren, bis die gewünschte Funktion erreicht wird. Mit fortschreitender Softwarevervollständigung werden zunächst die Softwareteile eines Entwicklers und später dann aller Entwickler zusammengefügt. Es folgen wiederum umfangreiche Funktionstests. Natürlich findet man auch bei diesen Tests häufig Fehler, die korrigiert werden müssen.

In der Realisierungsphase von Software benötigt der Entwickler die Editoren, Cross-Assembler und Cross-Compiler. Tests führt er mit Softwaresimulatoren oder mit Emulatoren, die im Simulationsmode betrieben werden, durch.

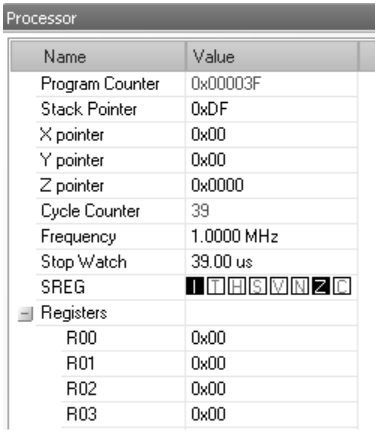


Abb. 1.28: Übersicht des Prozessors für den ATtiny2313

Abb. 1.28 zeigt eine Übersicht des Prozessors für den ATtiny2313. In der ersten Zeile wird der Wert im Programmzähler PC abgefragt. Es handelt sich um den Zählerstand 0x00003F. Der Stackpointer hat den Zählerstand 0xDF. Der X-, Y- und Z-Pointer ist nicht gesetzt. Der Zykluszähler hat den Wert 39. Die Frequenz beträgt 1,0000 MHz. Die Stoppuhr zeigt 39 µs an und beim Statusregister ist die Bitstelle 7 mit dem I-Flag und Bitstelle 1 mit dem Z-Flag gesetzt. Dann folgen die Inhalte der Register R00 bis Rx31. Die folgenden Register sind gesetzt:

R16: 0xFF
R17: 0x80
R25: 0x14

Hierzu hat man folgende Möglichkeiten für den Debugger:

- RUN (F5): Der Debugger erhöht schrittweise sein Programm und dient dem Test bzw. der Korrektur von assemblierten Programmen.

- BREAK (Strg + F5): Mit BREAK wird das Programm abgebrochen und dann wieder mit RUN gestartet.
- RESET (Umschalter + F5): Der Debugger wird zurückgesetzt.
- Show Next Statement (Alt + (Zehnerblock)): Pfeil zeigt auf die nächste Anweisung.
- Step Into (F11): Schrittweise wird der Debugger hochgezählt und der Pfeil zeigt auf die nächste Anweisung.
- Step Over (F10): Schrittweise Abarbeitung des Debuggers und der Pfeil zeigt auf die nächste Anweisung.
- Step Out (Umschalter + F11): Schrittweise Abarbeitung des Debuggers bis zur nächsten Pfadfunktion.
- Run to Cursor (Strg + F10): Abarbeitung des Programms bis zum Cursor.
- Autostep (Alt + F5): Automatische Abarbeitung des Programms.
- Toggle Breakpoint (F9): Setzen und Rücksetzen einer Unterbrechung.
- Remove all Program Breakpoint: Rücksetzen aller Unterbrechungen im Programm.

Mit diesen Funktionen können Sie das Debuggern durchführen.

1.5 Blinker mit Timerfunktion

Mit der Taste an Pin PB0 kann man den Blinker starten oder unterbrechen. Normalerweise liegt an Pin PB0 ein 1-Signal und mit „not_pressed“ wird dieser Zustand ausgewertet. Drückt man den Taster, erkennt der Eingang ein 0-Signal an und der Blinker gibt für etwa 500 ms am Ausgang von Pin PB7 ein 0-Signal ab. Bei einem 1-Signal an Pin PB7 ist die Funktion für etwa 500 ms auf 1-Signal und die Leuchtdiode kann nicht mehr emittieren. *Abb. 1.29* zeigt das Programm.

```
; mainprogram
main:
    sbic PINB,0
    rjmp not_pressed

    sbrs stavec, 0
    rjmp main

    ; 500 ms interrupt occured
    cbr stavec,$01
    ; toggle PORTB(7)
    in acc0, PORTB
    ldi acc1, $80
    eor acc0, acc1
    out PORTB, acc0
    rjmp main

not_pressed:
    sbi PORTB, 7
    rjmp main
```

Abb. 1.29: Programm für den Blinker mit Timerfunktion

Das Programm soll auf seine Funktionalität untersucht werden. Rechts neben dem Button „Assembler“ befindet sich der Button „Assembler and run“. Wenn Sie diesen anklicken, erscheint *Abb. 1.30*.

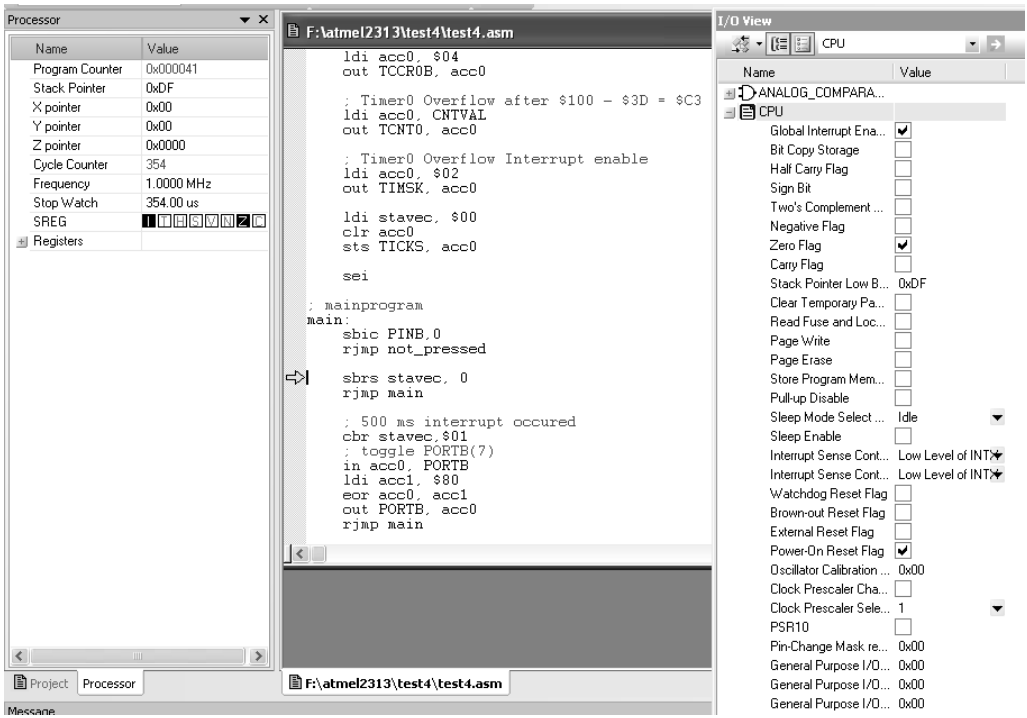


Abb. 1.30: Assembliertes Programm für den Blinker mit Timerfunktion

Programme, die in Assemblersprache geschrieben wurden, sind in dieser Form nicht direkt vom Mikrocontroller zu verarbeiten. Die Assemblerprogramme müssen erst mithilfe eines geeigneten Übersetzungsprogramms in maschinenverständliche Form (sogenannten Code) übersetzt werden. Das Übersetzungsprogramm bezeichnet man allgemein als Assembler. Der Begriff kommt aus dem angelsächsischen Sprachgebrauch und hat die Bedeutung „Zusammenbau“. Abhängig von der Art des Rechners, auf dem das Assemblerübersetzungsprogramm implementiert ist, unterscheidet man zwischen dem Assembler und dem Cross-Assembler. Der Assembler ist die Version des Übersetzungsprogramms, die auf einem Rechner implementiert ist, der mit dem gleichen Mikrocontroller arbeitet, für den auch der Code generiert werden soll. Der Cross-Assembler hingegen ist die Version, die auf einem Rechner implementiert ist, die nicht mit dem Mikrocontroller ausgestattet ist, für den der Code generiert werden soll. Da Assembler heute vorwiegend auf PCs

mit den Entwicklungsprogrammen und speziellen Mikrocontroller-Entwicklungssystemen implementiert sind, hat man es vorwiegend mit Cross-Assemblern zu tun. Lediglich in den Fällen, in denen ein mit dem gleichen Prozessor ausgerüstetes Entwicklungskit zur Assemblierung benutzt wird, hat man es noch mit einem Assembler zu tun. Die Unterscheidung Cross-Assembler bzw. Assembler hat daher in der heutigen Zeit nur noch geringe Bedeutung. Da zwischen Assembler und Cross-Assembler keine funktionellen Unterschiede bestehen, wird im Folgenden nur noch vom Cross-Assembler gesprochen.

In den seltensten Fällen ist der Cross-Assembler nur ein einfaches Übersetzungsprogramm. Meist wird der Übersetzungsprozess der mnemotechnischen Mikrocontrollercodes und Pseudoanweisungen in mehreren Stufen durchgeführt. Die klassische funktionale Untergliederung ist Assembler, Linker und Locater. Der Assembler kann einzelne Dateien eines umfangreicheren Softwareprojektes (das Hauptprogramm und einzelne Unterprogramme) getrennt verarbeiten. Da zwischen den einzelnen Programmteilen diverse Querverweise existieren (z. B. Sprünge zu Programmlabels in anderen Modulen oder die Benutzung von Variablen, die in anderen Programmteilen deklariert wurden) und auch die Adressvergabe des zu assemblierenden Programmteils zu diesem Zeitpunkt noch nicht feststeht, ist es einleuchtend, dass im Assemblerlauf nur sehr unvollständig übersetzt werden kann. Nur die mnemotechnischen Befehle, die keinen Bezug zu einer externen Speicherstelle aufweisen, werden vollständig übersetzt. Bei absoluten Sprüngen zu einer externen Adresse hingegen wird nur der Befehlscode selbst, nicht aber das Sprungziel übersetzt. Der gesamte vom Assembler erzeugte Programmcode bleibt frei verschiebbar (relocatibel) und wird für jedes Modul fiktiv bei Adresse Null beginnend abgelegt. Die Assemblerphase erfordert in der Regel drei Durchläufe, um alle relevanten Anweisungen und Sprungziele zu verarbeiten.

Die Linkerstufe hat die Aufgabe, das Hauptprogramm und alle zu dem Projekt gehörenden Unterprogramme zusammenzubinden. Beim Aufruf der Linkerstufe ist es also erforderlich, alle Dateien anzugeben, die zum Entwicklungsprojekt gehören. Alle aus der Assemblerstufe noch unbefriedigt gebliebenen Sprungziele und Referenzen müssen also spätestens in der Linkerstufe ordnungsgemäß übersetzt werden. Sollten aufgrund von Programmierfehlern trotzdem noch unbefriedigte Referenzen auftreten, werden diese beim Linkprozess als fehlend gemeldet. Der Anwender wird also spätestens an dieser Stelle auf seine Programmierfehler hingewiesen. Eine absolute Adressangabe findet im Linkprozess noch nicht statt. Diese ist Aufgabe des „Locatorprozesses“.

Der Locatorprozess nimmt schließlich die endgültige Adressvergabe vor. Hierzu werden entweder die hierfür vorgesehenen Pseudoinstruktionen (ORG bzw. .base) des Assemblerquelltextes ausgewertet oder die speziell bei der Linkerphase anzuge-

benden Sektionsstartadressen verwendet. Die Linkerstufe unterscheidet in der Regel zwischen dem Codesegment (für das Programm) und dem Datensegment. Der gesamte Programmtext des Hauptprogramms und aller Unterprogramme wird in der Reihenfolge aneinandergereiht, wie dies beim Linkervorgang angegeben wurde. In der gleichen Art werden auch die Datenbereiche des Hauptprogramms und aller Unterprogramme aneinandergereiht. Ergebnis der Locatorstufe ist schließlich ein direkt vom Mikrocontroller abarbeitbares Programm. Es wird entweder zur Integration von Hard- und Software in den Echtzeitemulator geladen oder nach erfolgtem Debugging mit in das EEPROM des Mikrocontrollers programmiert. Da die Adressvergabe in der Locatorstufe keine sonderlich umfangreichen Operationen erfordert, wird die Locator- und Linkerstufe bei einigen Cross-Assemblern zu einer kombinierten Linkerstufe zusammengefasst.

Die Vorteile der Unterteilung des Cross-Assemblers in drei bzw. zwei aufeinanderfolgende Übersetzungsstufen werden deutlich, wenn man große Softwareentwicklungsprojekte betrachtet. Bei solchen Projekten ist es üblich, das gesamte Programm zu unterteilen und in vielen Dateien auf dem Rechner oder Entwicklungssystem abzulegen. Das ablauffähige Mikrocontrollerprogramm entsteht durch das Assemblieren und Zusammenbinden all dieser Dateien. Muss man während des Entwicklungsprozesses Änderungen an einem Unterprogramm vornehmen, so müsste bei einem einstufigen Assemblierungsprogramm der Übersetzungsprozess nach dem Ändern des einen Unterprogramms auf alle Unterprogramme und damit auf alle Dateien des Softwareprojektes ausgedehnt werden. Unglücklicherweise gehören Änderungen der beschriebenen Art zum täglichen Arbeitsgeschehen des Softwareentwicklers. Dies macht also umfangreiche Assemblerläufe nötig. Verwendet man hingegen einen Cross-Assembler mit getrennter Assembler-Linkerstufe, muss nur die eine bearbeitete Datei neu assembliert werden. In dem nachfolgenden Linkerlauf wird dann diese neu assemblierte Datei mit den anderen, unveränderten Dateien zusammengebunden. Das Gesamtprogramm wird mit wesentlich geringerem Aufwand an Rechenzeit auf den neuesten Stand gebracht.

Trotz der Vorteile, die mehrstufige Cross-Assembler bieten, gibt es auch Gründe, die einen einstufigen Cross-Assembler rechtfertigen. Hierzu gehören die Einfachheit solcher Übersetzungsprogramme und der damit günstige Anschaffungspreis. Außerdem sind bei kleineren Softwareprojekten die einstufigen Assembler im Zeitbedarf für die Übersetzung den mehrstufigen Cross-Assemblern überlegen. Einige Hersteller von Entwicklungstools liefern daher bei Bestellung eines Cross-Assemblers sowohl eine kombinierte als auch eine in Assembler und Linker unterteilte Version.

Ein besonderer Aspekt bei der Auswahl eines Cross-Assemblers ist seine Fähigkeit, Informationen, die für die Fehlersuche mit dem Emulator nötig sind, zur Verfügung

zu stellen. Zu diesen Informationen gehören die im Assemblerprogramm definierten Labels und natürlich ein auf Papier ausdruckbares Listing des Programms mit dem zugehörigen, vom Cross-Assembler generierten Code. Falls das Assemblerprogramm durch einen Compilerlauf aus einem Hochsprachenprogramm entstanden ist, müssen für das Hochsprachen-Debugging auch Informationen über die korrespondierenden Hochsprachenzeilen, über Variable des Hochsprachenprogramms, über die Namen der Unterprogramme usw. zur Verfügung gestellt werden. Interessant bei der Debugphase ist weiterhin eine sogenannte Symboltabelle. Hierunter versteht man eine Tabelle aller Labels eines Programms, mit den von der Linkerstufe zugewiesenen Adressen.

Sie starten mit der Funktionstaste F5 oder dem Button „Run“ und nach und nach können Sie die Struktur des Programmablaufs prüfen. Links in *Abb. 1.30* ist das Innenleben des Prozessors mit einer Frequenz von 1 MHz gezeigt. Sie können mit dem Button „Break“ den Ablauf der Schrittfolge unterbrechen und mit dem Button „Reset“ alles auf die Adresse 0 stellen. Rechts sehen Sie die CPU des Mikrocontrollers.

1.6 Taster mit Einschaltverzögerung

Für die Steuerungstechnik benötigt man Taster mit Einschaltverzögerung. Wenn man die Schaltung von *Abb. 1.30* erweitert, z. B. um ein 4-poliges Mäuseklavier, lassen sich 16 Verzögerungszeiten erzeugen. *Abb. 1.31* zeigt das Programm für eine Verzögerungszeit.

```
; mainprogram
main:
    sbic PINB,0
    rjmp not_pressed

    ; Timer0 normal mode with CK/256 (256us)
    ldi acc0, $04
    out TCCR0B, acc0

    sbrs stavec, 0
    rjmp main

    ; 1 sec interrupt occured
    cbr stavec,$01

    ; LED on
    cbi PORTB, 7
    rjmp main

not_pressed:
    ; Timer0  stopped
    clr acc0
    out TCCR0B, acc0

    ; Timer0 Overflow after $100 - $3D = $C3 = 195 (*256us) = (50ms)
    ldi acc0, CNTVAL
    out TCNT0, acc0

    cbr stavec,$01
    clr acc0
    sts TICKS, acc0

    ; LED off
    sbi PORTB, 7
    rjmp main
```

Abb. 1.31 (test5): Programm für einen Taster mit Einschaltverzögerung von einer Sekunde

Wird der Taster nicht gedrückt, hat Eingang PB0 ein 1-Signal und die Schaltung reagiert nicht. Betätigt man den Taster, leuchtet die LED nach einer Sekunde auf, vorausgesetzt, der Taster wird kontinuierlich gedrückt. Lässt man den Taster vor Ablauf der Verzögerungszeit los, kann die LED nicht aufleuchten.

Sie wollen den Assemblerkopf von Programm „test4“ in das Programm „test5“ übernehmen. Wenn die Einstellungen durchgeführt worden sind, klicken Sie „Minimieren“ an. Dann klicken Sie den Windows-Explorer an, klicken den Arbeitsplatz an und danach die Festplatte, wo Sie „Atmel Studio“ abgespeichert haben. Es erscheint ein Ordner mit „atmel2313“ und in dem Ordner sind zwei Ordner mit „test4“ und „test5“ vorhanden. Klicken Sie den Ordner „test4“ an und markieren Sie „test5.asm“.

Das Hauptprogramm setzt zuerst das I/O-Register mit „sbic PINB, 0“ und dann erfolgt der Rücksprung mit „rmp not_pressed“. Solange diese Bedingung gilt, kann das Programm nicht arbeiten und die Schleife verlassen.

Ist das der Fall, wird der Befehl „ldi acc0, $04“ ausgeführt. Der Akkumulator hat den Wert

0 0 0 0 0 1 0 0

geladen. Mit dem Befehl „out TCCR0B“ erfolgt das Laden des Zeitgebers. Die Frequenz des Mikrocontrollers wird von 1 MHz auf CLK/256 geteilt und hat eine Frequenz von 3,906 kHz. Anschließend wird mit „cbr stavec, $01“ im Statusregister die Bitstelle 1 gelöscht. Dann wird noch mit dem Befehl „cbi PORTB, 7“ der Wert mit einer Konstanten verglichen.

Der Programmteil „not_pressed“ stoppt den Timer 0. Mit dem Befehl „clr acc0“ wird das Register bzw. der Akkumulator gelöscht. Danach kommt der Befehl „out TCCR0B, acc0“ und der Inhalt des Akkumulators wird in das Timer-Register übernommen.

Multipliziert man 195 * 256 µs, erhält man 50 ms und damit wird der Zähler TCNT0 geladen. Vom Register für den Statusvektor wird die Bitstelle 1 und anschließend der Akkumulator gelöscht. Dann wird der Akkumulator unter „TICKS“ direkt mit „sts“ gespeichert. Zum Schluss wird im I/O-Register das Bit gesetzt.

Ein Mikrocontrollerprogramm, bei dem der Programmzähler unbeeinflusst stetig wächst, bezeichnet man als lineares Programm. Verzweigungen und Sprünge kommen in einem linearen Programm also nicht vor. Die meisten praktischen Anwendungsfälle sind allerdings so geartet, dass ein lineares Programm die Aufgabe kaum lösen kann. Erst das Einführen von Schleifen und Programmverzweigungen führt

zur Lösung der Aufgabe. Auf den Mikrocontroller projiziert bedeutet dies, dass der Programmzähler nicht nur linear wachsen darf, sondern abhängig von bestimmten Bedingungen zurück- oder vorgesetzt werden muss. Bei allen diesen Fällen ist es beispielsweise erforderlich, einfache Additionen oder Subtraktionen mit dem Inhalt des Programmzählers durchzuführen bzw. den Programmzähler eventuell auch mit einem neuen Wert zu laden.

Auch für Programmverzweigungen und Schleifen in einem Mikrocontrollerprogramm gibt es spezielle Befehle. Programmverzweigungen werden mithilfe sogenannter Branchanweisungen realisiert. Wenn der Inhalt des Akkumulators oder eines vom Anwender definierbaren Registers den Wert Null annimmt, wird die Verzweigung eingeleitet (Branch on equal). Auf Wunsch kann auch verzweigt werden, wenn der Inhalt verschieden von Null ist (Branch not equal). Die Entscheidung, ob der Akkumulator bzw. das Register den Wert Null enthält, leitet der Mikrocontroller vom Statusregister ab. Grundsätzlich werden alle Verzweigungsbefehle geprüft und mit dem Inhalt des Statusregisters verglichen. Auch für die anderen Flags des Statusregisters existieren spezielle Verzweigungsbefehle. Durch Kombination mehrerer dieser Befehle kann bei nahezu jeder denkbaren Konstellation im Inneren des Mikrocontrollers eine Programmverzweigung realisiert werden.

Programmschleifen werden beim Mikrocontroller durch eine Kombination aus einer Branchanweisung und einem Register, das als Schleifenzähler genutzt wird, realisiert. Das Register wird einfach mit der Zahl der gewünschten Schleifendurchläufe vorbesetzt. Es folgt der Programmteil, der bei jedem Schleifendurchlauf ausgeführt werden soll. Abgeschlossen wird die Programmschleife durch einen Dekrementbefehl, der das als Schleifenzähler arbeitende Register herunterzählt. Als letztes folgt dann die Branchanweisung. Die Schleife wird solange durchlaufen, bis die Branchbedingung erfüllt ist und erst dann wird die Schleife verlassen.

Bei der bisherigen Betrachtungsweise wurde angenommen, dass der Mikrocontroller ausschließlich Befehle abarbeitet. Sinn und Zweck eines Mikrocontrollerprogramms ist es aber auch, dass Daten verarbeitet werden. Wie der Mikrocontroller mit diesen Daten verfährt und wie sie zwischen den einzelnen Funktionseinheiten transportiert werden, wird nachfolgend gezeigt. Es ist einleuchtend, dass sowohl Daten als auch Befehle über den Datenbus zwischen Mikrocontroller und Speicherbausteinen bzw. Peripheriebausteinen transportiert werden müssen. Die Verfahrensweise bei der Bearbeitung von Daten ist im Prinzip ähnlich wie bei der Verarbeitung von Befehlen.

Zunächst einmal wird der Befehl, der die Bearbeitung der Daten zur Folge haben soll, durch den Programmzähler adressiert und in den Befehlsdecoder geladen. Der Befehlsdecoder erkennt aus der Art des Befehls, ob dem Befehl Daten folgen und in

welcher Zahl die Daten folgen. Erkennt der Befehlsdecoder, dass Daten erforderlich sind, legt er die für den Datentransfer erforderliche Adresse auf den Adressbus und adressiert dadurch die gewünschte Speicherzelle. Der Datentransfer zwischen Speicher und Mikrocontroller kann mithilfe des Kontrollbusses in der üblichen Art stattfinden. Sind die Daten vollständig übertragen, wird mit dem nächsten Befehl fortgefahren. Hier bestimmt dann wieder der Programmzähler, welcher Befehl vom Programmspeicher geladen und dekodiert wird. Bei einem üblichen Mikrocontrollerprogramm wechseln Befehls- und Datenzugriffe einander in der Regel ab.

Die Quelle für Daten können in einem Mikrocontrollersystem der Daten- oder Programmspeicher, ein Peripheriebaustein, die ALU und auch interne Register des Mikrocontrollers sein. Alle Mikrocontroller stellen Befehle zur Übertragung bzw. Verarbeitung dieser Daten zur Verfügung. Sind in dem Befehl selbst bereits Angaben über Quelle und Ziel des Datentransfers enthalten, folgt kein weiteres Byte. In den Fällen, in denen keine Angaben über Quelle und Ziel im Befehlswort enthalten sind, folgen dem Befehlswort ein oder mehrere Bytes mit diesen Angaben.

Abhängig von der Art, wie die Speicherstelle adressiert werden soll, mit der der Datentransfer stattfindet bzw. deren Inhalt verarbeitet werden soll, unterscheidet man zwischen:

- der direkten Adressierung: Hier wird in den beiden Bytes, die dem Befehl folgen, die Adresse direkt angegeben, mit der der Datentransfer stattfinden soll.
- der Registeradressierung: Hier wird durch das Befehlsbyte selbst ein Register spezifiziert, mit dem der Datentransfer bzw. die Operation ausgeführt werden soll.
- der Registeradressierung indirekt: Bei dieser Adressierungsart wird durch das Befehlsbyte ein Registerpaar spezifiziert, dessen Inhalt auf eine Speicherzelle zeigt, mit der der Datentransfer stattfinden soll.
- der immediate Adressierung: Bei dieser Adressierungsart folgt dem Befehlsbyte direkt das Datenbyte bzw. das Datenwort, mit dem der Datentransfer stattfinden soll.

1.7 Taster mit Ein- und Ausschaltverzögerung

Mit dem Programm für den Taster mit Ein- und Ausschaltverzögerung soll ein professioneller ISP-Programmer, den man per Internet bestellen kann, kurz beschrieben werden.

Wesentlich einfacher und professioneller ist der Kauf einer fertigen USB-Schnittstelle (ISP-Programmer), die zwischen PC und dem 10-poligen Wannenstecker eingeschaltet wird. *Abb. 1.32* zeigt den ATtiny2313 an einer genormten Steckverbindung.

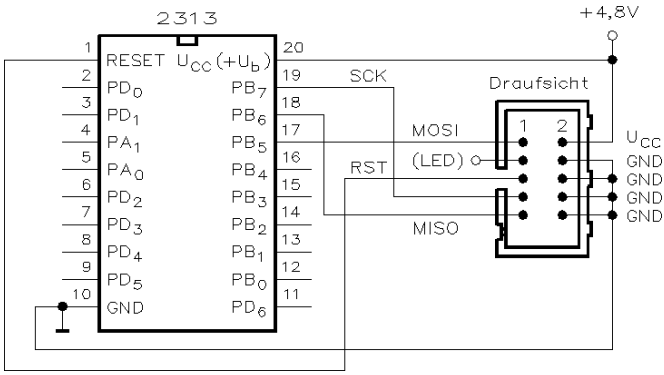


Abb. 1.32: ATtiny2313 an der genormten Steckverbindung, wo der USB-Programmer angeschlossen wird

Die Anschlüsse des Wannensteckers sind

Ebene 1 – Pin 1: Anschluss für MOSI
Ebene 1 – Pin 2: LED (ist nicht unbedingt erforderlich)
Ebene 1 – Pin 3: RESET
Ebene 1 – Pin 4: SCK
Ebene 1 – Pin 5: MISO
Ebene 2 – Pin 1: VCC (+4,8 V), ist nicht unbedingt erforderlich
Ebene 2 – Pin 2: Masseanschluss
Ebene 2 – Pin 3: Masseanschluss
Ebene 2 – Pin 4: Masseanschluss
Ebene 2 – Pin 5: Masseanschluss

Der ISP-Programmer ist für Atmel-Mikrocontroller geeignet und kann über die einfache Drei-Draht-SPI-Schnittstelle programmiert werden. Normalerweise sind diese Programmer mit einem eigenen Mikrocontroller und spezieller Firmware ausgestattet, die einen sehr schnellen Programmierzyklus erlauben. Der Programmer wird zwischen PC mit USB-Schnittstelle und dem 10-poligen Wannenstecker eingeschaltet. Die Besonderheit ist die dynamische Nutzung des auf dem Programmer vorhandenen Speichers, der bereits Teile vom PC übernehmen kann, während noch programmiert wird.

Über zwei DIL-Schalter ist eine Target-Spannungsversorgung (Zielsystem) möglich. Diese ist wahlweise auf +5 V oder +3,3 V einstellbar sowie zu- und ausschaltbar. Wenn mit dem Akkumulatorblock gearbeitet wird, ist die Stromversorgung vom USB-Programmer auf unserer Platine abzuschalten.

Zwei Status-Leuchtdioden signalisieren den momentanen Zustand des Programmers:

- USB-LED: Softblinken im Leerlauf oder Blinken bei USB-Aktivität
- Target-LED: Aus im Leerlauf oder Blinken bei Targetzugriffen

USB-Programmer sind beim SR-Tronic-Versand erhältlich.

Bitte laden Sie sich die „*inf-Datei“ für den Treiber des ISP-Programmers dort herunter bzw. unter www.obd-diag.de – downloads – Downloadcenter – Treiber – „Stange ISP Prog“ oder DX_ISP.inf für Vista oder WIN2000 auf das kleine Festplattensymbol klicken. Alternativ können Sie sich diese Datei auch per Mausklick über Internet zusenden lassen, die Anforderung ist an „spm@cybercyclone.de“ zu senden.

Die „*inf“-Datei auf Festplatte oder einem Suchpfad ablegen. Programmer USB-seitig anschließen und automatisch ausführen. Bei der Installation fragt Windows nun nach einem Treiber. Es wird der Windows-eigene Treiber verwendet, was dem System durch die spezielle „Stange_ISP_Prog.inf-Datei“ mitgeteilt wird. Dazu Häkchen setzen und „Weiter“ klicken. Im nächsten Dialog angeben, wo sich die „.inf-Datei“ befindet.

Die ISP-SPI-Geschwindigkeiten ergeben sich durch die USB-Anbindung und sie zeigt leicht abweichende ISP-Frequenzen (Tabelle 1.7) gegenüber einem STK500.

Tabelle 1.7: ISP-SPI-Geschwindigkeiten

STK500	ISP-Programmer
921,6 kHz	1 MHz
230,4 kHz	250 kHz
57,6 kHz	62,5 kHz
28,8 kHz	28,2 kHz
4 kHz	4 kHz
603 Hz	779 Hz

Der Zustand der beiden LEDs signalisiert:

- USB-LED: Softblinken im Leerlauf, Blinken bei USB-Aktivität
- Target-LED: Aus im Leerlauf, Blinken bei Targetzugriffen

Folgende Funktionen sind enthalten:

- FLASH, EEPROM, Fusebits, Lockbits schreiben, lesen, Chip erase, OSCCAL-Register lesen

Folgendes Protokoll ist vorhanden:

- STK500v2

Softwareunterstützung:

- AVR Studio (COM1...COM9)
- AVRDUDE
- Bascom mit der Einstellung Options → Programmer = STK500:

Spannungsversorgung der Versuchsplatine (Target):

- 3,3 V maximal mit 120 mA, 5 V maximal mit 150 mA bis 500 mA abhängig vom PC
  Schalter 1 off = 3,3V/on = 5,0V
  Schalter 2 off = Target-Spannung aus/on = ein
- Benötigt zum Programmbetrieb keine Stromversorgung vom Target

Nachdem Sie die USB-Schnittstelle für den ISP-Programmer auf der Platine angebracht und verlötet haben, können Sie mit der Erstellung des Programms beginnen.

Das Programm für den Taster hat zwei Funktionen:

- Wird der Taster länger als eine Sekunde gedrückt, schaltet der Mikrocontroller um und speichert den Schaltzustand. Es ergibt sich eine Einschaltverzögerung von einer Sekunde.
- Hat man den Taster beispielsweise länger gedrückt und lässt ihn los, bleibt der Schaltzustand noch für drei Sekunden an. Es ergibt sich eine Ausschaltverzögerung von drei Sekunden.

```
; mainprogram
main:
    sbrc button,0
    rjmp not_pressed

    sbrs stavec, 0
    rjmp chk_press_event

    ; 1 sec interrupt occured
    andi stavec, $f0

    ; LED on
    cbi PORTB, 7

chk_press_event:
    sbrc button,1
    ; start 1 sec interrupt
    ori stavec,$02
    rjmp main

not_pressed:
    sbrs stavec, 0
    rjmp chk_release_event

    ; 3 sec interrupt occured
    andi stavec, $f0

    ; LED off
    sbi PORTB, 7

chk_release_event:
    sbrs button,1
    ; start 3 sec interrupt
    ori stavec,$04
    rjmp main
```

Abb. 1.33 (test6): Programm für Taster mit Ein- und Ausschaltverzögerung

Abb. 1.33 zeigt das Programm für den Taster mit Ein- und Ausschaltverzögerung. Das Hauptprogramm hat am Anfang eine Tastenabfrage mit „sbrc button, 0“.

Solange der Taster 0 am Eingang PB0 nicht gedrückt ist, springt das Programm zurück. „sbrc“ steht für „skip if in Register is Cleared“, also springe, wenn das Register gelöscht ist. Danach kommt der Befehl „sbrs stavec, 0“ und hier steht „skip if in Register is Set“. In den Statusvektor wird ein Wert null geladen und dadurch erfolgt eine UND-Verknüpfung zwischen Register (Statusvektor) und „$f0“. Entsprechend wird der Ausgang PB7 für die Leuchtdiode mit dem Befehl „cbi“ (Clear Bit in I/O-Register) angesteuert. Die Einschaltverzögerung ist abgeschlossen, denn der Interrupt setzt den Zähler auf null zurück.

Mit „chk_press-event“ folgt das Programm für den Taster an PortB0. Ist der Taster gedrückt, hat der Eingang PortB0 ein 0-Signal. Wird der Taster losgelassen, reagiert der ATtiny2313 auf die positive Flanke an PortB0 und löst den Interrupt für drei Sekunden aus. Die Leuchtdiode am Ausgang PB7 leuchtet drei Sekunden, bevor sie der Interrupt abschaltet.

1.8 UND-Verknüpfung zwischen zwei Tastern

Bisher sind Sie mit einem Taster ausgekommen, ab jetzt benötigen Sie einen zweiten Taster. An Port PB1 sind Taster, Widerstand und Leuchtdiode anzuschließen.

```
; mainprogram
main:
    andi button,$11
    brne led_off

    ; LED on
    cbi PORTB, 7
    rjmp main

led_off:
    ; LED off
    sbi PORTB, 7
    rjmp main
```

Abb. 1.34: Programm für eine UND-Verknüpfung zwischen zwei Tastern

Das Programm für eine UND-Verknüpfung zwischen zwei Tastern ist in *Abb. 1.34* gezeigt. Sind beide Taster nicht gedrückt, erkennen PortB0 und PortB1 am Eingang ein 1-Signal an. Ist einer der beiden Taster gedrückt, ist die UND-Bedingung nicht erfüllt. Erst wenn beide Taster gedrückt sind, ist die UND-Bedingung erfüllt, der Ausgang PortB7 hat ein 0-Signal und die LED leuchtet.

Es soll nun die Datenübertragung zwischen Atmel-Studio und der Versuchsplatine beschrieben werden. Textdateien sollten mit dem Standardformat nach dem ASCII-Code auskommen und somit ohne Probleme per Datenfernkommunikation zu übertragen sein. Dies ist jedoch eher die Ausnahme als die Regel. So verwenden die meisten Text-Editoren Bit 7 jedes Zeichens, um spezielle Formatierungsinformationen zu speichern. Die mit diesen Editoren erzeugten Dateien haben somit durch-

gängig ein 8-Bit-Format und erfordern eine codetransparente Übertragung oder die Konvertierung in ein 7-Bit-Format.

Ob eine Textdatei über eine codegebundene Übertragungsstrecke transferiert werden kann, lässt sich durch einen einfachen Test feststellen. Der Inhalt der Datei braucht nur mit einem gewöhnlichen LIST-Kommando auf dem Bildschirm ausgegeben zu werden. Erscheinen dann viele Sonderzeichen, muss die Datei vor der Übertragung umformatiert werden. Wird der Inhalt der Datei korrekt und ohne zusätzliche Sonderzeichen angezeigt, bedeutet dies allerdings immer noch nicht, dass sie – zumindest beim erweiterten ASCII – ohne Weiteres übertragen werden kann. Bei Verwendung des erweiterten ASCII ist eine codetransparente Übertragung oder eine Konvertierung erforderlich, wenn die Datei nationale Sonderzeichen enthält, da diese mit Werten oberhalb von 127 codiert sind und daher ein 8-Bit-Format aufweisen. Wird auf Umlaute verzichtet oder werden die Umlaute vor der Übertragung in standardisiertes ASCII-Format umgewandelt, können auch mit dem erweiterten ASCII erstellte Textdateien problemlos übertragen werden. Der Empfänger muss – sofern er ebenfalls das erweiterte ASCII verwendet – eventuell eine Rückformatierung durchführen, bevor er die Datei verwenden kann.

Zur Umwandlung von Textdateien in ein 7-Bit-Format kann ein einfaches Hilfsprogramm verwendet werden, das bei allen Zeichen Bit 7 auf 0-Signal setzt. Ein solcher Filter basiert darauf, dass zum jeweiligen ASCII-Code der Wert 127 binär hinzuaddiert wird. Da sich der Wert 127 im Binär-Code als 01111111 darstellt, bewirkt dies, dass die Inhalte der Bits 0 bis 6 unverändert bleiben, während Bit 7 auf „0“ gesetzt wird. Dieses Prinzip wird anhand des ASCII-Codes 183 angewendet, der bei Rücksetzung von Bit 7 in den Wert 55 überführt wird.

	dezimal	binär
	183	1 0 1 1 0 1 1 1
	127	0 1 1 1 1 1 1 1
binäres UND	—	—
	55	0 0 1 1 0 1 1 1

Man hat eine Filterung von Bit 7 durch binäre Addition. Auf diese Weise kann nicht nur sichergestellt werden, dass die Ausgabe eines Terminal-Programms auf jeden Fall 7-Bit-Codes liefert. Mit diesem Verfahren kann auch eine mit einem Text-Editor erstellte Datei zur 7-Bit-Übertragung vorbereitet werden.

Das alte Betriebssystem CP/M enthält in seinem Hilfsprogramm PIP (Peripheral Interchange Program) bereits eine spezielle Zusatzoption zur Durchführung dieser Umformatierung. Eine Datei wird mit PIP durch Aufruf mit der Option „Z“ in ein

7-Bit-Format umgewandelt. Hat die zu konvertierende Datei z. B. den Namen „TEST.TXT“, dann kann mit dem folgenden Kommando eine Datei namens „TEST.CON“ erzeugt werden, die den Inhalt von „TEST.TXT“ im 7-Bit-Format enthält:

```
PIP TEST.CON = TEST.TXT [Z]
```

Für moderne Betriebssysteme gibt es leider keine ähnlich einfache Möglichkeit, Bit 7 zurückzusetzen. Für die erforderliche Konvertierungsprozedur lässt sich jedoch mit wenigen Zeilen ein kleines Hilfsprogramm schreiben, das bei dem Texteditor vorhanden ist. Vor der Umwandlung einer Textdatei ist jedoch darauf zu achten, dass die Umlaute zunächst im Standard-ASCII-Format umcodiert werden müssen. Die auf diese Weise vorgenommene Konvertierung einer Datei ist eine Art Einbahnstraße. Nach der Konvertierung kann das ursprüngliche 8-Bit-Format nicht wiederhergestellt werden. Wenn keine Weiterverarbeitung der betreffenden Datei beabsichtigt ist, ist dies ohne Bedeutung. Soll die Datei dagegen vom Empfänger mit einem Text-Editor weiterverarbeitet werden, müssen die entsprechenden Formatierungsinformationen erhalten bleiben, was nur bei einer codetransparenten Übertragung mit einer Bytelänge von acht Bit oder bei der Verwendung des Kermit-Protokolls mit der entsprechenden Zusatzoption möglich ist.

Sollen binär codierte Programmdateien übertragen werden, kann das oben für Textdateien beschriebene Verfahren nicht angewendet werden, da es die einzelnen Prozessorinstruktionen verfälschen würde. Steht zur Übertragung von binären Programmdateien keine codetransparente Übertragung zur Verfügung, hat eine Umwandlung in ein 7-Bit-Format nur dann Sinn, wenn das ursprüngliche Format nach der Übertragung wiederhergestellt werden kann. Für diese Zwecke wurde ein spezielles standardisiertes 7-Bit-Format entwickelt, das die anschließende Wiederherstellung des 8-Bit-Formats beim Empfänger gestattet. Das hierzu benutzte Format wird auch als Intel-Hex-Format bezeichnet.

Das Grundprinzip des Intel-Hex-Formats beruht darauf, dass das Byte nicht binär durch seine einzelnen Bits repräsentiert wird, sondern durch seinen hexadezimalen Wert, der als Zeichenfolge mit zwei gewöhnlichen, mit einer Bytelänge von sieben Bit darstellbaren ASCII-Zeichen, codiert wird. So hat das oben abgebildete Byte mit dem dezimalen Wert 183 in hexadezimaler Notation den Wert B7H. Die Übertragung kann nun durch Senden der Standard-ASCII-Codes für „B“ und „7“ erfolgen:

```
10110111 (B7H = 183)
```

wird übertragen als:

```
1000010 („B“) 0110111 („7“)
```

Das führt zu einem wesentlich längeren Code, doch stellt dieses Verfahren oft die einzige Möglichkeit dar, einen Dateitransfer von Programmdateien über codegebundene Übertragungsstrecken durchzuführen. Zusätzlich spezifiziert das Intel-Hex-Format noch bestimmte Regeln zur Fehlererkennung.

Zur Umwandlung von Dateien in Intel-Hex-Format und umgekehrt werden Zusatzprogramme benötigt, die kostenlos im Internet erhältlich sind. Diese Programme verwenden meistens den Namen „LOAD“ (zur Umwandlung einer Intel-Hex-Datei in eine 8-Bit-Datei) und „UNLOAD“ (zur Erzeugung einer Intel-Hex-Datei). Vor der Übertragung wandelt der Sender die zu übertragende Datei in ein Intel-Hex-Format um, indem er das Programm UNLOAD aufruft. Dieses Programm fertigt eine Kopie des ursprünglichen Programms an, die mit der Erweiterung „.HEX“ versehen wird. Der Sender überträgt nun die konvertierte Datei. Bevor der Empfänger mit dieser Datei arbeiten kann, muss er das Programm LOAD aufrufen, das eine Kopie der Datei in ihrer ursprünglichen Form erzeugt. Dabei wird gleichzeitig auch eine Fehlerprüfung durchgeführt.

Eine Datei im Intel-Hex-Format besteht aus mehreren Zeilen oder „Records“. Jeder dieser Records besteht aus maximal 60 Spalten und beginnt mit einem Doppelpunkt. Das nachfolgende Beispiel zeigt den Anfang einer Datei im Intel-Hex-Format:

```
:180000001A084144562E444F430003000004BE6C0400004F0C7D06DBF9
:180018003E800700000C418858710145CA131069E6800803A2E01326A5
:180030004F8E808023E7CD193961DA80A083260C1D846DE0B029D3A662
:180048008C1B3A0AC3B80111D04A112754AA482902E24CC6322E4024AD
:180060003103A281823C6FEA80E868A70C084803DCBCF998A7CC4791D0
:18007800619A9259B8320C99A227EBC829D3824E1E38466D960431F2ED
```

Bereits am äußeren Erscheinungsbild einer Datei ist somit ersichtlich, ob eine Umcodierung in Intel-Hex erfolgte. Jeder Record hat ein festes Format, das in Tabelle 1.8 beschrieben ist.

Da jedes Byte durch zwei Zeichen repräsentiert wird, entspricht ein Zeichen einem „Halbbyte“. Die Prüfsumme berechnet sich aus der Summe aller übertragenen Bytes (einschließlich Länge und Adresse, aber ausgenommen des Startzeichens „:“) modulo 256.

Mit dem Programm lässt sich eine ODER-Verknüpfung mit dem Befehl „OR“ oder „ORI“ realisieren. Mit dem Programm kann man auch eine EXOR-Verknüpfung mit dem Befehl „EOR“ realisieren. Folgt nach einer UND-Verknüpfung ein EXOR (Antivalenz), kann man eine NAND-Verknüpfung programmieren. Folgt nach einer ODER-Verknüpfung ein EXOR, kann man eine NOR-Verknüpfung program-

Tabelle 1.8: Aufbau eines Records im Intel-Hex-Format

Position	Bedeutung	
1	Startzeichen: immer	
2	oberes Halbbyte	Länge
3	unteres Halbbyte	des Recordinhalts
4	höchstes Halbbyte	Adresse
5	zweithöchstes Halbbyte	des ersten
6	dritthöchstes Halbbyte	Bytes des
7	vierthöchstes Halbbyte	Recordinhalts
8	0 Trennzeichen	
9	0 (immer „0“)	
10		
.		
.		Daten
.		
n		
n + 1	oberes Halbbyte	Zweierkomplement
n + 2	unteres Halbbyte	der Prüfsumme

mieren. Folgt nach einer EXOR-Verknüpfung ein EXOR, kann man eine INOR-Verknüpfung (Äquivalenz) programmieren.

1.9 Wechselschaltung mit zwei Tastern

Mit dem Programm „test8“ kann man eine Wechselschaltung realisieren. Drückt man den Taster am Eingang PB0, lässt sich das programmierbare Flipflop mit dem Mikrocontroller setzen und rücksetzen, d. h. die Leuchtdiode ein- oder ausschalten. Drückt man den Taster am Eingang PB1, kann man das programmierbare Flipflop mit dem Mikrocontroller setzen und rücksetzen. Die Leuchtdiode am Ausgang PB7 lässt sich auch vom Eingang PB0 oder vom Eingang PB1 ein- und/oder ausschalten. *Abb. 1.35* zeigt das Programm.

Zuerst wird mit dem Befehl „sbrs stavec, 0“ das Bit im Statusvektor auf 0 gesetzt. Dann erfolgt der relative Sprung „rjmp main“ und ist die Bedingung erfüllt, dann wird der Statusvektor gelöscht. Mit „mov acc0, button“ wird der Zustand der beiden Schalter in den Akkumulator geschrieben und dann mit „andi acc0, $03“ mit UND verknüpft. Dann folgt mit „cpi acc0, $02“ ein Löschen des Bits im I/O-Register. Anschließend erfolgt der Befehl „breq led_toggle“. Ist der Inhalt des Akku-

```
; mainprogram
main:
    sbrs stavec,0
    rjmp main
    clr stavec

    mov acc0, button
    andi acc0, $03
    cpi acc0, $02
    breq led_toggle

    mov acc0, button
    andi acc0, $30
    cpi acc0, $20
    brne main

led_toggle:
    ; toggle PORTB(7)
    in acc0, PORTB
    ldi acc1, $80
    eor acc0, acc1
    out PORTB, acc0
    rjmp main
```

Abb. 1.35: Programm für eine Wechselschaltung mit zwei Tastern

mulators gleich, springt das Programm auf „breq led_toggle“, d. h. die Leuchtdiode wird entweder ein- oder ausgeschaltet.

Mit „mov acc0, button“ beginnt der nächste Programmabschnitt. Der Inhalt des Akkumulators wird mit „$30“ über ein UND verknüpft. Dann folgt mit „cpi acc0, $20“ ein Löschen des Bits im I/O-Register. Mit dem Befehl „brne main“ erfolgt ein Rücksprung, wenn der Inhalt des Akkumulators ungleich „$20“ ist.

Das Programm „led_toggle“ setzt oder rücksetzt den Ausgang PB7. Bei einem 0-Signal leuchtet die LED auf, bei einem 1-Signal bleibt sie dunkel. Der Akkumulator wird mit „in acc0, PortB“ geladen und dann mit „ldi acc1, $80“ der andere Akkumulator geladen. Anschließend erfolgt ein Vergleich mit EXOR-Verknüpfung durch „eor acc0, acc1“, wobei der Akkumulator „acc0“ das Ergebnis der logischen Verknüpfung beinhaltet. Mit „out PORTB, acc0“ wird der Setz- oder Rücksetzvorgang der LED bestimmt. Zum Schluss erfolgt der Rücksprung in das Hauptprogramm.

1.10 Steuerbarer Blinker

Die Peripherie des steuerbaren Blinkers besteht aus zwei Tastern und einer Leuchtdiode. Drückt man den Taster am Eingang PB0, blinkt die LED mit 1 Hz, d. h. die LED ist etwa 500 ms an und 500 ms aus. Drückt man den Taster am Eingang PB1, blinkt die LED mit 0,33 Hz, d. h. die LED ist etwa 1,5 s an und 1,5 s aus. *Abb. 1.36* zeigt das Programm.

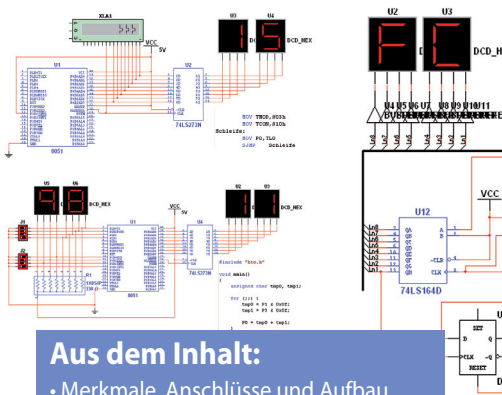
Das Hauptprogramm beginnt mit dem Befehl „sbrc button, 0“ und damit wird das Register gelöscht. Dann erfolgt der Befehl „rjmp not_pressed“, ein Sprung auf das

Sachverzeichnis

FRANZIS
PC & ELEKTRONIK

Herbert Bernstein

Mikrocontroller in der Elektronik



Die beiden Mikrocontroller ATtiny2313 und ATtiny26 von ATMEL sind zwei leistungsfähige 8-Bit-Mikrocontroller. Beide Mikrocontroller verwenden für die Speicherung des Programms einen 2-KByte-Flashspeicher, der über eine „In-System-Programmierung" (3-Draht-ISP-Schnittstelle) programmiert wird. Die Programmierung übernehmen diese Bausteine selbstständig durch eine interne „High Voltage"-Einheit.

Mit dem bekannten „digitalen" 8-Bit-Mikrocontroller ATtiny2313, einem Industriestandard in einem praktischen 20-poligen DIL-Gehäuse, steht für den Elektroniker ein kostengünstiger Mikrocontroller zur Verfügung. Mit diesem Baustein lassen sich zahlreiche Aufgaben aus dem Bereich der Hobbyelektronik und aus der Praxis lösen. Zahlreiche digitale Versuche lassen sich aufbauen und programmieren.

Der „analoge" 8-Bit-Mikrocontroller ATtiny26, ebenfalls ein Industriestandard in einem praktischen 20-poligen DIL-Gehäuse, hat mehrere 10-Bit-AD-Wandler zur Verfügung. Mit dem AD-Wandler lassen sich zahlreiche Versuche aufbauen, programmieren und durchführen.

Die in diesem Buch vorgestellten Versuche mit dem ATtiny2313 und dem ATtiny26 sind einfach auf einer Lochrasterplatine aufgebaut, und man benötigt für die einzelnen Aufbauten etwa eine Stunde Lötzeit. Die Bauelemente erhalten Sie im Versandhandel.

Die Versuche werden alle im AVR-Assembler programmiert. Entwickler, Studenten und Schüler können auf komfortable Weise die ATMEL-Software für ihre Anwendungen auf dem PC implementieren und testen. Die sehr leistungsfähige Entwicklungsumgebung „AVR-Studio" bündelt alle benötigten Funktionen – vom Assembler bis zum Simulator.

Hardware- und Softwarevoraussetzungen:
PC mit CD-Laufwerk ab Windows 2000/NT/Vista/7

Aus dem Inhalt:

ISBN 978-3-645-65014-4

Euro **29,95** [D]