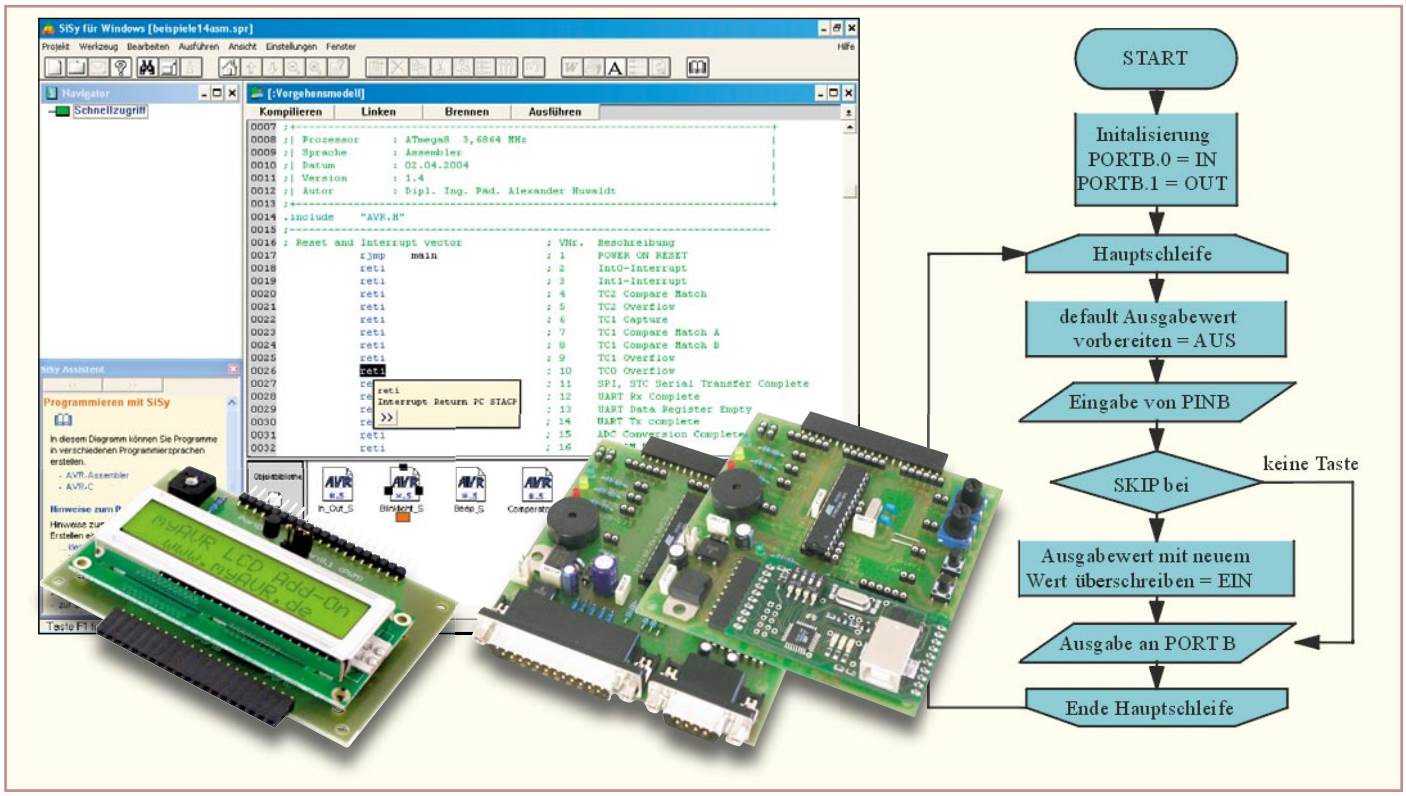


Mikrocontroller-Einstieg mit myAVR Teil 2

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, Software-Paket, Kabel, sämtliches Zubehör. Im zweiten Teil unserer Serie zum Einstieg in die AVR-Programmierung betrachten wir den AVR-Assembler, gehen auf die Grundlagen der Ein- und Ausgabe-Register ein und zeigen das erste praktische Programmierbeispiel mit dem myAVR-Board.

Der AVR-Assembler

Der Assembler ist ein Bestandteil der im ersten Teil bereits kurz betrachteten Entwicklungsumgebung SiSy. Er dient dazu, so genannte Maschinenprogramme in das für den Controller allein direkt lesbare Binärformat umzusetzen und die korrekten Adressen für Sprungbefehle und Speicherzugriffe auszurechnen. Denn der Controller kann nur Befehle im Binärformat interpretieren. Da sieht dann der bekannte Rücksprungbefehl, der in der Befehlstabelle mit der Mnemonik „RET“ beschrieben ist, so aus: 1001 0101 0000 1000. In dieser Form wäre es zwar prinzipiell möglich, ein Programm zur direkten Eingabe in den Controller zu schreiben, aber selbst erfahrene Programmierer würden sich hier sehr schwer tun, ein überblickbares Programm zu schreiben. Deshalb hat man o. g. Mnemoniks entwickelt, die je nach Befehl noch mit entsprechenden Operanden ergänzt werden. Alle von den AVR-Mikrocontrollern unterstützten Befehle sind in der Tabelle 1 (siehe Teil 1) zusammengefasst. Sie werden vom Assembler übersetzt („compiliert“ und „gelinkt“) und liegen schließlich als Programm im Hex-Format für das Übertragen in den AVR-Controller vor.

Programmeditor

Zur Programmieroberfläche von SiSy gehört auch ein Editor (Abbildung 9), in dem man das Programm, den so genannten Quellcode, wie mit einem normalen Texteditor schreibt. Es kann natürlich auch in einem beliebigen Texteditor verfasst werden, Bedingung ist lediglich, dass der Text im ASCII-Format abgespeichert werden kann. Die ordentliche Formatierung, die man im Quellcode sieht, dient eigentlich nur der besseren Übersicht über die einzelnen Elemente der Programmzeilen. Diese werden übersichtlich in Spalten angeordnet, können mit beliebigen Kommentaren versehen werden und werden einfach mit Text-Tabulatoren oder ganz profan mit der Leertaste erzeugt.

Wollen wir die Struktur des Quellcodes kurz anhand Abbildung 9 und ff. näher betrachten. Ausführliche Abhandlungen hierzu, z. B. zur Interrupt-Behandlung oder zum Stack, finden sich im Kursmaterial des „myAVR“ sowie in den Assistenzfunktionen von „SiSy“, denn ein ausführlicher Assemblerkurs ist im Rahmen dieses Artikels nicht unterzubringen. Hier soll es ja lediglich um den Einstieg gehen. Innerhalb der praktischen Beispiele werden wir bei Bedarf allerdings auf jeden Programmschritt und jeden Befehl ausführlich eingehen.

Kommentare und Programmkopf

Jedes Programm beginnt mit dem Programmkopf, der zahlreiche Kommentarzeilen enthält. Eine Kommentarzeile beginnt stets mit dem Semikolon-Zeichen (;). Hier trägt man im Programmkopf grundsätzliche Angaben zum Programm wie Name, Funktionen, Autor, Version, Datumsangaben, vorgesehener Prozessor usw. zur Dokumentation ein.

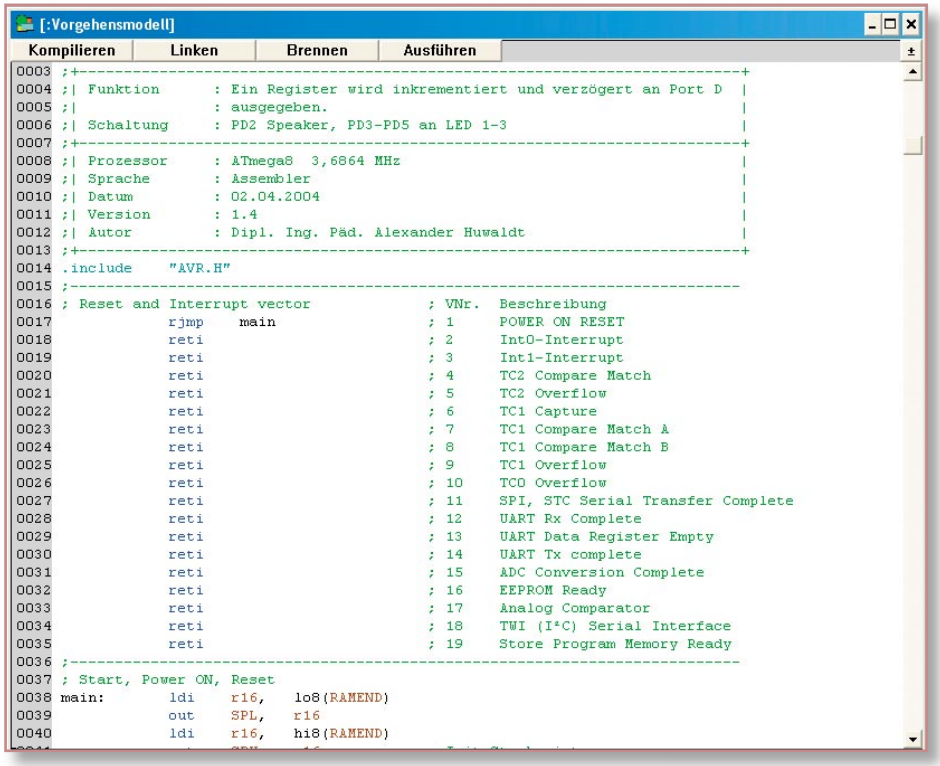
Die Zeile „.include "AVR.H"“ bezeichnet die Einbindung der (für alle AVRs feststehenden) Datei, die die I/O-Register (Ports) des AVR verschlüsselt, in das Programm. Sie wird automatisch generiert und erscheint im fertigen Projektordner als so genannte H-Datei.

Weitere Kommentare sind beliebig innerhalb des Programms einschreibbar, sie sind nur jeweils mit dem Semikolon zu kennzeichnen.

Reset- und Interrupt-Tabelle

Als Nächstes folgt die so genannte Reset- und Interrupt-Tabelle, ein fester Bestandteil jedes Programms. Hier werden entsprechend des verwendeten Controllers Ansprungpunkte für bestimmte Ereignisse (Interrupt) während des Programmlaufs festgelegt. Betrachtet man Abbildung 10, erkennt man, dass in unserem einfachen Beispiel nur ein Sprungbefehl zum Hauptprogramm (main), das mit der Power-on/Reset-Phase des Controllers beginnt, aktiv ist (Befehl RJMP mit dem Operanden „main“). Alle restlichen Zeilen der Tabelle sind als inaktive Platzhalter mit dem RETI-Befehl (bei einem entsprechenden Interrupt soll keine Reaktion erfolgen) belegt. In der Beschreibungs-Spalte kann man ersehen, wofür die jeweiligen Interrupts genutzt werden. Jetzt erkennt man, warum die komplette Tabelle aufgeführt werden muss – würde man nicht alle eintragen, „wüsste“ der Controller nicht, welcher Interrupt-Vektor gemeint ist.

Über die Vollständigkeit der Tabelle muss man sich beim SiSy-System (wie auch bei vielen anderen Entwicklungsumgebungen) keine Gedanken machen, sie liegt fest und wird einfach mit dem so genannten




```
0003 ;+--------------------------------------------------------------------+
0004 ;| Funktion      : Ein Register wird inkrementiert und verzögert an Port D  |
0005 ;|               : ausgegeben.                                        |
0006 ;| Schaltung     : PD2 Speaker, PD3-PD5 an LED 1-3                    |
0007 ;+--------------------------------------------------------------------+
0008 ;| Prozessor     : ATmega8  3,6864 MHz                                |
0009 ;| Sprache       : Assembler                                          |
0010 ;| Datum         : 02.04.2004                                         |
0011 ;| Version       : 1.4                                                |
0012 ;| Autor         : Dipl. Ing. Päd. Alexander Huwaldt                  |
0013 ;+--------------------------------------------------------------------+
0014 .include    "AVR.H"
0015 ;---------------------------------------------------------------------
0016 ; Reset and Interrupt vector              ; VNr.  Beschreibung
0017             rjmp    main                 ; 1     POWER ON RESET
0018             reti                         ; 2     Int0-Interrupt
0019             reti                         ; 3     Int1-Interrupt
0020             reti                         ; 4     TC2 Compare Match
0021             reti                         ; 5     TC2 Overflow
0022             reti                         ; 6     TC1 Capture
0023             reti                         ; 7     TC1 Compare Match A
0024             reti                         ; 8     TC1 Compare Match B
0025             reti                         ; 9     TC1 Overflow
0026             reti                         ; 10    TC0 Overflow
0027             reti                         ; 11    SPI, STC Serial Transfer Complete
0028             reti                         ; 12    UART Rx Complete
0029             reti                         ; 13    UART Data Register Empty
0030             reti                         ; 14    UART Tx complete
0031             reti                         ; 15    ADC Conversion Complete
0032             reti                         ; 16    EEPROM Ready
0033             reti                         ; 17    Analog Comparator
0034             reti                         ; 18    TWI (I²C) Serial Interface
0035             reti                         ; 19    Store Program Memory Ready
0036 ;---------------------------------------------------------------------
0037 ; Start, Power ON, Reset
0038 main:       ldi     r16,    lo8(RAMEND)
0039             out     SPL,    r16
0040             ldi     r16,    hi8(RAMEND)
```

Bild 9: Im Editorfenster lässt sich der Quellcode übersichtlich anzeigen und editieren.

„Grundgerüst“ geladen, wie wir noch sehen werden. Je nach Programm sind nur noch die einzelnen Sprungbefehle einzutragen, also Befehl plus Operand, sofern nötig.

Initialisierung und Hauptprogramm

Der Interrupt- und Reset-Tabelle folgt das Hauptprogramm (main, Abbildung 11). Dieses beginnt immer damit, dass der Controller bei einem Reset oder beim Einschalten auf definierte Bedingungen zurückgesetzt werden muss. Das heißt vor allem, dass die bereits im ersten Teil beschriebenen wichtigen Steuerregister bei jedem Neustart initialisiert werden müssen. Vor allem der Speicher für die Rücksprungadressen von Unterprogrammen (STACK) muss immer initialisiert sein, sonst gerät der Programmablauf – also z. B. die Sprünge zwischen Programmteilen, zu Interrruptroutinen usw. – durcheinander.

```
; Start, Power ON, Reset
main:     ldi     r16, lo8(RAMEND)
          out     SPL, r16
          ldi     r16, hi8(RAMEND)
          out     SPH, r16
          ; Hier Init-Code eintragen.
;-----------------------------------------
mainloop: wdr
          ; Hier den Quellcode eintragen.
          rjmp mainloop
```

Bild 11: Das Grundgerüst von Initialisierung und Hauptprogramm

Im bereits angesprochenen Grundgerüst befindet sich also diese Stack-Initialisierung. Hier wird das Register SP als Stackpointer auf seine Startadresse am Ende des SRAM gelegt (RAMEND). Details dazu finden sich wiederum im Kursmaterial des „myAVR“.

Die Zeile „; Hier Init-Code eintragen“ markiert, dass dort, neben der eben beschriebenen Grundinitialisierung, weitere Definitionen entsprechend dem jeweiligen Programm einzutragen sind. So legt man hier etwa fest, welche Ports als Ein- oder Ausgänge definiert werden sollen, wo evtl. ein Pull-up-Widerstand zu aktivieren ist, auf welche Anfangswerte ein Zähler zu legen ist usw. Beispiele dazu werden wir noch kennen lernen.

Nach der Initialisierungs-Sequenz ist nun das eigentliche Programm zu schreiben. Das Grundgerüst besteht hier lediglich aus der Definition, dass der Programmablauf

```
; 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
```

Bild 10: Die Reset- und Interrupt-Vektor-Tabelle

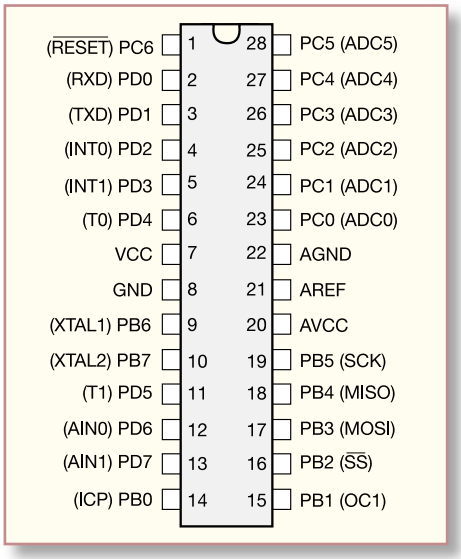


Bild 12: Anhand der Pin-Belegung erkennt man die Doppelfunktion vieler Ports.

ohne Unterbrechung ständig von Neuem beginnen soll. So wird zunächst zu jedem Beginn dieser Programmschleife (loop) die Watchdog-Schaltung des Controllers definiert zurückgesetzt, damit diese nicht versehentlich einen Reset auslöst und so ungewollt Daten löscht bzw. Vorgänge unterbricht. Dann folgen die eigentlichen Programmschritte, hier mit dem Kommentar „Hier den Quellcode eintragen“ als Platzhalter markiert.

„rjmp mainloop“ schließlich erzwingt wieder eine Rückkehr zum Beginn des Hauptprogramms. So einfach erzeugt man einen ununterbrochenen Programmablauf.

Damit man bei umfangreicheren Programmen, die vor allem von mehreren Programmteilen aus immer wieder benötigte Standardroutinen anspringen, die Übersicht behält, folgen dem Hauptprogramm, übersichtlich einzeln geordnet und kommentiert, Unterprogramme, auf die jeweils mit Sprungbefehlen aus dem Hauptprogramm zugegriffen wird.

Apropos, kommen wir noch einmal zum Thema Übersichtlichkeit. Die eingangs erwähnte Spaltenanordnung der einzelnen Textteile des Programms dient wesentlich der Übersicht über das Gesamtprogramm und sollte im eigenen Interesse strikt eingehalten werden. Noch einige Erläuterungen dazu:

In der ersten Spalte finden sich die so genannten Marken bzw. Bezeichner für Standard-Adressen, die von den Sprungbefehlen im Programm genutzt werden. Schreibe ich also „rjmp mainloop“ ins Programm, erfolgt ein Sprung zur Adresse, die sich hinter dem Bezeichner „mainloop:“ verbirgt. Marker/Bezeichner sind immer als erste Spalte einzuschreiben, sie müssen mit einem Buchstaben beginnen und mit einem Doppelpunkt enden.

In der zweiten und dritten Spalte sind die Maschinenbefehle sowie deren Operanden mit Ziel und Quelle (in dieser Reihenfolge!) der Operation einzutragen.

Schließlich kann zu jeder Programmzeile wiederum ein Kommentar (Beschreibung) eingetragen werden.

So, und in deutliche Programmteile aufgeteilt strukturiert, bleibt jedes Programm übersichtlich und einzelne Bestandteile sind klar definiert abgegrenzt.

Hat man das Programm fertig geschrieben, tritt der Assembler überhaupt erst in Aktion. Er übersetzt den geschriebenen Quellcode, prüft diesen dabei auf Plausibilität und exakte Programmierung und stellt nach dem Linken eine auf den Controller übertragbare Hex-Datei zur Verfügung.

Rein und raus – I/O-Grundlagen

Jeder Mikrocontroller kommuniziert mit seiner Umwelt über so genannte Ports. Dies sind der jeweiligen Aufgabe angepasste Ein- und Ausgangsstufen, die bei kleineren Controllern aus Mangel an mechanisch realisierbaren Pins schon einmal so ausgeführt sind, dass man sie wahlweise als Ein- oder Ausgang programmieren kann. Die AVR-Controller sind darüber hinaus so konfiguriert, dass man auch die Steuerregister, die für die unterschiedlichsten Standardaufgaben des Controllers zuständig sind, aus der Sicht des Programmierers bzw. Rechenwerks als I/O-Ports realisiert hat. Deshalb werden diese Register auch wie Ports angesprochen bzw. die hier als I/O-Lines bezeichneten physischen Ports wie Register. Alle Ports haben eine feste Registeradresse, über die sie angesprochen werden können. Bei der Programmierung verwendet man hierzu so genannte Alias-

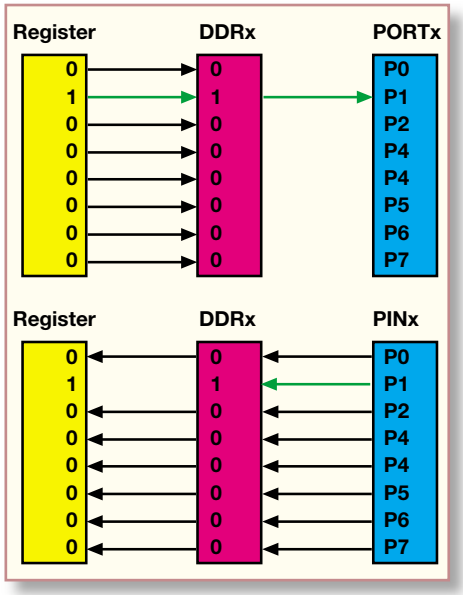


Bild 13: Das Wirkungsschema des Steuerregisters DDRx bei out- (oben) und in-Befehlen (unten)

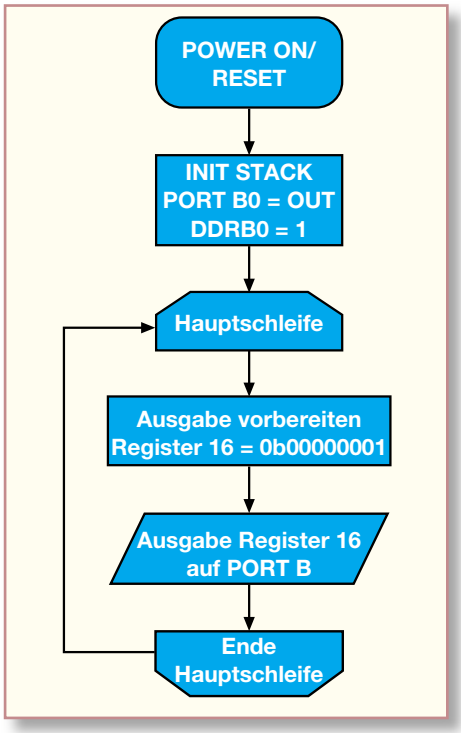


Bild 14: Im Flussdiagramm ist die zu lösende Aufgabe übersichtlich skizziert.

Namen, die sich jeweils in der bereits erwähnten Datei „AVR.H“ finden.

Widmen wir uns aber zunächst den physischen Ports unseres AVR-Controllers. Der auf dem myAVR-Board verbaute Controller verfügt über 3 digitale 8-Bit-I/O-Ports, d. h. über insgesamt 24 so genannte I/O-Lines:

Port B0...B7 (Registeradresse 0x18)
Port C0...C7 (Registeradresse 0x15)
Port D0...D7 (Registeradresse 0x12)

Betrachtet man nun einmal die vollständige Anschlussbelegung des auf dem myAVR-Board verbauten Controllers (Abbildung 12), erkennt man auf Anhieb, dass aufgrund der geringen Baugröße des Controllers fast alle Pins doppelt belegt sind. Damit ergeben sich bei der Programmierung jeweils einige Einschränkungen, etwa die, dass bei unserem myAVR die Ports PD0 und PD1 nicht frei zur Verfügung stehen, da sie fest mit dem RS232-Baustein für die Datenkommunikation verdrahtet sind. Weitere Besonderheiten und auch die vollständige Registertabelle mit allen Adressen sind im myAVR-Lehrgang detailliert aufgeführt. Das führt schließlich dazu, dass je Port noch 6 I/O-Lines frei verfügbar bleiben, wobei man PC0...C5 als A/D-Wandler-Ports (ADC) berücksichtigen muss.

Entsprechend der genannten AVR-Port-Philosophie erfolgt die Programmierung der Ports über jeweils 3 so genannte I/O-Register (siehe Abbildung 13). Zunächst ist hier das Steuerregister DDRx (Data Direction Register Port x [B/C/D]) zu besprechen, das die Richtung des Daten-

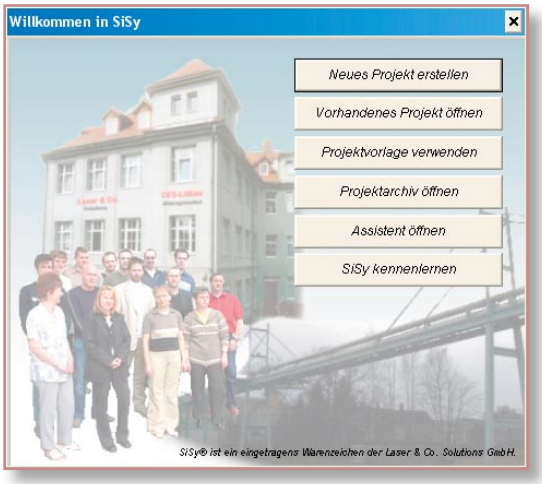


Bild 15: Das SiSy-Startfenster

flusses festlegt. Hier finden wir also die Schaltstelle, die über die Funktion jedes einzelnen Port-Pins entscheidet. Über das Register PORTx erfolgt die Ausgabe, über PINx die Eingabe von Daten. So zeichnen sich bereits erste Konturen von Programmansätzen ab, die I/O-Aufgaben betreffen: Es ist das entsprechende Portregister laut vorliegender Aufgabe anzusprechen bzw. zu konfigurieren und dann die gewünschte I/O-Line anzusteuern bzw. auszuwerten.

Genau diesen Aufgaben widmet sich nun unser erster Einstieg in die Programmierpraxis mit anschließender Ausführung auf dem myAVR-Board.

Er lebt! Der erste Ausgabebefehl

Die erste Aufgabe lautet für uns, einen Port-Pin des myAVR-Controllers als Ausgang zu konfigurieren und eine der drei LEDs auf dem Board vom Controller aus einschalten zu lassen. Da der Ausgang des AVRs bis zu 20 mA (bei 5-V-High-Pegel) treiben kann, ist also der direkte Anschluss einer LED, natürlich mit entsprechendem Vorwiderstand, einfach möglich.

Also müssen wir „nur“ noch einen Port-Pin als Ausgang konfigurieren und hierüber einen Ausgabebefehl geben. Wir legen fest, dass die Ausgabe über Port B, I/O-Line 0 (PB0) erfolgen soll.

1. Initialisierung

Erinnern wir uns dazu zunächst an den eingangs beschriebenen Aufbau des Programm-Grundgerüstes. Hier fand sich im Initialisierungsteil der Hinweis auf weitere Initialisierungsdefinitionen neben der Grundinitialisierung des Controllers. Genau hier gehört nun die Port-Konfiguration über das Steuerregister DDRB hinein. Jedes Bit dieses Steuerregisters steht für die Datenrichtung der einzelnen I/O-Lines, wobei logisch null den Pin als Eingang, logisch eins hingegen den Pin als Ausgang definiert.

Für die Konfiguration stehen zwei Befehle zur Verfügung, die sich in ihrer Wirkung unterscheiden:

Der out-Befehl konfiguriert immer alle Bits des Ports auf einmal, einzelne Bits (Pins) sind nicht differenzierbar. Hier kann es also vorkommen, dass es zu Kollisionen, sprich Überschreiben, kommt, wenn etwa Bits des Ports durch andere Programmteile bereits gesetzt sind.

Der sbi-Befehl hingegen setzt auf Nummer Sicher – er spricht gezielt nur ein Bit des jeweiligen Ports an, ohne andere Bits des Ports zu beeinflussen.

Beide Methoden haben ihre jeweilige Berechtigung, wie wir noch sehen werden.

Also lernen wir beide kurz kennen:

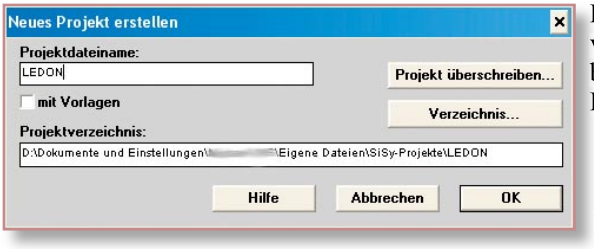


Bild 16: Ein neues Projekt wird angelegt.

Konfiguration mit dem out-Befehl

Wir verwenden ein allgemeines Register, hier r16 (verfügbar: r0...r31) als temporäre Variable (im Prinzip ein Zwischenspeicher), um es mit der gewünschten I/O-Konfiguration (Konstante, Bit 0 von Port B auf Ausgang, Bits 1 bis 7 auf Eingang) zu laden (ldi-Befehl, siehe Befehlsübersicht:

ldi Register, Konstante) und geben die Port-Konfiguration des allgemeinen Registers mittels des out-Befehls an Port B aus (out Ziel, Quelle):

```
ldi     r16,    0b00000001
out     DDRB,   r16
```

Konfiguration mit dem sbi-Befehl

Deutlich „eleganter“ gestaltet sich die Konfiguration mit dem sbi-Befehl. Hier wird das Steuerregister direkt angesprochen (ein Bit direkt im I/O-Register/Port gesetzt), wobei die Bits B1 bis B7 unbeeinflusst bleiben:

```
sbi     DDRB, 0
```

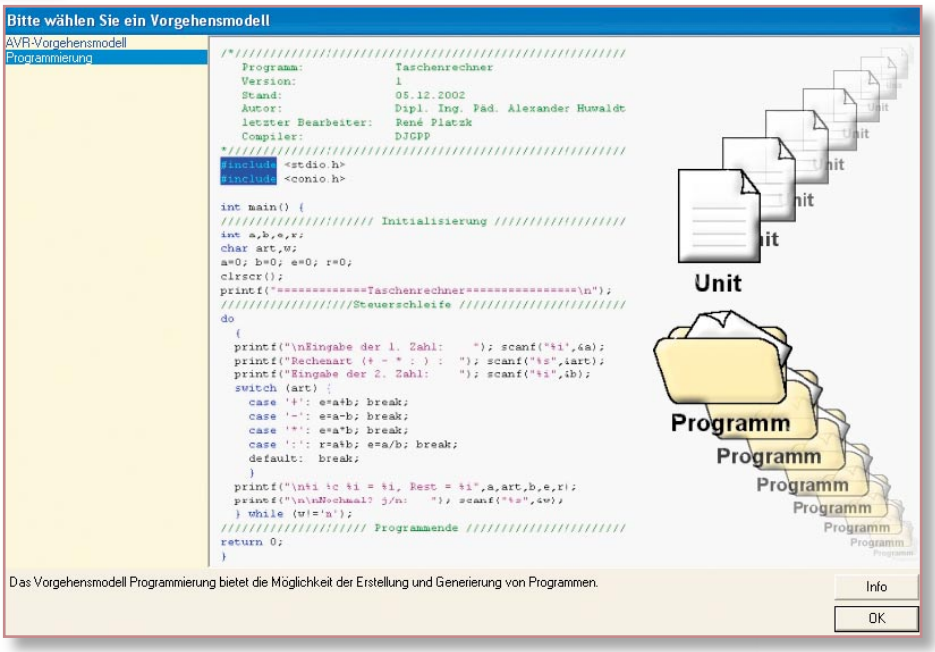


Bild 17: Als Vorgehensmodell wählen wir hier „Programmierung“.

Damit erweist sich der sbi-Befehl für unseren Zweck am besten für die Initialisierung des Ports geeignet, da wir ja zunächst nur die Line B0 beeinflussen wollen.

2. Ausgabe

Wir erinnern uns, von der Programmierung her werden die physischen Ports genau so behandelt wie die eben bei der Initialisierung behandelten Steuerregister. Deshalb können wir zur Ansteuerung der LED genau die gleichen Befehle einsetzen wie eben besprochen, wir beschränken uns hier auf den out-Befehl.

Dazu ist wieder ein allgemeines Register mittels ldi-Befehl mit den gewünschten Ausgabedaten zu laden, und dann folgt die Ausgabe des Register-Inhalts an Port B0. Je nachdem, ob wir das Bit 0 auf null oder eins setzen, wird später die angeschlossene LED an Ausgang B0 ein- oder ausgeschaltet. Wir sollen die LED einschalten:

```
ldi     r16,    0b00000001
out     PORTB,  r16
```

Bild 18: Es gibt in der Software schon zahlreiche vorgefertigte Teilprogramme, hier wird zunächst „keine Vorlage verwenden“ angewählt.

Dies bildet unser erstes Hauptprogramm! Die beiden Zeilen werden also als Quellcode im Programmteil „Mainloop“ eingetragen. Sollte das Ganze funktionieren, schaltet der Controller (sobald alles assembliert, gelinkt und auf den AVR gebrannt ist), die LED ein, sobald die Betriebsspannung an myAVR angeschaltet ist.

Das werden wir jetzt testen!

Zuvor rufen wir uns aber noch einmal komplett ins Gedächtnis, was wir bis jetzt getan haben. Das kann man am besten in einem Flussdiagramm erkennen, wie es Programmierer zur besseren Übersicht über ihre Programmplanungen entwerfen. Das zu unserem Programm passende Diagramm ist in Abbildung 14 zu sehen.

Licht an! Das erste Programm

Jetzt gehen wir wie beschrieben vor: SiSy starten und „Assistent starten“ anwählen (Abbildung 15), dann im Assistenten „neues Projekt“ anwählen. Hier (Abbildung 16) erscheint das Eingabefenster

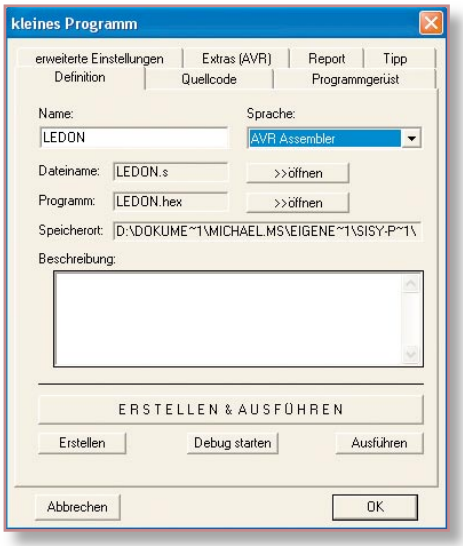


Bild 19: Das Dialogfenster für Grundeinstellungen und Programmierung

für den Projektnamen und den Standort der Projektdatei (der erfahrene Benutzer gelangt hierhin direkt aus dem Startmenü über „Neues Projekt“). Nach Eingabe des Projektnamens erscheint der Abfragedialog zum Vorgehensmodell (Abbildung 17), hier ist „Programmierung“ auszuwählen. Aus dem dann folgenden Vorschlag „Diagrammvorlagen“ (Abbildung 18) wird zunächst „keine Vorlagen“ angewählt.

Jetzt erscheint das bereits bekannte SiSy-Editorfenster, und es ist aus der Objektbibliothek das Icon „kleines Programm“ per Drag & Drop in das Diagrammfenster zu ziehen.

Daraufhin öffnet sich ein Dialogfenster, in das unter „Definition“ zunächst noch einmal der Programmname einzutragen und anschließend unter „Sprache“ die Option „AVR-Assembler“ auszuwählen ist (Abbildung 19).

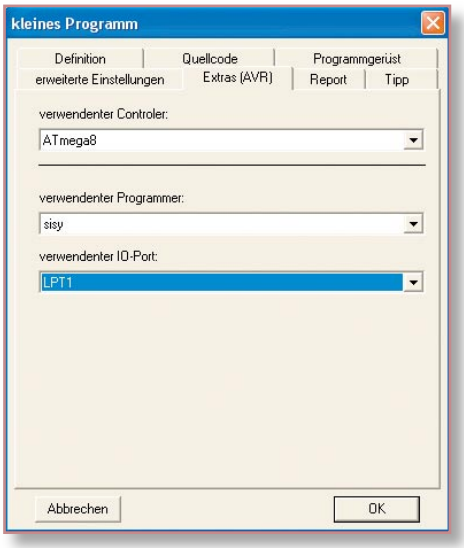


Bild 20: Auch der AVR-Typ und die Art des Programmiersystems lassen sich hier einstellen.

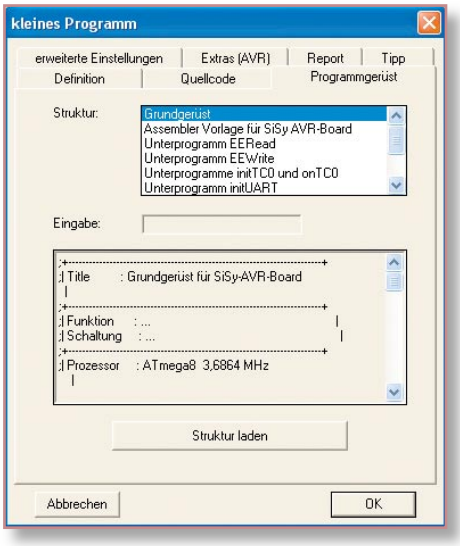


Bild 21: Das Grundgerüst des Programms wird geladen.

Unter „Extras (AVR)“ (Abbildung 20) wird sodann der eingestellte Controller-Typ (ATmega8), der Programmer (sisy) und der verwendete I/O-Port (hier LPT1) kontrolliert bzw. eingestellt.

Über die Option „Programmgerüst“ lädt man nun sehr bequem das bereits mehrfach diskutierte Grundgerüst unseres Programms (Abbildung 21). Über „Struktur laden“ wird dieses Grundgerüst in den Editor geladen, es erscheint im Hintergrund. Jetzt kann man nach Bestätigung über „OK“ dazu übergehen, im Editor zu arbeiten oder, nach Wechsel auf „Quellcode“, diesen dort bearbeiten.

Quellcode ergänzen

Nach der Bestätigung erscheinen der Programmname „LEDON“ unter dem Icon im Diagrammfenster und der Quellcode des Grundgerüstes im Editorfenster. Zur guten Dokumentation gehört zunächst das

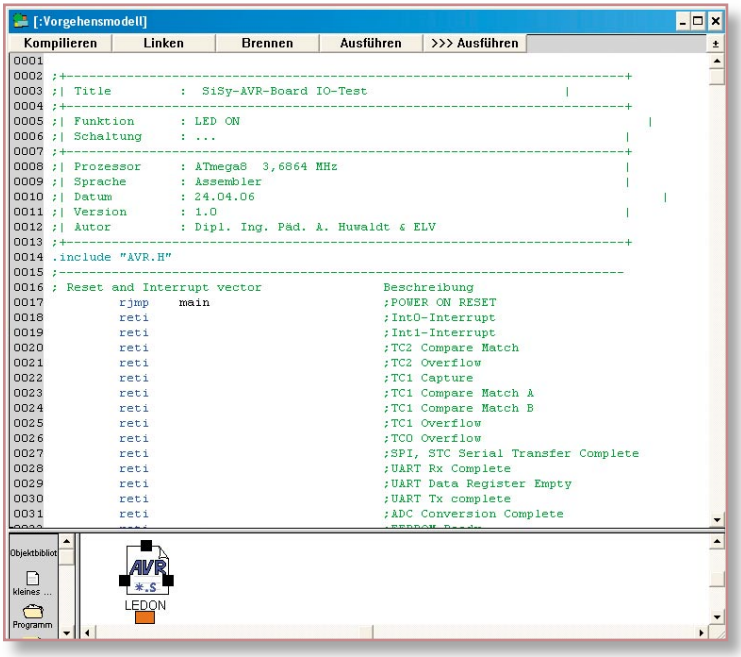


Bild 22: Der Editor mit dem Programmkopf unseres ersten Programms, unten die Objektbibliothek und das Diagrammfenster mit unserem Projekt

```
;+------------------------------------------------------------------------+
;| Title          :  SiSy-AVR-Board IO-Test                               |
;+------------------------------------------------------------------------+
;| Funktion       : LED ON                                                |
;| Schaltung      : ...                                                   |
;+------------------------------------------------------------------------+
;| Prozessor      : ATmega8  3,6864 MHz                                   |
;| Sprache        : Assembler                                             |
;| Datum          : 24.04.06                                              |
;| Version        : 1.0                                                   |
;| Autor          : Dipl. Ing. Päd. A. Huwaldt & ELV                      |
;+------------------------------------------------------------------------+
.include "AVR.H"
;-------------------------------------------------------------------------
; Reset and Interrupt vector                Beschreibung
        rjmp    main                        ;POWER ON RESET
        reti                                ;Int0-Interrupt
        reti                                ;Int1-Interrupt
        reti                                ;TC2 Compare Match
        reti                                ;TC2 Overflow
        reti                                ;TC1 Capture
        reti                                ;TC1 Compare Match A
        reti                                ;TC1 Compare Match B
        reti                                ;TC1 Overflow
        reti                                ;TC0 Overflow
        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
        reti                                ;TWI (I²C) Serial Interface
        reti                                ;Strore Program Memory Redy
;-------------------------------------------------------------------------
; Start, Power ON, Reset
main:   ldi     r16, lo8(RAMEND)
        out     SPL, r16
        ldi     r16, hi8(RAMEND)
        out     SPH, r16
        sbi     DDRB, 0
;-------------------------------------------------------------------------
mainloop: wdr
          ldi    r16, 0b00000001           ; LED ON
          out   PORTB, r16
          rjmp mainloop
;-------------------------------------------------------------------------
```

Bild 23: Der komplette Quelltext unseres Programms „LEDON“

Eintragen des Projekttitels und weiterer projektbezogener Angaben in den Programmkopf (Abbildung 22).

Danach wird das Programm-Grundgerüst um die besprochenen Programmteile zur Initialisierung und zum Hauptprogramm ergänzt. Der Quellcode des Programms sieht dann aus wie in Abbildung 23 gezeigt. Sehr angenehm für den Einsteiger ist, dass jede Quellcodeeingabe von einem (abschaltbaren) Assistenzfenster begleitet wird, das alle wichtigen Parameter erläutert und z. B. auch Vorschläge für die Registerwahl macht (siehe auch Abbildung 8 im Teil 1).

Kompilieren, Linken, Brennen

So komplettiert, wird der Quelltext nun mit dem Assembler kompiliert und gelinkt. Ergebnis ist ein im zuvor gewählten Projektverzeichnis abgelegtes File „LEDON.hex“, das nun noch auf den ATmega-Controller des myAVR-Boards zu übertragen ist. Jede Aktion wird, wenn sie fehlerfrei verlaufen ist, mit einer Ende-Meldung quittiert.

Jetzt kommt endlich das myAVR-Board ins Spiel. Es wird über das mitgelieferte Parallelportkabel (Programmierkabel) mit dem LPT1-Port des Rechners verbunden (wir

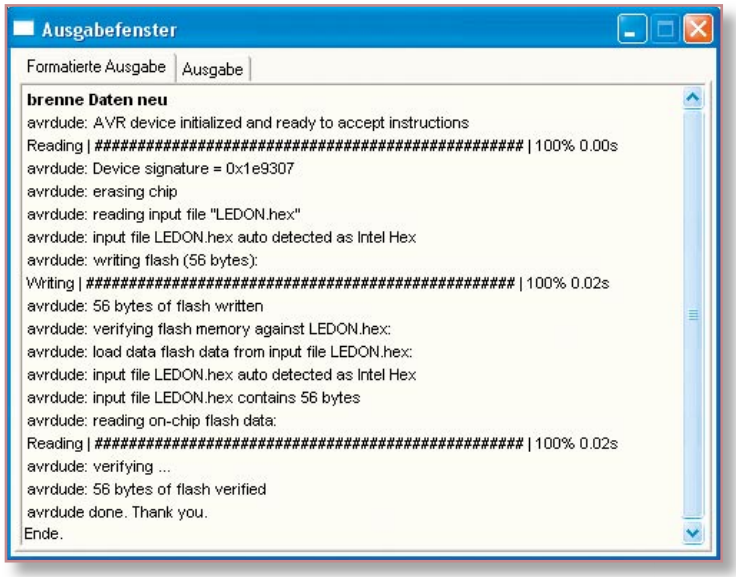


Bild 24: Im Ausgabefenster werden alle Vorgänge beim Brennen des AVR-Flash kontrolliert.

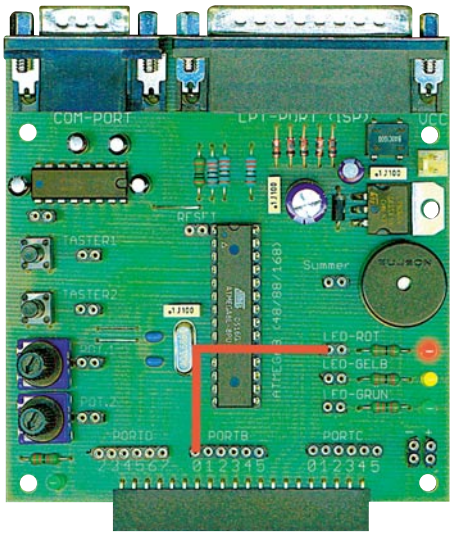


Bild 25: Schnell verdrahtet – so wird PB0 mit der roten LED verbunden.

beziehen uns hier auf diese LPT-Version, bei der USB-Version ist die Vorgehensweise in der mitgelieferten Dokumentation beschrieben). Der LPT-Port des Rechners kann hier auch die erforderliche Betriebsspannung liefern, dies erkennt man am Aufleuchten der grünen Betriebs-LED.

Ist die Verbindung hergestellt, geht es ans „Brennen“ des eben hergestellten Hex-Files in den Flash-Speicher des AVR. Nach Anwahl der Schaltfläche „Brennen“ startet der Vorgang unter Protokollierung in einem Ausgabefenster (Abbildung 24). Erscheint die hier zu sehende Meldung, sind Übertragung und Verifizierung erfolgreich verlaufen.

Geht's? Der erste Testlauf

Nun kommt der spannende Moment: Läuft unser erstes Programm auf dem AVR?

Dazu trennt man zunächst das Programmierkabel vom myAVR-Board und verbindet mittels einer der mitgelieferten Patchleitungen den Buchsenleistenkontakt von Port B0 mit der Buchsenleiste einer der LEDs, in unserem Beispiel (Abbildung 25) der roten LED. Schließt man nun eine 9-V-Spannungsquelle an das myAVR-Board an, leuchtet die LED – unser Programm funktioniert!

Jetzt ist der Experimentierdrang geweckt! Bevor wir in der nächsten Folge u. a. zur Eingabeprogrammierung kommen, probieren Sie doch zunächst einmal aus, wie Sie andere Ports entsprechend als Ausgabeports programmieren oder gar mehrere LEDs gleichzeitig zum Leuchten bringen! Schon nach kurzer Zeit gehen einem die kleinen Programmieralgorithmen in Fleisch und Blut über und man ist gespannt auf die nächsten Aufgaben …