Ausgewählte Kapitel eingebetteter Systeme
Die Atmel AVR Mikrocontrollerfamilie

Johannes Bauer

10. Mai 2006

1 Einführung

1.1 Grundsätzliche Überlegungen

In viel mehr Geräten, als man eigentlich annehmen würde, befinden sich heutzutage Mikrocontroller. In allen Bereichen, in denen Regel- und Steueraufgaben zu erledigen sind oder zeit- oder ereignisgesteuert Schaltvorgänge ausgelöst werden sollen, sind Mikrocontroller das Mittel der Wahl. Zusätzlich werden sie von Entwicklern gegenüber diskret aufgebauten Steuerplatinen aufgrund der einfacheren Wartbarkeit, den komplexen Debugmöglichkeiten und nicht zuletzt wegen der Platzersparnis auf einer Platine bevorzugt.

Die Firma Atmel bietet mit Ihrer AVR-Mikrocontroller-Serie eine breite Palette an 8-Bit RISC MCUs[1] an. Der Aufgabenbereich reicht von einfachsten Zeitschaltuhren bishin zu komplexen Produkten, die beispielsweise Messwerte protokollieren und diese auf Flash-Bausteine schreiben. Auch für Heimanwender stellen AVR-Mikrocontroller ein beliebtes Gebiet dar, da die meisten gebräuchlichen Controller der ATtiny und ATmega-Serie alle Bedingungen erfüllen, die es selbst Laien ermöglicht, in kürzester Zeit mit AVRs vertraut zu werden:

- Nahezu alle ATtiny und ATmega-Prozessoren sind in dem gut handlötbaren DIP-Gehäuse[2] beziehbar.
- Es gibt viele Distributoren, von denen die Mikrocontroller auch in Kleinstmengen bezogen werden können.
- Die Prozessoren sind erschwinglich - einfache Typen fangen preislich schon bei unter einem Euro an, der komplexeste ATmega liegt in der Kategorie um etwa zehn Euro.
- Eine extrem vielseitige, komplexe OpenSource-Toolchain auf Basis der GCC[3] ist für AVRs vorhanden, welche die Bereiche Assemblierung, Kompilierung, Simulation, Debugging und den Flash-Vorgang abdeckt.
- Keine teuere Hardware muss angeschafft werden, um mit den Prozessoren zu arbeiten. Ein einfaches Flash-Gerät besteht nur aus wenigen Bauteilen und ist für unter einem Euro selbst zusammenzubauen.

1.2 Überblick über die Familie

Die AVR-Prozessorfamilie umfasst drei große Klassen von Prozessoren: die AT90S-Varianten, die ATtiny und die ATmega. Die älteste der Familien, die AT90S, ist bereits von Atmel abgekündigt und sollte nicht mehr für Neuentwicklungen benutzt werden. Dennoch sind diese

[1] Micro Controller Unit
[2] Dual Inline Package
[3] GNU Compiler Collection

Prozessoren gerade aufgrund des geringen Preises für Heimanwender attraktiv und zum Lernen gut geeignet. Die AT90S-Familie sollte, so Atmel, von den ATtiny und ATmega-Familien abgelöst werden. Die ATtinys sind hierbei die leistungsschwächeren Prozessoren mit in der Regel 8 Pins, die besonders auf Energiesparmechanismen optimiert sind. Sie sind also prädestiniert für batteriebetriebene Geräte, in denen nicht viel Platz für große Hardware vorhanden ist. Die ATmegas hingegen sind leistungsstärkere Prozessoren mit 28 bis 64 Pins, viel Flash (bis zu 128kB) und interessanter Zusatzhardware (wie Hardware-PWMs[4], USARTs[5], USB-Controllern, CAN-Bus-Controller, ADCs[6], I^2C-Bus und vielem mehr).

Bis auf sehr wenige Ausnahmen sind alle Prozessoren voll binärkompatibel zueinander. Eine solche Ausnahme bildet beispielsweise der älteste Baustein der AT90S-Familie, der AT90S1200. Dieser verfügt über kein integriertes SRAM und kann demnach natürlich auch keine Opcodes ausführen, welche beispielsweise für Speicherzugriffe gedacht sind. Über diese Opcode-Kompatiblität hinaus sind auch die Pins für das ISP-Verfahren[7] für alle Prozessoren gleich: dies ist ein enormer Vorteil, da beim anfertigen einer Leiterplatte mehrere Footprints für verschiedene Prozessoren vorgesehen werden können. Somit kann, falls sich herausstellen sollte, dass die Speicherkapazität des einen Prozessors nicht mehr genügt, dieser einfach durch ein größeres Pendant ausgetauscht werden.

Die neueren AVRs bieten darüberhinaus die Möglichkeit eines sogenannten debugWIRE: Ein externes Debugging-Tool wird lediglich an den `/RESET`-Pin des Controllers und die Versorgungsspannung angeschlossen. So kann ein Host (also beispielsweise ein PC) den Prozessor zu jedem beliebigen Zeitpunkt anhalten, die Register analysieren oder verändern und die MCU somit wirkungsvoll debuggen. Besonders praktisch hierbei ist, dass tatsächlich kein weiterer Portpin für die Verwendung des Debugging-Interfaces verbraucht werden muss.

---

[4]Pulse Width Modulation, also die Ausgabe eines pulsbreitenmodulierten Signals mit einstellbarem Dutycycle

[5]Universal Synchronous/Asynchronous Receiver and Transmitter, beispielsweise bei RS232 verwendet

[6]Analog Digital Converter

[7]In System Programming, also das Flashen des Prozessors *in* der fertigen Schaltung

2 Hardware

2.1 Features

Ein typisches Beispiel für einen flexiblen Prozessor der AVR-Familie ist der ATtiny2313 (gedacht als direkten Ersatz für den AT90S2313). Er verfügt über 2kB Flash-Speicher (in den das Programm geladen wird - auch Programmspeicher genannt), 128 Bytes EEPROM (in dem beispielsweise Konfigurationsdaten des Programms liegen können) und 128 Bytes SRAM (Stack, Variablen). Sowohl FLASH, EEPROM als auch SRAM sind bei allen AVR-Controllern direkt mit auf den Mikrocontroller-IC aufgebracht. Somit werden externe Komponenten eingespart. Das FLASH kann laut Datenblatt mindestens 10.000 Mal programmiert werden, der EEPROM 100.000 Mal.

Die AVRs sind ein Beispiel für Prozessoren, die nach der Harvard Architektur gefertigt werden: Im Gegensatz zur Von Neumann-Architektur gibt es separate Speicherbereiche für SRAM, Datenspeicher (EEPROM) und Programmspeicher (FLASH). Das eigentlich auszuführende Programm liegt also nicht im RAM des Controllers, sondern einzig im FLASH-Speicher. In der Praxis führt dies leider manchmal zu recht obskuren Problemen, da AVRs mit dem Ziel entwickelt wurden, hauptsächlich in der Programmiersprache C programmiert zu werden. C bietet jedoch keine explizite Trennung der Speicherbereiche an; es ist also dem Compiler zunächst nicht klar, *welcher* Speicherbereich gemeint ist, wenn mit Pointern gearbeitet wird. Beim GCC referenzieren Pointer standardmäßig den SRAM-Speicher - ungewohnt, wenn mit String-Konstanten gearbeitet wird (die eigentlich einen bessern Platz im EEPROM finden würden).

Jeder AVR verfügt über sogenannte Ports. Ein Port ist eine Gruppe von maximal 8 I/O-Leitungen über die der Prozessor mit der Außenwelt kommunizieren kann. Der ATtiny2313 beispielsweise verfügt über 18 programmierbare I/O-Leitungen an 3 Ports: `PORTB` stellt 8 Leitungen in den Bits `PB0` bis `PB7`, `PORTD` stellt 7 Leitungen und `PORTA` die restlichen 3. Jede einzelne Leitung kann hierbei als Eingang oder Ausgang konfiguriert werden. Werden Leitungen als Ausgang konfiguriert bieten die Mikrocontroller zusätzlich die Möglichkeit an, interne Pullup-Widerstände zu aktivieren. Somit hat ein Ausgang immer einen definierten Pegel (`HIGH`), auch wenn keine Last an dem Portpin hängt. Diese sind allerdings relativ schwach (ca. 20-50kΩ) und müssen so je nach Anwendungsfall eventuell durch externe Komponenten ergänzt werden. Vorteil der internen Pullup-Widerstände ist eine Reduktion externer Bauteile.

Die meisten Portpins der AVRs sind mit mehrfachen Funktionen belegt: So sind beispielsweise beim ATtiny2313 die Portpins `PB0` und `PB1` Eingänge für den internen ADC. Die ADC-Funktion ist hier - wie standardmäßig bei allen Spezialfunktionen - deaktiviert. Will man einen oder beide Portpins als Eingang für den internen ADC nutzen, so muss man einfach im Programm einige Register so konfigurieren wie im Datenblatt angegeben. Genauso teilt sich andere interne Hardware Portpins: das USART benutzt Pins `PD0` und `PD1`, externe Interrupts können an `PD2` und `PD3` abgefragt werden.

2.2 Grundschaltung

Sämtliche AVR-Prozessoren benötigen zur Programmierung eine Versorgungsspannung, die sich zwischen 2,7V und 5,5V bewegen darf (genauere Daten sind den elektrischen Charakteristika im Datenblatt des Prozessors zu entnehmen). In der Regel werden hier für einfachere Anwendungen stabilisierte 5,0V verwendet, bei professionellen Anwendungen kommt 3,3V-Technologie zum Einsatz, weil hierbei weniger Verlustleistung entsteht, als bei 5,0V-Technologie. Für die Spannungsversorgung alleine werden also zwei Leitungen (GND und Vcc verwendet). Zusätzlich muss die `/RESET`-Leitung[1] des Prozessors beim Flashen angesteuert werden. `/RESET` liegt üblicherweise über einen 10kΩ-Widerstand an Vcc, sodass der Prozessor im Normalbetrieb läuft, anstatt einen Reset durchzuführen. Zusätzlich empfiehlt Atmel einen 100nF-Kondensator nach Masse, damit durch den entstehenden RC-Tiefpaß bei Aktivierung der Schaltung der Prozessor kurzzeitig[2] das Signal zum Reset bekommt. Darüberhinaus werden drei weitere Signalleitungen mit dem Prozessor verbunden: `MISO`, `MOSI` und `SCK`. Diese sind für die Kommunikation von der MCU zum Host und zurück verantwortlich. Den Programmiertakt `SCK` stellt hierbei immer der Host.

Nun da sämtlichen Signalleitungen erklärt sind, muss noch erwähnt werden, welche zusätzliche externe Beschaltung notwendig ist: zunächst einmal ist es generell bei Digitalbausteinen empfehlenswert, einen 100nF Abblockkondensator so nah wie möglich am Baustein zwischen Vcc und GND anzubringen. Bei den AVR MCUs ist es obligatorisch: fehlt der Kondensator kann es zu sehr seltsamen Störungen während Programmier- und Betriebsphase kommen. Der Einfachheit halber gibt es sogar extra DIP-Sockel, welche diesen 100nF-Blockkondensator bereits integriert haben.

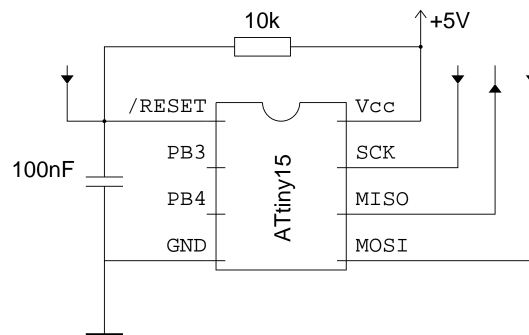

Abbildung 2.1: ATtiny15 in Grundschaltung

Als zusätzliche externe Beschaltung muss bei älteren AT90S-Modellen ein Quarz an die Pins `XTAL1` und `XTAL2` angebracht werden und zwei Bürdekondensatoren[3] gegen Masse geschaltet werden. Die neueren Varianten ATtiny sowie ATmega bringen alle einen internen RC-Oszillator mit, der zwar nicht die Präzision eines Quarzes mitbringt, jedoch die externe Beschaltung um eben diese drei Bauteile (zwei Kondensatoren, ein Quarz) verringert.

[1] sprich: „not RESET", also ein invertiertes `RESET`-Signal. Liegt hier ein `HIGH`-Pegel an, arbeitet der Prozessor im Normalbetrieb, bei `LOW` geht er in den Reset/Programmiermodus

[2] $3 \cdot \tau = 3 \cdot R \cdot C = 3\text{ms}$

[3] in der Kapazität der Bürdekapazität des Quarzes angepasst, normalerweise sind das 27-33pF

2.3 Kritik

Nicht alle Features, die eine moderne CPU bietet können in einem kleinen Mikroprozessor integriert werden. Hier sollen die wichtigsten Punkte aufgeführt werden, um abzuschätzen, ob ein AVR für ein bestimmtes Problem nicht etwa doch die falsche Wahl ist:

- Die AVRs verfügen über keinen Hardwaredividierer. Damit müssen also die Integerdivision und die Modulo-Operation jeweils in Software emuliert werden. Dies ist um Größenordnungen langsamer als ein „echter“ Dividierer.
- AVRs bieten keine Gleitkommaunterstützung an. Sämtliche Gleitkommaoperationen müssen also wiederum in Software nachgebildet werden. Dies ist nicht nur langsam, sondern benötigt auch noch viel Flash-Speicher (die Gleitpunktzahl-Bibliothek der GCC benötigt mit den vier Grundrechenarten etwa 3kB).
- Kein Prozessor der AVR-Familie bietet DMA[4] an. Dies ist insbesondere dann tragisch, wenn größere Datenmengen (also in der Größenordnung von 0,5-1,0 MB/s) zwischen dem Prozessor und einem anderen Gerät ausgetauscht werden sollen, da der Prozessor dann vollständig mit dem Lesen und Abspeichern der Daten beschäftigt ist.
- Die Prozessoren verfügen über keine MMU[5]. Es kann also keine Separierunng der Speicherbereiche einzelner Tasks gewährleistet werden.
- Die Handhabung der sogenannten Fuses, also der in das EEPROM einbrennbaren Konfigurationsbits ist kritisch: So ist es beispielsweise möglich, die Fuses des Prozessors so zu manipulieren, dass kein SPI[6] mehr möglich ist, oder dass er *immer* versucht, einen externen Quarz zu benutzen. Bei anderen Prozessorfamilien ist dies besser gelöst, da man zum Beispiel durch eine MCU-Instruktion den Quarz während des laufenden Betriebs aktivieren kann. So ist eine SPI-Programmierung *immer* möglich und die MCU muss nicht aus einer etwaigen Schaltung entfernt werden, wenn die Fuses verstellt wurden.
- Interrupts sind im AVR nicht beliebig priorisierbar, sie werden immer in Reihenfolge der Interruptvektoren abgearbeitet. Damit ist die Reihenfolge in der Hardware fest verdrahtet. Dies kann zu Problemen führen, wenn „wichtige“ Interrupts in der Vektortabelle an einer höheren Stelle plaziert sind als „unwichtige“ Interrupts.
- Die Firmenpolitik von Atmel sieht es leider vor, neue Prozessoren schnell auf den Markt zu bringen und bereits relativ junge Modelle schnell abzukündigen. So wurde die AT90S-Familie bereits nach gut drei Jahren nach ihrer ersten Ankündigung als obsolet markiert und verkündet, dass sie nicht mehr für Neuentwicklungen eingesetzt werden sollte. Dies birgt für Entwickler professioneller Hardware enorme Schwierigkeiten: Das unkalkulierbare Risiko, möglicherweise ein Produkt neu entwickeln zu müssen, weil es sich auf einen Prozessor verlassen hat, der abgekündigt wurde, geht niemand gerne ein. Außerdem gibt es für die AVRs keine Second Source, also einen Hersteller, der Klone der Prozessoren herstellt. Um hier Entwicklungssicherheit zu haben, werden lieber relativ alte Prozessoren verwendet wie der über 25 Jahre bestehende 8051 - diese werden auch von dutzenden Firmen (interessanterweise unter anderem auch von Atmel) gefertigt.

[4]Direct Memory Access
[5]Memory Management Unit
[6]Serial Programming Interface

3 Software

3.1 Register

Die AVR MCUs verfügen, wie man für eine RISC-Architektur erwarten würde, über die relativ hohe Anzahl von 32 allgemein verwendbaren Registern (`r0`-`r31`). Diese haben alle eine Breite von 8 Bit. Sechs Register davon haben eine spezielle Bedeutung: Jeweils zwei dieser sechs können als 16-Bit Pointer verwendet werden und werden dann das `X` (`r26:27`), `Y` (`r28:29`) und `Z`-Register (`r30:31`) genannt. Darüberhinaus gibt es 64 Register, die für I/O und Kontrollfunktionen der MCU verwendet werden. Hier wird beispielsweise die Konfiguration der I/O-Ports gespeichert, die Condition Codes oder die Einstellung der Timer.

Das wichtigste Register der AVR-Prozessoren ist das Kontrollregister `SREG`. In `SREG` werden eben die gerade angesprochenen Condition Codes und das Bit gespeichert, welches global steuert, ob Interrupts zugelassen oder gesperrt sind. Die Condition Codes werden von konditionalen Sprungbefehlen verwendet, um beispielsweise einen Codeteil anzuspringen, falls das Ergebnis einer vorhergehenden Subtraktion negativ oder Null war.

3.2 Ports

Um über die Ports mit der Außenwelt zu kommunizieren, gibt es für jeden Port drei Register: `PORTx`, `PINx` und `DDRx`. Standardmäßid sind alle Ports nach einem Reset hochohmig (tristated). Um dies nun zu ändern müssen die `PORTx` und `DDRx` Register gesetzt werden:

DDRx	PORTx	Funktion des Portpins
0	0	Input (hochohmig)
0	1	Input mit internem Pullup
1	0	Output: `LOW`
1	1	Output: `HIGH`

Die Syntax für das Programmieren der Ports unterscheidet sich hier von Compiler zu Compiler - hier wird jedoch die des avr-gcc verwendet. Der Grund ist recht einfach zu erklären: die `PORT`-Register sind keine Memory-Mapped I/O Register, sondern werden über die CPU-Instruktionen `out` und `in` gesetzt. Der gcc-Compiler vermittelt jedoch dem Programmierer (aufgrund der höheren Intuitivität) den Eindruck, es seien Memory-Mapped I/Os.

In einem C-Programm in dem die Pins `PB4` und `PB6` als Inputs und `PB2` als Output konfiguriert werden sollte, sähe das also so aus (nur `PB4` benutzt den internen Pullup, der Output soll auf `LOW` gesetzt werden):

```
DDRB &= ~((1<<PB4) | (1<<PB6));
DDRB |= (1<<PB2);
PORTB &= ~((1<<PB6) | (1<<PB2));
PORTB |= (1<<PB4);
```

Nun noch die Erklärung zu `PINx`: Dieses Register spiegelt den Eingang eines Ports wieder. Um beispielsweise zu prüfen, ob das Bit `PB6` gesetzt ist, kann folgendes Konstrukt benutzt werden:

```
if (((PINB & (1<<PB6)) != 0) {
    /* Einang ist gesetzt (HIGH) */
    ...
}
```

3.3 Interrupts

Interrupts können dazu benutzt werden, Polling zu vermeiden. So muss also beispielsweise nicht in einer Endlosschleife gewartet und ständig ein `PINx`-Signal abgefragt werden, sondern es kann ein entsprechender Portpin ausgewählt werden, der Hardware-Interrupts unterstützt. Sobald eine bestimmte Pegeländerung an diesem Portpin geschieht (also wahlweise eine `LH`-Flanke, `HL`-Flanke oder beides), wird der Programmfluss angehalten und der entsprechende Interruptvektor ausgeführt. Einen Interrupthandler für den `avr-gcc` zu definieren ist denkbar leicht:

```
ISR(INT0_vect) {
    /* Hier kommt der Code für den Handler */
}
```

Auch Software-Interrupts sind mit AVRs möglich: So kann derjenige Portpin, für den ein entsprechender Handler installiert ist (beim ATtiny2313 wäre das `PD2` für Interrupt 0) einfach per Software in einen entsprechenden Zustand geschaltet werden:

```
PORTD &= ~(1<<PD2);
PORTD |= (1<<PD2);
```

3.4 Timer

Timer sind besondere Interrupts: Jeder AVR hat einen internen Zähler (entweder 8 oder 16 Bit breit), der in einem definierten Takt inkrementiert wird. Die Taktgeschwindigkeit kann per Software bestimmt werden und liegt bei wahlweise $\frac{1}{1024}, \frac{1}{256}, \frac{1}{64}, \frac{1}{8}$ des Prozessortaktes. Dieser Prescaler kann auch vollständig deaktiviert werden, womit dann eine Timer-Inkrementierung zu jedem Taktzyklus stattfindet.

Sollte nun der Zähler überlaufen (also wieder auf Null zurückfallen), wird ein `TIM0_OVF`-Interrupt ausgelöst. Damit nun das Zeitintervall genauer steuerbar ist, in dem Timer-Interrupts ausgelöst werden, bedient man sich eines Tricks: Man rechnet vorher aus, wieviele Zyklen (nach dem Prescaler) durchlaufen werden sollen. Dann lädt man den errechneten Wert in dem Interrupthandler wieder. Zunächst einmal die Initialisierung des Timers:

```
TCCR0A = 0x00;           /* Normaler Timer, kein PWM */
TCCR0B = 0x04;           /* 1/256stel Prescaler */
TCNT0  = 178;
TIMSK = (1<<(TOIE0));    /* Timer0 Overflow Interrupt Enable */
```

Zur Verdeutlichung ein Beispiel: Sei der Prozessortakt 20 MHz und es würde ein Interrupt jede Millisekunde gewünscht. Den Prescaler würde man auf $\frac{1}{256}$ konfigurieren, somit wäre der Takt nach dem Prescaler $\frac{20\text{MHz}}{256} \approx 78,1\text{kHz}$. Für die Wahl des optimalen Prescalers muss ein

wenig herumprobiert werden: Teilt man den Takt zu stark herunter können zwar längere Intervalle getimt werden, die Granularität wird allerdings geringer. Bei einem zu schwach geteilten Takt ist zwar die Auflösung höher, möglicherweise ist jedoch die Zeit bis zu einem vollen Registerüberlauf zu kurz: Hierbei können natürlich auch nur recht kurze Intervalle getimt werden.

Um auf das Beispiel zurück zu kommen: Eine Prescaler-Einstellung von $\frac{1}{256}$ entspricht nun einer Zählerinkrementierung alle $12,8\mu$s. Somit werden also $\frac{1\text{ms}}{12,8\mu\text{s}} \approx 78$ Inkrementierungen benötigt, um eine Millisekunde voll zu bekommen. Damit der 8-Bit Timer nach 78 Inkrementierungen auch überläuft, muss also im Interrupthandler der Wert $256 - 78 = 178$ in das Timer-Counter Register (`TCNT0`) geladen werden, damit der Timer korrekt funktioniert:

```
ISR(TIMER0_OVF_vect) {
    /* Timer ausgelöst */
    TCNT0 = 178;
}
```

3.5 PWM

Die Timer der AVRs können durch setzen eines Bits auch als PWM-Ausgänge geschaltet werden: So kann relativ einfach ein digitaler Wert in einen analogen umgesetzt werden. An den Ausgang des PWM (beim ATtiny2313 wäre das `PB2` beziehungsweise `PB3`) muss lediglich ein entsprechend der Frequenz des Signals angepasster RC-Tiefpaß geschaltet werden. Ein klein dimensioniertes RC-Glied hat hier zur Folge, dass sich das Signal schneller am Zielwert einpendelt - jedoch wird damit auch die Ripplespannung (also der Wechselspannungsanteil der Ausgangsspannung) größer.

Im Prinzip funktioniert ein PWM wie ein Timer (und wird auch vom Prescaler exakt so konfiguriert). Zusätzlich muss jedoch noch ein Compare-Wert übergeben werden: Der Timer-Counter wird nun stetig inkrementiert. Kommt es zum *Compare Match* (also wenn der Zählerstand den Compare-Wert erreicht), so wird der Output auf `LOW` geschaltet. Beim *Overflow* wird das Ausgabesignal auf `HIGH` gezogen. Somit können also bei Comparewerten von 0-255 Signale erzeugt werden, die exakt den $\frac{\text{Compare}}{255}$ Teil einer PWM-Periode `LOW` und die restliche Zeit `HIGH` sind. Wird der Wert mitten in einer Periode geändert, so tut der AVR das, was man „intuitiv“ als richtig empfinden würde: Er aktualisiert den Wert erst mit Beginn der nächsten Zählperiode (phasenrichtiger PWM).

3.6 USART

Durch das USART können auf direktem Weg zwei AVRs miteinander verbunden werden. Viel interessanter, gerade bei den ersten Entwicklungen mit AVRs, ist jedoch das Verbinden des AVRs mit einem PC. Hierfür bietet sich die serielle Schnittstelle an (die meist ein UART des Typs 16550 beinhaltet). Eine kleine Hürde gibt es dabei noch zu nehmen: Für den AVR ist eine logische `0` eine Spannung von 0-0,8V, eine logische `1` eine Spannung von 2,0-5,0V. Bei RS232 hingegen ist eine logische `0` eine Spannung von 3-12V, eine `1` hingegen eine Spannung von -12 bis -3V.

Da meist auch keine negative Versorgungsspannung vorhanden oder gewünscht ist, gibt es eine sehr einfache Lösung: Den Maxim MAX232, der universelle „Dolmetscher“-IC zwischen RS232 und einem UART mit TTL-Pegeln[1]. Er wird mit fünf Elektrolytkondensatoren beschal-

[1] Transistor-Transistor Logic

tet und benutzt diese als Ladungspumpe und erzeugt somit -8 und 8V ohne weitere Probleme. Der Ausgang des MAX232 kann direkt an ein serielles Kabel gelötet werden, der Eingang direkt an den AVR.

Um das USART zu konfigurieren muss man sich zuallererst auf eine Baudrate einigen. Ein Standardwert für den PC beträgt 115200 Baud. Diese sollen für die Konfiguration verwendet weren. Da RS232 ein taktloses Protokoll ist, muss die Baudrate möglichst exakt stimmen, damit es nicht zu Übertragungsfehlern kommt. Der AVR gewinnt seinen USART-Takt aus dem Prozessortakt, nachdem dieser einen separaten Prescaler durchlaufen hat. Wie stark der Prozessortakt heruntergeteilt wird erfährt der AVR durch das Register `UBRR`. Dieses muss laut Datenblatt nach folgender Formel berechnet werden:

$$\text{Baudrate} = \frac{f\text{OSC}}{16(\texttt{UBRR}+1)}$$

oder umgeformt

$$\texttt{UBRR} = \frac{f\text{OSC}}{16 \cdot \text{Baudrate}} - 1$$

Damit die Berechnung des `UBRR` möglichst glatt aufgeht gibt es spezielle „krumme“ Quarze: Diese sind schon für die Verwendung mit einem UART und dem vorhandenen Prescaling-Problem konzipiert worden. Verwendet man also einen 14,7456 MHz-Quarz kann für eine Datenübertragung von 115.200 Baud ein `UBRR`-Prescaler von 15 verwendet werden.

In dem UART-Kontrollregister `UCR` kann zusätzlich noch eingestellt werden, ob möglicherweise nur der Empfangsteil oder nur der Sendeteil des UART verwendet werden soll. Überdies können in diesem Register Interrupts für Sende- oder Empfansereignisse konfiguriert werden. Der UART kann bis zu drei Interrupts auslösen: „Daten gesendet“, „Daten empfangen“, „Bin bereit zum Senden“. Im Folgenden soll nur der Interrupt verwendet werden, wenn Daten empfangen wurden:

```
UBRR = 15;
UCR = (1<<TXEN) | (1<<RXEN) | (1<<RXCIE);
```

Sobald nun Daten empfangen wurden, löst der `UART_RX`-Interrupt aus. Um das empfangene Byte zu verwerten, muss einfach nur lesend auf das UART-Datenregister `UDR` zugegriffen werden:

```
ISR(UART_RX_vect) {
    /* Ein Byte wurde empfangen */
    LastByte = UDR;
}
```

Fast genauso einfach geht das Schreiben auf das UART: Durch schreibenden Zugriff auf `UDR`. Hier muss lediglich vorher ein Flag überprüft werden, das signalisiert, ob Sendebereitschaft besteht:

```
while ((USR & (1<<UDRE)) == 0) { };
UDR = WriteByte;
```

3.7 EEPROM

Das Schreiben und Lesen von Konfigurationsdaten, die im EEPROM liegen, wird von avr-gcc vollständig gekapselt, sodass nicht mehr klar ist, was eigentlich für Operationen geschehen. Hierfür werden die Funktionen `eeprom_read_byte` und `eeprom_write_byte` angeboten:

```
Wert = eeprom_read_byte((char*)123);
Wert++;
eeprom_write_byte((char*)123, Wert);
```

Um auch andere Compiler nutzen zu können, ist es hilfreich zu verstehen, was eigentlich passiert. Zunächst muss in dem EEPROM-Kontrollregister geprüft werden, ob der letzte Schreibvorgang (der im Vergleich zum MCU-Takt wirklich lange dauert - ganze 3,4ms) erfolgreich beendet wurde. Danach wird die Adresse übergeben und der Lesevorgang gestartet:

```
while ((EECR & (1<<EEWE)) != 0) { };
EEAR = 123;
EECR |= (1<<EERE);
Wert = EEDR;
```

Der Schreibvorgang läuft in etwa genau so ab, nur dass es eine zusätzliche Sicherheitsmaßnahme gibt: Um versehentliches Schreiben in das EEPROM zu verhindern, muss zuerst im EEPROM-Kontrollregister das Bit `EEMPE` (EEPROM Master Program Enable) gesetzt werden. Danach muss innerhalb von vier Taktzyklen das Bit `EEPE` (EEPROM Program Enable) gesetzt werden. Geschieht dies nicht, wird `EEMPE` durch die Hardware wieder auf `0` gesetzt. Die beiden Bits dürfen *nicht* in derselben Operation gesetzt werden. Ein Schreibvorgang sieht dann so aus:

```
while ((EECR & (1<<EEWE)) != 0) { };
EEAR = 123;
EEDR = Wert + 1;
EECR |= (1<<EEMPE);
EECR |= (1<<EEPE);
```

Um nun noch einmal kurz auf das Problem der getrennten Speicherbereiche zurück zu kommen, das nun durch das folgende Beispiel verständlicher werden sollte: Angenommen wir haben eine Funktion, die einen String über das USART ausgibt:

```
void schreibe_sram_string(char *string) {
    unsigned char i = 0;
    while (string[i] != 0) {
        while ((USR & (1<<UDRE)) == 0) { };
        UDR = string[i];
        i++;
    }
}
```

Zusätzlich sei ein String im EEPROM definiert:

```
#define EEPROM  __attribute__ ((section (".eeprom")))
static unsigned char GrussFormel[] EEPROM = "Hallo!";
```

Wird nun der direkte Zugriff versucht:

```
schreibe_sram_string(GrussFormel);
```

Dann schlägt dies Fehl: Die Funktion `schreibe_sram_string` erwartet einen `char*` auf einen SRAM-Speicherbereich und interpretiert den übergebenen Zeiger als solchen. Der String liegt jedoch in einer ganz anderen Speicherdomäne! Eine funktionierende Lösung wäre also zum Beispiel:

```
void schreibe_eeprom_string(char *string) {
    unsigned char sram_string[32];
    unsigned char pos, read;
    pos = 0;
    do {
        read = eeprom_read_byte((char*)pos);
        sram_string[pos] = read;
        pos++;
    } while (read != 0);
    schreibe_sram_string(sram_string);
}
```

Diese kann dann auch intuitiv aufgerufen werden:

```
schreibe_eeprom_string(GrussFormel);
```

Referenzen

- „Atmel AVR Document 2543 - ATtiny2313 Data Sheet“:
  http://www.atmel.com/dyn/resources/prod_documents/doc2543.pdf
- „Atmel AVR Document 2467 - ATmega128 Data Sheet“:
  http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf
- „Atmel AVR Document 0856 - AVR Family Instruction Set“:
  http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf
- „avr-libc: AVR Libc User Manual“:
  http://www.nongnu.org/avr-libc/user-manual/