Hardwarenahe Programmierung

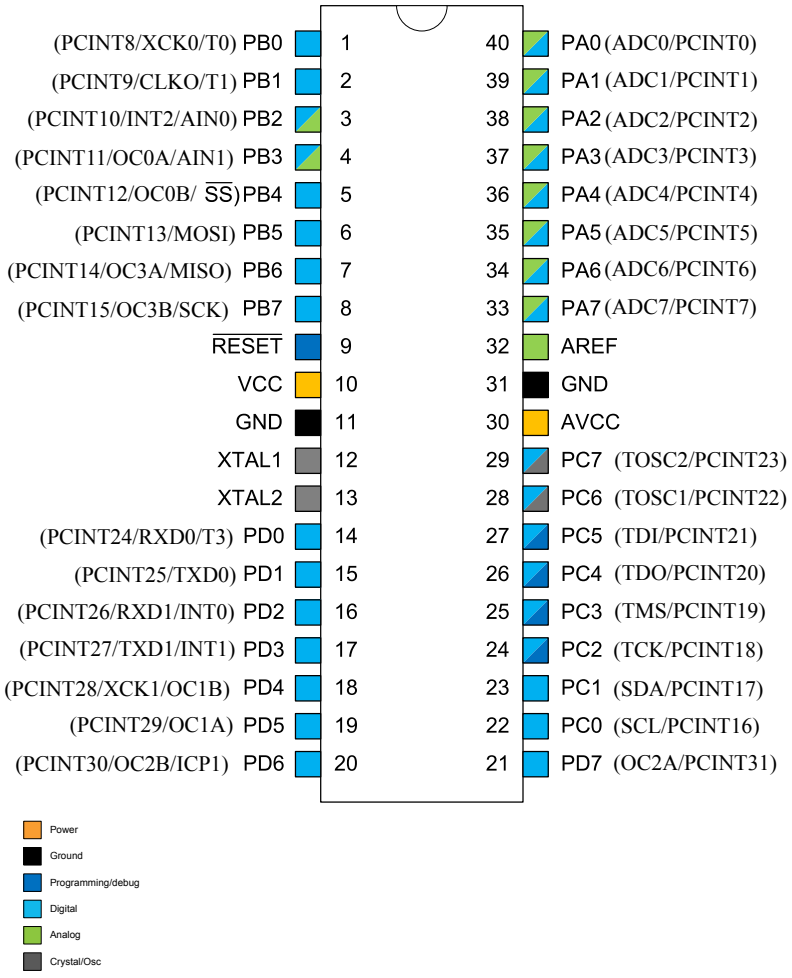
Technische Universität München

Inhalt

1 Allgemein

1.1 Prozessor

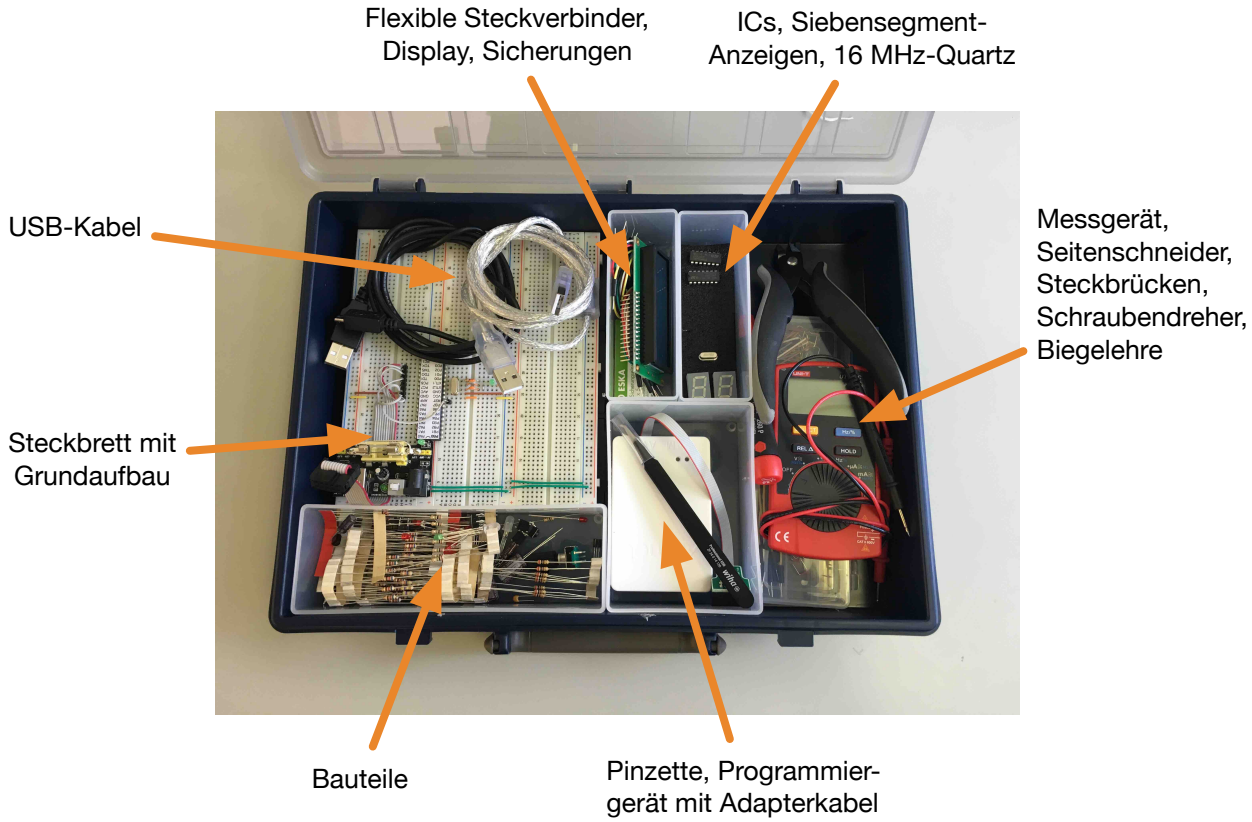
Im Praktikum verwenden wir den Prozessor Atmel AVR 1284p. Der Prozessor ist eine Zweiadress-Maschine und wird von uns mit 20 MHz getaktet und über JTAG programmiert. Die Wortbreite beträgt 8 Bit.
Auf der Webseite finden Sie einen Link zum entsprechende Datenblatt sowie zum Befehlssatz. Laden Sie sich beide Dateien herunter und legen Sie sie im Ordner PICAVR/Datasheets ab. Machen Sie sich mit dem Inhaltsverzeichnis der Dokumente vertraut, so dass Sie wissen, wo Sie nachschlagen müssen. Verwenden Sie am besten einen PDF-Viewer, der Ihnen in einer Seitenleiste das Inhaltsverzeichnis darstellen kann. So können Sie gezielt suchen und schnell navigieren.



Die einzelnen Pins des Prozessors sind mehrfach belegt. Die genaue Funktionsweise der einzelnen Pins können Sie dem Datenblatt entnehmen.

1.2 Sortimentskasten, Grundaufbau

Im Praktikum verwenden Sie einen Sortimentskasten, in dem Sie das für die Versuche notwendige Zubehör finden. Räumen Sie die Kasten bitte immer wieder so ein, wie auf dem Bild gezeigt und zeigen Sie ihn den Tutoren unaufgefordert vor.



Unter dem Steckbrett finden Sie ein Bild, dem Sie den Inhalt der einzelnen Fächer entnehmen können.

1.3 Kompilieren, Ausführen und Assembler-Code einsehen

Wir verwenden den *avr-gcc*-Compiler, um aus C-Code in Maschinencode für den AVR-Prozessor (Objektdatei mit Endung *.o) zu übersetzen und um (ggf. mehrere) Objektdateien mit den benötigten Bibliotheken zu verbinden (linken). Das Resultat liegt zunächst im ELF-Format vor und wird dann mit dem Programm avr-objcopy in eine Datei im Intel-Hex-Format konvertiert (Dateiendung *.hex). Diese Hex-Datei wird mit dem Programm *avrdude* über den Programmieradapter mittels JTAG-Protokoll auf den Prozessor übertragen (‘‘geflasht’’).
Die oben genannten Prozesse werden durch das von uns verwendete Makefile automatisch ausgeführt, so dass sich der Übersetzungs-, Konvertierungs- und Flash-Prozess auf folgende auf der Kommandozeile im Aufgaben-Verzeichnis einzugebende Befehle reduziert:

- `make`: Übersetzt den Quelltext.
- `make clean`: Löscht während des Übersetzungsprozesses erzeugte Dateien.
- `make program`: Übersetzt den Quelltext, lädt die erzeugte Binärdatei in den Prozessor und startet die Programmausführung.
- `make programlsg`: Musterlösung (hex-Datei) auf den Prozessor laden.
- `make show`: Übersetzt den Quelltext und zeigt an, wie der C-Code in Assembler-Befehle übersetzt wurde.
- `make fuses`: Stellt Taktfrequenz u.a. ein. Da das bereits bei allen Prozessoren erledigt ist, sollten Sie diesen Befehl nicht benötigen.

1.4 Byte- und Bit-Zuweisungen

Im folgenden ist *x* ein 8 Bit breites Prozessor-Register bzw. eine Variable, die in einem 8 Bit breiten Prozessor-Register abgelegt ist.

Byte- bzw. Wort-Zuweisungen

Zuweisungen mit ‘‘=’’ wirken immer auf ganze Byte bzw. Datenworte. Beispiel:

- `x = 1` setzt in *x* Bit 0 auf 1 und Bits 1 - 7 auf 0.
- `x = 4` setzt in *x* Bit 2 auf 1 und Bits 0, 1, 3, 4, 5, 6 und 7 auf 0.

Sollen Bits unverändert bleiben, verwendet man in C bitweise logischen Verknüpfungen.

Einzelne Bits setzen

Sollen nur einzelne Bits gesetzt werden und die restlichen Bits unverändert bleiben, verwendet man eine bitweise ODER-Verknüpfung. Beispiele:

- `x |= 1` setzt in *x* Bit 0 auf 1 und lässt Bits 1 - 7 unverändert.
- `x |= 7` setzt in *x* die Bits 0, 1 und 2 auf 1 und lässt Bits 3, 4, 5, 6 und 7 unverändert.

Einzelne Bits löschen

Sollen einzelne Bits gelöscht werden und die restlichen Bits unverändert bleiben, verwendet man eine bitweise UND-Verknüpfung. Beispiele

- `x &= 0xFE` löscht in *x* Bit 0 und lässt Bits 1 - 7 unverändert.
- `x &= ~1` macht das gleiche, ist aber ggf. leichter lesbar/codierbar. Der Tilde-Operator berechnet das Einer-Komplement, d.h. es werden alle Bits invertiert.

1.5 Das Schlüsselwort *volatile*

Compiler versuchen häufig, den zu übersetzenden Code bzgl. bestimmter Kriterien wie Ausführungszeit oder Speicherbedarf zu optimieren. Die Kriterienauswahl sowie den gewünschten Grad der Optimierung kann dem Compiler beim Start als Aufruf-Parameter übergeben werden.
Betrachten Sie folgenden Code:

```
int main(void)
{
    unsigned int i;

    ...

    for(i = 0; i < 20000; i++)
    {
    }

    ...
}
```

Aus Sicht des Compilers passiert in der Schleife nichts und kann somit entfernt werden. Wird zudem die Variable *i* in den restlichen Zeilen nicht verwendet, dann wird die Variable auch keinen Registern zugewiesen. Der Compiler übersetzt dann das Programm so, als ob die Anweisungen ‘‘`unsigned int i`’’ und ‘‘`for(i = 0; i < 20000; i++)`’’ nicht vorkommen würden.
Tatsächlich wollte der Programmierer mit seiner Schleife einfach nur Zeit verbrauchen.
Um dem Compiler nun anzuweisen, die Variable *i* und die for-Schleife trotzdem zu verwenden/übersetzen, kann die Variable *i* mit dem Schlüsselwort *volatile* gekennzeichnet werden:

```
int main(void)
{
    volatile unsigned int i;

    ...
}
```

Der Compiler wird die Variable und die for-Schleife dann nicht weg-optimieren, so dass sich das gewünschte Verhalten ergibt.

1.6 Vorbereitungen

- Legen Sie in Ihrem Home-Verzeichnis einen Ordner *PICAVR* an (Kommandozeilenbefehl: `mkdir PICAVR`)
- Laden Sie die Datei *Makefile.GNUmakefile* herunter und legen Sie diese im Ordner *PICAVR* ab.
- Erstellen Sie den zur Aufgabe gehörigen Aufgabenordner mit dem Kommandozeilenbefehl `mkdir xxx`, wobei `xxx` der Aufgabenname ist, z.B. *01_led*.
- Laden Sie die Datei *Makefile* herunter und legen Sie sie im jeweiligen Aufgabenordner (z.B. *01_led*) ab.
- Laden Sie die Aufgabe herunter und legen Sie sie im Aufgabenordner ab. Die Aufgaben bestehen in der Regel aus einer Angabe-Datei, in der Sie C-Code ergänzen müssen, und einer Binären Lösungsdatei im Intel-Hex-Format. Sie können die Lösungsdatei auf den Prozessor laden, um einen Eindruck davon zu bekommen, wie sich Ihre Lösung verhalten soll.
- Wechseln Sie in den jeweiligen Aufgabenordner (Kommandozeilenbefehl `cd xxx`, wobei `xxx` der Aufgabenname ist).
- Geben Sie an der Kommandozeile folgendes ein:
  - `make`, um Ihr Programm zu übersetzen.
  - `make program`, um Ihr Programm zu übersetzen und danach in den Flash-Speicher des Prozessors zu übertragen und auszuführen.
  - `make programlsg`, um die Musterlösung in den Flash-Speicher des Prozessors zu übertragen und auszuführen.
  - `make show`, um Ihr Programm zu übersetzen und danach anzuzeigen, wie der C-Code in Assembler-Befehle übersetzt wurde.

2 Ein- und Ausgabe über Ports

2.1 Allgemein

Der Prozessor hat vier Ports A, B, C und D, über die Eingaben und Ausgaben getätigt werden können.
Dazu sind für jeden Port *n* drei Register interessant:

- *DDRn*: Mit diesem Register stellen Sie für jeden Pin des Ports *n*, ob dieser als Eingang (Bit in DDRn ist gesetzt) oder als Ausgang (Bit in DDRn ist gelöscht) dienen soll. (DDR = Data Direction Register)
  - Ein gesetztes Bit in DDRn konfiguriert den entsprechenden Pin als Ausgang. Am Pin liegt dann der in PORTn spezifizierte Pegel an.
  - Ein gelöschtes Bit in DDRn konfiguriert den entsprechenden Pin als Eingang.
    - Ist das entsprechende Bit im PORTn-Register gelöscht, so ist der Eingang hochohmig (tristate).
    - Ist das entsprechende Bit im PORTn-Register gesetzt, so wird der Pin über einen Widerstand (5 - 50 k) auf High-Pegel gezogen.
- *PORTn*: Mit diesem Register können Sie Werte (0 oder 1) ausgeben (DDRnx = 1) bzw. den Pull-Up-Widerstand ein-/ausschalten (DDRnx = 0). Bei einer Konfiguration als Ausgang führt ein im Register PORTn gesetztes Bit zu High-Pegel, ein gelöschtes Bit zu Low-Pegel.
- *PINn*: Mit dem PINn-Register können Sie am Prozessor-Pin anliegende Pegel einlesen.

2.2 Beispiele

High-Pegel ausgeben

An Pin 14, d.h. Bit 0 von Port D (= PD0) High-Pegel ausgeben.

```
DDRD = 1;  // 1 = 0000 0001 binär => Bit 0 gesetzt => PD0 ist Ausgang
PORTD = 1; // 1 = 0000 0001 binär => Bit 0 gesetzt => an PD0 1 ausgeben
```

An Pin 16, d.h. Bit 2 von Port D (=PD2) High-Pegel ausgeben.

```
DDRD = 4;  // 4 = 0000 0100 => Bit Nr. 2 gesetzt => PD2 ist Ausgang
PORTD = 4; // an PD0 High-Pegel ausgeben
```

Low-Pegel ausgeben

An Pin14, d.h. Bit 0 von PORT D Low-Pegel ausgeben.

```
DDRD  = 1;  // PD0 ist Ausgang, PD1, PD2, ... PD7 ist Eingang
PORTD = 0;  // alle Bits sind 0 (aber nur PD0 ist Ausgang, s.o.)
```

Pegel einlesen – kein Pull-Up

PD0 als Eingang konfigurieren und Richtung aller anderen Pins unverändert lassen. Kein Pull-Up, d.h. entsprechendes Bit in Register PORTD auf 0 setzen (entspricht Voreinstellung). Pegel an PD0 in Variable *a* einlesen.

```
DDRD &= 0xFE;   // PD0 als Ausgang konfigurieren
PORD &= 0xFE;   // PD0 explizit löschen; nicht notwendig, da
                // PORTD beim Start 0x00 entspricht (Voreinstellung)
a = PIND & 1;   // Pegel am Pin PD0 einlesen
```

Pegel einlesen – mit Pull-Up

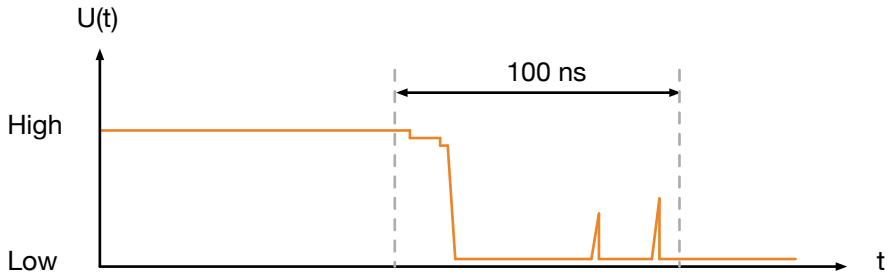
PD0 als Eingang konfigurieren und Richtung aller anderen Pins unverändert lassen. Pull-Up an PD0 einschalten. Pegel an Pin PD0 in Variable *a* einlesen.

```
DDRD &= 0xFE;   // PD0 als Ausgang konfigurieren
PORTD |= 1;     // Pull-Up für PD0 einschalten
a = PIND & 1;   // Pegel am Pin PD0 in Variable a einlesen
```

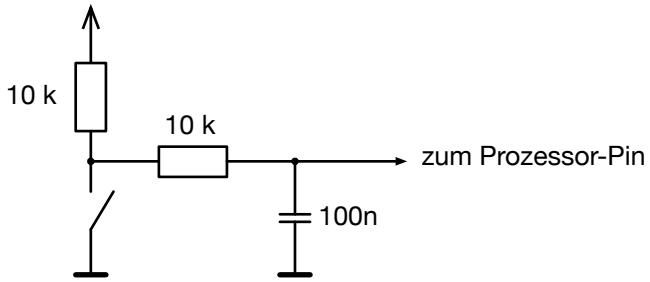
Einschalten des Pull-Up-Widerstands bedeutet, dass der entsprechende Pin intern über einen Widerstand (5 - 50 k) hochohmig auf High-Pegel gezogen wird. Wird der Pin über einen Taster auf Masse gezogen, können so auf einfache Weise Zustandsänderungen am Pin realisiert werden.

2.3 Entprellen

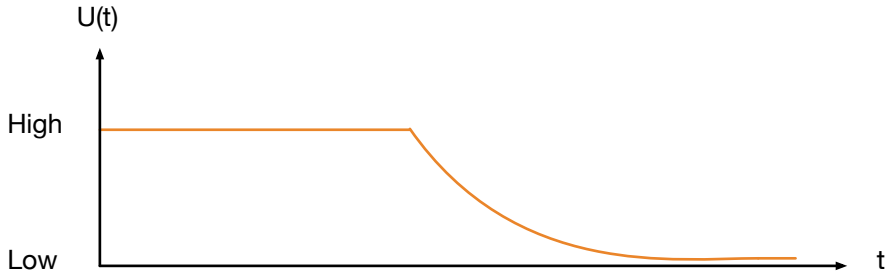
Mit Tastern können Informationen eingegeben werden. Im Moment des Herunterdrücken eines Tasters ist die Verbindung zwischen den Kontakt gebenden Metallen nicht ‘‘ganz sauber’’, d.h. der Zustand kann im Mikro-/Millisekundenbereich zwischen *gedrückt* und *nicht gedrückt* wechseln.
Nachfolgende Abbildung zeigt den Spannungsverlauf an einem Prozessor-Pin, der mit einem Pull-Up-Widerstand auf High-Pegel gezogen wird und über einen Taster mit Masse verbunden wird.



Über Widerständen und Kondensatoren kann man dafür sorgen, dass sich der Spannungspegel der durch den Taster auf Masse gezogenen Leitung nicht abrupt ändern kann. Nachfolgende Abbildung zeigt ein Beispiel für eine solche Hardware-Entprellung.



Das resultierende Signal ergibt sich dann in etwa wie folgt:



3 Interrupts

Interrupts unterbrechen den Programmfluss durch Verzweigen in eine sog. *Interrupt Service Routine* (ISR), um auf aufgetretene Ereignisse reagieren zu können.
Unser Atmel-Prozessor kann auf viele verschiedene Ereignisse reagieren, beispielsweise

- zurücksetzen des Prozessors durch Low-Pegel am Reset-Pin,
- bestimmte Pegel oder Pegelwechsel an einem Prozessor-Pin, z.B. externer Interrupt, Pin Change Interrupt,
- Ereignisse betreffend interne Zähler/Timer (Überlauf, Null, Gleichzeit mit einem Wert, ...),
- Analog/Digital-Wandlung abgeschlossen etc.

Um Interupts zu verwenden, muss

- die Verwendung von Interrupts global freigeschaltet sein (Register SREG),
- der Interrupt für das entsprechende Ereignis freigeschaltet sein (z.B. Register EIMSK für externe Interrupts),
- das Ereignis durch Register-Einstellungen ggf. noch weiter spezifiziert werden (bei externem Interrupt z.B. fallende Flanke, steigende Flanke, Pegeländerung, ..., vgl. Register EICRA),
- die Interrupt-Service-Routine angegeben werden.

Eine Interrupt-Service-Routine wird durch Implementierung der Funktion *ISR* realisiert. Als Parameter enthält die Funktion den Kurz-Namen des Interrupts (vgl. Spalte ‘‘Source’’ der Interrupt-Vektoren im Datenblatt), gefolgt von ‘‘_vect’’. Beispiel: `INT0_vect`.

4 Externe Interrupts

4.1 INT0, INT1, INT2

Bei den externen Interrupts INT0, INT1 und INT2 kann bei

- Low-Pegel,
- einer Pegel-Änderung,
- einer steigende Flanke oder
- einer fallende Flanke

an dem entsprechenden Pin in die entsprechende Interrupt-Service-Routine (ISR) verzweigt werden.
Die Interrupt-Service Routine wird als Funktion `ISR(vector)` implementiert, wobei vector den Interrupt spezifiziert. Gültige Vektor-Werte für externe Interrupts sind `INT0_vect`, `INT1_vect`, `INT2_vect`.
Um in eine Interrupt-Service-Routine verzweigen zu können, müssen folgende Bedingungen erfüllt sein:

- Interrupts müssen im Spezialregister `SREG` global freigeschaltet sein.
- Der jeweilige externe Interrupt muss im Register `EIMSK` (External Interrupt Mask Register) freigeschaltet sein.
- Das Ereignis muss auftreten.

Die Spezifizierung des Ereignisses (steigende Flanke, fallende Flanke, ...) erfolgt über das Register `EICRA` (External Interrupt Control Register A).
Beispiel für eine Interrupt Service Routine:

```
ISR(INT0_vect)
{
    PORTD ^= 1; // Pegel an PD0 ändern
}
```

4.2 Pin Change Interrupts

Über Pin-Change-Interrupts ist es für alle Ports A, B, C und D möglich, im Falle einer Pegel-Änderung an einem ausgewählten Pin in eine Port-spezifische Interrupt-Routine zu verzweigen.

5 Timer

Timer sind im Prozessor integrierte Hardware-Schaltungen, die den Inhalt sog. Timer-Register beim Auftreten bestimmter Ereignissen erhöhen oder erniedrigen, weshalb werden sie manchmal auch Zähler (Counter) genannt werden. Auf welche Ereignisse ein Timer reagieren soll, kann über Steuer-Register eingestellt werden. Auslöse-Ereignisse können beispielsweise das Auftreten eines Prozessor-Takts (oder eines Vielfachen davon) oder eine Pegel-Änderung an einem Pin sein.
Das Erhöhen/Erniedrigen der in den Timer-Registern gespeicherten Werte wird durch im Hardware-Schaltungen realisiert. Damit laufen Timer im Hintergrund, d.h. echt parallel zum auf dem Prozessor gerade ausgeführten Programm.
laufenden Programm und lösen nach einer einstellbaren Anzahl an Zeit-Einheiten einen Interrupt aus. In der entsprechenden Interrupt-Service-Routine können dann Aktionen getätigt werden (z.B. LED an- oder ausschalten). Mit den Timern können Hardware-unterstützt PWM-Signale erzeugt werden oder Pulsweiten gemessen werden.
Der Prozessor verfügt über vier Timer: T0, T1, T2 und T3.

- Timer T0 und T2 sind 8-Bit Timer,
- Timer T1 und T3 sind 16-Bit Timer.

Bei der Verwendung von Timern muss u.a.

- über das Spezialregister `SREG` Interrupts global freigeschaltet werden,
- eine Takt-Quelle ausgewählt werden,
- ein Takt-Teiler (Prescaler) ausgewählt werden,
- eine Timer-Funktion ausgwählt werden (einfacher Zähler, PWM, ...)
- ausgewählt werden, ob das Timer-Register beim Auftreten des Timer-Takts erhöht oder erniedrigt werden soll,
- geeignete Startwerte für die Timer-Register angegeben werden.

Beispiel:

```
ISR(TIMER1_OVF_vect)
{
    TCNT1 = 20000;  // Neuen Timer-Wert setzen
    // ...
}

int main(void)
```

```
{
 // ...
    SREG = 0x80;        // Interrupts global freischalten
    TIMSK1 = 0x01;      // T1 interrupt freischalten
    TCCR1B = 0x01;      // kein prescaler
    TCNT1 = 20000;      // Start-Wert
    while(1);           // Warten
}
```