Mikrocontroller

die mächtigen Helfer bei

Bastelprojekten

Inhaltsverzeichnis

1. Vorwort

Im Laufe der letzten Jahre sind Mikrocontrollerschaltkreise immer mächtiger und billiger geworden sodass sie in Eigenbauprojekten vielen Fällen eine gute Alternative zu diskret aufgebauter Logik geworden sind. In diesem Vortrag sollen solche Mikrocontroller vorgestellt werden. Die hier beschriebenen Funktionen treffen vorwiegend auf die ATMEL-Serie von Mikrocontrollern zu wobei andere Serien wie z.B. PIC wohl ähnliche Eigenschaften haben.

In dem Vortrag soll es nicht um komplette Rechner (wie z.B. Rasberry-PI mit einem vollständigen Betriebssystem) gehen sondern um reine Mikrocontroller ohne eigenes Betriebssystem.

2. Historie

2.1 Schaltungen mit Logik-ICs

Manche kennen die Logikschaltungen sicher noch, die mit einzelnen Transistoren aufgebaut wurden. Mit dieser Aufbautechnik musste man schon für aus heutiger Sicht einfache logische Aufgaben ganze Schränke mit Leiterplatten füllen. In den 60er und 70er Jahren begann dann der Siegeszug der Logik-ICs. Mit den TTL-ICs aus der 74er-Reihe konnte man die Aufbauten gegenüber denen mit diskreten Transistoren extrem verkleinern. Trotz dieser Verbesserungen gab es bei dieser Art des Aufbaus zwei Probleme:

- Der Stromverbrauch
- Die Komplexität des Aufbaus

Je nach Komplexität des verwendeten Logikgatters lag der Stromverbrauch bei einigen zehn bis einigen hundert Milliwatt pro IC. Das hat sich bei komplexeren Logiken schnell zu mehreren Ampere bei 5V Versorgungsspannung aufaddiert. Waren die Gatter für die vorgesehene Anwendung nicht schnell genug, musste man anstatt der „normalen“ 74er-Bausteine welche aus der 74S-Serie nehmen was den Stromverbrauch noch weiter in die Höhe getrieben hat. Bei kleineren Geschwindigkeiten konnte man auf CMOS-Bausteine ausweichen. Um den Geschwindigkeitsverlust auszugleichen wurde die Low-Power-Schottky Serie und später die HCT (High speed CMOS) Serie aufgelegt.

Auch mit höherwertigen Teilfunktionen pro Baustein (Zählern, Flip-Flops) wurde die Entflechtung von Platinen (besonders beim Eigenbau) sehr anspruchsvoll. In vielen Fällen waren dann auch doppelseitige Platinen nicht mehr ausreichend sodass man ohne zusätzliche Drahtbrücken nicht auskam.

Um dieses Problem in den Griff zu bekommen wurde, wurde eine neue Verdrahtungstechnik, die „Wraptechnik“ entwickelt. Damit konnte man die Anschlusspunkte der integrierten Schaltkreise mit Drähten direkt verbinden. Eine Entflechtung irgendwelcher Platinen war damit nicht mehr nötig. Auch bei diesem Verfahren gab es natürlich Nachteile:

- Der Platzverbrauch auf der Verdrahtungsseite war deutlich höher als bei der Platinenlösung
- Das Verhalten der Gesamtschaltung war nicht unbedingt reproduzierbar weil die gegenseitige Beeinflussung von Signalen von der Lage der Drähte abhing. Das konnte bei größeren Aufbauten und höheren Geschwindigkeiten zu sehr unangenehmen – weil

hochsporadischen – Fehlern führen.

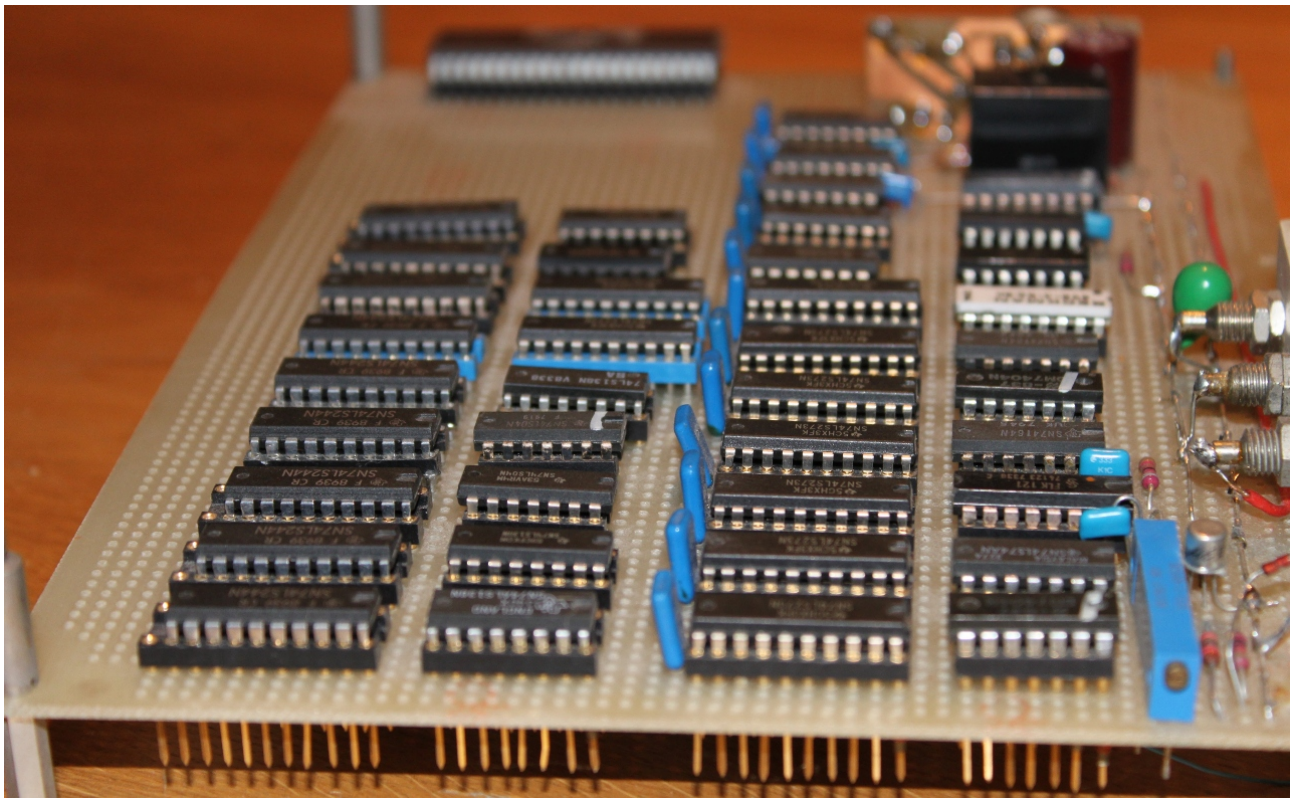


Bild 1: Aufbau einer Logikschaltung in Wraptechnik von der Bauteileseite gesehen

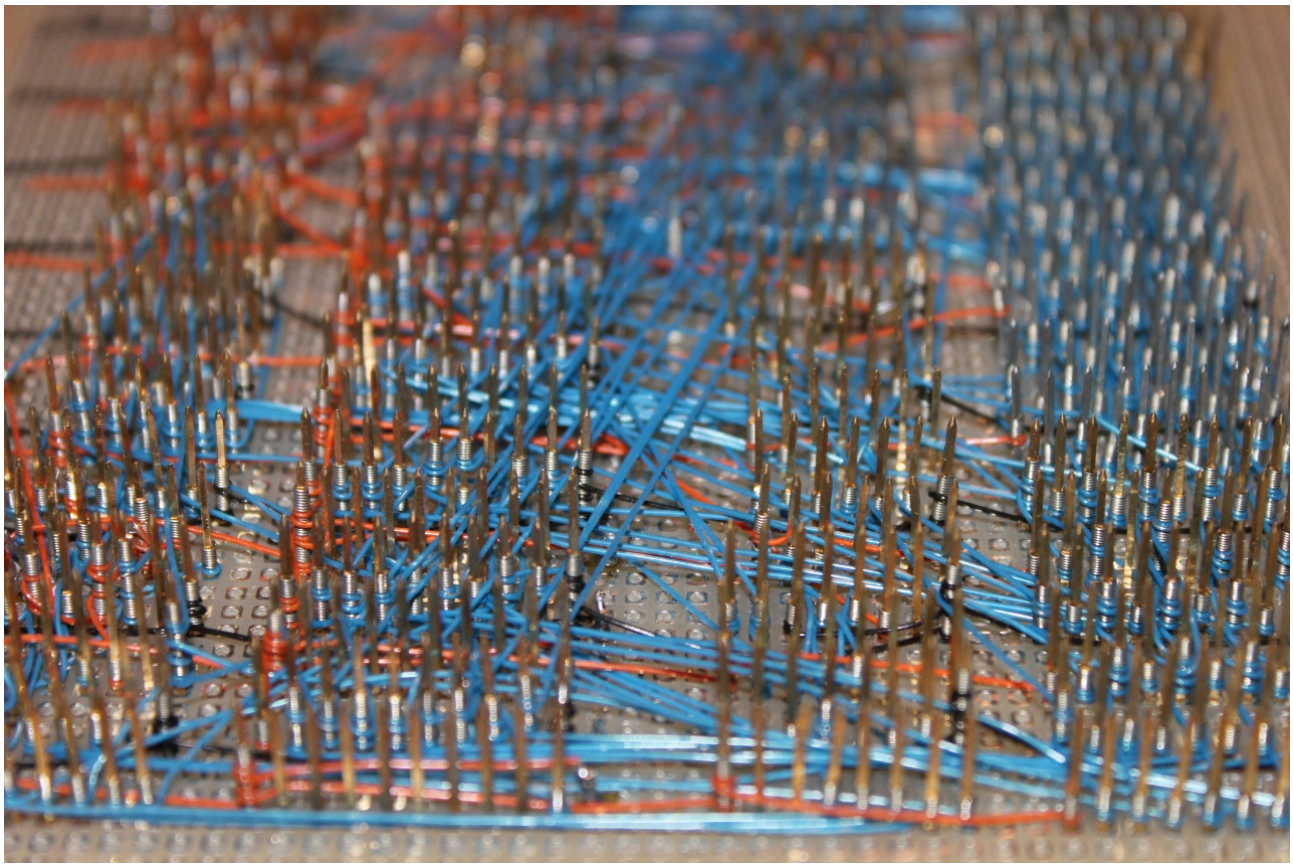


Bild 2: Aufbau derselben Logikschaltung von der Verdrahtungsseite aus gesehen.

2.2 Logikschaltungen mit Mikroprozessoren

Mit den oben beschriebenen Aufbauten kam man bei komplexeren Aufgaben schnell an die Grenzen. Auf der anderen Seite führte die steigende Integrationsdichte neuer integrierter Schaltkreise letztendlich zu den ersten Mikroprozessoren. Diese ersten Mikroprozessoren (TMS9900, 8080, 6502...) hatten noch einen großen Nachteil: Für sich alleine konnte man sie nicht betreiben. Damit sie funktionierten waren eine Menge zusätzlicher Funktionen zu realisieren:

- Ein Taktgeber (Oszillator)
- Ein Programmspeicher (meist in Form eines EPROMs)
- Ein Hauptspeicher (RAM)
- Eine Rücksetzlogik
- Ein/Ausgabe Schaltkreise

Um irgendeine Logikfunktion realisieren zu können war also wieder eine größere Anzahl von Einzel-ICs nötig. Außerdem brauchte es zusätzlich Geräte um die Programme erstellen, die EPROMs programmieren und notfalls auch wieder löschen zu können. Diese Mikroprozessorschaltungen hatten außerdem den Nachteil, dass sie weit langsamer waren als diskret aufgebaute Logikschaltungen. Weiterhin war das Zeitverhalten nicht immer deterministisch.

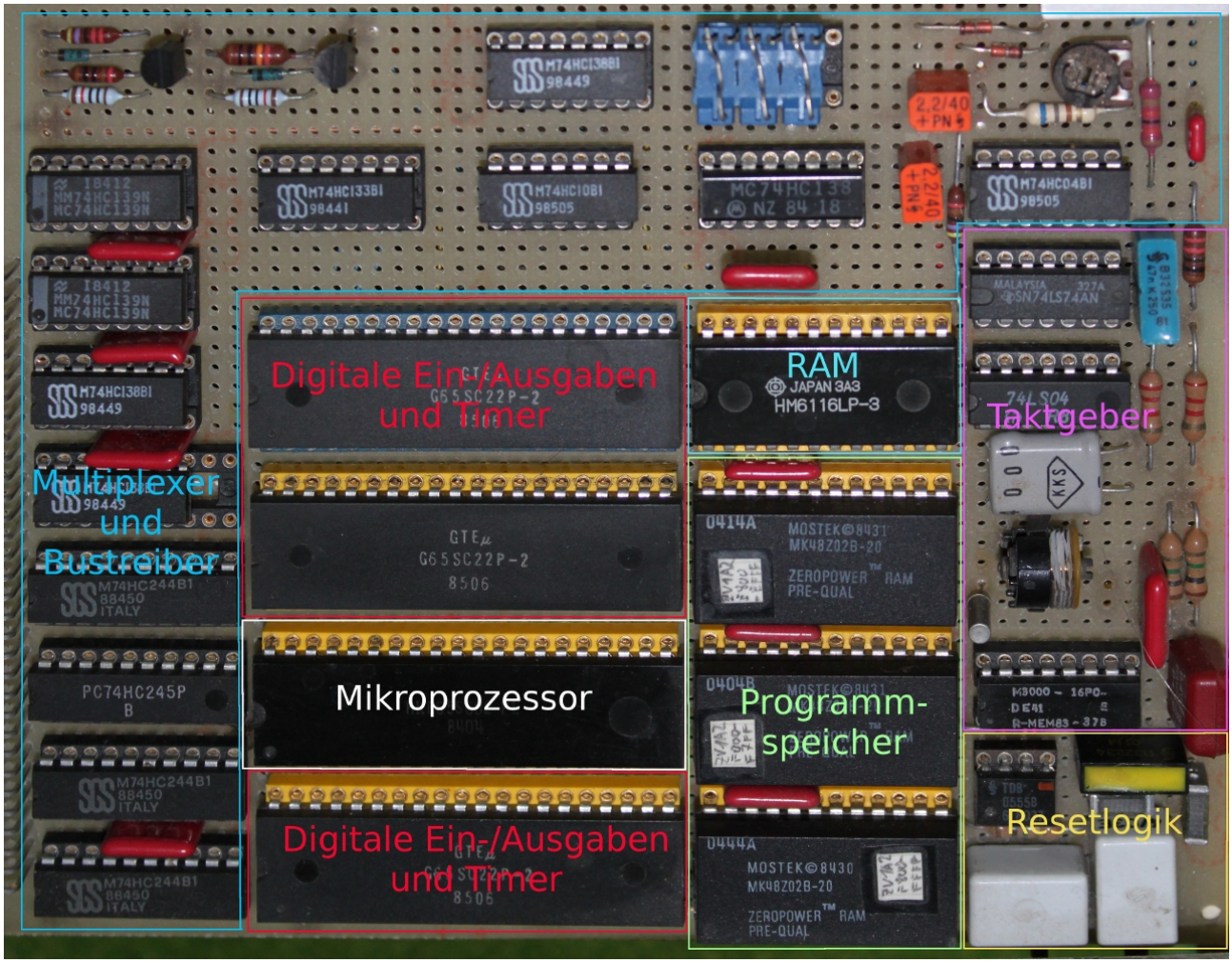


Bild 3: Schaltung mit einem Mikroprozessor des Typs 6502

Wie im Bild 3 zu erkennen ist der Mikroprozessor selbst nur ein kleiner Teil der Gesamtschaltung.

Seit der Zeit der ersten Mikroprozessoren sind die hier beschriebenen Zusatzfunktionen immer mehr in die Prozessoren integriert worden. Inzwischen lassen sich diese Chips auch ohne zusätzliche externe Bauteile betreiben.

3. Die Beschreibung eines Mikrocontrollers

Die folgende Beschreibung bezieht sich vorwiegend auf Atmel-Prozessoren. Der Grund für diese Beschränkung liegt darin, dass ich mit diesen Controllern die meiste Erfahrung habe. Bei anderen Controllerserien wie z.B. den PIC-Controllern sind Architekturen vergleichbar.

3.1 Der prinzipielle Aufbau

Wie schon beschrieben, sind die früheren, externen Funktionen alle in die Mikrocontroller integriert worden. Speziell die Ein- und Ausgabefunktionen können verschiedenen Controlleranschlüssen in weiten Bereichen frei zugeordnet werden.

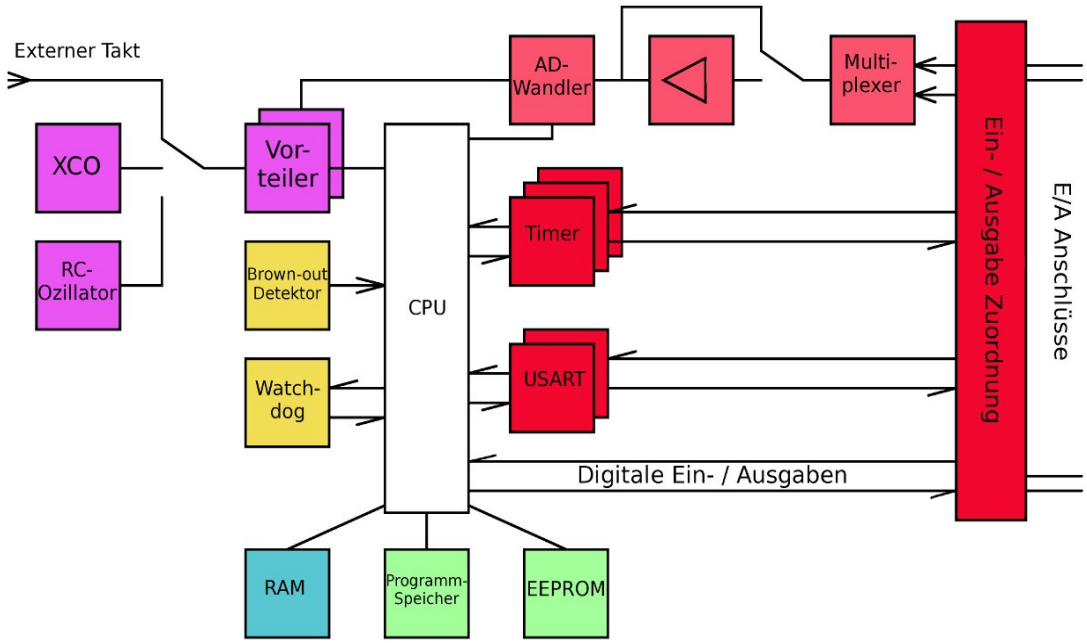


Bild 4: Prinzipieller Aufbau eines Mikrocontollers.

Die einzelnen Teilfunktionen sollen in den folgenden Punkten näher beschrieben werden.

3.2 Das zweistufige Startprinzip

Um die Controller so universell betreiben zu können ist ein zweistufiges Konzept erforderlich: Im ersten Schritt werden zentrale Funktionen eingestellt wie die Art der Takterzeugung (Betrieb mit externem Ozillator, mit einem Quarz, einem keramischen Resonator, oder dem internen laserabgeglichenen RC-Oszillator). Solche zentralen Funktionen können natürlich nicht erst im laufenden Betrieb aktiviert werden sondern müssen von Anfang an eingestellt sein. Dazu gibt es im Controller sogenannte „Fuse-Bits“, die am Ende der Programmierung gesetzt werden. Ab dem nächsten Controllerstart sind dann die damit definierten Einstellungen aktiv.

Ist der Prozessor dann angelaufen, sind vorerst alle E/A-Anschlüsse als hochohmige Digitaleingänge eingestellt. Auf diese Weise ist sichergestellt, dass direkt nach dem Controllerstart keine elektrischen Schäden entstehen können. Während des Starts muss deshalb eine Routine durchlaufen werden, die die E/A-Anschlüsse entsprechend ihrer Zielfunktion einstellt. Solche Zielfunktionen können sein:

- Unipolarer Analogeingang
- Differentieller Analogeingang
- Digitalausgang

- Digitaleingang mit/ohne „pull-up“-Widerstand
- USART – Ausgang (nur logisch, nicht elektrisch)
- USART - Eingang (nur logisch, nicht elektrisch)
- direkte Verbindung zu einer Zusatzfunktion (Timer, Taktgeber...)

3.3 Die Speicher des Controllers

Auch die Speicher der Controller sind inzwischen integriert. Bei den Atmel-Controllern sind es drei Speicherarten:

- der Arbeitsspeicher (RAM) der die änderbaren Variablen enthält
- der Programmspeicher (Flash-RAM) der das Programm enthält
- der EEPROM der Werte enthält, die über einen Spannungsausfall „gerettet“ werden müssen.
- 32 Register. Das sind 32 RAM-Zellen die neben der Adressierung über RAM-Zellen auch durch die Registernummern R0 bis R31 adressierbar sind.

Während der Arbeitsspeicher direkt nach dem Start keinen definierten Inhalt hat, enthält der Programmspeicher die Bitmuster die bei der letzten Programmierung gesetzt worden sind. Während des normalen Betriebs des Controllers bleibt dieser Speicherbereich unverändert. Nach Herstellerangaben ist dieser Bereich mehr als 10000 mal per Programmierung änderbar.

Der EEPROM enthält Werte, die über einen Spannungsausfall gerettet werden müssen. Diese Werte können entweder in einem Programmiervorgang ähnlich wie dem der Programmierung des Programmspeichers gesetzt werden oder per Programm im laufenden Controllerbetrieb. Nach Herstellerangaben sind dabei 100000 Programmiervorgänge möglich.

Neben den „normalen“ frei verfügbaren Speicherzellen enthält der Adressbereich des RAMs zusätzlich

- die Prozessorregister
- die IO-Zellen
- den Stack

3.3.1 Die Register

Die Register sind Speicherzellen, die zwar auch über „normale“ RAM-Adressen ausgelesen und beschrieben werden können aber fast ausschließlich über deren Registernummern adressiert werden.

Beispiel:

```
mov r16, r18
```

In diesem Fall wird der Inhalt des Registers 18 in das Register 16 kopiert.

Wo diese Register im RAM stehen ist bei dieser Art der Adressierung nicht weiter relevant. Natürlich muss man als Programmierer schon wissen welcher Bereich von diesen Registern beansprucht wird um deren Inhalte nicht versehentlich zu überschreiben.

3.3.2 Die I/O-Zellen

Wie schon beschrieben werden die Funktionen der E/A-Anschlüsse während des Programmstarts eingestellt. Neben diesen Zuordnungen können im Anlauf noch eine Menge andere Dinge (Parameter des AD-Wandlers, Geschwindigkeit des USARTs...) eingestellt werden. Dazu existiert eine Reihe von RAM-Adressen in deren Zellen sich die Eigenschaften der Controllerhardware definieren lässt. In der Programmierung werden anstatt der zahlenmäßig angebbaren Adressen gerne Adressnamen verwendet. Der Grund dafür ist, dass sich die Adressen von Controllertyp zu Controllertyp ändern können, die Namen meist aber unverändert bleiben.

Beispiel:

Kommando mit reinen Zahlenangaben:

```
out $1A, $FF
```

Das gleiche Kommando mit dem Namen der I/O-Zelle

```
out DDRA, $FF
```

DDRA ist das Datenrichtungsregister des Port A. Mit diesem Kommando werden alle 8 Bits des IO-Registers DDRA auf „1“ gesetzt was bedeutet, dass alle Anschlüsse des Port A ab sofort Digitalausgänge sind. Das Zeichen „$“ gibt an dass es sich bei der folgenden Zahl um eine Hexadezimalzahl handelt.

3.3.3 Der Stack

Die Atmel-Prozessoren haben 32 Register. Demnach kann man in diesen 32 Registern 32 verschiedene Variablen speichern. Das reicht in vielen Fällen aber nicht aus. Oft will man einfach nur dasselbe Register mehrfach verwenden. Man braucht also eine Möglichkeit den jeweiligen Registerinhalt „auszulagern“ und später wieder zu restaurieren. Um so etwas möglichst einfach zu gestalten wurde der Registerstapel (der sogenannte Stack) erfunden. Dazu wird eine spezielle „I/O“-Zelle, der Stackpointer gesetzt – meist auf die höchste verfügbare RAM-Adresse. Möchte man z.B. den Inhalt des Registers r16 retten, genügt das Kommando

```
push r16
```

Der Prozessor kopiert den Inhalt von r16 in die Zelle auf die der Stackpointer zeigt und erniedrigt den Stackpointer anschließend um 1.

Jetzt kann man das Register für andere Aufgabe nutzen. Möchte man den ursprünglichen Inhalt wieder restaurieren genügt in unserem Beispiel das Kommando

```
pop r16
```

Der Prozessor erhöht jetzt den Stackpointer um 1 und kopiert den Inhalt der Zelle auf die er jetzt zeigt in r16 zurück.

Bei diesem Verfahren muss man natürlich strikt darauf achten, dass man das Register zuerst restauriert das man zuletzt gerettet hat, also zu einer Folge

```
push r16
push r17
push r18
```

die Befehlsfolge

```
pop r18
pop r17
pop r16
```

gehört.

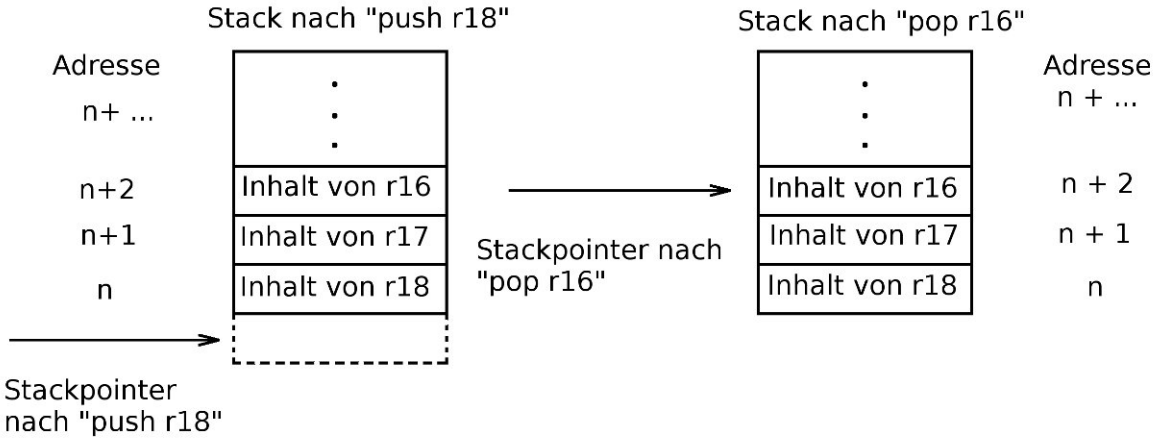


Bild 5: Inhalte des Stacks (Registerstapels) nach den oben beschriebenen Befehlsfolgen

3.3.4 Frei verfügbare Speicherzellen

Alle anderen Zellen im RAM sind frei verfügbar. Nachdem der Stack von den höchst möglichen RAM-Adresse nach unten wächst muss man natürlich drauf achten dass man mit seinen Zellen nicht in diesen dynamischen Stack-Bereich gerät.

3.4 Der Programmspeicher

Wie der Name schon sagt, enthält der Programmspeicher die Folge der Programmkommandos. Auch dieser Speicherbereich hat eine Struktur. Er besteht aus

- einer Interruptleiste und
- der Befehlsfolge

3.4.1 Die Interruptleiste

Der Prozessor muss wissen auf in welcher Zelle der erste Befehl steht der auszuführen ist. Dabei gibt es unterschiedliche Ereignisse auf er mit unterschiedlichen Programmen reagieren kann. Diese unterschiedlichen Ereignisse werden als „Interrupts“ bezeichnet. Zu jedem dieser Ereignisse gibt es eine Interruptadresse. Tritt nun ein solches Ereignis ein, führt der Prozessor das Kommando aus, das auf der für dieses Ereignis zuständigen Adresse steht. Beispiel:

Die Adresse des „RESET“-Interrupts ist die Adresse „0“. Wird die Versorgungsspannung des Controllers eingeschaltet, so wird automatisch ein „RESET“-Interrupt generiert. Der Prozessor sieht

deshalb in der Zelle „0“ nach und führt den dort stehenden Befehl aus. Auf dieser Adresse „0“ steht meist ein Befehl der Art „Springe zur Adresse xxx“. Ab der Adresse xxx steht dann die eigentliche für den Reset zuständige Befehlsfolge.

Die möglichen Interruptadressen stehen in den untersten Zellen des Programmspeichers wobei der „RESET“-Interrupt die Adresse „0“ hat. In dem hier gezeigten Beispiel sind drei der möglichen Interrupts ausprogrammiert.

Adresse	Befehl	Kommentar
0	rjmp ANLAUFROUTINE	Sprung in die Anlaufroutine
2	rjmp E/A_INTERRUPT_0	Änderung eines Digialeingangs
8	rjmp TIMER_1	Zeitablauf im Timer 1

3.4.2 Die Befehlsfolge

Alle Zellen, die nicht zur Interruptleiste gehören können für das Programm und genauer für die unterschiedlichen Interruptroutinen verwendet werden.

3.5 Der EEPROM

Der EEPROM enthält Werte, die über Spannungsausfälle hinweg gesichert bleiben sollen. Das können z.B. die aktuellen Bedienzustände einer Anwendung sein. Wird die Stromversorgung aus und danach wieder eingeschaltet ist eben dieser Bedienzustand wieder verfügbar. Nachdem der Zugriff auf den EEPROM verglichen mit dem RAM deutlich länger dauert, ist es üblich den EEPROM-Inhalt während des Programmstarts (also in der RESET-Routine) in den RAM zu kopieren und vor dem Ausschalten der Stromversorgung wieder zu sichern.

3.6 Die Ein- / Ausgabefunktionen

Die unterschiedlichen Ein- / Ausgabefunktionen wurden im Punkt „3.2 Startprinzip“ schon kurz erwähnt. Hier sollen sie etwas genauer beschrieben werden. Zu den reinen E/A-Funktionen kommen noch welche dazu, die etwas übergeordnet sind und deren Ein-und Ausgänge nicht direkt mit einem Eingang oder Ausgang verbunden sein müssen (z.B die Timer).

3.6.1 Der Digitaleingang

Hochohmiger Digitaleingang ist die Standardeinstellung von E/A-Anschlüssen die der Mikrocontroller nach jedem Neustart (Reset) einnimmt. Auf diese Weise kann sichergestellt werden dass der Controller durch anstehende Hardwarepegel (sofern sie im erlaubten Bereich von 0V bis Vcc liegen) nicht zerstört wird und umgekehrt auch keine andere dort angeschlossenen Bauteile zerstören kann. Das Verhalten dieses Eingangs lässt sich hardwaremäßig ändern: Man kann einen controllerinternen „Pull-Up“-Widerstand aktivieren. Der Eingang wird damit intern auf Vcc-Potential gezogen sodass man z.B. mit nur einem Schalter ohne weitere externe Bauelemente auskommt einen solchen Eingang auf den Bedienzustand 0 oder 1 zu setzen.

Softwaremäßig kann man den Zustand des Eingangs ermitteln, indem man die zugehörige E/A-

Zelle ausliest wobei jede E/A-Zelle die Zustände von 8 Eingängen anzeigt. Das Kommando

```
in r16, PINA
```

kopiert die Zustände des E/A-Blockes A in das Register 16. In dem Register kann man den Zustand des gewünschten einzelnen Eingangs dann isolieren:

```
andi r16, $01
```

In diesem Beispiel wird die Information des Eingangs PA0 isoliert, alle anderen Informationen (die von PA1 bis PA7) werden gelöscht. Es gibt noch eine weitere Möglichkeiten nur einen dieser Eingänge einzulesen:

```
sbis PINB, 1
```

Steht in einem Programm ein solches Kommando, dann wird das Folgekommando des Programms übersprungen wenn der Eingang 1 des E/A-Blockes B (also PB1) den Zustand „1“ hat.

Dieses Verfahren, den Zustand eines oder mehrerer Eingänge auszulesen hat einen Nachteil. Zustandswechsel werden nur erkannt, wenn das Programm durch eine Abfragesequenz (wie in dem Beispiel oben) läuft. Möchte man auf Zustandswechsel sofort reagieren, kann man sogenannte Interrupts aktivieren. Bei einem Zustandswechsel eines Eingangs eines E/A-Blockes wird der „normale“ Programmlauf unterbrochen und kann dann durch eine Sequenz ersetzt werden, die den Zustandswechsel bearbeitet. (Die Programmierung mit Interrupts wird weiter unten noch genauer beschrieben).

Neben diesen Interrupts gibt es noch die Möglichkeit, Zähler (Timer) über Eingänge direkt zu starten oder anzuhalten um so z.B. genaue Impulsbreitenmessungen machen zu können.

Einer oder mehrere Eingänge können auch auf komplexere digitale Eingabefunktionen (USARTs) geschaltet werden. Damit sind Datenübertragungen möglich wie sie von den COM-Ports von PCs bekannt sind.

3.6.2 Der Digitalausgang

Mit Hilfe des „Datenrichtungsregisters“ lassen sich einzelne Anschlüsse eines E/A-Blocks als Ausgänge schalten. So stellt z.B. das Kommando

```
out DDRB $55
```

Die Anschlüsse PB0, PB2, PB4 und PB6 auf Ausgänge um, die anderen Anschlüsse bleiben Eingänge. Der Zustand der Anschlüsse lässt ich dann durch setzen der zugehörigen Ausgabezelle einstellen.

```
out PORTB $05
```

setzt die Ausgänge PB0 und PB2 auf Vcc, der Rest bleibt auf „0“.

Ähnlich wie bei den Eingängen gibt es auch bei einigen der Ausgänge Sonderfunktionen. Diese Ausgänge mit Sonderfunktionen können z.B.

- auf Zählerausgänge geschaltet werden (für pulsbreitenmodulierte Signale)
- auf einen USART geschaltet werden (COM-Port)
- auf den Oszillatortakt geschaltet werden.

3.6.3 Der Analogeingang

Einer der E/A-Blöcke bietet neben der Möglichkeit Anschlüsse als Digitaleingänge oder Digitalausgänge zu nutzen außerdem die Möglichkeit sie als Analogeingänge zu verwenden. Dazu können die als Analogeingänge definierten Anschlüsse über einen Analogmultiplexer auf einen A/D-Wandler geschaltet werden. Zusätzlich gibt es noch die Möglichkeit Signalverstärker einzuschleifen.

3.7 Übergeordnete Funktionen

Um einen Mikrocontroller effektiv einsetzen zu können ist neben der reinen Rechenfunktion der CPU und den schon beschriebenen digitalen und analogen Ein/Ausgabefunktionen noch eine größere Zahl zusätzlicher Funktionen realisiert. Diese Sonderfunktionen lassen sich (je nach Einstellung) entweder direkt über Ein- und Ausgänge erreichen oder können programmgesteuert betrieben werden.

3.7.1 Zähler (Timer)

Eine der zentralen übergeordneten Funktionen sind die Zähler (Timer). Zusammen mit der Interruptfunktion (Beschreibung siehe unten) werden sie meist benutzt um die Vorgänge im Controller zeitlich zu strukturieren. Die Zähler können programmgesteuert auf einen Anfangswert gesetzt werden. Auf den Zählereingang können verschiedene Signale geschaltet werden wie z.B.

- das Signal des Taktoszillators
- das Signal eines Digitaleingangs.

Der augenblickliche Wert den Zählers kann mit hardwaremäßig realisierten Vergleichern verglichen werden deren Ausgänge dann Interrupts auslösen können oder direkt auf Digitalausgänge geschaltet werden können. Zusätzlich kann der Zähler mit diesem Signal wieder auf den Anfangswert zurückgesetzt werden.

Beispiel:

Der Startwert eines Zählers wird auf „0“ gesetzt

Der Zähler direkt wird mit dem Taktoszillator im 1us-Takt hochgezählt

Der Vergleicher wird auf den Wert 999 gestellt.

Mit dem Ausgang des Vergleichers wird der Zähler wieder auf „0“ gesetzt, außerdem wird ein Interrupt generiert.

Hat man den Zähler so programmiert, wird alle 1000us also jede Millisekunde ein Interrupt erzeugt mit dessen Hilfe sich ein bestimmter Programmteil ausführen lässt.

Dieses Verfahren hat den Vorteil, dass die Zeitabläufe in einem exakten Zeitraster von 1ms ablaufen, die von den übrigen Vorgängen im Controllerprogramm weitgehend unabhängig sind.

3.7.2 USART

Die USART-Funktion (Universal Synchronous and Asynchronous Serial Receiver and Transmitter) ist eine Funktion mit deren Hilfe sich serielle Daten senden und empfangen lassen: Die Art wie diese Daten übertragen werden kann in weiten Bereichen über spezielle I/O-Zellen eingestellt

werden. Über diese I/O-Zellen sind Eigenschaften einstellbar wie

- die Baudrate (Übertragungsgeschwindigkeit)
- die Anzahl der Bits pro Zeichen
- die Anzahl der Stop-Bits (also der Zeichenabstand)
- ob und welche Sicherungsinformation mit übertragen werden soll (Parity-Bit)
- ob die Übertragung asynchron oder synchron erfolgen soll.
- ob der USART Interrupts generieren soll.

Abhängig vom Controllertyp können auch mehrere USARTs verfügbar sein, die sich unabhängig voneinander einstellen und betreiben lassen.

3.7.3 A/D-Wandler

Für den 10 Bit A/D-Wandler können verschiedenen Betriebsarten eingestellt werden:

- Kontinuierliche Konvertierung
- Einmalige Konvertierung ausgelöst durch unterschiedliche Ereignisse wie Befehle durch die CPU oder ein externes Signal.
- Messung gegen eine interne Referenz, die Versorgungsspannung oder eine externe Referenzspannung.

Der Eingang des A/D-Wandlers kann über Multiplexer auf „einfache“ Eingänge oder differentielle Eingänge geschaltet werden. Zusätzlich kann ein Signalverstärker in den analogen Signalweg eingeschleift werden.

3.7.4 Brown-out Detektor (Betriebsspannungsüberwachung)

Die Elektronik eines solchen integrierten Schaltkreises funktioniert nur in einem bestimmten Spannungsbereich (z.B. 2.7V bis 5.5V). Sinkt die Spannung unter die erlaubte Untergrenze, arbeitet die CPU nicht mehr zuverlässig. In dem Grenzbereich kurz unter der Untergrenze lässt sich nicht mehr vorhersagen was der Schaltkreis wirklich tut. Es kann also passieren, dass in diesen Grenzbereichen fehlerhafte Ausgangssignale gesetzt werden, die im ungünstigsten Fall zu Schäden oder zur Zerstörung von Bauteilen führen können. Um das zu vermeiden lässt sich in Schaltkreis eine Betriebsspannungsüberwachung aktivieren, die den Prozessor ab einer Spannungsuntergrenze die noch deutlich über der Ausfallgrenze des Chips liegt in einer geordneten Weise anhält. Steigt die Spannung wieder über die definierte Grenze (+ einer Hysterese) wird nach einer kurzen Wartezeit ein RESET-Interrupt ausgelöst, die CPU startet damit neu.

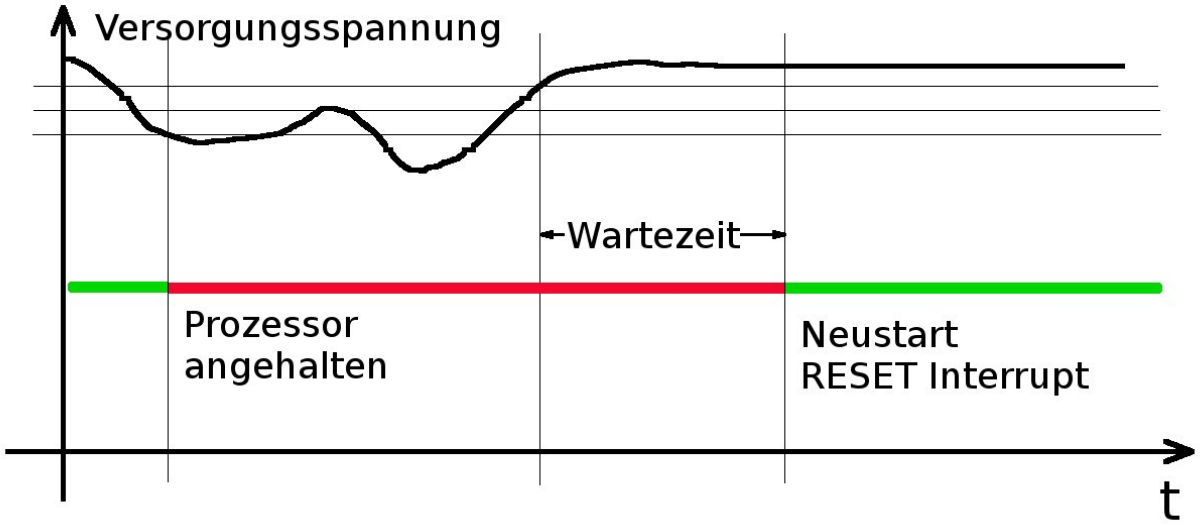


Bild 6: Die Funktion des Brown-out Detektors

3.7.5 Watchdog (Programmüberwachung)

Neben Ausfällen durch zu geringe Betriebsspannung können auch Programmierfehler zu einem Ausfall der Controllers führen. So kann es z.B. passieren, dass das Programm in eine ungewollte Endlosschleife läuft oder durch Überläufe (z.B. Stacküberlauf) ausfällt. Das Ergebnis ist ein Mikrocontroller der auf Eingangssignale nicht mehr reagiert. Vor solchen Totalausfällen kann man sich durch einen „Watchdog“ schützen. Dabei handelt sich um einen Zähler der von einem vorgebbaren Wert aus abwärts zählt. Getaktet wird dieser Zähler durch den Taktgeber des Controllers – also unabhängig vom Programm. Im Anwenderprogramm müssen an einer oder mehreren Stellen Sequenzen eingebaut werden mit denen der Zähler des Watchdog immer wieder auf seinen Anfangswert gesetzt wird. Arbeitet das Programm korrekt, so erreicht der Zähler nie den Wert Null weil er immer rechtzeitig hochgesetzt wird. Gerät das Anwenderprogramm in eine Endlosschleife und wird untätig, wird der Zähler nicht mehr hochgesetzt und erreicht damit den Wert „0“. Wenn der Zähler die Null erreicht wird ein „Reset“-Interrupt ausgelöst, was dazu führt, dass das Anwenderprogramm wieder neu startet.

Das Verfahren hat auf der einen Seite den Vorteil dass keine Totalhänger mehr passieren können. Auf der anderen Seite wird dadurch ein Programmierfehler „versteckt“. Das kann im schlimmsten Fall dazu führen, dass das Programm scheinbar unlogisch reagiert und durch den Reset die Spuren des Fehlers verwischt werden sodass man die eigentliche Fehlerursache nicht mehr findet.

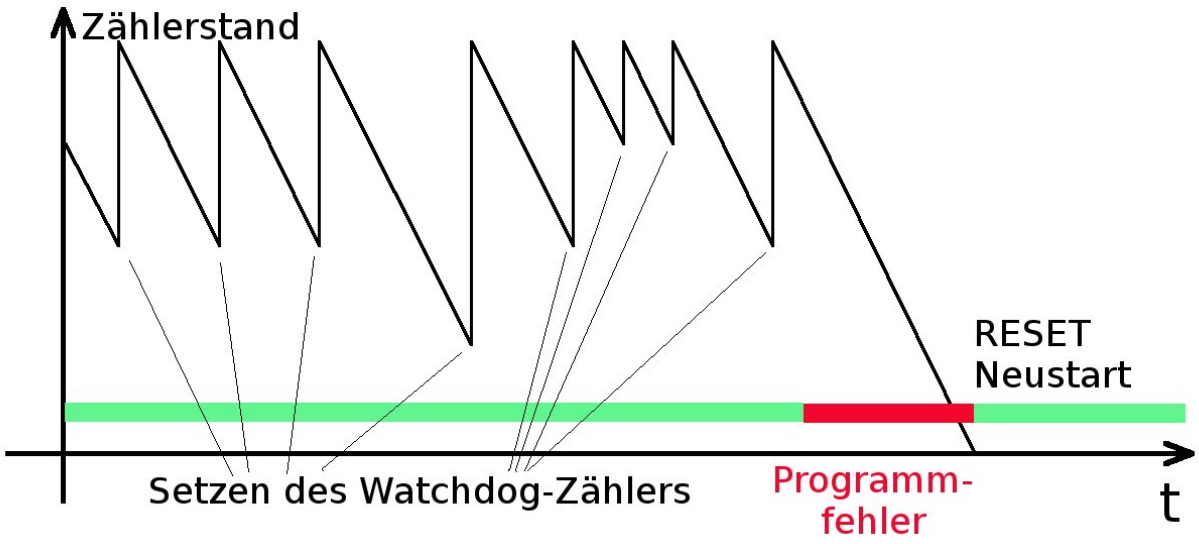


Bild 7: Programmüberwachung mit Hilfe des „Watchdogs“

4. Das Programm

4.1 Interrupts

Interrupts sind sehr wirksame Hilfsmittel um das zeitliche Verhalten des Prozessors und damit des gesamten Controllers zu steuern. Sie sind Unterbrechungen des gerade laufenden Programmteils. Ausgelöst werden solche Unterbrechungen durch Ereignisse wie z.B.

- Einschalten des Mikrocontrollers (RESET-Interrupt)
- Ablauf eines Timers (Zählers)
- Abschluss einer A/D-Umwandlung
- Eintreffen eines Zeichens im USART
- Erfolgreich abgeschicktes Zeichen durch den USART
- Signalwechsel an einem Digitaleingang.

Wurde ein Interrupt ausgelöst, so wird

- das gerade noch laufende Prozessorkommando abgeschlossen,
- die Adresse des nächsten Kommandos im Stack gesichert,
- der zu dem aktuellen Interrupt gehörende Programmteil (die Interruptroutine) durchlaufen und
- nach deren Ende das ursprüngliche Programm fortgesetzt.

Mit einem solchen Verfahren kann man erreichen, dass der Controller spätestens nach der Restbearbeitungszeit des aktuellen Kommandos beginnt auf das ausprogrammierte Ereignis zu reagieren. Die Reaktionszeit setzt sich dann zusammen aus der Restlaufzeit des gerade ausgeführten Kommandos und der konstanten Reaktionszeit auf den Interrupt.

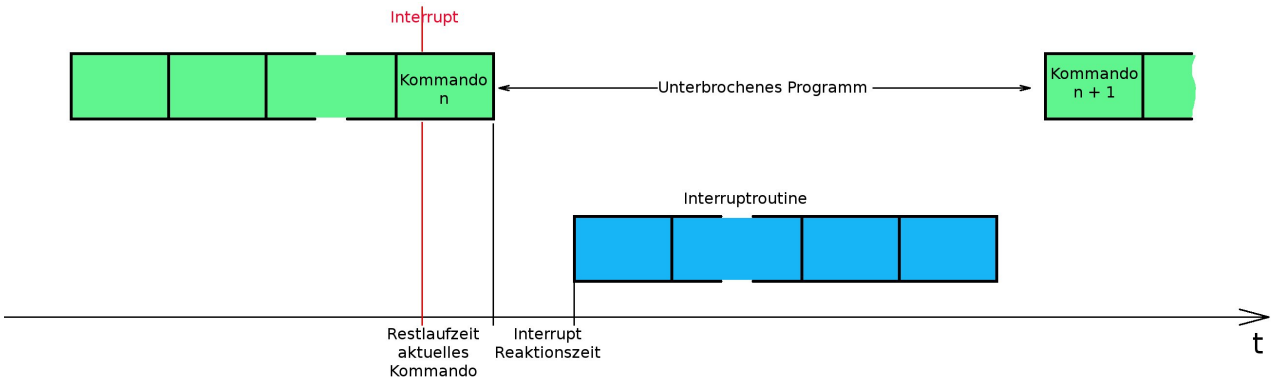
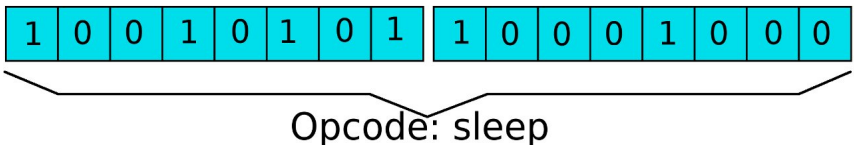


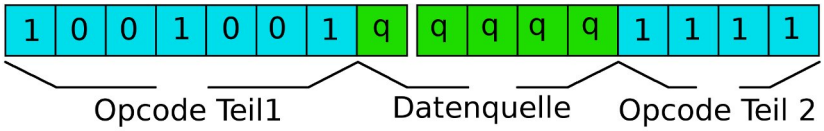
Bild 8: Zeitlicher Ablauf bei einem Interrupt

4.2 Datenformate der Kommandos

Die Kommandos für die CPU (das Rechenwerk) des Controllers bestehen aus 2 oder 4-Byte langen Hexazahlen, die den Opcode (das eigentliche Kommando) und optional Angaben zur Datenquelle und zum Datenziel enthalten. Um die zur Verfügung stehenden 16 oder 32 Bits möglichst gut auszunutzen, hängt das Datenformat vom Opcode (Operationscode) des Kommandos ab. Im ersten Fall besteht das gesamte Kommando nur aus dem Operationscode, es gibt weder Quelle noch Ziel weil das Kommando die gesamte CPU betrifft.



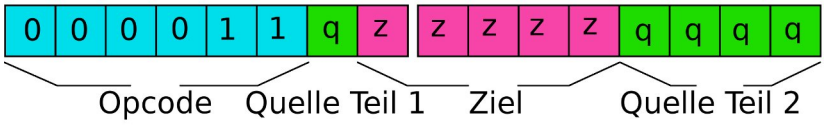
Im zweiten Beispiel – dem oben schon beschriebenen „push“-Kommando werden in dem 16Bit Kommando der Operationscode und die Angabe des Quellregisters untergebracht:



Wie man hier erkennen kann, ist der Operationscode (10010011111) in zwei Teile aufgespalten. In dem 5 Bits der Datenquelle steht die Nummer des Registers dessen Daten gesichert werden sollen. 00000 steht für das Register 0, 11111 für das Register 31. Das Kommando

`push r16`

würde damit durch die Bitfolge 1001001100001111 oder in Hexadezimalzahlen ausgedrückt 930F kodiert sein.
Im dritten Beispiel ist das „add“-Kommando dargestellt



Wie man hier sieht, ist in diesem Fall die Nummer der Quellregisters aufgeteilt. Würde man z.B. das Kommando

`add r17, r16` (addiere den Inhalt des Registers 16 zu dem des Registers 17)

kodieren, so wäre das zugehörige Bitmuster 0000111100010000 oder als Hexazahl geschrieben 0F10.
Man kann nun Programme schreiben indem man die Hexamuster von Opcode, Quelle und Ziel von Hand zusammensetzt und so erhaltenen Bitmuster untereinanderschreibt. Das Verfahren ist allerdings extrem aufwendig und fehleranfällig. Hat man im gesamten Programm auch nur ein Bit

falsch gesetzt wird das Programm nicht so reagieren wie erwartet, meist ist es dann völlig unbrauchbar.

Eine für ganz kurze Programme mögliche „Abhilfe“ ist ein Disassembler, also ein Programm das die Bitmuster in lesbare Kommandos umsetzt. Problem solcher Disassembler ist, dass sie die Bitmuster 1:1 in Text umwandeln was bedeutet, dass Sprungadressen als Zahlen auftauchen und das Sprungziel damit nicht direkt anzeigen. Im Beispiel (Springe 27 Befehle zurück) müsste man im Programm ab der aktuellen Stelle 27 Befehle zurückzählen um das Sprungziel zu finden.

4.3 Programmiersprachen

Um das oben beschriebene Problem zu umgehen, schreibt man Programm gleich in einer lesbaren Form. Dafür gibt es inzwischen eine unüberschaubare Zahl von Programmiersprachen, jede auf bestimmte Gruppen von Anwendungsfällen zugeschnitten. Diese Programmiersprachen lassen sich in drei Typen einteilen

- Assembler,
- Compiler und
- Interpreter (in Mikrocontrollern nicht verwendbar)

4.3.1 Assembler

Assembler setzen die in Textform geschriebenen Prozessorkommandos direkt in die prozessorspezifischen Bitmuster in Zahlenform um. Damit erfordert jeder Prozessor – oder Controllertyp seinen eigenen Assembler. Um die beim Disassembler beschriebenen Probleme mit der direkten Zahleneingabe z.B. für Sprungadressen nicht zu haben, gibt es bei Assemblern meist die Möglichkeit, Werte und Namen einander fest zuzuordnen oder Sprungziele im Programm direkt zu markieren. Das fehleranfällige Abzählen von Kommandos entfällt damit. Um Programme wirklich „lesen“ zu können haben Assembler die Möglichkeit die Kommandos zu kommentieren.

Nachteile von Assemblern:

- Jeder Controllertyp braucht seinen eigenen Assembler. Umgekehrt heißt das, dass man ein Programm neu schreiben muss wenn es auf einem anderen Prozessortyp ablaufen soll.
- Man muss sich um alles selbst kümmern. Die Adresse einer Speicherzelle selbst oder die Inhalte der Register selbst festlegen. Programmierung in Assemblern ist deshalb sehr fehleranfällig.

Vorteile von Assemblern

- Assemblerprogramme sind im Allgemeinen deutlich schneller als solche, die von höheren Programmiersprachen generiert wurden.
- Assembler brauchen weit weniger Speicherplatz als Compilerprogramme.

Beispiel:

Möchte man die Berechnung von 1 + 2 + 3 + 4 … + 10 in Assembler schreiben, so könnte das so aussehen:

```
       clr r16          ; Ergebnisregister löschen
       ldi r17, 10      ; Schleifenzähler setzen
:SCHLEIFE               ; Rückspungadresse für die Schleife
       add r16, r17     ; Addition
       dec r17          ; Schleifenzähler aktualisieren
       brne SCHLEIFE    ; Rücksprung wenn Zähler ungleich 0
```

4.3.2 Höhere Programmiersprachen

Höhere Programmiersprachen verstecken den Hardwarehintergrund. In solchen Sprachen ist es nicht mehr nötig irgendwelchen Variablen Register zuzuordnen. Anders gesehen, sind höhere Programmiersprachen (Compiler) so geschrieben, dass die zu programmierende Problemlösung im Vordergrund steht und nicht der zu programmierende Prozessor. Dazu gibt es Konstrukte wie Schleifen, bedingte Abfragen, oder versteckte Konvertierungen von Zahlenformaten.

Nachteile von Compilern:

- Compilerprogramme sind deutlich langsamer als Assemblerprogramme
- Compilerprogramme brauchen deutlich mehr Platz als Assemblerprogramme
- Aus dem vom Compiler abgesetzten Maschinencode lässt sich der eigentliche Progammablauf nur noch schwer rekonstruieren.

Vorteile von Compilern:

- Compilerprogramme sind prozessorunabhängig. Soll ein vorhandenes Programm auf einen anderen Prozessortyp umgesetzt werden, ist deshalb nur ein anderer Compiler nötig, das Programm selbst kann bleiben.
- Probleme lassen sich mit Compilermitteln einfacher, anschaulicher und kompakter formulieren.
- Bedingt durch die verfügbaren Sprachmittel können bei Compilern Programmierfehler vermieden werden, die sich bei Assemblern nicht erkannt werden können. Abhängig vom Compiler lässt sich damit sehr sicherer (fehlerarmer) Code erzeugen. (z.B. in der Programmiersprache Ada).

Für die Programmierung von Mikrocontrollern wird häufig die Sprache „C“ verwendet. In C für Mikrocontroller geschriebene Programme sind nicht gänzlich vom Controllertyp unabhängig weil durch das Programm ja eine Menge Hardware „eingestellt“ werden muss wie z.B. die Funktionen der einzelnen E/A_Anschlüsse.

Beispiel:

Möchte man die in Punkt 3.4.1 beschriebene Berechnung 1 + 2 + … in der Sprache C formulieren, dann kann das so aussehen:

```
Ergebnis = 0;                              /* Startwert           */
for (i = 10; i > 0; --i) Ergebnis += i;  /* Berechnungsschleife */
```

4.4 Programmiertechniken

Möchte man mit einem Controller eher einfache Aufgaben lösen, reichen meist recht einfache Ablaufstrukturen aus. Eleganter ist es allerdings die Interruptfunktionen der Controller auszunutzen. Bei komplexeren Problemen kommt man im diese etwas ausgefeiltere Technik nicht herum.

4.4.1 Die Endlosschleife

Die Endlosschleife ist die einfachste Art eine Controllerfunktion auszuprogrammieren wenngleich man auch diesem Fall nicht ganz um Interrupts herumkommt. Wenn der Controller startet oder über den zugehörigen Eingang zurückgesetzt wird, wird ein Reset-Interrupt ausgelöst. Dieser Reset-Interrupt bewirkt, dass die CPU den Befehl ausführt, der in der Programmadresse „0“ steht. In die Adresse „0“ lädt man deshalb einen Befehl, der einen Sprung auf den eigentlichen Programmcode beschreibt. Ab der Adresse dieses Sprungziels steht dann der eigentliche Programmcode.

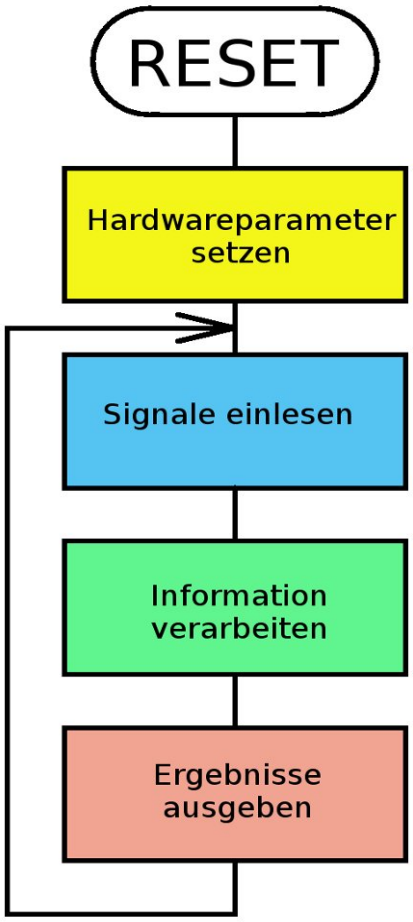


Bild 9: Programmstruktur „Endlosschleife“

Wie aus dem Bild 9 zu sehen ist, besteht der Programmcode

- aus einem Teil in dem die Hardware konfiguriert wird. Dieser Teil wird nur nach einem RESET (Signal am RESET-Eingang oder Start des Controllers) durchlaufen;
- einem Teil in dem die Eingänge ausgelesen werden, die Information verarbeitet und das Ergebnis ausgegeben wird. Dieser Teil wird in einer Endlosschleife dauernd durchlaufen.

4.4.2 Programmierung mit Interrupts

Das oben gezeigte Beispiel mit der Endlosschleife hat einen gravierenden Nachteil: Die Reaktionszeit des Controllers ist nicht vorhersagbar. Sie ist davon abhängig wo das Programm während des Zustandswechsels an einem der Eingänge gerade steht.

Möchte man ein zeitlich vorhersagbares Verhalten muss man die elegantere Methode, das ausnutzen der Interrupts anwenden. Dazu nutzt man das schon beschriebene Controllerverhalten beim Interrupt:

Bei einem Interrupt unterbricht die CPU das gerade laufende Programmstück, merkt sich die Stelle an der das Programm unterbrochen wurde, springt die dem Interrupt zugeordnete Adresse an, führt den dort stehenden Programmcode – die Interruptroutine – aus und springt wieder in das ursprüngliche Programm zurück. Interrupts werden meist in der Startroutine scharfgeschaltet. Im Controller gibt es eine ganze Reihe möglicher Interrupts wie

- Zustandswechsel an einem Digitaleingang
- Der Zahlenvergleich eines Zählers mit einem Vergleichsregister
- die Ankunft eines kompletten Zeichens in dem USART
- das Freiwerden eines USART Sendepuffers nach dem senden eines Zeichens
- Der Abschluss eines Konvertierungszyklus des A/D-Wandlers

Damit sieht die Programmierung unseres Beispiels jetzt so aus:

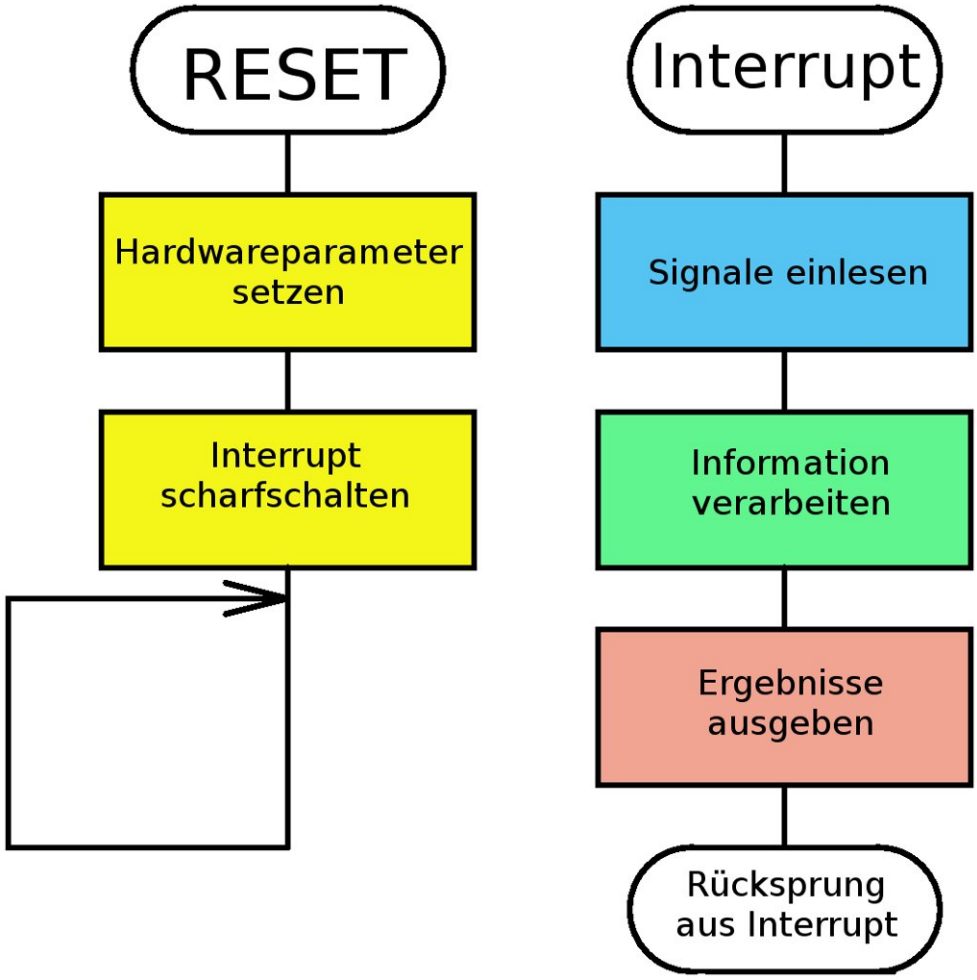


Bild 10: Programmierung mit Hilfe von Interrupts

Wie in Bild 10 zu sehen, taucht in der Startroutine jetzt auch die Aktivierung eines Signalwechselinterrupts auf. Das Auslesen der Eingänge die Verarbeitung dieser Informationen und die Ausgabe stehen jetzt in der Interruptroutine. Die Endlosschleife aus dem letzten Beispiel existiert zwar noch, besteht aber nur noch aus einem „Rücksprung auf sich selbst“. Jetzt ist das zeitliche Verhalten vorhersagbar weil im Fall einer Zustandsänderung die Folge „auslesen – verarbeiten - ausgeben“ sofort durchlaufen wird. Bei dieser Struktur ist noch eine weitere Verbesserung möglich. Setzt man in die Endlosschleife ein „SLEEP“-Kommando, dann setzt sich die CPU zur Ruhe und wird erst von dem Interrupt geweckt. In der Ruhephase ist der Stromverbrauch des Controllers deutlich geringer.

5. Energie sparen

Mikrocontroller werden häufig in batteriebetriebenen Geräten eingesetzt. Dort ist die Betriebsdauer mit einer Akkuladung oder einem Satz Batterien ein ganz zentrales Qualitätskriterium. Aus diesem Grund wurden in den Mikrocontrollern einige Funktionen realisiert, die helfen den Energieverbrauch zu minimieren.

5.1 Der Einfluss der Taktfrequenz

Die Mikrocontroller sind MOS-Bauteile bei denen der Stromverbrauch vorwiegend durch den Transport von Ladungen bedingt ist. Verdoppelt man den Takt, wird auch die transportierte Ladung verdoppelt. Damit verdoppelt sich der Stromverbrauch linear mit der Taktfrequenz und linear mit der Versorgungsspannung. Während sich die Versorgungsspannung nur in Grenzen absenken lässt, hat man beim Prozessortakt mehr Möglichkeiten. Man kann z.B. die Taktfrequenz durch einen Quarz mit niedrigerer Frequenz absenken. Eine andere Möglichkeit bieten interne Teiler mit denen die Taktfrequenz auf einen Bruchteil der Oszillatorfrequenz eingestellt werden kann. Natürlich steigen damit auch die Reaktionszeiten des Controllers linear an. Meist reichen aber deutlich niedrigere Taktfrequenzen für die gewünschte Funktion aus.

5.2 Abschalten von Teilfunktionen

Im Allgemeinen braucht man von den in einem solchen Controller realisierten Funktionen nur einen Bruchteil (z.B. nur einen Teil der E/A-Anschlüsse, nicht alle Zähler (Timer), keinen A/D-Wandler...). Nachdem diese Zusatzfunktionen in Form von Hardware auf dem Controllerchip realisiert sind, brauchen sie natürlich auch Strom. Um den Stromverbrauch ungenutzter Funktionen zu vermeiden kann man diese komplett abschalten. Dieses Abschalten geht wie die Konfiguration der Schnittstellen: Durch Löschen von Bits in speziell dafür vorgesehenen Speicherzellen. Dieses Abschalten programmiert man am besten in der Reset-Routine, also im Anlaufteil des Controllerprogramms.

5.3 Der Schlafmodus

In vielen Fällen hat der Controller eigentlich nichts anders zu tun als auf ein bestimmtes Ereignis zu warten. Das kann eine Bedienung sein oder der Ablauf einer bestimmten Zeitspanne. Ist das Ereignis dann eingetreten muss er schnell reagieren um danach wieder auf das nächste Ereignis zu warten. Anders ausgedrückt – einer der „Großverbraucher“ im Controller, nämlich die CPU wird die meiste Zeit überhaupt nicht benötigt. Diese Tatsache kann man ausnutzen indem man in der Anlaufroutine (im Reset-Programmteil) alle Ereignisse auf die reagiert werden muss scharf schaltet indem man die dazugehörigen Interrupts aktiviert und den Controller dann in den „Schlafmodus“ versetzt. Dadurch wird das Rechenwerk, die CPU angehalten sodass nur noch die Teile Strom verbrauchen, die Interrupts auslösen können. Tritt irgendwann ein parametriertes Ereignis ein, wird die CPU aktiviert und die zugehörige Interruptroutine durchlaufen. Am Ende der Interruptroutine versetzt man die CPU durch ein entsprechendes Kommando wieder in den Schlafmodus.

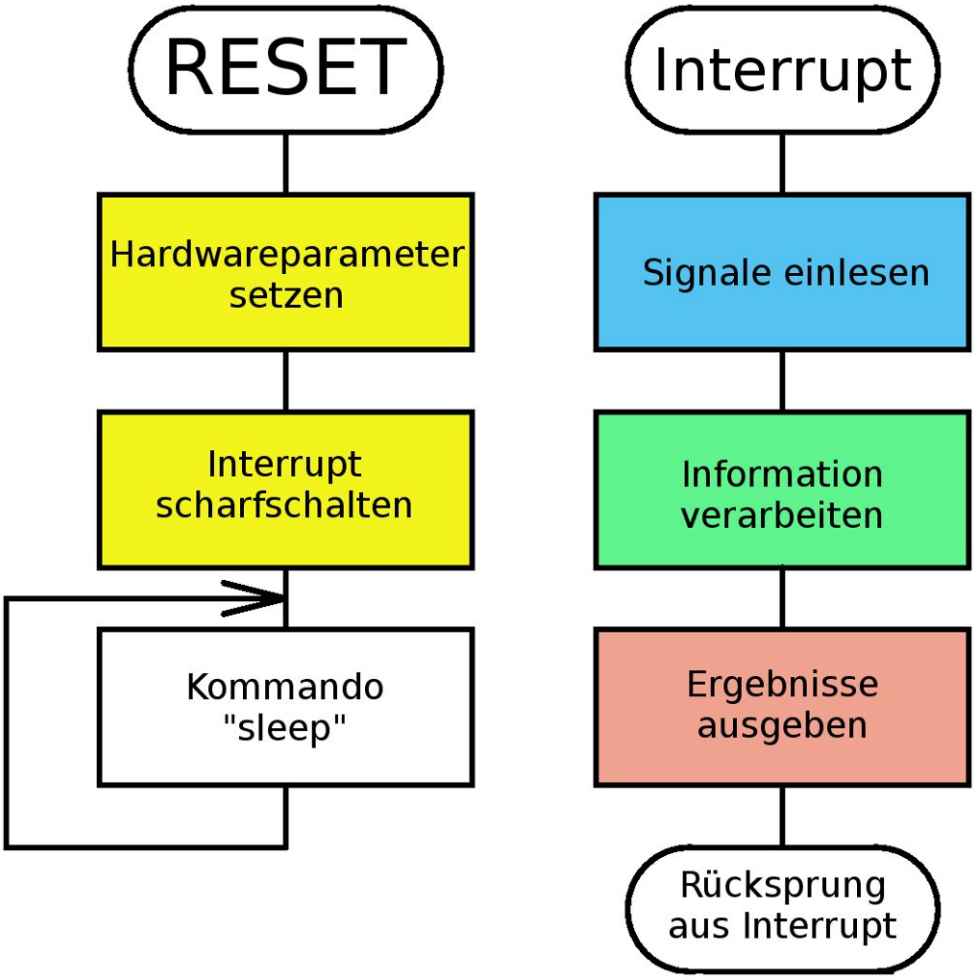


Bild 11: Ausnützen der Schlafmodus der CPU um Energie zu sparen

7. Der Test von Programmen

8. Programmierung der Controller

Bisher wurde beschrieben, wie der Controller parametriert wird und wie die Programme erstellt werden. Diese Programme – genauer die Folge von Kommandos in Zahlenform – muss auf irgendeine Weise in den Zielcontroller geladen werden.

8.1 Programmierung mit kommerziellen Programmiergeräten

Vom Hersteller werden Programmierboards angeboten mit deren Hilfe sich eine Reihe von unterschiedlichen Typen aus der Controllerreihe programmieren und testen lassen. Für die Programmierung ist in dem hier beschriebenen Fall ein PC mit einer seriellen Schnittstelle nötig. Auf dem PC kann dann eine Software geladen werden, mit deren Hilfe sich die Controllerprogramme in C oder Assembler erstellen und laden lassen. Diese Software (AVR-Studio) läuft leider nur unter Mikrosoft Betriebssystemen. Nachdem ich zuhause kein Mikrosoft verwende musste ich diese Software unter Linux nachprogrammieren.

8.2. Programmierung mit eigenen Mitteln

Die Controller lassen sich aber auch ohne spezielles Board programmieren. Dazu muss der Controller erst in den Programmiermodus gebracht werden (Die Stromversorgung muss bei auf „0“ gehaltener Reset-Leitung sowie auf „0“ gehaltener SCK (serial clock) Leitung eingeschaltet werden). In diesem Programmiermodus bekommen drei der E/A-Anschlüsse des Controllers eine feste Funktion:

- eine Leitung für den Datentakt (SCK)
- eine Leitung für die zu programmierenden Daten (MOSI)
- eine Leitung für die Rückmeldung (MISO)

Die zugehörige Logik lässt sich natürlich auch mit den alten Logikbausteinen der 74er-Serie aufbauen, für das zusammenstellen der Kommandos und die Überwachung der Programmierung anhand der Rückmeldungen wird ein PC nötig sein.

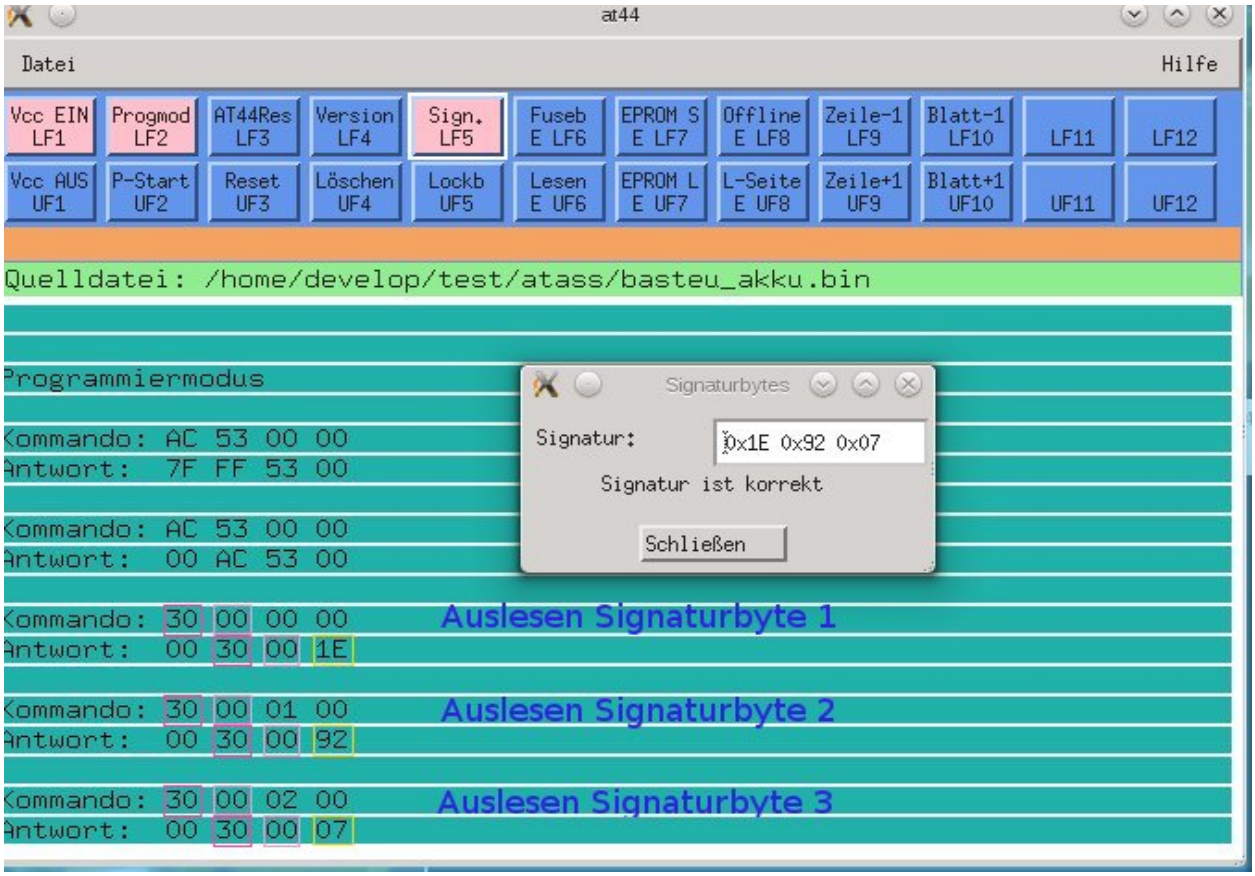


Bild 12: Beispiel für eine Programmierfunktion – dem Auslesen der Signaturbytes

In dem Bild 12 ist der prinzipielle Programmierablauf anhand der Funktion „Auslesen der Signaturbytes“ zu sehen: In den ersten beiden Blöcken wird der Programmiermodus angefordert. Die Wiederholung der Folge AC 53 im Block 2 zeigt an, dass der Programmiermodus ab jetzt aktiv ist. In den letzten drei Blöcken werden mit den Bitmustern 30 00 00, 30 00 01 und 30 00 02 die drei Signaturbytes angefordert. In der jeweiligen Antwort wird der erste Teil der Kommandos wiederholt und der Wert des jeweiligen Signaturbytes (gelb umrandet) ausgegeben.

9. Beispiele für fertige Geräte

9.1 Steuergerät für einen MW- oder Langwellensender.

Mit dem Gerät können Nachrichten in WSPR, OPERA und QRSS ausgegeben werden.

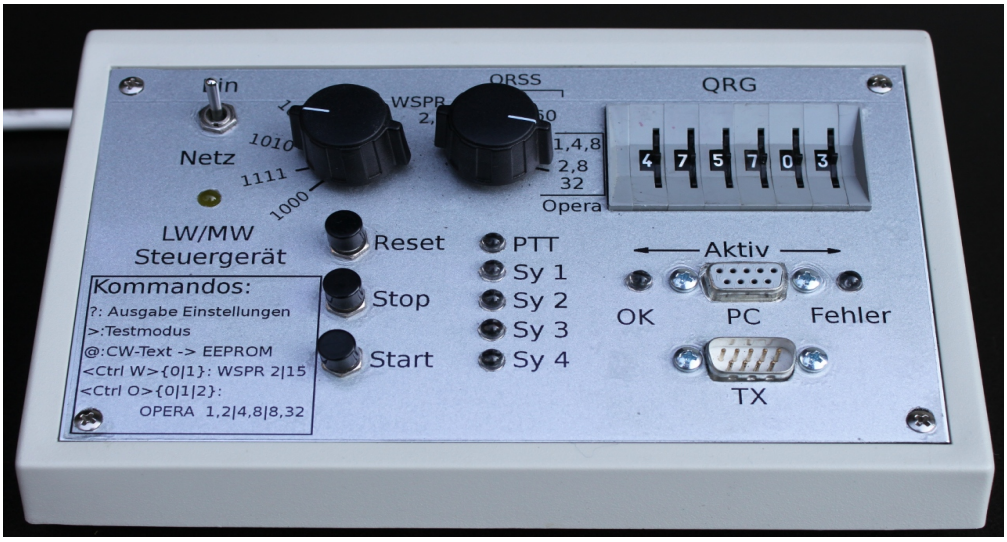


Bild 12: Steuergerät für einen MW- oder Langwellensender für die Betriebsarten WSPR, OPERA und QRSS.

Controllertyp: ATMEGA 162

Codeumfang: 9854 Byte

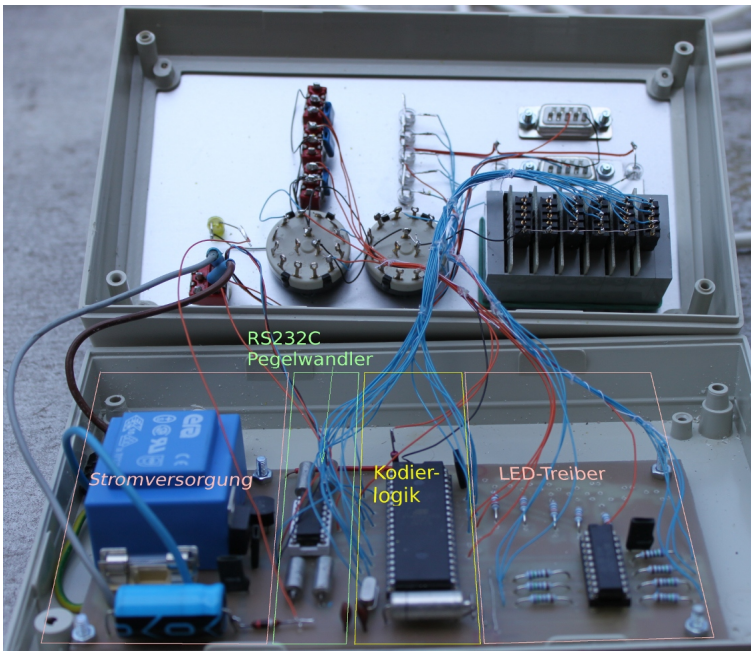


Bild 13:
Innenansicht des Steuergerätes. Um die erforderliche Genauigkeit im WSPR-Betrieb zu garantieren wurde der Taktgeber des Mikrocontroller mit einem Quarz stabilisiert. (Vorne links im Rahmen „Kodierlogik“

9.2 Steuergerät für einen Antennenrotor

Mit diesem Gerät können Azimuth und Elevation einer Antenne per habe eingestellt werden oder über einen Rechner gesteuert werden. Dazu gibt der Controller die Antennenposition auf Anforderung des Rechners aus und startet/stoppt die beiden Rotorantriebe.

Bild 14: Vorder- und Rückansicht einer für Satelliten- und EME-Betrieb nutzbaren Rotorsteuerung.

Controllertyp: ATMEGA 16

Codeumfang: 240Byte

9.3 Programmiergerät für den ATtiny 44

Mit diesem Gerät kann der ATtiny 44 über eine serielle Schnittstelle programmiert und getestet werden.

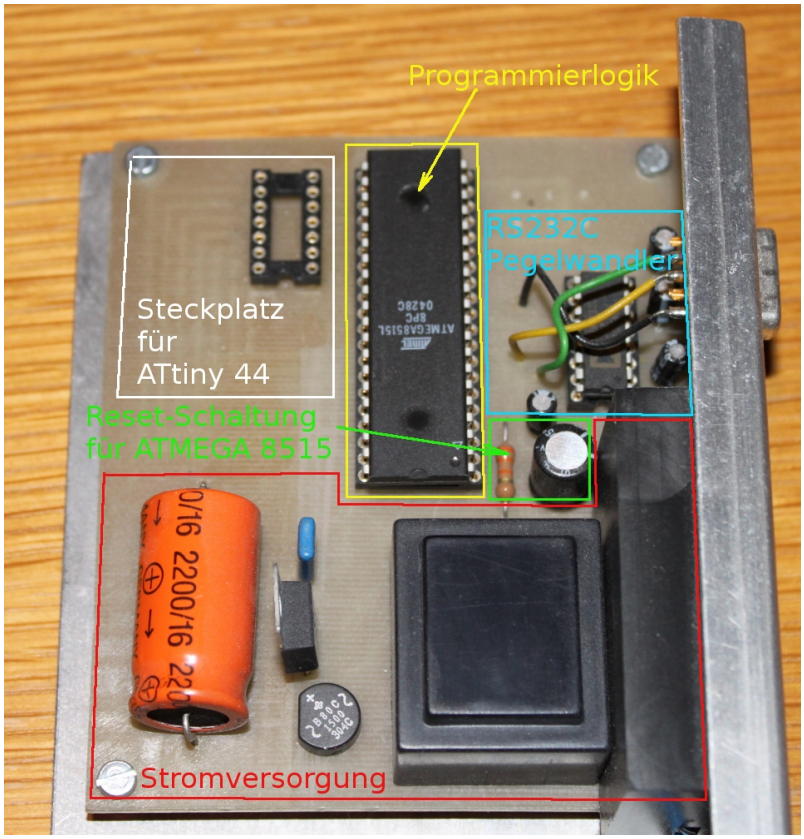


Bild 15: Programmiergerät für den ATtiny 44.

Controllertyp: ATMEGA 8515

Codeumfang: 2680 Byte

Wie man aus dem Bild erkennt, benötigt der Mikrocontroller ATMEGA 8515 außer der Stromversorgung und der Pegelanpassung der Seriellen Schnittstelle aus +/-12V nur ein RC-Glied für das RESET-Signal. Der zu programmierende Schaltkreis braucht keine zusätzlichen externen Elemente.

9.4 Signalkodierung für eine Funkfernbedienung

Mit diesem Controller wird das zu sendende Bitmuster für eine Funkfernbedienung erzeugt. Das Signal besteht aus einem Header, der eigentlichen Information und einer Kontrollinformation mit der sich auf der Empfangsseite die empfangende Nachricht prüfen lässt.

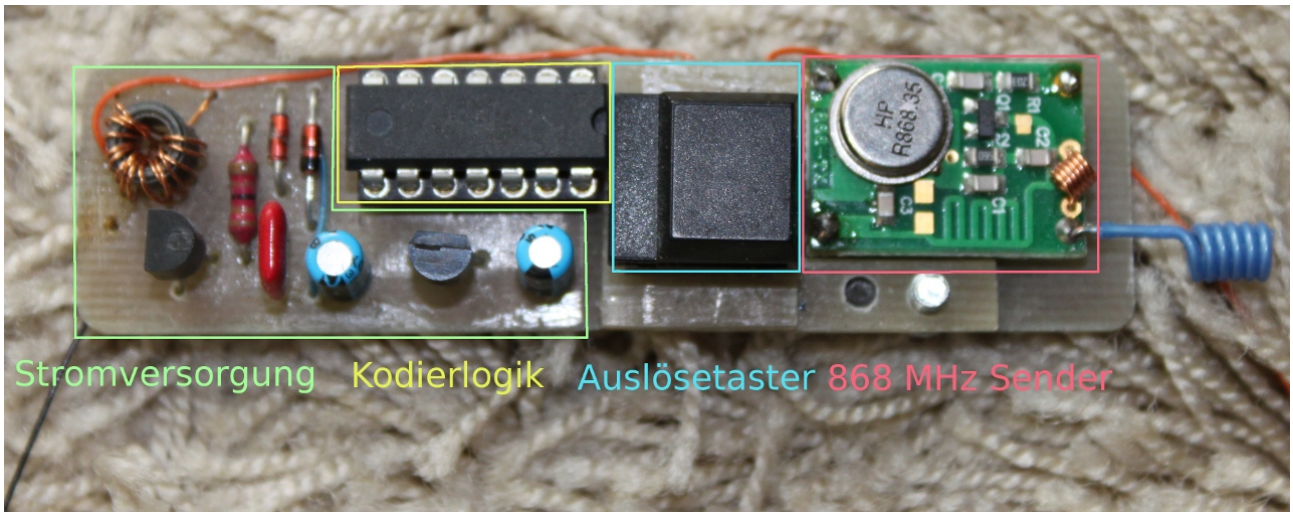


Bild 16: Einsatz eines ATtiny 44 in einer Funkfernbedienung

Controllertyp: ATtiny44

Codeumfang: 412 Byte

10. Quellenverzeichnis

ATMEL 8-bit AVR Instruction Set Rev. 0856D-AVR-08/02

ATMEL 8-bit AVR Mikrocontroller with 2/4/8k Bytes In-System Programmable Flash
ATtiny24, ATtiny44, ATtiny84. Rev. 8006K-AVR-10/10