Assemblerkochbuch für Arduino und Atmel AVR Micro Controller

Felix Morgner and Manfred Morgner

1. Mai 2012

Dieses Buch ist ein lebendes Werk. Wenn Du Fehler findest kannst Du uns kontaktieren, wir versuchen dann, die Beule auszubeulen. Wir freuen und immer, wenn uns jemand hilft, unser Buch zu verbessern, uns bessere Wege zeigt, verständlichere Beschreibungen demonstriert, schnellere Lösungen einbringt, Fehler beseitigt ... was immer nötig ist um das beste Kochbuch aller Zeiten zu entwickeln.

Wenn Du in mit diesem Buch arbeiten willst, empfehlen wir mindestens die folgende Dokumentation herunter zu laden ‚ATMEL 8bit AVR instruction set manual':

`http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf`

In diesem Buch werden wir die verwendeten Befehle nicht grundlegend erklären. Diese Erläuterungen liegen bereits in der o.g. Dokumentation vor. Natürlich werden wir erläutern wieso wir bestimmte Befehle einsetzen wenn es, im Rahmen des Buches, offensichtliche Alternativen gäbe.

Dieses Buch verwendet für die Umsetzung entweder durchgehend oder überwiegend *ATmega8*. Diese Einschränkung dient der Vereinfachung der Codebeispiele. Wir wollen und nicht in den Spezifika der Micro Controller Modelle verlieren und unnötig komplexen Quelltext produzieren.

Es geht uns auch darum zu zeigen, was bereits mit kleinen Micro Controllern erreicht werden kann. Es ist uns klar, dass grössere, schnellere Micro Controller für ähnliche Preise angeboten werden. Wir sind allerdings der Meinung, dass es viel eindrücklicher ist, mit einem sehr kleinen Micro Controller eine überraschend grosse Aufgabe zu lösen, wie zum Beispiel ein Musikinstrument aus einem *ATmega8*ein paar Drähten und ein paar Widerständen.

Dieses Buch dreht sich um Assemblerprogrammierung. Wir glauben nicht, das Assemblerprogramme die Welt retten werden. Auch wenn es einfacher ist, die Welt zu retten wenn man in Assembler programmiert ;-) Jede Programmiersprache hat ihren Zweck und manche sind sogar sinnvoll. Der Leser dieses Buches möchte Assemblerprogrammierung anhand von Beispielen erlernen und experimentiert gern mit echter Hardware. Wir versuchen diese Lesergruppe zu bedienen indem wir versuchen, die Schaltungen einfach zu halten und die Kosten dafür niedrig.

Inhaltsverzeichnis

I Einfache Beispiele **9**

1 Licht **11**

1.1 Es werde Licht! 12

1.2 Mach' mich an, lass' mich aus 16

1.3 Stable Decisions 19

1.3.1 WHAT to do 21

1.3.2 HOW to do it 21

1.3.2.1 Slow users 23

1.3.2.2 Fast feedback 23

1.3.3 Finding the event 24

1.3.4 And some modesty 25

1.4 Stable Decisions Triggered 25

1.5 Light Shift 26

2 Time **31**

2.1 Instable Elements 31

II External devices **33**

3 Shift Registers **35**

3.1 SRAM to Shift Register 35

Abbildungsverzeichnis

1.1 Arduino Uno 11
1.2 'Licht' Schaltplan 12
1.3 Arduino zu *ATmega8*Anschlusszuordnung 13
1.4 Mach' mich an, lass' mich aus - Schaltplan 17
1.5 Stable Decisions - Flow Diagram 22
1.6 Stable Decisions - Signal Diagram 24
1.7 Light Shift - Schema 27

Tabellenverzeichnis

Teil I

Einfache Beispiele

Kapitel 1

Licht

In diesem Kapitel demonstrieren wir die Grundlagen der Assemblerprogrammierung mit AVR Micro Controllern. In den Beispielen werden wir nur das Mindeste an Hardware verwenden. Für die ersten Beispiele wirst Du lediglich ein Stück Draht und Deinen mit Strom versorgten Arduino benötigen

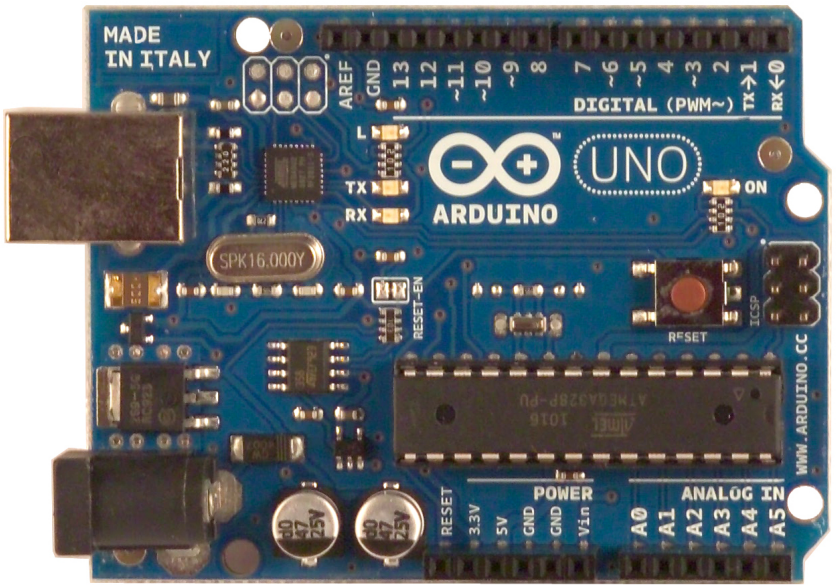


Abbildung 1.1: Arduino Uno

Du kannst auch Deine eigene Schaltung aufbauen, wie in Abbildung 1.4 auf Seite 17 gezeigt. Das Breadboard Layout kannst Du der Datei `LED/S000_LED-Basic-Circuit.fz` entnehmen, eine Fritzing-Datei.

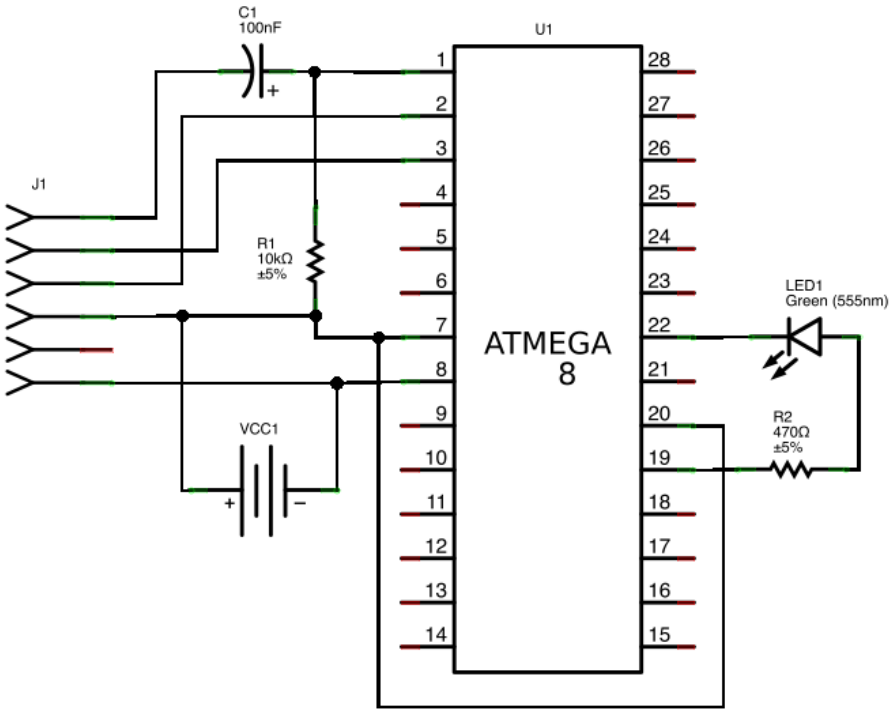


Abbildung 1.2: 'Licht' Schaltplan

Wie Du feststellen wirst, konzentrieren wir und darauf, nach Möglichkeit Freie und Opens Source Software zu verwenden. Aller Programmcode, den wir hier zeigen ist Freie Software. Das Buch und alles darin ist so frei wie es uns erlaubt wird es zu machen.

Ausserdem versuchen wir, 'Dinge' zu verwenden, die nicht frei sind. Wir möchten niemanden zwingen nicht freie Produkte zu benutzen nur um diesem Buch zu folgen. Eins unserer wichtigeren Ziele ist ein Beitrag zur grossen Welt des Freien Denkens, der Libre Software und der Zusammenarbeit.

1.1 Es werde Licht!

Das erste Beispiel in diesem Kapitel ist das kleinste Programm, das ich mir vorstellen kann, das tatsächlich auch etwas sichtbares tut.

Es schaltet die LED am Arduinoanschluss 13 ein.

In Assembler Programmieren heisst, Du hast die Macht! Aber wie wir wissen, erfordert Macht Wissen und Verantwortungsbewusstsein. Jemand der Micro Controller in Assembler programmiert, hat die absolute Macht, ob er es will oder nicht! Demzufolge ist ein Mindestmass an Fachwissen erforderlich um verantwortlich zu handeln.

Als erstes musste Du wissen, was Anschluss 13 am Arduino in Wirklichkeit heisst. Infolge des Arduinodesigns ist Anschluss 13 am Arduino nicht Pin 13 am *ATmega8*. Das zu wissen ist erst die halbe Miete. Das tatsächliche Pin am *ATmega8*zu kennen ist erforderlich um mit dem blanken Chip zu arbeiten. Was Du ausserdem wissen musst ist, wie dieser Anschluss im Micro Controller adressiert werden muss. Um das alles heraus zu finden gibt es ein sehr schönes Schaubild: `http://www.arduino.cc/hu/Hacking/PinMapping`

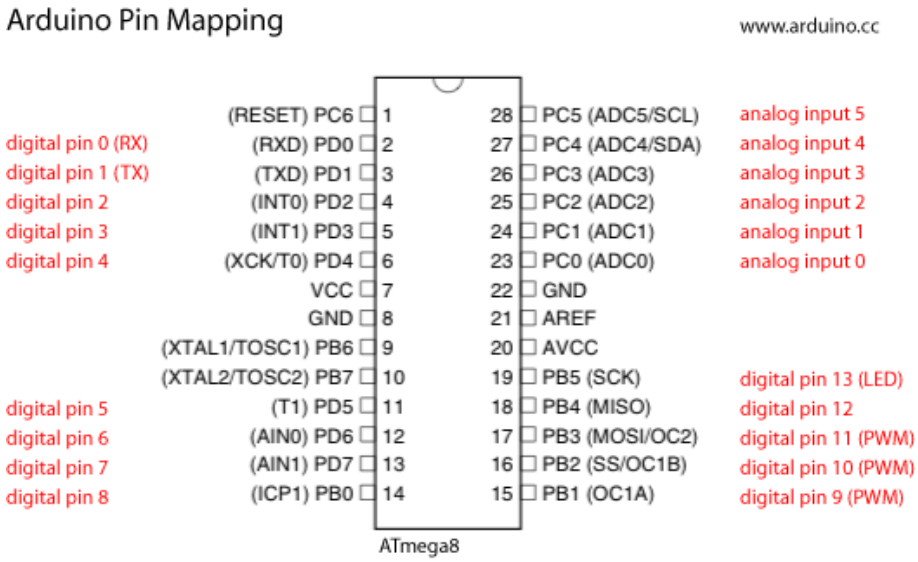


Abbildung 1.3: Arduino zu *ATmega8*Anschlusszuordnung

Nachdem wir zunächst vermutlich genug wissen um verantwortungsvoll zu handeln, programmieren wir unser 8 Byte grosses Programm, dass 'Adruino LED 13' erleuchtet.

```
; LED/S000_let-there-be-light.asm

.DEVICE atmega8

.org 0x0000
            rjmp    start

start:
            sbi     DDRB,       5
            sbi     PORTB,      5

main:
            rjmp    main
```

So einfach dieses Programm auch ist, es gibt doch ein paar Kleinigkeiten zu beleuchten.

Zunächst müssen wir angeben, welchen konkreten Micro Controller wir verwenden. Das ist erforderlich weil die verschiedenen Modelle verschiedene Adressen für ihre Elemente aufweisen. Dem Assembler wird auf diese Weise mitgeteilt, welche Werte er für welche Elemente verwenden muss. In unserem Beispiel für DDRB und PORTB. DDRB und PORTB sind Platzhalte oder

auch 'logische Namen' für Zahlen. Welche Zahlen verwendet werden, entscheidet die Microcontrollerangabe. Das geschieht durch:

```
.DEVICE atmega8
```

Als nächstes müssen wir den Anfang der Welt benennen. Der Witz hierbei ist, dass wir die volle Wahrheit nie wirklich erfahren! Wir verwenden Symbole um mit dieser Anforderung umzugehen. Wie bereits beschrieben, haben verschiedene Micro Controller verschiedene innere Werte. Aber nicht nur das. Wo exakt sich unser Programm am Ende tatsächlich befinden wird ist eine kaum beantwortbare Frage. Später werden wir nochmals darauf zurück kommen.

Da wir also zur Verwendung von Symbolen gezwungen sind, werden wir dementsprechend handeln. Wir werden mit einem Symbol den Startpunkt unseres Programms markieren. Dieses Symbol werden wir 'start' nennen. Was immer auch unser Programm starten wird, es muss diesen ominösen Startpunkt kennen:

```
.org 0x0000
            rjmp    start
```

Mit '`.org`' (bitte den Punkt am Anfang nicht vergessen) eröffnen wir eine Sequenz von Befehlen, die an der bezeichnete Position (hier 0x0000) beginnt. Diese Sequenz, die auch Tabelle genannt wird, ist eine Liste von Aktionen, die für bestimmte Anforderungen ausgeführt werden sollen. Für unser Programm besteht die einzige Anforderung darin, unser Programm zu starten. Und glücklicher Weise findet sich der Eintrag für diese Anforderung, also 'Starte das Programm', an der ersten Stelle diese Tabelle.

Für die Neugierigen unter uns: Die Adressierung innerhalb dieser Tabelle, ist immer relativ zum Anfang der Tabelle. Die Tabelle befindet sich in Wirklichkeit kaum an der Position `0x0000`, aber darauf müssender keine Rücksicht nehmen. Genau genommen betreten wir hier bereits eine Art Traumwelt: Wir wissen nicht wirklich, was passiert! Aber in manchen Fällen, wie hier, müssen wir das nur sehr selten wissen. Um unseren ersten 'Magischen Weltpunkt' zu bestimmen, formulieren wir lediglich:

```
start:
```

'`start:`' ist eine Marke, ein Label. In unserem Fall ist es eine Sprungmarke. Sie repräsentiert die Adresse der ersten Speicherstelle nach ihrem Erscheinen. In unserem Fall die Adresse des ersten Befehls unseres Programms.

Die nächste Sprungarke befindet sich bereits hinter dem eigentlichen Ende unseres Programms. Sie ist der Beginn einer unbedingten unendlichen Schleife. Diese Schleife ist erforderlich, weil der Prozessor (CPU) unseres Micro Controllers (MC) operiert, solange er Strom hat. Diese Aussage stimmt nicht, in Wirklichkeit kann man nicht nur die CPU anhalten. Das Anhalten der CPU ist aber bereits ein recht komplizierter Vorgang. Wichtig, aber kompliziert. Darum tue ich hier so als wäre es nicht möglich. Da wir die CPU also (momentan) nicht stoppen können, müssen wir ihr etwas zu tun geben was das Ergebnis unseres Programms nicht beeinträchtigt.

Zwischen '`start:`' und '`main:`' befindet sich momentan unser eigentliches Programm. Ich nenne das ein Programm der 'Ersten Form'. Ein solches Programm mag nur begrenzten Nutzen

haben, aber es ist ganz sicher nicht völlig sinnlos. Diese 'Erste Form' ist die Basis aller erweiterten Programmformen. Ein solches Programm

- startet
- tut etwas
- tritt in eine Endlosschleife, tut nichts mehr

Sofern das Programm im inneren der Endlosschleife etwas tut, nenne ich das die 'Zweite Form' eines Programms. Eine 'Dritte Form' darf für später erwartet werden. Sei es wie es sei, unser aktuelles Programm wurde speziell entworfen um einige wichtige Regeln guter MC Programmierung zu demonstrieren.

Die beiden Befehle, die die Aufgabe unseres Programms erfüllen tun das folgende:

- Bit 5 an PORTB als Ausgabepin festlegen
- Bit 5 an PORTB einschalten um die LED zu erleuchten

```
            sbi     DDRB,         5
            sbi     PORTB,        5
```

Am Ende die nicht enden wollende Schleife:

```
main:
            rjmp    main
```

Das ist alles was das Programm tut. Allerdings wirst Du feststellen, dass dieses Programm bedacht und sparsam operiert.

Ein PORT eines 8bit Micro Controllers verwaltet 8 Bits um 8 Beine in der Aussenwelt zu steuern. An unserem *ATmega8* kann jedes dieser Beine verwendet werden um eine der folgenden Funktionen zu erfüllen:

- Ein Signal ausgeben
- Ein Signal lesen, dass +5V oder GND ist als Repräsentation für 1 or 0
- Ein Signal lesen, dass 'nicht GND' oder GND ist als Repräsentation für 1 or 0

Jedes Bein kann unabhängig von jedem anderen Bein konfiguriert werden um eine dieser Operationen auszuführen. Alles am gleichen Port!

An einigen Beinen sind weitere Funktionen möglich wie

- Energiesparmodus (das Bein wird abgeschaltet)

- Pulsbreitenpodulierter Signalgenerator
- Analog-Digital-Wandlung
- Interruptempfang

In unserem Programm verwenden wir den Befehl `sbi` um das Bit 5 anzusteuern anstatt den Befehl `out` mit dem Parameter `1 « 5` zu verwenden, was auf den ersten Blick zum gleichen Ergebnis führen würde. Wir wollen aber einzig bit 5 ansteuern! `out 1 « 5` würde dummer Weise aber nicht nur Bit 5 auf `1` setzen, sondern gleichzeitig alle anderen Bits, ihrer 7 Stück, auf `0`!

Auch wenn wir - für den Augenblick - wissen, dass alle anderen Bits nicht benutzt sind, werden wir bald mit den eher typischen Situationen konfrontiert:

1. Wir werden zunehmen unsicher werden, welche Bits tatsächlich benutzt werden, während wir etwas bestimmtes programmieren. Auch und besonders wenn es sich um Bibliotheken handelt. Dann sowieso nicht!
2. Wir wissen nicht was passiert wenn wir `0` an Bits senden, die wir momentan gar nicht betrachten.

Ein wichtiges Konzept guten Programmierstils ist, so wenig wir irgend möglich zu tun.

Wenn nichts zu tun ist, dann tu's nicht! Das gilt besonders bei Micro Controllern, wo jede Aktion Energieverbrauch bedeutet. Wenn wir also ein Bit ansteuern wollen, sollten wir genau ein Bit ansteuern und nicht mehr, ausser wir haben gut Gründe, diese Regel nicht zu beachten. Momentan haben wir die nicht.

Wir wollen nicht Beine Aufwecken, wenn sie auch in Ruhe schlafen könnten. Denn möglicher Weise würden diese Anschlüsse alle zur Verfügung stehende Energie verheizen, im Sinn von heizen!

1.2 Mach' mich an, lass' mich aus

Zuerst die erweiterte Schaltung

Anschliessend das Programm

```
; LED/S002_get-me-on-get-me-off.asm

.DEVICE atmega8

.org 0x0000
            rjmp     start

start:
            sbi      DDRB,          5
            cbi      DDRB,          0
```

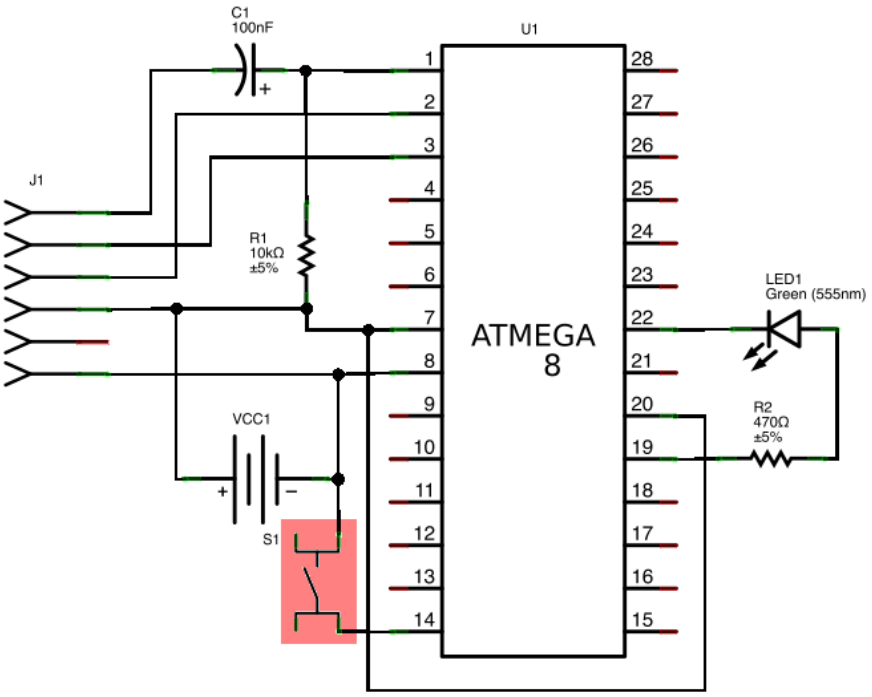


Abbildung 1.4: Mach' mich an, lass' mich aus - Schaltplan

```
            sbi     PORTB,      0

main:
            sbic    PINB,       0
            rjmp    led_on
            cbi     PORTB,      5
            rjmp    led_ok
led_on:
            sbi     PORTB,      5
led_ok:
            rjmp    main
```

Wie Du sicher schon vermutet hast ist dies bereits ein Programm der 'Zweiten Form'. Es tut nicht nur etwas *vor* der Endlosschleife, sondern ebenfalls etwas darin. Dieses Programm hat wirklich viel zu tun, wie noch sehen werden.

Der erste Unterschied zum vorherigen Programm ist, dass wir ein zusätzliches Bein benutzen, diesmal um ein Eingangssignal zu erkennen. Um dieses Bein, Bit 0 an Port B = Bein 14 am MC, auf den Input Modus zu schalten, senden wir eine `0` an das entsprechende 'Data Direction Register' (`DDR`). Ausserdem senden wir eine `1` auf das Bit am entsprechenden Port. Was im Ausgabemodus bedeuten würde 'Schalte +5V an das Bein', heisst im Eingabemodus: 'Schalte den pull-up-Widerstand[1] zu'.

[1]Ein pull-up-Widerstand ist ein Widerstand, der dazu benutzt wird, um den Signalpegel an einem Bein 'hoch' zu legen. Hier auf die Betriebsspannung, VCC = +5V = Logisch `1`. Er dient dazu um an einem Eingangsbein auch dann einen stabilen Zustand zu erreichen, wenn kein Element angeschlossen ist.

Danach können wir vom Eingangsbein den Status 'EIN' oder '1' lesen. Um eine Reaktion am MC zu erreichen und das Eingangssignal auf 'AUS' oder '0' zu setzen, müssen wir das Beim auf Masse legen.

Wenn wir das machen, schaltet unser Programm das Licht aus solange wir das Eingangsbein auf Masse legen. Das ist nicht direkt eine Anwendung mit der wir gross angeben können. Für uns ist es dennoch grossartig weil wir verstehen was passiert!

1. Initialisiere das System und die angeschlossenen Geräte
2. Lese Bit 0
3. Setze Bit 5 entsprechend
4. Weiter bei (2)

Du wirst vielleicht fragen wollen, wieso ständig das gleiche Signal auf das Ausgabebit ausgegeben wird. Wenn wir einmal grob annehmen, dass unser Programm das Eingangsbit ca. eine Million mal pro Sekunde liest (eher 8 Mio. mal), ist einzusehen, dass ein Mensch mit einem Tasten kaum 'Last' für unseren MC erzeugen kann. Wenn Du auf dem Schalter einhämmerst so schnell Du kannst, passiert aus der Sicht des Micro Controllers so gut wir gar nichts, das Signal ändert sich dann für den MC in historischen Zeiträumen.

Kann man das optimieren? Wir glauben nicht.

Gibt es andere Lösungen? Ja, die gibt es!

Wir könnten den zuletzt ausgegebenen Status speichern. Dann könnten wir in der nächsten Programmschleife prüfen ob sich der eingelesene Status zum vorherigen Status geändert hat und nur dann ein Signal an das Bein ausgeben, wenn eine solche Änderung vorliegt.

Das klingt einfach, ist es aber leider nicht! Es ist nicht nur nicht einfach, es ist gefährlich, kostspielig und kompliziert. Und es ist effektiv sinnlos weil wir eine Menge Aufwand treiben würden, ohne dass sich etwas verbessert.

Das Konzept ist gefährlich weil das Programm aus dem Rhythmus kommen könnte. Danach würde es falsch herum arbeiten oder überhaupt nicht mehr auf Eingangssignale reagieren.

Es ist kostspielig weil das Programm nicht nur viel grösser würde, wir würden ausserdem ein CPU Register verbrauchen (um den Status zu speichern) und wir haben total nur 32 Stück!

Und es ist kompliziert weil wir zwei unabhängige Einheiten zusammenhalten müssen (das Licht und das Statusregister) um nur einen Effekt zu erzielen. Das ist ein grosses Risiko und ein Nachteil gegenüber der vorliegenden Lösung.

Deswegen liegen wir vielleicht nicht falsch mit der Vermutung, dass die Entwickler unseres *ATmega8*Micro Controllers ihren Chip so entwickelt haben, dass in Wirklichkeit gar nichts gemacht wird, wenn wir eine '1' auf ein Steuerbit senden, dass bereit auf '1' gesetzt ist.

Ein anderer Vorloche könnte sein, das Programm anzuhalten solange wir auf ein Eingangssignal warten. Das sollte möglich sein, liegt aber momentan weiter über unseren Kenntnissen.

Es gibt keinen Befehl, der einen MC veranlasst, auf ein Eingangssignal zu warten. Aber es ist durchaus möglich, ein Programm zu entwickeln, dass so wirkt als gäbe es einen solchen.

das heisst, momentan haben wir keine Wahl und müssen bei der vorliegenden Lösung bleiben. Aber bald werden wir auch ins Reich der Statusverwaltung vordringen.

1.3 Stable Decisions

For the next step we need to do something more useful. But we stay with the 'second form of a standard program': A program that does something in its infinite loop.

This program stores one of two states ;-) and sets the light on or off, keeping it according to the stored status. This explanation is slightly wrong. I try it again.

This program recognises an input signal consisting of two phases. A complete input signal consist of a LOW phase starting from a HIGH phase followed by the next HIGH phase. The signal ends with entering the HIGH phase again. We are interested only in change, as we always should.

If the Input signal changes from HIGH to LOW, our program changes the light from its current status to the opposite status. If the light was ON it will be set to OFF and vice versa. The status is only changed as reaction of changing the Input from HIGH to LOW because 'no action' at the Input device leads to HIGH status in the input bit as result of using an internal pull-up resistor who does what he is called - he pulls the input signal up to HIGH.

The electrical signal we are waiting for with our micro controller on our input bit is: Pulling the status down to LOW (GND). We may phrase:

The *Signal* we are waiting for is the *Change* form HIGH to LOW.

So for the first time we have to be aware of a dynamic process. If the change of the status is the signal, then we have to recognise if the status is change. If a status appears after the opposite status was recognised - in the past - which is then no more present (!) we have to deal with 'historical' status management and so for the first time we remember a status from one processing cycle to the next.

Finally this here is the Code to do it:

```
; LED/S004_stable-decisions.asm

.DEVICE atmega8

.org 0x0000
            rjmp    start

start:
            sbi     DDRB,         5
```

```
            cbi     DDRB,           0
            sbi     PORTB,          0

            ldi     r16,            1

main:
            sbic    PINB,           0
            rjmp    led_keep
            tst     r16
            breq    led_ok
            clr     r16
            sbis    PINB,           5
            rjmp    led_on
            cbi     PORTB,          5
            rjmp    led_ok
led_on:
            sbi     PORTB,          5
            rjmp    led_ok
led_keep:
            ldi     r16,            1
led_ok:
            rjmp    main
```

As you can see, this Code is not easy to understand. To do better, we add symbolic names to the soup. The basics are easy:

- `.equ` means: 'a name for a value'
- `.def` means: 'a name for an entity'

So for example `DDRB` already is a number. This number is defined in an include file chosen by you device selection. In our case, whatever number is hidden behind the name DDRB it will be our Input/Output control port. So the name will be `ctl` as prefix for 'control' and `IO` short for Input&Output.

In another sample `bit` stands for 'bit number' and `Input` for 'Input' which makes `bitInput`, the bit where we read the input status.

You may define you own naming convention which better should hold throughout your project.

```
; LED/S005_stable-decisions+symbols.asm

.DEVICE atmega8

.equ ctlIO     = DDRB    ; DDRB  is our I/O control register
.equ prtIO     = PORTB   ; PORTB is our I/O output port register
.equ pinIO     = PINB    ; PINB  is our I/O input pin register

.equ bitOutput = 5       ; bit 5 is our output bit
.equ bitInput  = 0       ; bit 0 is our input bit

.equ FALSE     = 0       ; 0 will be FALSE or OFF
.equ TRUE      = 1       ; 1 will be TRUE  or ON

.def bStatus   = r16     ; the last state will be stored in 'r16'
```

As you may not have expected, this makes the soup - or code - somehow better readable and much easier to understand. Now it looks more like a higher language:

```
.org 0x0000
            rjmp    start

start:
            sbi     ctlIO,          bitOutput
            cbi     ctlIO,          bitInput
            sbi     prtIO,          bitInput

            ldi     bStatus,        HIGH

main:
            sbic    pinIO,          bitInput
            rjmp    led_keep
            tst     bStatus
            breq    led_ok
            clr     bStatus
            sbis    pinIO,          bitOutput
            rjmp    led_on
            cbi     prtIO,          bitOutput
            rjmp    led_ok
led_on:
            sbi     prtIO,          bitOutput
            rjmp    led_ok
led_keep:
            ldi     bStatus,        HIGH
led_ok:
            rjmp    main
```

Also it provides us with the possibility of changing things with reduced risk. Shortly we had to change the schema a bit to support additional ideas and we changed `bitInput` from 4 to 0. Using symbolic names this was much easier and much less risky to do because we only needed to change the definition for `bitInput`!

Even if symbols make a better reading than constants, it seems not really to be easy to follow the program flow. So at first, we should introduce a program flow chart. And for good measure two of them. We need two of them to demonstrate a major point in assembler programming.

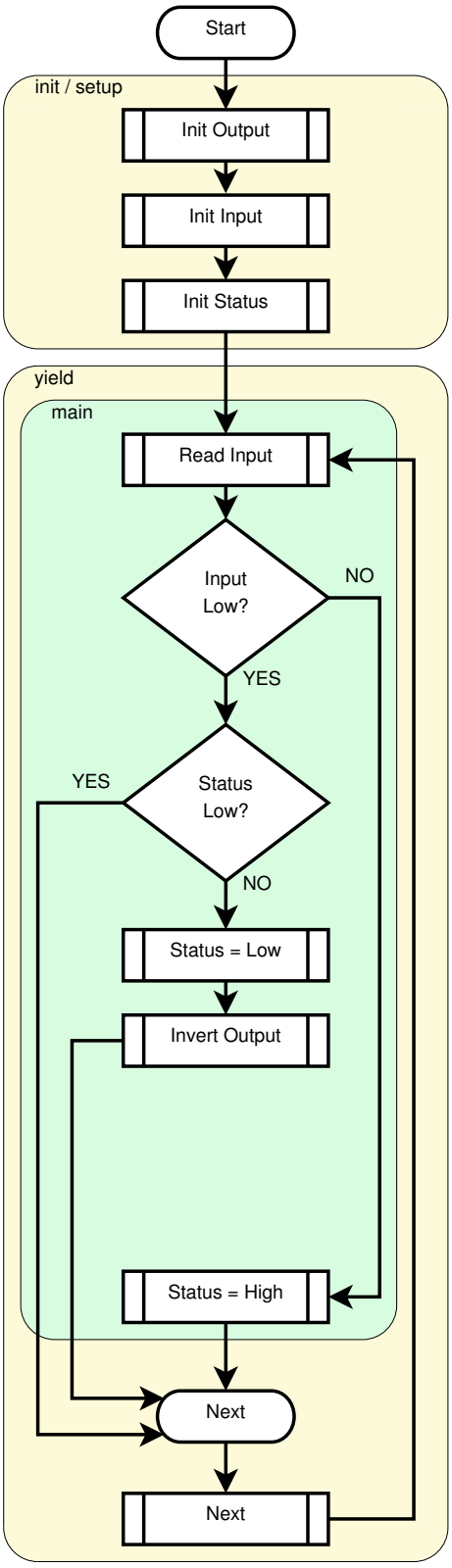
We have to take watch about WHAT we wish to do, but equally to about HOW we are going to do it.

1.3.1 WHAT to do

1. Initialise system and devices
2. Wait for the change ïnput bit was HIGH and became LOW"
3. Invert LED status
4. Restart at (2)

1.3.2 HOW to do it

To get an impression on how to it, at first, we take a look at a flow diagram. This diagram shows the program flow.




```
start:
        sbi     ctlIO,   bitOutput

        cbi     ctlIO,   bitInput
        sbi     prtIO,   bitInput

        ldi     bStatus, HIGH

main:

        sbic    pinIO,   bitInput
        rjmp    led_keep

        tst     bStatus
        breq    led_ok

        clr     bStatus

        sbis    pinIO,   bitOutput
        rjmp    led_on
        cbi     prtIO,   bitOutput
        rjmp    led_ok
led_on:
        sbi     prtIO,   bitOutput
        rjmp    led_ok

led_keep:
        ldi     bStatus, HIGH

led_ok:

        rjmp    main
```

Abbildung 1.5: Stable Decisions - Flow Diagram

As you may have found out until now, dealing with the 'human device interface' is the most complicated thing in informatics. It starts with the unbearable slowness of bio entities and does not end with humans expectations against machines. Which means the user is problem number one in software development and for this has to be the main focus! Otherwise your product will not be accepted by those who should pay our wages!

1.3.2.1 Slow users

A living entity presses a button and our device has to react. Our device may ask the input pin about 1.000.000 times per second. A human for example may do a very short pressing of a button in about 0.2 seconds. Which means, our System reads the status 'button pressed' about 200.000 times during our humans action. But what the human expects is:

Change the light ones as I press (in my view) the button ones!

A problem we will not deal with in this phase of our development is, that electric switches may flicker during status change. Flickery leads to problems because it will generate random additional signals during the action phase. In general this is dealt with by electronically measures. Therefore we simply have to ensure that changing the LEDs status only happens ones per button press action and we will ignore any flickery event by not doing anything about it.

1.3.2.2 Fast feedback

Our feedback device is the light we control. We have no display oder acoustic device to us as feedback but the very same object that goes to be controlled. User feedback therefore simply means to manipulate the status of the light. Even if its cool in movies to do anything without any visible feedback, real living bio entities require 'immediate feedback' for every action. Otherwise they run into problems.

Figure 1.6 shows the signal we have to expect (idealised). You have to remember, that our micro controller reads the 'LOW' status of the signal 100.000 times or more often. We do not measure time! The natural timing unit in micro controllers is cycles. Read cycles, CPU cycles, sensor cycles. This is important to recognise especially because different clock frequencies and different voltage leads to different physical cycle durations. A program that measures a certain event by 1000 cycles will receive 2000 if the clock is doubled up.

Different clock frequencies lead to different CPU dependent cycles. Different voltage may lead to different sensor cycles, depending on the specification of the sensor in question and the characteristics of the signal to measure. Further, different voltage may lead to different absolute sensor signals, but this is another chapter.

The signal we need to recognise at the moment is the switch from HIGH to LOW and back again. The LOW level will stay for an undefined time / amount of measure cycles. What we have to focus on is the change of the signal!

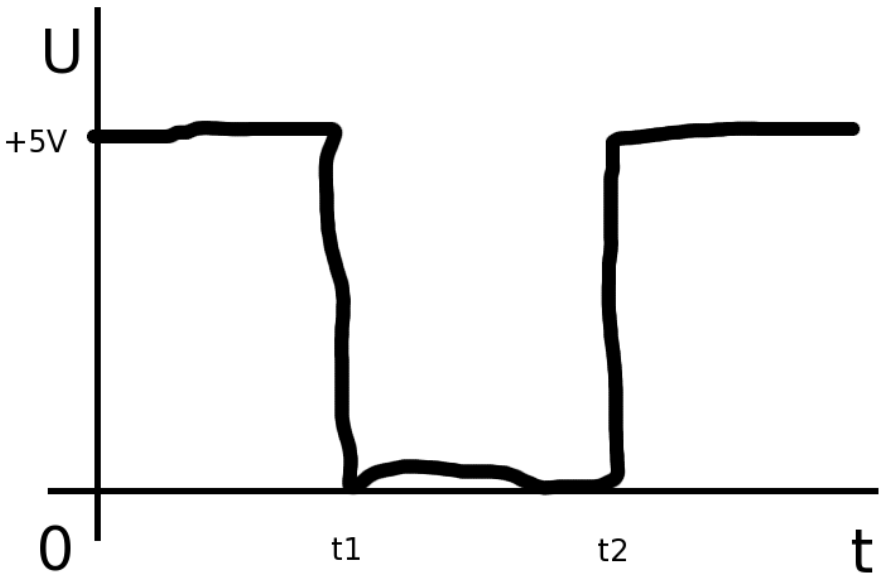


Abbildung 1.6: Stable Decisions - Signal Diagram

To deal with this there are only two moments in all this 'endless' button down phase - between 't1' and 't2' - where it is applicable to really change the LEDs status. At the beginning, as the signal changes from HIGH to LOW (t1) or at the end, where the signal changes from LOW to HIGH again (t2).

To give the human who uses our device (the user) immediate feedback about success or failure of his action, we decide to use the first phase (t1) to do all the action. So we have a simple mission: If the signal is LOW now and was HIGH the last time we looked, we change the LEDs state. This satisfies also the endless LOW state of the input signal and the change from LOW to HIGH where we have to do nothing. Not even accidentally! This way, the human feels his request immediately answered, even if he is unable to comprehend on which grade immediately it real is answered.

1.3.3 Finding the event

To find the event the signal has gone to ground, we have to be aware of the status of the signal from the previous query cycle. Which is easy but expensive. We simply use a register to store the previous input status.

We simply have to check if the last seen status was HIGH as log as the current status is LOW. The status accumulator register follows the status of the input signal after being used to query the change event.

If the one moment we are waiting for passes, we invert the LED status. That's all

1.3.4 And some modesty

In respect of things to come, we have to be modest in our style. The 'endless loop' which keeps the micro controller running and waiting without going wild, will be used for some important things:

- It mostly contains the main program
- It possibly consist of multiple parts
- It consist of an undefined amount of functional separate program parts

So we have to ensure, to don't make any shortcut back to the 'main' label. As there can only be one Highlander, there can only be one jump back to 'main'. This jump resides at the very end of the main sequence. Like this:

```
main:

  A_begin:
            block     A or goto to end of block A
  A_end:

  B_begin:
            block     B or goto to end of block B
  B_end:

  C_begin:
            block     C or goto to end of block C
  C_end

mein_end:
            finalise
            rjmp      main
```

In our flow diagram one part has to reach the next part regardless of how the program is flowing. The rounded rectangle 'next' represents the central point behind our (currently) one part. This is the position where the next program part would follow in the same manner.

1.4 Stable Decisions Triggered

If you feel a bit uncertain about our solution or if you have the feeling that all this should be done better you may be right. We would not ensure you that the following solution is better under every circumstance, but for some application, there is a much better way.

For this way we need to introduce the concept of interrupts.

The magic behind interrupts in micro controllers is much more as in simple CPUs!

Interrupts are special mechanics in processing units to enable the execution of code at a certain time or after a certain event by interrupting the normal processing, doing something else and after this continue whatever was interrupted.

In micro processors this mechanic is much more sophisticated. In micro controllers you may interrupt the system from nothing! Meaning, it is possible to nearly shut off the whole system, interrupt it from his deep sleep, let it do something and send it back to sleep again.

Such applications are useful if energy resources are small. For example, if you wish to drive your weather station one year on a single AAA cell or less, possibly supported by solar power.

In such applications you wish to reduce power consumption of your system as far as possible. There is no need to let your micro controller do eight million cycles per second if you take a measurement every ten minutes! It may be much better so stop the whole system until the next measurement is to start.

You already may expect it. Such code does not need to loop with full processing power to do nothing, such code really does nothing while waiting:

```
; LED/S005_stable-decisions-trigger.asm

.DEVICE atmega8

.org 0x0000
            rjmp    start
            rjmp    trigger0

start:
            ...

main:
            sleep
            rjmp    main

trigger0:
            do_it
            reti
```

This approach also has the advantage to don't need the otherwise necessary status accumulator register and consequently no comparison between former and current status. It simple needs to change the current status of the light if called.

1.5 Light Shift

Next we will start with showing off. A little bit at least. We will put three lights in a row and lighten one of them up. Each time we press the button the light will shift to the next LED. After the last light, we start with the first one.

It would be much easier to do this with 8 lights, but here we are. We will keep the existing circuit as untouched as possible and, not at last, we are constantly on he search for a challenge.

So here is the enhanced circuit:

As you can see, we added the three LEDs to the remaining pins on the right side of the chip, so it is easy to build in Hardware on a breadboard. For this we have to pay with a higher degree of complexity in our code

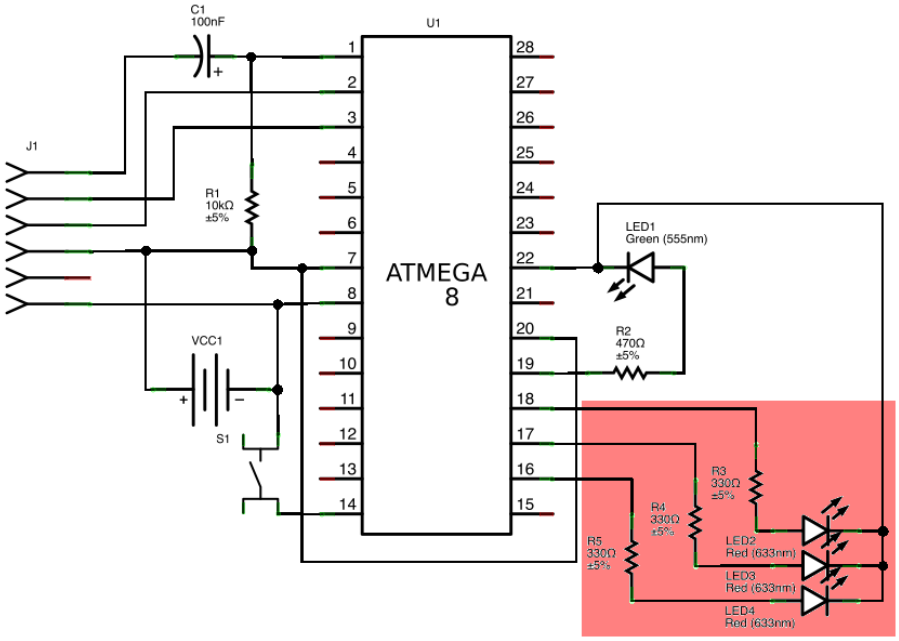


Abbildung 1.7: Light Shift - Schema

And this is the code:

```
; LED/S010_light-shift.asm

.DEVICE atmega8


.equ ctlIO          = DDRB
.equ prtIO          = PORTB
.equ pinIO          = PINB

.equ bitSignal      = 5
.equ bitInput       = 4
.equ bitLightStart  = 3

.equ mskLightShift  = 0x0E

.equ LOW            = 0
.equ HIGH           = 1

.def bStatus        = r16
.def bTemp          = r17
.def bData          = r18


.org 0x0000
            rjmp    start


start:
            ldi     bTemp,      mskLightShift | 1 << bitSignal
            out     ctlIO,      bTemp
            ldi     bTemp,      1 << bitInput | 1 << bitSignal
            out     prtIO,      bTemp
```

```
            ldi     bStatus,        HIGH

main:
            sbic    pinIO,          bitInput
            rjmp    led_keep
            tst     bStatus
            breq    led_ok
            clr     bStatus

            in      bData,          pinIO
            mov     bTemp,          bData
            andi    bData,          0xFF - mskLightShift
            ori     bData,          1 << bitInput

            andi    bTemp,          mskLightShift
            lsr     bTemp
            andi    bTemp,          mskLightShift
            brne    shift_ok
            ldi     bTemp,          1 << bitLightStart
shift_ok:
            or      bData,          bTemp
            out     prtIO,          bData

            rjmp    led_ok
led_keep:
            ldi     bStatus,        HIGH
led_ok:
            rjmp    main
```

As you can see, we have some changes to the former code. This is, in the hole, because we have to deal with four instead of one lights. One light (the one we toggled last time) will serve as signal light. It is lighting up after our micro controller has finished booting. The other three LEDs will be used to shift the light. In our case this means we start off with one of the three lights ON and each time we press our button the next light lights up whilst the former light shuts off.

Our code reflects the higher complexity at the first look with more constants and more named registers. Already we are using 10% of all available registers and 20% of all registers usable for generic purpose. This is bad news! Hopefully our resource consumption will not grow with constant speed.

Names for constants (not resources) we need besides the ones from the former sample are

```
.equ bitLightStart = 3     ; as in  0b00001000

.equ mskLightShift = 0x0E  ; builds 0b00001110
```

`bitLightStart` is the number of the pin on the output port where the LED light of our light chain resides. A bit will be shifted this time to the left. The other LEDs are expected to use the two lower pins (2 and 1). Pin B0 on the *ATmega8*can be found on the other side of the chips, so for our physical breadboard circuit it's a bit too far away to use it.

Please remember, such decisions are no fun. In reality you may sometimes be driven to compensate in software for simplification of hardware as PCB layout, different chip pinouts and so on. So we accept this situation as example of what may happen in real life.

`mskLightShift` is a bit mask containing bits at all pins of the output port where our light chain is connected to. We need such a bit mask because we need to shift the 'LED ON' bit around but

won't shift any other bits found in the port data. So we use this bit mask to

- mask out the relevant light chain bits from the rest
- mask out a too far shifted 'LED ON' bit (logical overflow)
- mask out the light chain bits in input data after querying our PORT

Named registers (resources) we need in addition to the former program are:

```
.def bTemp          = r17
.def bData          = r18
```

`bTemp` will be used to shift our 'LED ON' bit around but `bData` has to give us the 'big picture' about our ports status. First before, second after shifting the 'LED ON' bit.

We start, as usual, by initialising the micro controller where it needs to be initialised. This time, we need to set all pins on our port the same time because the alternative is (for the moment) much too expensive. So we build a bit mask to send it to our IO ports data direction register (DDR):

`mskLightShift` combined with `1 « bitSignal` makes `0b00101110`. Sending this byte to DDRx will set pins 1, 2, 3, 5 to output and all other pins to input mode.

```
start:
            ldi     bTemp,          mskLightShift | 1 << bitSignal
            out     ctlIO,          bTemp
            ldi     bTemp,          1 << bitInput | 1 << bitSignal
            out     prtIO,          bTemp
```

Then we need to do three things by sending a byte to PORTx:

- Switch ON LED on pin 5 (`bitSignal`)
- Set pin 4, our input pin, to 'pulled up' (`bitInput`)
- Set all other pins to off/offline

All this is done by mixing all active bits together and sending the mix `(0b00110000)` to the port. After doing so the signal LED is on, the light chain is off. Until `clr bStatus` nothings more has changed. Also the rest with and after label `led_keep` is kept the same.

The changed part is this:

```
            in      bData,          pinIO
            mov     bTemp,          bData
            andi    bData,          0xFF - mskLightShift
            ori     bData,          1 << bitInput

            andi    bTemp,          mskLightShift
            lsr     bTemp
            andi    bTemp,          mskLightShift
            brne    shift_ok
            ldi     bTemp,          1 << bitLightStart
shift_ok:
            or      bData,          bTemp
            out     prtIO,          bData
```

All magic resides in these lines. It's not too much, so we simply explain it sequentially.

We need to get the ports state and unfortunately we have to understand more about this than is good for the program. We have to send back the whole data byte, modified only in the part where the LED chain state changed. BUT (!) we also need to ensure, our input pin keeps his pull up resistor active. Otherwise we loos our connection to the outer world!

Next we need to split the read byte/data in two parts. The one containing constant data and the other one containing the bits to be shifted.

Phase one therefore copies `bData` to `bTemp`, then masks out all bits dealt with in the bit shifting process and finally adding/regenerating the 'pull up' bit for port `bitInput`.

Phase two masks out all bits not related to the bis shifting operation, shifts the remaining bit and masks the result again with the same mask. If this operation leads to a result of ZERO/EQ/NULL, then we have shifted out our bit and need to insert a new one at the start position inside the byte where the bit starts its decent, at: `bitLightStart`.

Phase three then combines our static bits with our manipulated one and output all together to our combined input/output port.

Kapitel 2

Time

2.1 Instable Elements

The longest time we tried to avoid the simplest demo in most Arduino beginners sets. The blinking light demo! Some of our readers may have ask themselves where the problem should be. Now is the moment to explain.

We do not want to make a fuss with code we would have to be ashamed of but couldn't bring us to start with timers and interrupts before introducing the most basic principles of programming. Please remember, using assembler language brings us in a position of power which forces us into reliability.

There is no way a reliable programmer would use 4.000.000 NOPs ('no operation' operations) twice to let a light blink ones per second. Also, we hope, no one reading until this point would keep it as responsible to enter a loop, busying our poor micro controller to wait half a second by wasting four million CPU cycles.

So we had to experiment enough to enter the reign of timers and interrupts. Staying our ground not demonstrating bad code, keeping the book pure, we don't show only a part of a single bad example in code. You may remember it in your dreams!

To cut a long story short. Here its the code:

```
; LED/S020_instable-elements.asm

.DEVICE atmega8

.org 0x0000
            rjmp    start

start:
            ...
main:
            rjmp    main
```

Teil II

External devices

Kapitel 3

Shift Registers

3.1 SRAM to Shift Register

Sending SRAM content/data to shift registers has some applications. Some of them are

- Light chains
- Raster displays