Cookbook for Assembly Programming with Arduino and plain 8bit Atmel AVR Micro Controllers

Felix Morgner and Manfred Morgner

February 6, 2012

This book is ongoing work. If you find errors, typos, other mistakes, please get in contact with us. We are always happy to be shown better ways, clearer views, faster solutions, lesser errors ... you name it.

If you wish to follow our ways, it would be a good idea to get a copy of ‚ATMEL 8bit AVR instruction set manual':

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

Herein we will not explain the commands we are using because all explanation is already worded out in the named document. Clearly we will discuss why we are using specific commands if there are known (as in the context of the book) alternatives.

This book is mainly based on *ATmega8* because we need to push aside micro controller model specifics. At some point we have to start dealing with this problem but we try to prevent this from happening as hard as we possibly can.

We are more destined to show what can be made with a small micro controller. It's clear for us, that more powerful platforms are available to a similar price. But our believe is, that learning Assembler Programming leaves deeper traces if the result of our work is amazing in face of the used hardware, like a music instrument out of a micro controller some wire and some resistors.

This book is about Assembler Programming. We do not share the view, that Assembler Programming will rescue the world. Each programming language has its purpose and some of them are useful too. The reader of this book wishes to learn Assembler Programming by example and has a tendency to experiment with the real thing. We try to satisfy this kind of reader by keeping the circuits simple and price and requirements low.

Contents

List of Figures

Part I

Simple Samples

Chapter 1

Light

In this chapter we will demonstrate the basics of assembly programming with AVR Micro Controllers. These samples will only require the simplest of additional hardware. If you use an Arduino, you will need only a simple wire for the first examples. For the first example nothings more than a powered Arduino.

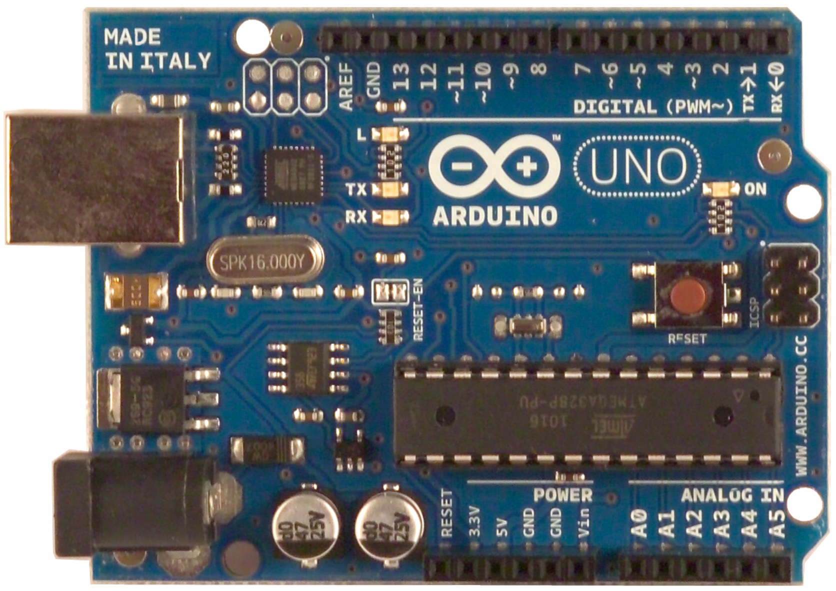


Figure 1.1: Arduino Uno

If not, you may build a circuit like in figure 1.2 on page 10. The breadboard layout is shown in file `LED/S000_LED-Basic-Circuit.fz` in Fritzing file format.

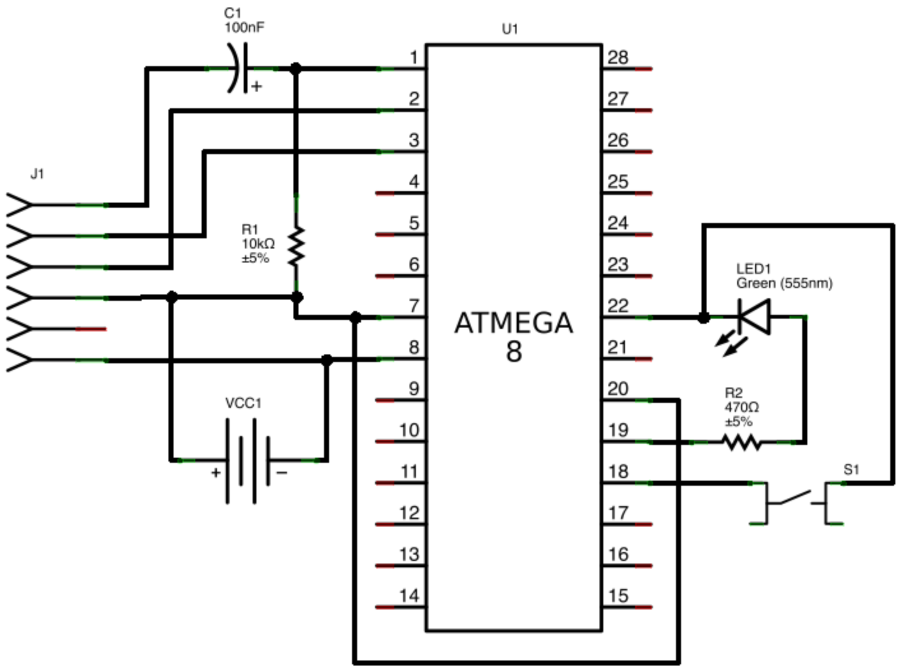


Figure 1.2: Basic Schema

1.1 Let there be light

The first sample in this chapter is the smallest program I can think of which does something. This program will switch on the LED on your Arduino on pin 13.

Programming in Assembler means you're in control. But as you know, power requires knowledge and responsibility. If you're entering the world of assembly programming, you have absolute power if you wish to or not. Consequently you need both, knowledge to become responsible.

At first you need to know what 'Arduino pin 13' really means! As consequence of the Arduino design it is not pin 13 on your *ATmega8* . Knowing this is yet the half way thru. Knowing the pin is a step you need to know if are working with a plain chip. What you need to know is the MC internal addressing for Arduino pin 13. To find out, look at `http://arduino.cc/hu/Hacking/PinMapping`. After we know everything we have to know to be responsible, we generate our 8byte machine program that will set our 'LED 13' under power.

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

.DEVICE atmega8

.org 0x0000
            rjmp    start

start:
            sbi     DDRB,         5
            sbi     PORTB,       5

main:
            rjmp    main
```

As simple as the program is, I believe there is some need for explanation.

At first we have to declare the type of micro controller we are intuited to use with the program. This is necessary because different micro controllers do have different assignments in their inner structure and need different addressing for their components. We do this by:

```
.DEVICE atmega8
```

Next we need to declare where our world will start. Funny thing is, we done really know! So we are forced to use symbols to deal with this necessity. As indicated below, different micro controllers will have different inner values. But not only this, to be honest, where our program lives is due to additional effect a most uncertain thing. We come to this later in the book.

As we are forced to use symbols, we have to do so. We will set a symbol to our programs starting point later and this symbol will be named 'start' in out code. Whatever starts our program it must be informed where to goto to do so:

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

With '.org' (don't forget the leading dot!) we build up a sequence of command positioned at the addressed position. This sequence, some times named a table, is a list of action to be taken for different requests. The only request we have is to start our program and fortunately the entry for this request is expected as the first in our table.

For those who really want to know: The addressing in this table is relative to wherever it is placed in real life! So it always starts with 0. Strictly speaking, at this point we already enter the realms of a dreamworld. We don't know what really happens! But sometimes this is irrelevant

To declare our first magic point, we only need to postulate:

```
start:
```

'start:' is a label and represent the final address for the first memory position used afterwards. In our case the address of the first command in our program.

The next label is already behind all things we need to do in our program. It is the starting point of an unconditional infinite loop. This sequence is necessary because the processor (CPU) of our MC is running as long as it has power. We can't stop it, so we have to lead it into a controlled way of doing nothing because we don't wish that our MC does anything after it did all things we expected it to do.

Between 'start:' und 'main:' is our program. I call this the 'first form of a standard program'. A program which runs ones after the system awakes. Such program may be of limited use, but not completely useless. And the schema of this 'first form' is the basic schema of all derived form. The program

- starts
- does something
- loops forever, possibly doing something

If the program does something in the 'infinite loop' then this may be called the 'second form of a standard program'. A third form should be expected to pop into existence later on. But anyway. Our program of the first form is specifically designed to show some important rules for good programming.

The two commands which full fill our programs mission will do two things:

- declare pin 5 at PORTB as output pin
- set pin 5 at PORTB under power to enlighten our LED

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

Finally the never ending loop:

```
main:
            rjmp    main
```

This is all the program does and there is nothing more about it. You will discover, that this program demonstrates prudence and thrift. But for the moment, our knowledge is insufficient to explain it.

1.2 Get me on - Get me off

At first the code

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

.DEVICE atmega8

.org 0x0000
           rjmp     start

start:
            sbi     DDRB,          5
            cbi     DDRB,          4
            sbi     PORTB,         4

main:
            sbic    PINB,          4
            rjmp    led_on
            cbi     PORTB,        5
            rjmp    led_ok
led_on:
            sbi     PORTB,        5
led_ok:
            rjmp    main
```

1.3 Stable Decisions

Next step we need to do something more useful. And we reach the 'second form of a standard program', which means: A program that does something in it infinite loop.

This program stores one of two states and sets the LED on or off, keeping it according to the stored state. 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 followed by a HIGH phase. The signal ends with entering the HIGH phase. We are interested only in change, as we always should.

If the Input signal changes from HIGH to LOW, our program changes the LED from its current state to the other state. If the LED was ON it will be set to OFF and vice versa. The state is only changed as reaction of changing the Input state from HIGH to LOW because 'no action' at the Input device leads to HIGH state in the input register as the 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 pin is: Pulling it down to LOW (GND). And finally this is the Code to do it:

```
; LED/S004_stable-decisions.asm

.DEVICE atmega8

.org 0x0000
            rjmp    start

start:
            sbi     DDRB,         5
            cbi     DDRB,         4
            sbi     PORTB,        4

            ldi     r16,          1

main:
            sbic    PINB,         4
            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 number'
- `.def` means: 'a name for an entity'

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

In another example `bit` stands for 'bit number' and `Input` for 'Input bit' which makes `bitInput`, the bit where we red the input state.

You may define you own naming convention which 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       ; pin 5 is our output bit
.equ bitInput  = 4       ; pin 4 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
```

As you may not have expected, this makes the soup - or code - somehow better readable and so 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
```

Even if it's a better reading, it seems no 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 the. 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 too about HOW we are going to do it.

1.3.1 WHAT to do

- Initialise system and devices
- Wait for the change "input pin was HIGH and became LOW"
- Invert LED status
- 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 outside the 'invert LED status' part. Changing the LED status currently is out of our focus. It is the very thing we try to deal with, but it is the more easy part of the solution.

As you may found out until now, dealing with the 'human device interface' is the most complicated thing in informatics. It starts by the unbearable slowness of humans and does not end with humans expectations against machines.

At the moment out project restricts the 'human device problem' the unbearable slowness of humans.

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 to a short pressing of the button in about 0.2 seconds. Which means, our System reads the state 'button pressed' about 500.000 times during our humans action. But what the human expects is:

"Change the LED status ones as I press (in my view) the button ones!"

A second problem we will not deal with in this phase of our development is, that electric switches may flicker during status change.

So we simply have to ensure, that changing the LEDs status only happens ones per button press action. There are only two moments in all this endless button down phase where it is suitable to really change the LEDs status. At the beginning, as the signal changes from HIGH to LOW or at the end, where the signal changes from LOW to HIGH again.

To give the human who uses our device immediate feedback about success or failure of his action, we decide to use the first phase 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 how immediately it real is answered.

1.3.3 Finding the right moment

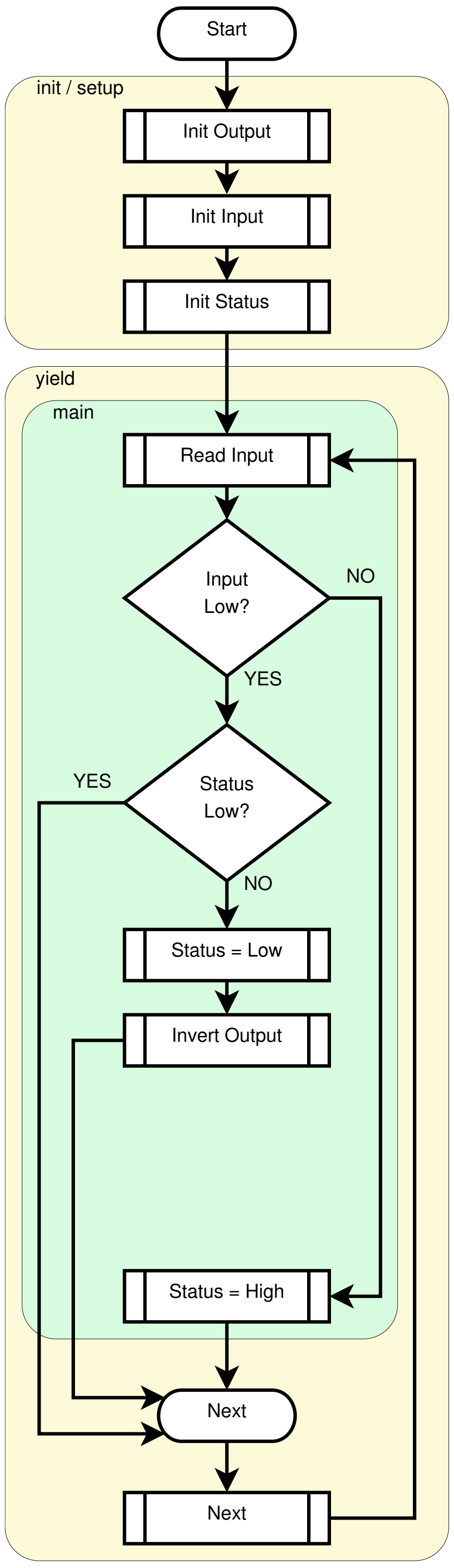
To find the moment the signal has gone to ground, we have to be aware of the status of the signal from the previous query cycle. Which is easy but expensive. We simply use a register to store the previous input status.

So 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:



```
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
```

Figure 1.3: Stable Decisions Flow Diagram

- It mostly contains the main program
- It possibly consist of multiple parts to do
- It consist of an unknown amount of 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.

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

1.4 Light Shift

Next we will start with showing off. A little bitt at least. We will put three lights in a row and lighten one of them up each time we press the button.

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 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 will deal with four instead of one lights. One light (the one we toggled last time) will serve as signal light, lighting up after out micro controller has finished booting. The other will used to shift the light. In our case, this means, we start off with one light ON and each time we press our button the next light lights up whilst the former active light shuts off.

Our code reflects this at the first look with more constants and more named registers. Already we are using 10% of all registers present 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 values (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 first LED of our LED chain resides. 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 complicated 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 chips and so on. So we accept this situation as example of what may happen in real life.

`mskLightShift` is a bit mask containing bits ate 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.

Named registers (resources) we need in addition 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) not acceptable. 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 dealer 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 bit at the start position inside the byte `bitLightStart`.

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

Chapter 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
```

Part II

External devices

Chapter 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

It is an important way to communicate with the technical environment in certain situations. Most importantly if you have more bit to output as pins on your micro controller.